Ethereum
Proof of Stake:
Casper FFG 2017 Chronicles
1. Casper Overview
2. The Last Year of Development
3. Building Your Own Protocol
1. Starting with Proof of Work

Proof of work currently secures the Ethereum blockchain
How does PoW Work?

Ethereum miner

Miner mining on top of genesis block

Miner receives transactions from normal users

Time 1
How does PoW Work?

Miner receives block reward

But mining the block costs lots of energy
How does PoW Work?

Time 1

Miner receives transactions from normal users
How does PoW Work?
How does PoW Work?

Current state of Ethereum
How does PoW Work?

New miner larger arrives
How does PoW Work?

A fork has occurred!
How does PoW Work?
How does PoW Work?
How does PoW Work?

Time 1

Block not included in Ethereum history.

Note: The miner doesn’t receive their block reward.
How does PoW Work?

Small miner gives up on their fork, and instead mines the longest known chain.
How does PoW Work?

Small miner finds the next block, and this time receives block reward.
How does PoW Work?

Mining on shorter forks does not provide block rewards.

Notice the incentive to mine on the longest chain.
How does PoW Work?

Another miner joins

Time 1
How does PoW Work?

Time 1
How does PoW Work?
How does PoW Work?
How does PoW Work?

Yay 🌿

Proof of work... works!
Not so fast!

Proof of work isn’t perfect...
2. Problems with Proof of Work

- Wasted energy
- Vulnerable to ASICs and centralization
- Lacks "finality"
- No clearly defined validator set
Problem 1: Bad for the Environment

PoW mining already burns more energy than 160 countries... Insane.
Problem 2: Vulnerable to ASICs & Miner Centralization

ASICs and miner centralization make chain reversions relatively low cost
Problem 2: Vulnerable to ASICs & Miner Centralization

ASICs and miner centralization make chain reversions relatively low cost
Problem 2: Vulnerable to ASICs & Miner Centralization

ASICs and miner centralization make chain reversions relatively low cost.
Problem 2: Vulnerable to ASICs & Miner Centralization

ASICs and miner centralization make chain reversions relatively low cost
Problem 2: Vulnerable to ASICs & Miner Centralization

ASICs and miner centralization make chain reversions relatively low cost
Problem 2: Vulnerable to ASICs & Miner Centralization

ASICs and miner centralization make chain reversions relatively low cost
Problem 2: Vulnerable to ASICs & Miner Centralization

ASICs and miner centralization make chain reversions relatively low cost
Problem 2: Vulnerable to ASICs & Miner Centralization

ASICSs and miner centralization make chain reversions relatively low cost.
Problem 2: Vulnerable to ASICs & Miner Centralization

ASICs and miner centralization make chain reversions relatively low cost.
Problem 2: Vulnerable to ASICs & Miner Centralization

Blockchain reversion!
Problem 2:
Vulnerable to ASICs &
Miner Centralization

- No in protocol penalty for reversion
- Difficulty recovering from 51% attacks
- Explicitly lacks **finality**

*Finality guarantees that history will never be changed.*
Why do we want finality?
Case 1: Withdrawing from Exchanges

Blockchain is reverted.
The user’s payment is not included in the new chain, but inside the exchange their balance is still zero.

User receives withdrawal payment.
And exchange updates their ETH balance to zero.
This is real!
UASF exploited this vulnerability!
Problem 3: No Active Validator Set

PoW has no concept of an “active validator set”

Active validator sets can be useful for:
- Main chain consensus (Casper)
- Sharding
- Cryptoeconomic light clients
- & more!
3.

Our Solution: Hybrid Casper

Layering PoS on top of PoW for with Casper the Friendly Finality Gadget
What is a “Finality Gadget”

Casper the Friendly Finality Gadget is the phase 1 deployment of proof of stake on Ethereum. It adds:

- PoS, layered on top of PoW
- Finality every 50 blocks [one epoch]
- 51% attack resistance
- Reduction in energy waste from PoW
- Staked active validator set

In later iterations, PoW will be replaced completely.
Becoming a Validator

Any ETH holder can become a validator by simply depositing ETH into the Casper smart contract.
Becoming a Validator

The Casper incentive logic lives in a smart contract!

Any ETH holder can become a validator by simply depositing ETH into the Casper smart contract.
Why Deposits?
- *Gives us larger incentives to work with*
- *Impose larger economic penalties to bad actors*
Why Deposits?

- *Gives us larger incentives to work with*
- *Impose larger economic penalties to bad actors*
Why Deposits?
- Gives us larger incentives to work with
- Impose larger economic penalties to bad actors

Slashed deposit!
Casper Votes & Finality
Casper Votes & Finality

The chain is chunked in 50 block segments called epochs
Casper Votes & Finality

The chain is chunked in 50 block segments called epochs.

