# Upgrading Ethereum

A technical handbook on Ethereum's move to proof of stake and beyond.

**Ben Edgington**

**Altair Edition**

1 July 2023 - 8fa708b

# Contents

# Preface

## Work in progress!

I am writing this book backwards. Bottom up. Starting with the details and working towards the big picture.

The first pretty much complete part is Part 3: The Annotated Spec. These are the guts of the machine. Like the innards of a computer, all the components are showing and the wires are hanging out: everything is on display. But with the guts in place, everything else can be built around them with the messiness all neatly tucked away.

I'm now working on Part 2: Technical Overview which wraps a first, hopefully more accessible, layer around the Annotated Spec. Again, I'm writing this backwards, starting with the protocol's Building Blocks and its Incentive Mechanisms and working forwards towards a higher level narrative of how it all fits together.

In an ideal world, my plan is as follows:

- Deliver *Edition 1.0: Altair* at some point before The Merge (the point at which Ethereum moves to proof of stake). By then, I hope to have done the following:
  - Completed Part 2: Technical Overview
  - Completed Part 1: Building (probably too ambitious)
  - Made a start on Part 4: Future (unlikely before The Merge)
- Some while after The Merge, I'll publish a fully revised *Edition 2.0: The Merge*.
- Editions *2.5* (with post-Merge clean-ups) and *3.0* (a full revision for sharding) are also in view. This thing's going to keep me busy for a while.

Meanwhile, I might get round to making it prettier, ensuring it is accessible and mobile-friendly, adding search, navigation and other rich information, PDF versions, maybe NFTs… who knows?

**Warning:** until Edition 1.0 is out, anything may change. I'll try not to change URLs and anchors in the Annotated Spec part, but no promises. Anything else, including entire chapters and sections, should be considered unstable.

## What to expect

This is a book for those who want to understand Ethereum 2.0 – Ethereum on proof of stake – at a technical level. What does it do? How does it work? Why is it like this?

Who am I writing for? For people like me! People who enjoy understanding how things work. But more than that, who like to know *why* things are the way they are.

Although I am an Ethereum staker and an Ethereum user, I am not writing primarily for stakers or users here. Some of the generic material on Staking may be relevant (once I have written it), but you will find better help in places like the excellent EthStaker community.

The scope of the book concerns (what I consider to be) the Ethereum 2.0 protocol. Ethereum 2.0 has become a less well-defined term recently. But for me, it broadly includes,

- all things proof of stake and the beacon chain,
- the process of The Merge by which Ethereum moved to proof of stake,
- in-protocol data sharding, and
- an array of potential future enhancements.

I will not be covering any of the historic Ethereum 1.0 protocol, except as it touches upon The Merge. The Mastering Ethereum book is an excellent resource, and there is no point in duplicating it. Although roll-ups and other so-called layer 2 solutions have rapidly become part of the overall Ethereum 2.0 narrative, they are by definition not in-protocol, and I will not be covering them here. I will not be discussing, DeFi, DAOs, NFTs, or any of the wonderful things that can be built on top of this amazing technology.

It's a chunky list of exclusions, but there's still plenty to talk about.

## Altair

This edition covers the Altair version of the deployed Ethereum 2.0 beacon chain. The beacon chain went live with Phase 0 on December 1st, 2020. It was upgraded to Altair on October 27th, 2021.

Specifically, any reference to the consensus specifications is to the version tagged v1.1.1.

## A note on Terminology

The "Ethereum 2.0" terminology is out of favour in some circles, but I don't really care. I will be happily using the terms "Ethereum 2.0", "Ethereum 2", "Ethereum 1", "Eth1", and "Eth2" throughout this book where it makes sense to me, and I'm pretty sure you'll know what I mean. I have more to say about this in the first chapter.

You will also notice that I unapologetically use British English spelling, punctuation, and quaint idioms. It's a feature, not a bug.

## Acknowledgements

First and foremost, I want to thank my employer, ConsenSys. Much of the work has been in my own time, but ConsenSys has also been very cool with me working on this in the course of my day job. ConsenSys is a wonderful employer, a terrific force for good in the ecosystem, and an incredible place to work.

So much of what I do involves writing about other people's work, and pretty much everything in this book is other people's work. I deeply value the openness and generosity of the Ethereum community. For me, this is one of its defining characteristics. Many people's contributions are cited throughout this book, and I am indebted to all of you. Being part of the Eth2 dev community has been the best experience of my life.