Every 50th block is considered a checkpoint.
Casper Votes & Finality

The chain is chunked in 50 block segments called epochs.

Every 50th block is considered a checkpoint.

Finality is achieved when two consecutive checkpoints receive \( \frac{2}{3} \) votes.
Casper Votes & Finality
Casper Votes & Finality
Casper Votes & Finality
Casper Votes & Finality
Casper Votes & Finality
Casper Votes & Finality

Justified checkpoint!
Casper Votes & Finality
Casper Votes & Finality
Casper Votes & Finality

1 epoch
Casper Votes & Finality
Casper Votes & Finality

VOTE: [validator_index, target_hash, target_epoch, source_epoch]
Casper Votes & Finality

VOTE: [validator_index, target_hash, target_epoch, source_epoch]
Casper Votes & Finality

VOTE: [validator_index, target_hash, target_epoch, source_epoch]
Casper Votes & Finality

VOTE: [1, target_hash, target_epoch, source_epoch]
Casper Votes & Finality

VOTE: [1, 0x34ba31, target_epoch, source_epoch]
Casper Votes & Finality

VOTE:

\[
\begin{bmatrix}
1, \\
0x34ba31, \\
2, \\
\end{bmatrix}
\]

VOTE RESTRICTION:
A vote’s source_epoch must be justified (greater than \( \frac{2}{3} \) votes)
Casper Votes & Finality

VOTE RESTRICTION:
A vote’s source_epoch must be justified (greater than $\frac{2}{3}$ votes)

VOTE: $[1, 0x34ba31, 2, 1]$
Casper Votes & Finality

Epoch 1

Epoch 2
Casper Votes & Finality

Epoch 1

Epoch 2
Casper Votes & Finality

Epoch 1

Epoch 2

Justified checkpoint!
Casper Votes & Finality

Epoch 1

Epoch 2

Supermajority link
Casper Votes & Finality

Two in a row!
Casper Votes & Finality

Epoch 1

Epoch 2

Finalized checkpoint!
Why two in a row?

To be finalized, you must know that everyone else knows that this block is included in the main chain.

1. Announce you believe block is in main chain
2. Receive same message from \( \frac{2}{3} \) of validators
3. Announce that you heard \( \frac{2}{3} \) validators announcements
4. Receive same message from \( \frac{2}{3} \) of validators

Now you know that everyone else knows that the block is part of the main chain.
Why two in a row?

1. Announce you believe block is in main chain
2. Receive same message from \( \frac{2}{3} \) of validators
3. Announce that you heard \( \frac{2}{3} \) validators announcements
4. Receive same message from \( \frac{2}{3} \) of validators

Each vote counts as a “preparation” of the target epoch, and a “commitment” on the previous epoch
Casper Votes & Finality

I believe this is the target for epoch 2!
Casper Votes & Finality

I have seen \( \frac{2}{3} \) votes on this block for epoch 1!
4. Slashing Conditions

Burning validator deposits if malicious behavior detected
Slashing Condition #1: NO DOUBLE VOTE
Slashing Condition #1: **NO DOUBLE VOTE**

An individual validator $v$ must not publish two distinct votes,\[\langle v, s_1, t_1, h(s_1), h(t_1) \rangle \quad \text{AND} \quad \langle v, s_2, t_2, h(s_2), h(t_2) \rangle\]

Such that 1. $h(t_1) = h(t_2)$.

Equivalently, a validator must not publish two distinct votes for the same target height.
Slashing Condition #1: **NO DOUBLE VOTE**

An individual validator $v$ must not publish two distinct votes,

\[
\langle v, s_1, t_1, h(s_1), h(t_1) \rangle \quad \text{AND} \quad \langle v, s_2, t_2, h(s_2), h(t_2) \rangle
\]

such that

1. $h(t_1) = h(t_2)$.

   Equivalently, a validator must not publish two distinct votes for the same target height.
Slashing Condition #1: NO DOUBLE VOTE

An individual validator \( \nu \) must not publish two distinct votes,

\[
\langle \nu, s_1, t_1, h(s_1), h(t_1) \rangle \quad \text{AND} \quad \langle \nu, s_2, t_2, h(s_2), h(t_2) \rangle
\]

Such that

1. \( h(t_1) = h(t_2) \).
   Equivalently, a validator must not publish two distinct votes for the same target height.
Slashing Condition #1: NO DOUBLE VOTE

An individual validator \( v \) must not publish two distinct votes,

\[
\langle v, s_1, t_1, h(s_1), h(t_1) \rangle \quad \text{AND} \quad \langle v, s_2, t_2, h(s_2), h(t_2) \rangle
\]

Such that

1. \( h(t_1) = h(t_2) \).

   Equivalently, a validator must not publish two distinct votes for the same target height.

Epoch 1

Epoch 2
Slashing Condition #1: NO DOUBLE VOTE

An individual validator \( v \) must not publish two distinct votes,

\[
\langle v, s_1, t_1, h(s_1), h(t_1) \rangle \quad \text{AND} \quad \langle v, s_2, t_2, h(s_2), h(t_2) \rangle
\]

Such that

1. \( h(t_1) = h(t_2) \).

   Equivalently, a validator must not publish two distinct votes for the same target height.
Slashing Condition #1: NO DOUBLE VOTE

An individual validator $v$ must not publish two distinct votes,

$$\langle v, s_1, t_1, h(s_1), h(t_1) \rangle \quad \text{AND} \quad \langle v, s_2, t_2, h(s_2), h(t_2) \rangle$$

such that

1. $h(t_1) = h(t_2)$.

Equivalently, a validator must not publish two distinct votes for the same target height.
Slashing Condition #1: NO DOUBLE VOTE

An individual validator $v$ must not publish two distinct votes,

$$\langle v, s_1, t_1, h(s_1), h(t_1) \rangle \quad \text{AND} \quad \langle v, s_2, t_2, h(s_2), h(t_2) \rangle$$

such that

1. $h(t_1) = h(t_2)$.

Equivalently, a validator must not publish two distinct votes for the same target height.
Slashing Condition #1: NO DOUBLE VOTE

AN INDIVIDUAL VALIDATOR \( \nu \) MUST NOT PUBLISH TWO DISTINCT VOTES,

\[
\langle \nu, s_1, t_1, h(s_1), h(t_1) \rangle \quad \text{AND} \quad \langle \nu, s_2, t_2, h(s_2), h(t_2) \rangle
\]

SUCH THAT

1. \( h(t_1) = h(t_2) \).

Equivalently, a validator must not publish two distinct votes for the same target height.
Slashing Condition #1: NO DOUBLE VOTE

An individual validator $v$ must not publish two distinct votes,

$$\langle v, s_1, t_1, h(s_1), h(t_1) \rangle \quad \text{AND} \quad \langle v, s_2, t_2, h(s_2), h(t_2) \rangle$$

Such that

1. $h(t_1) = h(t_2)$.

   Equivalently, a validator must not publish two distinct votes for the same target height.
Slashing Condition #1: NO DOUBLE VOTE

An individual validator \( v \) must not publish two distinct votes,

\[
\langle v, s_1, t_1, h(s_1), h(t_1) \rangle \quad \text{AND} \quad \langle v, s_2, t_2, h(s_2), h(t_2) \rangle
\]

Such that

1. \( h(t_1) = h(t_2) \).

Equivalently, a validator must not publish two distinct votes for the same target height.
Slashing Condition #1: NO DOUBLE VOTE

An individual validator $v$ must not publish two distinct votes,

$$\langle v, s_1, t_1, h(s_1), h(t_1) \rangle \quad \text{AND} \quad \langle v, s_2, t_2, h(s_2), h(t_2) \rangle$$

such that

1. $h(t_1) = h(t_2)$.

   Equivalently, a validator must not publish two distinct votes for the same target height.
Slashing Condition #1: NO DOUBLE VOTE

AN INDIVIDUAL VALIDATOR $v$ MUST NOT PUBLISH TWO DISTINCT VOTES,

$$\langle v, s_1, t_1, h(s_1), h(t_1) \rangle \quad \text{AND} \quad \langle v, s_2, t_2, h(s_2), h(t_2) \rangle$$

SUCH THAT I. $h(t_1) = h(t_2)$.

Equivalently, a validator must not publish two distinct votes for the same target height.
Slashing Condition #1: **NO DOUBLE VOTE**

An individual validator $v$ must not publish two distinct votes,

$$\langle v, s_1, t_1, h(s_1), h(t_1) \rangle \quad \text{AND} \quad \langle v, s_2, t_2, h(s_2), h(t_2) \rangle$$

Such that

1. $h(t_1) = h(t_2)$.

Equivalently, a validator must not publish two distinct votes for the same target height.
Slashing Condition #1: NO DOUBLE VOTE

An individual validator $\nu$ must not publish two distinct votes,

$$\langle \nu, s_1, t_1, h(s_1), h(t_1) \rangle \quad \text{AND} \quad \langle \nu, s_2, t_2, h(s_2), h(t_2) \rangle$$

Such that

1. $h(t_1) = h(t_2)$.