Thank you to the many GitCoin grant supporters who donated in support of the original annotated specification and my regular What's New in Eth2 newsletter. And to generous crypto friends, anon and otherwise, for your kind gifts over the years. Your support has encouraged me hugely as I've wrestled with the minutiae of the spec. I bloody love this community.

Shout-out to the EthStaker community: you rock!

Finally, to circle back to ConsenSys: working daily with such brilliant, talented, generous, and knowledgeable people is a joy. The Protocols group, PegaSys, has been my home for the past five-plus years. It is where I helped establish the fabulous Protocols R&D team, and later kicked off the project that became Teku. Thank you for all your support and encouragement. I love working with all you wonderful people.

# Part 1: Building

# Introduction

TODO

### Why Ethereum 2.0?

TODO

### The Cathedral and the Bazaar

TODO

### A Brief History of Ethereum's Future

TODO

### Who's who

TODO

### Outline of the Book

TODO

# Goals

## Introduction

TODO

## Design Goals

TODO

## Attacks and Defences

TODO

# Making the Sausage

## Introduction

TODO

## The Specifications

TODO

## The Implementations

TODO

# Part 2: Technical Overview

## Introduction

TODO: Intro

# The Beacon Chain

## Introduction

TODO

## Terminology

TODO

## Design Overview

TODO

## Architecture of a Node

TODO

## Genesis

TODO

# Consensus

Here's the opening sentence of a paper about attacks on the Ethereum 2.0 consensus protocol:

> The Proof-of-Stake (PoS) Ethereum consensus protocol is constructed by applying the finality gadget Casper FFG on top of the fork choice rule LMD GHOST, a flavor of the Greedy Heaviest-Observed Sub-Tree (GHOST) rule which considers only each participant's most recent vote (Latest Message Driven, LMD).

If that makes perfect sense to you then feel free to skip this chapter entirely. Otherwise, read on!

Our aim is to understand that sentence in all its parts. There's a lot to unpack, but we'll take time over it. We'll begin with some Preliminaries covering the basics of consensus. Then we will look in turn at each of the two consensus mechanisms used by Ethereum's proof of stake protocol, starting with Casper FFG, which is used to achieve finality, and then LMD GHOST which provides slot by slot liveness. After considering them individually we will look at how they work together as the combined consensus protocol that's become known as Gasper.

## Preliminaries

- Consensus is a way to build reliable distributed systems with unreliable components.
- Blockchain-based distributed systems aim to agree on a single history of transactions.
- Proof of work and proof of stake are not consensus protocols, but enable consensus protocols.
- Many blockchain consensus protocols are "forkful".
- Forkful chains use a fork choice rule, and sometimes undergo reorganisations.
- In a "safe" protocol, nothing bad ever happens.
- In a "live" protocol, something good always happens.
- No practical protocol can be always safe and always live.

### Introduction

In this section we'll cover the basics of consensus, fork choice, and finality. Most of this section is not specific to Ethereum and is for general background understanding.

### Coming to consensus

The Ethereum network comprises a large number of individual nodes. Each node acts independently, and nodes communicate over an unreliable, asynchronous network, the Internet. Any individual node might be honest – behaving correctly at all times – or faulty in any arbitrary way: simply down or non-communicative, following a different version of the protocol, actively trying to mislead other nodes, publishing contradictory messages, or any manner of other fault.

Users submit transactions to this network of nodes, and the goal of the consensus protocol is that all correct nodes eventually agree on a single, consistent view of the history of transactions. That is, the order in which transactions were processed and the outcome of that processing. So, if I have 1 ETH and I simultaneously tell the network that I am sending that 1 ETH to Alice and also to Bob, we expect that eventually the network will agree that either I sent it to Alice or I sent it to Bob. It would be a failure if both Alice and Bob received my Ether, or if neither received it.

A consensus protocol is the process by which this agreement on the ordering of transactions comes about.

The consensus protocol in Ethereum 2 actually "bolts together" two different consensus protocols. One is called Casper FFG, the other LMD GHOST. The combination has become known as Gasper. In subsequent pages we will be looking at these both separately and in combination.

### Byzantine generals

In a 1982 paper Leslie Lamport described in rather whimsical terms the fundamental problem that consensus systems are trying to solve - building reliable distributed systems.

> We imagine that several divisions of the Byzantine army are camped outside an enemy city, each division commanded by its own general. The generals can communicate with one another only by messenger. After observing the enemy they must decide on a common plan of action.

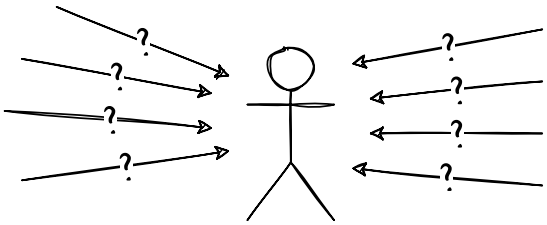
This formulation makes clear that there is no overall holistic view, no God-mode in which we can see the whole situation in one glance and make a decision. We are simply one of the generals, and our only source of information about the other generals is the messages that we receive - messages that may be correct, or lies, or mistakes based on limited information, or delayed, or modified in transit. We have only a very limited local view, yet we must come to a view about the state of the whole system.

It is important to keep this in mind at all times. When we draw diagrams of block chains and block trees, it is easy to assume that this is somehow "the state" of the whole system. But these diagrams only ever represent the local view of a single participant in the system. My node's view of the system is likely to differ from your node's view of the system, if only temporarily, because we operate over an unreliable network. For example, you will see blocks at different times from when I see them, or in a different order, or even different blocks from those that I see.

Lamport captures the faultiness of the system in the following way.

> However, some of the generals may be traitors, trying to prevent the loyal generals from reaching agreement.

These treacherous generals exhibit what we've come to call "Byzantine behaviour", or "Byzantine faults". They can act in any arbitrary way: delaying messages, reordering messages, outright lying, sending contradictory messages to different recipients, failing to respond at all, or any other behaviour we can think of.



*I receive a ton of messages from other nodes, but I have no idea which are accurate, what order they were sent in, or if any are missing or just delayed. Somehow, we need to reach agreement.*

The loyal generals need a method that reliably delivers an outcome on the following terms.

> A. All loyal generals decide upon the same plan of action [e.g. "attack" or "retreat"], and
>
> B. A small number of traitors cannot cause the loyal generals to adopt a bad plan.

Achieving consensus in such a Byzantine distributed system is not an easy problem to solve, but there have been several successful approaches over the years.

The first mainstream solution was the Practical Byzantine Fault Tolerance (PBFT) algorithm published by Liskov and Castro in 1999. This relies on a relatively small and limited set of known consensus

participants (called *replicas*). PBFT is always "safe", in the terms discussed below and does not have forks.

Nakamoto consensus, invented by Satoshi Nakamoto for Bitcoin in 2008, takes a fundamentally different approach. Rather than limiting participants to a known set it uses proof of work to permissionlessly select a temporary leader for the consensus. Unlike PBFT, Nakamoto consensus allows forks and is not formally "safe".

Many, many variants of these and other novel alternatives, such as the Avalanche family of protocols, have since sprung up. Section 7, Related Work, of the Avalanche white paper provides a good survey of the zoo of different consensus protocols currently in use in the blockchain world.

### Proof of Stake and Proof of Work

This is a good point at which to mention that neither proof of work nor proof of stake is a consensus protocol in itself. They are often (lazily) referred to as consensus protocols, but each is merely an enabler for consensus protocols.

For the main part, both proof of work and proof of stake are Sybil resistance mechanisms that place a cost on participating in the protocol. This prevents attackers from overwhelming the protocol at low or zero cost.[1]

Nevertheless, both proof of work and proof of stake are often fairly tightly coupled, via the fork choice rule, to the consensus mechanisms that they support. They provide a useful way to assign a weight, or a score, to a chain of blocks: in proof of work, the total work done; in proof of stake, the amount of value that supports a particular chain.

Beyond these basic factors, both proof of work and proof of stake enable many kinds of different consensus protocols to be built on them, each with its own dynamics and trade-offs. Once again, the survey in section 7, Related Work, of the Avalanche white paper is instructive.

### Block chains

The basic primitive that underlies blockchain technology is, of course, the block.

A block comprises a set of transactions that a leader (the block proposer) has assembled. A block's contents (its payload) may vary according to the protocol.