Equivalently, a validator must not publish two distinct votes for the same target height.
Slashing Condition #2: NO SURROUND VOTE

An individual validator $v$ must not publish two distinct votes,

$$\langle v, s_1, t_1, h(s_1), h(t_1) \rangle \quad \text{AND} \quad \langle v, s_2, t_2, h(s_2), h(t_2) \rangle$$

such that

II. $h(s_1) < h(s_2) < h(t_2) < h(t_1)$.

Equivalently, a validator must not vote within the span of its other votes.
Slashing Condition #2: NO SURROUND VOTE

AN INDIVIDUAL VALIDATOR $v$ MUST NOT PUBLISH TWO DISTINCT VOTES,

\[
\langle v, s_1, t_1, h(s_1), h(t_1) \rangle \quad \text{AND} \quad \langle v, s_2, t_2, h(s_2), h(t_2) \rangle
\]

SUCH THAT II. $h(s_1) < h(s_2) < h(t_2) < h(t_1)$.

Equivalently, a validator must not vote within the span of its other votes.
Slashing Condition #2: NO SURROUND VOTE

AN INDIVIDUAL VALIDATOR \( \nu \) MUST NOT PUBLISH TWO DISTINCT VOTES,

\[
\langle \nu, s_1, t_1, h(s_1), h(t_1) \rangle \quad \text{AND} \quad \langle \nu, s_2, t_2, h(s_2), h(t_2) \rangle
\]

SUCH THAT \( h(s_1) < h(s_2) < h(t_2) < h(t_1) \).

Equivalently, a validator must not vote within the span of its other votes.
Slashing Condition #2: NO SURROUND VOTE

AN INDIVIDUAL VALIDATOR \( \nu \) MUST NOT PUBLISH TWO DISTINCT VOTES,

\[
\langle \nu, s_1, t_1, h(s_1), h(t_1) \rangle \quad \text{AND} \quad \langle \nu, s_2, t_2, h(s_2), h(t_2) \rangle
\]

SUCH THAT \( \text{II. } h(s_1) < h(s_2) < h(t_2) < h(t_1) \).

Equivalently, a validator must not vote within the span of its other votes.
Slashing Condition #2: NO SURROUND VOTE

An individual validator $\nu$ must not publish two distinct votes,

$$\langle \nu, s_1, t_1, h(s_1), h(t_1) \rangle \quad \text{AND} \quad \langle \nu, s_2, t_2, h(s_2), h(t_2) \rangle$$

such that II. $h(s_1) < h(s_2) < h(t_2) < h(t_1)$.

Equivalently, a validator must not vote within the span of its other votes.
Casper Checkpoints:
Resist PoW attack

Time 1
Casper Checkpoints: Resist PoW attack
Casper Checkpoints: Resist PoW attack

Time 1
Casper Checkpoints: Resist PoW attack
Casper Checkpoints: Resist PoW attack

Time 1
Casper Checkpoints: Resist PoW attack

Time 1
Casper Checkpoints: Resist PoW attack

Time 1
Casper Checkpoints: Resist PoW attack

Time 1
Casper Checkpoints: Resist PoW attack

Time 1
Casper Checkpoints: Resist PoW attack
Casper Checkpoints: Resist PoW attack

Warning: PoW Attack!
Casper Checkpoints: Resist PoW attack
Casper Checkpoints:
Resist PoW attack
Casper Checkpoints: Resist PoW attack
Casper Checkpoints:
Resist PoW attack
Casper Checkpoints: Resist PoW attack
Casper Checkpoints: Resist PoW attack
Casper Checkpoints: Resist PoW attack
Casper Checkpoints:
Resist PoW attack
Casper Checkpoints: Resist PoW attack
Casper Checkpoints: Resist PoW attack

PoW Attack FAILED!
Casper Checkpoints: Resist PoW attack

PoW Attack FAILED!
Casper Checkpoints: Resist PoW attack

PoW Attack FAILED!

Go Casper!
5.

Designing Casper FFG

Casper is the result of years of research by Vlad Zamfir and Vitalik Buterin.

But the last year of design work looked a little something like this…
Protocol Definition Process

Propose Protocol → Define Properties

Define Properties → Implement / Simulate

Oh nose we’ve made a terrible mistake!
Protocol Proposal #1: Minimal Slashing Conditions

Vitalik Buterin
Mar 1, 2017 · 14 min read

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...