- The payload of a block on Ethereum's proof of work chain is an ordered list of user transactions.
- The payload of a block on the pre-Merge proof of stake beacon chain is (mostly) a set of attestations made by other validators.
- As and when EIP-4844 is implemented on Ethereum then blocks will contain opaque blobs of data alongside the ordered list of user transactions.

Except for the special Genesis block, every block builds on and points to a parent block. Thus, we end up with a chain of blocks: a blockchain. Whatever the contents of blocks, the goal of the protocol is for all nodes on the network to agree on the same history of the blockchain.


*A blockchain. Time moves from left to right and, except for the Genesis block, each block points to the parent block it builds on.*

The chain grows as nodes add their blocks to its tip. This is accomplished by temporarily selecting a "leader", an individual node that has the right to extend the chain. In proof of work the leader is the

[1] In proof of work, the "proof" you bring is a number that makes the block hash to a certain value. This proves that you did the work to calculate it. In proof of stake, your proof is a private key that is associated with a deposit of stake on the blockchain. Other proof mechanisms are available, such as proof of space and time.

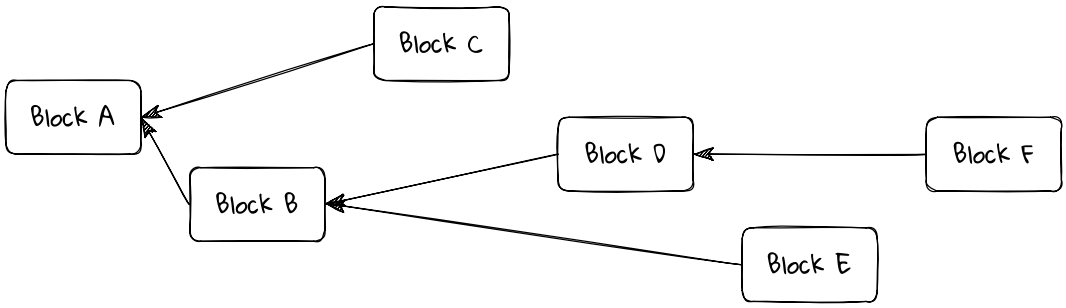
miner that first solves the proof of work puzzle for its block. In Ethereum's proof of stake the leader is selected pseudo-randomly from the pool of active stakers.

The leader (usually known as the block proposer) adds a single block to the chain, and has full responsibility for selecting and ordering the contents of that block.

The use of blocks is an optimisation. Each addition to the chain could in principle be a single transaction, but that would add a huge consensus overhead. So blocks are batches of transactions, and sometimes people argue about how big those blocks should be. In Bitcoin, the block size is limited by the number of bytes of data in the block. In Ethereum's proof of work chain, the block size is limited by the block gas limit (that is, the amount of work needed to run the transactions in the block). Beacon block sizes are limited by hard-coded constants.

### Block trees

Our neat diagram of a nice linear chain will for the most part reflect what we see in practice, but not always. Sometimes, due perhaps to network delays, or a dishonest block proposer, or client bugs, any particular node might see something more like the following.




*In general we might end up with a block tree rather than a block chain. Again, time moves from left to right and each block points to the parent block it builds on.*

In real networks we can end up with something more like a block tree than a block chain. In this example very few blocks are built on their "obvious" parent.

Why did the proposer of block $C$ build on $A$ rather than $B$?

- It may be that the proposer of $C$ had not received block $B$ by the time it was ready to make its proposal.
- It may be that the proposer of $C$ deliberately wanted to exclude block $B$ from its chain, for example to steal its transactions, or to censor some transaction in $B$.
- It may be that the proposer of $C$ thought that block $B$ was invalid for some reason.

The first two reasons, at least, are indistinguishable to the wider network. All we know is that $C$ built on $A$, and we can never know why for certain.

Similarly, why did the proposer of block $D$ build on $B$ rather than $C$? Any of the above reasons apply, and we can add another:

- The proposer of $D$ may have decided on some basis that there was more chance of the wider network eventually including $B$ than $C$. Thus, building $D$ on $B$ gives it more chance of making it into the eventual block chain, than building $D$ on $C$.

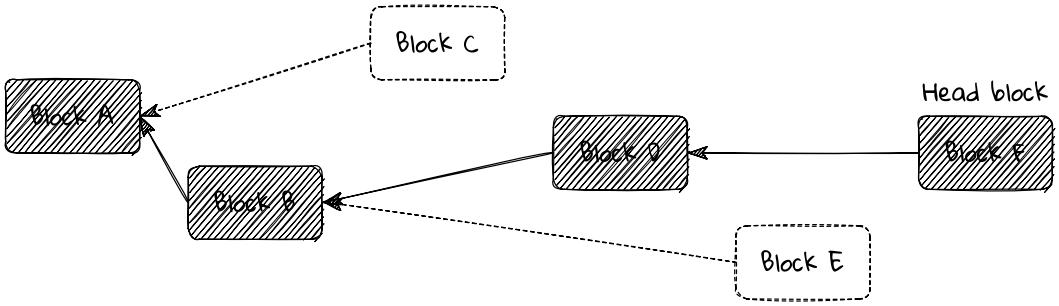
The various branches in the block tree are called "forks". Forks happen naturally as a consequence of network and processing delays. But they can also occur due to client faults, malicious client behaviour, or protocol upgrades that change the rules, making old blocks invalid with respect to the new rules. The last of these is often called a "hard fork".

The existence of forking in a consensus protocol is a consequence of prioritising liveness over safety, in the terms discussed below: if you were to consult nodes that are following different forks they would give you different answers regarding the state of the system. Non-forking consensus protocols exist, such as PBFT in the classical consensus world and Tendermint in the blockchain world. These protocols always produce a single linear chain and are thus formally "safe". However, they sacrifice liveness on asynchronous networks such as the Internet: rather than forking, they just stop entirely.

### Fork choice rules

Ultimately, we want every correct node on the network to converge on an identical linear view of history and hence a common view of the state of the system. This convergence is brought about by means of the protocol's fork choice rule.

Given a block tree and some decision criteria based on a node's local view of the network, the fork choice rule is designed to select, from all the available branches, the one that is most likely to eventually end up in the final linear, canonical chain. That is, it will choose the branch least likely to be later pruned out of the block tree as nodes attempt to converge on a canonical view.




*The fork choice rule selects a head block from among the candidates. This identifies a unique linear block chain running back to the Genesis block.*

The fork choice rule selects a branch implicitly by choosing a block at the tip of a branch, called the head block.

For any correct node, the first criterion for any fork choice rule is that the block it chooses must be valid according to the protocol's rules, and all its ancestors must be valid. Any invalid block is ignored, and any blocks built on an invalid block are themselves invalid.

Given that, there are many examples of different fork choice rules.

- The proof of work protocols in Ethereum and Bitcoin use a "heaviest chain rule"[2] (sometimes called "longest chain", though that's not strictly accurate). The head block is the tip of the chain that represents the most cumulative "work" done under proof of work.
- The fork choice rule in Ethereum's proof of stake Casper FFG protocol is "follow the chain containing the justified checkpoint of the greatest height", and to never revert a finalised block.
- The fork choice rule in Ethereum's proof of stake LMD GHOST protocol is specified in its name: take the "Greediest Heaviest Observed SubTree". It involves counting accumulated votes from validators for blocks and their descendent blocks. It also applies the same rule as Casper FFG.

We will properly unpack the second and third of these later in their respective sections.

You can perhaps see that each of these fork choice rules is a way to assign a numeric score to a block. The winning block, the head block, has the highest score. The idea is that all correct nodes, when they eventually see a certain block, will unambiguously agree that it is the head and choose to follow

[2]Contrary to popular belief, Ethereum's proof of work protocol does not use any form of GHOST in its fork choice. I really don't know why this misconception is so persistent - I eventually asked Vitalik about it and he confirmed to me (verbally) that although GHOST had been planned under PoW it was never implemented due to concerns about some unspecified attacks. The heaviest chain rule was simpler and well tested. It has served us well.

its branch whatever else is going on in their own views of the network. Thus, all correct nodes will eventually converge on a common view of a single canonical chain going back to genesis.