Read more...
Protocol Proposal #1: Minimal Slashing Conditions

1. PREPARE_REQ
2. COMMIT_REQ
3. NO_DBL_PREPARE
4. PREPARE_COMMIT_CONSISTENCY
Minimal Slashing Conditions
Property Definition #1: Safety Unless \( \frac{1}{3} \) Slashed

Formal methods on some PoS stuff

```
proof (cases "v1 = v2")
  case True
    show ?thesis using 1 4 2 5 6 True casper_axioms un
    one_third_slashed_def casper_axioms committed_def
    by metis
  next
  case False thus ?thesis using l3 1 2 4 5 6 by (met
  qed
qed

theorem safety:
  fork ?s \rightarrow one_third_slashed ?s
```
Property Definition #1: Safety Unless \( \frac{1}{3} \) Slashed

Formal Verification is great for proving **properties** of your protocol.
Property Definition #1: Safety Unless ⅓ Slashed

Formal Verification is great for proving properties of your protocol. Use it when human intuitions are not enough.
Property Definition #1: Safety Unless ½ Slashed

Minimal Slashing Conditions
Parametrizing Casper: the decentralization/finality time/overhead tradeoff

As Casper continues to reach an increasingly stabilized form, there has been increased interest in the various parameters that are going to be set in the protocol, including the interest rate, fees...

Read more...
Property Definition #2: **Decentralization Tradeoff**

\[ \text{finality\_time} \times \text{overhead} \geq \text{num\_validators} \times 2 \]

Parametrizing Casper: the decentralization/finality time/overhead tradeoff

As Casper continues to reach an increasingly stabilized form, there has been increased interest in the various parameters that are going to be set in the protocol, including the interest rate, fees…

Read more…

👍 170
Property Definition #2: Decentralization Tradeoff

\[ \text{finality_time} \times \text{overhead} \geq \text{num_validators} \times 2 \]

Parametrizing Casper: the decentralization/finality time/overhead tradeoff

As Casper continues to reach an increasingly stabilized form, there has been increased interest in the various parameters that are going to be set in the protocol, including the interest rate, fees…

Read more...
Property Definition #2: Decentralization Tradeoff

finality_time * overhead \geq num_validators * 2

Parametrizing Casper: the decentralization/finality time/overhead tradeoff

As Casper continues to reach an increasingly stabilized form, there has been increased interest in the various parameters that are going to be set in the protocol, including the interest rate, fees…

Read more…

170

3 responses
Property Definition #2: Decentralization Tradeoff

\[ \text{finality}_\text{time} \times \text{overhead} \geq \text{num}_\text{validators} \times 2 \]

Parametrizing Casper: the decentralization/finality time/overhead tradeoff

As Casper continues to evolve in increasingly sophisticated form, there has been increased interest in the various parameters that are going to be set in the protocol, including the interest rate fees...

Read more...
Property Definition #2: Decentralization Tradeoff

\[ \text{finality\_time} \times \text{overhead} \geq \text{num\_validators} \times 2 \]

Parametrizing Casper: the decentralization/finality time/overhead tradeoff

As Casper continues to reach an increasingly stabilized form, there has been increased interest in the various parameters that are going to be set in the protocol, including the interest rate, fees...

So, what are your tradeoffs?
Parametrizing Casper: the decentralization/finality time/overhead tradeoff

Property Definition #1: 

*Safety Unless ½ Slashed*

Minimal Slashing Conditions
Casper Implementation #1:  
*Prepare / Commit Casper*

```python
# Process a prepare message
def prepare(prepare_msg: bytes <= 1024):
    # Get hash for signature, and implicitly assert that it is an RLP list
    # consisting solely of RLP elements
    sighash = extract32(raw_call(self.sighasher, prepare_msg, gas=200000, outsize=32), 0)
    # Extract parameters
    values = RLPList(prepare_msg, [num, num, bytes32, bytes32, num, bytes32, bytes])
    validator_index = values[0]
    epoch = values[1]
    hash = values[2]
    ancestry_hash = values[3]
    source_epoch = values[4]
    source_ancestority_hash = values[5]
    sig = values[6]
    new_ancestority_hash = sha3(concat(hash, ancestry_hash))
```
Casper Implementation #1:
 Prepare / Commit Casper

```python
# Process a prepare message

# Check that the prepare is on top of a justified prepare
assert self.consensus_messages[source_epoch].ancestry_hash_justified[source_ancestry_hash]
# consisting solely of RLP elements
sighash = extract32(raw_call(self.sighasher, prepare_msg, gas=200000, outsize=32), 0)
# Extract parameters
values = RLPList(prepare_msg, [num, num, bytes32, bytes32, num, bytes32, bytes])
validator_index = values[0]
epoch = values[1]
hash = values[2]
ancestry_hash = values[3]
source_epoch = values[4]
source_ancestry_hash = values[5]
sig = values[6]
new_ancestry_hash = sha3(concat(hash, ancestry_hash))
```
Casper Implementation #1:
Prepare / Commit Casper

```python
# Check that the prepare is on top of a justified prepare
assert self.consensus_messages[source_epoch].ancestry_hash_justified[source_ancestry_hash]
```

Minimal Slashing Conditions

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

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

where `epoch_source != -1`, then unless, for some specific value

```
epoch_source_source , with -1 <= epoch_source_source < epoch_source ,
```

messages of the form

```
["PREPARE", epoch_source, ANCESTOR_HASH, epoch_source_source]
```

, where ANCESTOR_HASH is the \((\text{epoch} - \text{epoch_source})\)—degree ancestor of

```
HASH , have been signed and broadcasted by 2/3 of validators, then that
```
validator’s deposit is slashed.
Casper Implementation #1: Prepare / Commit Casper

```python
# Check that the prepare is on top of justified prepare
assert self.consensus_message[1].entry_hash_justified[source_ancestry_hash] == self.sighashentry_replay_msg(gas=200000, outsize=32), 0)
```

Minimal Slashing Conditions

[PREPARE_REQ] the validator sends a signed message of the form

```
["PREPARE", epoch_hash, epoch_source]
```

where `epoch_source` must be given unless, for a specific value `epoch_source_source`, `epoch_source < epoch_source_source < epoch_source`.

messages 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.
Parametrizing Casper: the decentralization/finality time/overhead tradeoff

Property Definition #1: Safety Unless \( \frac{1}{2} \) Slashed

Minimal Slashing Conditions
Protocol Proposal #2:
*Even More Minimal Slashing Conditions*
Protocol Proposal #2:  
Even More Minimal Slashing Conditions

**[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.
Protocol Proposal #2:
Even More Minimal Slashing Conditions

[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

[NO_DBL_PREPARE] 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 the validator is slashed
Even More...

Minimal Slashing Conditions

Parametrizing Casper: the decentralization/finality time/overhead tradeoff

Property Definition #1:

*Safety Unless \( \frac{1}{2} \) Slashed*

Minimal Slashing Conditions
Casper Implementation #2:  
*Fork Choice Revamp*
Casper Implementation #2:
Fork Choice Revamp

```python
135 - def apply_commits(self, state, casper, commits):
141 + def get_checkpoint_score(self, blockhash, commits):
142 +     state = self.mk_poststate_of_blockhash(blockhash)
143 +     casper = tester2.ABIContact(tester2.State(state), casper_util)
144 +     epoch = casper.get_current_epoch()
145 +     curr_dinsy_deposits = 0
146 +     prev_dinsy_deposits = 0
147 +     # Calculate the current & previous dynasty scores
136 148     for sig in commits:
137 -         try:
138 -             casper.commit(commits[sig], gasprice=0)
139 -         except tester2.TransactionFailed:
140 -             commit = self.get_decoded_commit(commits[sig])
141 -             print('Transaction failed! Commit was probably already
142 -             |Epoch:', commit['epoch'], '| validator_index:',
143 -             state.gas_used = 0
144 -
145 -             self.commit(commits[sig], gasprice=0)
```
Casper Implementation #2:
Fork Choice Revamp

```
def apply_commits(self, state, casper, commits):
    def get_checkpoint_score(self, blockhash, commits):
        state = self.mk_poststate_of_blockhash(blockhash)
        casper = tester2.ABCIContract(tester2.State(state), casper_util)
        epoch = casper.get_current_epoch()
        curr_dynasty_deposits = 0
        prev_dynasty_deposits = 0
        # Calculate...
        for sig in:
            try:
                hc = self.get_block(self.checkpoint_head_hash)
                fc = self.get_block(fork_hash)
                fork_score = self.get_checkpoint_score(fc.header.hash, sel
                # Loop over the hc & fc until they are equal...
                while fc != hc:
                    if fc.header.number > hc.header.number:
                        checkpoint_id = self.mk_checkpoint_id(utils.int_to
                        fork_score = self.get_checkpoint_score(fc.header.
                        fc = self.get_prev_checkpoint_block(fc)
                    else:
                        checkpoint_id = self.mk_checkpoint_id(utils.int_to
                        head_score = self.get_checkpoint_score(hc.header.
                        hc = self.get_prev_checkpoint_block(hc)
```
Casper Implementation #2: Fork Choice Revamp

```python
    def apply_commits(self, state, casper, commits):
        def get_checkpoint_score(self, blockhash, commits):
            state = self.mk_poststate_of_blockhash(blockhash)
            casper = tester2.ABICase(tester2.State(state), casper_util)
            epoch = casper.get_current_epoch()
            curr_dyn Deposits = 0
            prev_dyn Deposits = 0
            # Calculate the score for the checkpoint
            checkpoint_id = self.mk_checkpoint_id(utills.int_to_bytes(commit['hash']
            checkpoint_score = self.get_checkpoint_score(fc.header.hash, sel
            state =
            if fc.head
            check
            fork
            fc =
            else:
            check
            head
            if fc.head
            if b'cp_total_deposits' in checkpoint_hash:
                raise Exception('No deposits found for checkpoint hash: {}', format
                new_cp_score = self.get_checkpoint_score(checkpoint_hash)
                if new_cp_score <= self.get_checkpoint_score(checkpoint_hash):
                    log.info('*** block num: {} - new cp score: {} - subtree score:
```
It became clear the fork choice rule was too complex.
Casper Implementation #2:
Fork Choice Revamp