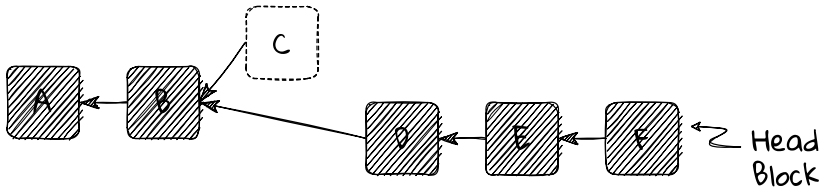
**Reorgs and reversions**

As a node receives new blocks (and, under proof of stake, new votes for blocks) it will re-evaluate the fork choice rule in the light of the new information. Most commonly, a new block will be a child of the block that it currently views as the head block. In this case the new block automatically becomes the updated head block (as long as it is valid).

However, sometimes the new block might be a descendent of some other block in the block tree. (Note that, if the node doesn't already have the parent block of the new block, it will need to ask its peers for it, and so on for any blocks it knows that it is missing.)

In any case, running the fork choice rule on the updated block tree might indicate a head block that is on a different branch from the previous head block. When this happens, the node must perform a reorg (short for reorganisation), also known as a reversion. It will kick out (revert) blocks that it had previously included in its chain, and will adopt the blocks on the new head's branch.

In the following diagram, the node has evaluated block $F$ to be the head block, hence its chain comprises blocks $A$, $B$, $D$, $E$, and $F$. The node knows about block $C$, but it does not appear in its view of the chain; it is on a side branch.

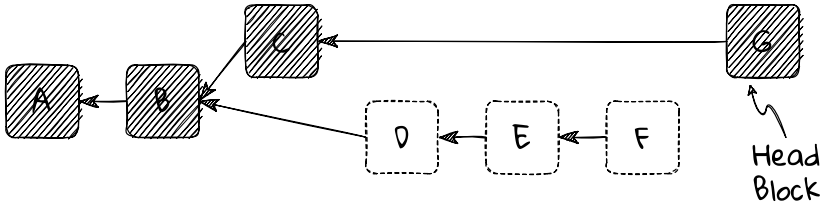



*At this point, the node believes that block $F$ is the best head, and therefore its chain is blocks $[A \leftarrow B \leftarrow D \leftarrow E \leftarrow F]$.*

Some time later the node receives block $G$ which is not built on its current head block $F$, but on block $C$ on a different branch. Depending on the details of the fork choice rule, the node might still evaluate $F$ to be a better head than $G$ and therefore ignore $G$. But in this case we will imagine that the fork choice rule indicates that $G$ is the better head block.

Blocks $D$, $E$, and $F$ are not ancestors of $G$, so they need to be removed from the node's canonical chain. Any transactions or information those blocks contain must be reverted, as if they were never received. The node must perform a full rewind to the state that it was in after processing block $B$.

After rewinding to $B$, the node can add blocks $C$ and $G$ to its chain and process them accordingly. After doing this, the node will have completed the reorganisation of its chain.




*Now the node believes that block $G$ is the best head, and therefore its chain must change to the blocks $[A \leftarrow B \leftarrow C \leftarrow G]$.*

Later, perhaps, a block $H$ might appear that builds on $F$. If the fork choice rule indicates that $H$ ought to be the new head, then the node will perform a reorg once again, reverting blocks back to $B$ and replaying the blocks on $H$'s branch.

Short reorgs of one or two blocks in both proof of work and Ethereum's proof of stake protocol are not uncommon due to network delays in block propagation. Much longer reorgs ought to be exceedingly rare, unless the chain is under attack, or there is a bug in the formulation of – or the clients' implementations of – the fork choice rule.

### Safety and Liveness

Two important concepts that crop up frequently when discussing consensus mechanisms are *safety* and *liveness*.

### Safety

Informally, an algorithm is said to be safe if "nothing bad ever happens"[3].

Examples of bad things that might happen in the blockchain context could be the double-spend of a coin, or the finalising of two conflicting checkpoints.

An important facet of safety in a distributed system is "consistency". That is, if we were to ask different (honest) nodes about the state of the chain at some point in its progress, such as the balance of an account at a particular block height, then we should always get the same answer, no matter which node we ask. In a safe system, every node has an identical view of the history of the chain that never changes.

Effectively, safety means that our distributed system "behaves like a centralized implementation that executes operations atomically one at a time." (to quote Castro and Liskov). A safe system is, in Vitalik's taxonomy of centralisation, logically centralised.

### Liveness

Again informally, an algorithm is said to be live if "something good eventually happens".