It became clear the fork choice rule was too complex.

```python
def add_block(self, candidate_block):
    if (self.get_score(self.head) < self.get_score(candidate_block) and
        not self.switch_reverts_finalized_block(self.head, candidate_block)):
        self.set_head(candidate_block)

def get_score(self, parent, block):
    casper = tester.AntContract(casper_address, block)
    return casper.get_first_justified_epoch() * 10**40 + self.get_pow_diff

def switch_reverts_finalized_block(self, old_head, new_head):
    while old_head.number > new_head.number:
        if b'finalized' + old_head.hash in self.db:
            log.info('[WARNING] Attempt to revert failed: checkpoint {} installed!
return True
        old_head = self.get_parent(old_head)
    while new_head.number > old_head.number:
        new_head = self.get_parent(new_head)
    while new_head.hash != old_head.hash:
        if b'finalized:' + old_head.hash in self.db:
            log.info('[WARNING] Attempt to revert failed; checkpoint {} installed!
return True
        old_head = self.get_parent(old_head)
        new_head = self.get_parent(new_head)
    return False
```
Even More...

Minimal Slashing Conditions

Property Definition #1:

*Safety Unless ½ Slashed*

Minimal Slashing Conditions
Casper FFG with one message type, and simpler fork choice rule

As it turns out, we can reduce the number of message types in Casper the Friendly Finality Gadget from two (prepares and commits) to just one ("votes"), increasing the algorithm’s simplicity to something which is arguably no longer that much more complex than, say, Nakamoto PoW.

A vote has three parameters:

- Epoch number
- Checkpoint hash
- Epoch source (MUST BE less than epoch number)

The two slashing conditions are as follows, using the notation \( (e1, h1, s1) \) or one prepare and \( (e2, h2, s2) \) for the second prepare. If a validator sends two prepares that satisfy either of these, they can lose their deposit.

- \( \text{NO\_DBL\_VOTE}: e1 = e2 \)
- \( \text{NO\_SURROUND}: e1 > e2 > s2 > s1 \)

Alternatively, this can be expressed as one three-clause condition (thanks Virgil for the suggestion!):
Protocol Proposal #3: One Message Casper FFG

As it turns out, we can reduce the number of message types in Casper the FFG (prepares and commits) to just one ("votes"), increasing the algorithm's simplicity and arguably no longer that much more complex than, say, Nakamoto PoW.

- NO_DBL_VOTE: e1 = e2
- NO_SURROUND: e1 > e2 > s2 > s1

Alternatively, this can be expressed as one three-clause condition (thanks Virgil for the suggestion!):
Protocol Proposal #3: 
One Message Casper FFG

As it turns out, we can reduce the number of message types in Casper the FFG (prepares and commits) to just one ("votes"), increasing the algorithm’s simplicity and arguably no longer that much more complex than, say, Nakamoto PoW.

- NO,DBL,VOTE: e1 = e2
- NO,SURROUND: e1 > e2 > s2 > s1

Alternatively, this can be expressed as one three-clause condition (thanks Virgil for the suggestion!):
Property Definition #3: 
*Partial Slashing provides decentralization incentive*
Property Definition #3:
Partial Slashing provides decentralization incentive

Yo!
We shouldn’t slash validators fully unless two checkpoints are actually finalized
Property Definition #3:
Partial Slashing provides decentralization incentive

If a staking pool gets too large, it is more likely to receive the full slashing penalty
Property Definition #3:
Partial Slashing provides decentralization incentive

If a staking pool gets too large, it is more likely to receive the full slashing penalty

So as a user I will choose smaller pools!
Parametrizing Casper: the decentralization/finality time/overhead tradeoff

Property Definition #1: Safety Unless ½ Slashed

Property Definition #3: Partial Slashing provides decentralization incentive
Casper Implementation #3: 
Casper Testnet
Casper Implementation #3: Casper Testnet

Casper testnet stats!

So much love to all the Casper implementers on the Ethereum research team!

@changwu_tw, @ChihChengLiang, @davidlknott, @jon_choi_, and of course @VitalikButerin 😊

8:21 AM - 31 Dec 2017
Casper Implementation #3:
Casper Testnet

Karl Floersch
@karl_dot_tech

Casper testnet stats!

So much love to all the Casper implementers on the Ethereum research team!

@changwu_tw, @ChihChengLiang, @davidlknot, @jon_choi_, and of course @VitalikButerin 💚

8:21 AM - 31 Dec 2017
Casper Implementation #3: 
Casper Testnet

Karl Floersch
@karl_dot_tech

Casper testnet stats!

@changwu_tw, @daviddknott, @johncarloscurrie, @VitalikButerin

8:21 AM - 31 Dec 2017
Casper Implementation #3: Casper Testnet

With high network latency, finality could not be reached.

@changwu_tw, @ChihChengLiang, @davidlknot, @jon_choi, and of course @VitalikButerin
Casper Implementation #3: Casper Testnet

With high network latency, finality could not be reached.

To solve this we progressively increase epoch length.
Casper Implementation #3: Casper Testnet

With high network latency, finality could not be reached.
To solve this, we progressively increase epoch length.
Casper Implementation #3:
Casper Testnet

With high network latency,
finality could not be reached.

To solve this, we progressively
increase epoch length.

Now we’re safe under partial synchrony!
Casper Implementation #3: Casper Testnet

With high network latency finality could not be reached. To solve this we progressively increase epoch length. Now we’re safe under partial synchrony!
Casper Implementation #3: Casper Testnet

With high network latency, finality could not be reached. To solve this, we progressively increase epoch length. Now we’re safe under partial synchrony!
Property Definition #1: Safety Unless ½ Slashed

Property Definition #3: Partial Slashing provides decentralization incentive

Casper Testnet
Best Block
#13,868

Minimal Slashing Conditions

Casper FFQ with one message type, and simpler fork choice rule
Parametrizing Casper: the decentralization/finality time/overhead tradeoff

Property Definition #1: Safety Unless ½ Slashed

Minimal Slashing Conditions

Property Definition #3: Partial Slashing provides decentralization incentive

Casper Testnet

Best Block #13,868
Even More...

Minimal Slashing Conditions

Property Definition #1: Safety Unless ½ Slashed

Property Definition #3: Partial Slashing provides decentralization incentive

Casper Testnet

#13,868
Property Definition #1: **Safety Unless 1/2 Slashed**

Even More...

Minimal Slashing Conditions

Parametrizing Casper: the decentralization/time/overhead tradeoff
Let’s a go!
Some Takeaways

1. Formal Verification is an important tool but doesn’t by itself provide an optimal protocol.
2. Identify and evaluate your tradeoffs.
3. The more you work on your protocol, the simpler it should become.
4. Each implementation and simulation will likely reveal something you didn’t expect.
5. Always stay open to new ideas.
Some Takeaways

1. Formal Verification is an important tool but doesn’t by itself provide an optimal protocol.
2. Identify and evaluate your tradeoffs.
3. The more you work on your protocol, the simpler it should become.
4. Each implementation and simulation will likely reveal something you didn’t expect.
5. Always stay open to new ideas.

Free and open source.  
Constantly collaborating.
6. Building Your Own Open Protocols

Learning to build these protocols is not just fun. It is also critical for ensuring this technology produces a more equitable world.
Casper is just the beginning
Casper is just the beginning.

We need Cryptoeconomics!
We need Cryptoeconomics!

Casper is just the beginning.
Casper is just the beginning.

We need Cryptoeconomics!
Casper is just the beginning
We need Cryptoeconomics!
Casper is just the beginning

We need Cryptoeconomics!

Cryptography!

Game theory!
Casper is just the beginning
We need Cryptoeconomics!
Cryptography!
Game theory!
Casper is just the beginning

We need Cryptoeconomics!

Cryptography!

Game theory!

Mechanism Design!
Casper is just the beginning. We need Cryptoeconomics! Cryptography! Game theory! Mechanism Design!
Casper is just the beginning
We need Cryptoeconomics!

Cryptography! Game theory!
Mechanism Design!
Math!
Casper is just the beginning. We need Cryptoeconomics! Cryptography! Game theory! Mechanism Design! Math! Econ!
Casper is just the beginning
We need Cryptoeconomics!
Math!
Econ!
Game theory!
Mechanism Design!

Peace and kindness!
Casper is just the beginning
We need Cryptoeconomics!
Cryptography!
Game theory!
Mechanism Design!
Econ!
Math!
Peace and kindness!
We need Cryptoeconomics!
Cryptography!
Game theory!
Mechanism Design!
Econ!
Math!
Peace and kindness!
Casper is just the beginning

We need Cryptoeconomics!

Cryptography!

Game theory!

Mechanism Design!

Econ!

Math!

Peace and kindness!

We Need you
love knows no borders
love knows no borders
love knows no borders
love knows no borders
love knows no borders
Thank you!

https://karl.tech/
@karl_dot_tech

ethresear.ch

https://karl.tech/
@karl_dot_tech