In a blockchain context we generally understand this to mean that the chain can always add a new block; it will never get into a deadlock situation in which it will not produce a new block with transactions in it.

"Availability" is another way of looking at this. I want the chain to be available, meaning that if I send a valid transaction to an honest node it will eventually be included in a block that extends the chain.

### You can't have both!

The CAP theorem is a famous result in distributed systems theory that states that no distributed system can provide all three of (1) consistency, (2) availability, and (3) partition tolerance. Partition tolerance is the ability to function when communication between nodes is not reliable. For example, a network fault might split the nodes into two or more groups that can't communicate with each other.

It is easy to demonstrate the CAP theorem in our blockchain context. Imagine that Amazon Web Services goes offline, such that all the AWS hosted nodes can communicate with each other, but none can talk to the outside world. Or that a country firewalls all connections in and out so that no gossip traffic can pass. Either of these scenarios divide the nodes into two disjoint groups, $A$ and $B$.

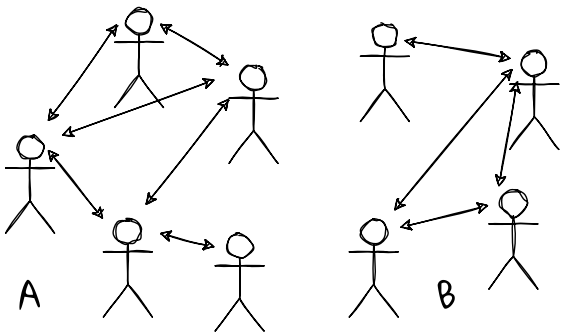
Let's say that somebody connected to the network of group $A$ sends a transaction. If the nodes in $A$ process that transaction then they will end up with a state that is different from the nodes in group $B$, which didn't see the transaction. So, overall, we have lost consistency between all the nodes, and therefore safety. The only way to avoid this is for the nodes in group $A$ to refuse to process the transaction, in which case we have lost availability, and therefore liveness.

In summary, the CAP theorem means that we cannot hope to design a consensus protocol that is both safe and live under all circumstances, since we have no option but to operate across an unreliable network, the Internet.[4]

---

[3] The helpful, intuitive definitions of safety and liveness I've quoted appear in short form in Lamport's 1977 paper, Proving the Correctness of Multiprocess Programs, and as stated here in Gilbert and Lynch's 2012 paper, Perspectives on the CAP Theorem.

[4] The CAP theorem is related to another famous result described by Fisher, Lynch and Paterson in their 1985 paper, Impossibility of Distributed Consensus with One Faulty Process, usually called the FLP theorem. This proves that, even




*The network is partitioned: the nodes in A can talk among themselves, but cannot talk to any node in B, and vice versa.*

### Ethereum prioritises liveness

The Ethereum 2 consensus protocol prioritises liveness: in the case of a network partition the nodes on each side of the partition will continue to produce blocks. However, finality (a safety property) will no longer occur on both sides of the partition. Depending on the proportion of stake managed by each side, either one side or neither side will continue to finalise.

Eventually, unless the partition is resolved, both sides will regain finality due to the novel inactivity leak mechanism. But this results in the ultimate safety failure. Each chain will finalise a different history and will become irreconcilable and independent forever.

It's worth noting that typical proof of work based algorithms also prioritise liveness over safety. In fact, Bitcoin and Ethereum's proof of work offer no safety guarantee at all; they have no concept of finality. At any time somebody might reveal a heavier chain that rewrites history. Even under non-adversarial conditions, minor forks happen frequently and there is no guarantee that different nodes you will give you the same answers. Exchanges typically use a proxy for safety that requires waiting for a certain number of blocks to be built on top of a transaction before it is considered final, but that's only a statistical guarantee, and is no guarantee at all in the face of a 51% attack.[5]

### Finality

Ethereum's proof of stake mechanism prioritises liveness, but unlike proof of work it also strives to offer a safety guarantee under favourable circumstances.

Safety in Ethereum 2 is called "finality", and is delivered by the Casper FFG mechanism that we'll explore shortly. The idea is that, as the blockchain progresses, all honest nodes agree on blocks that they will never revert. That block (a checkpoint) and all its ancestor blocks are then "final" - they will never change, and if you consult any honest node in the network about them or their ancestors you will always get the same answer. Thus, finality is a safety property: nothing bad ever happens.

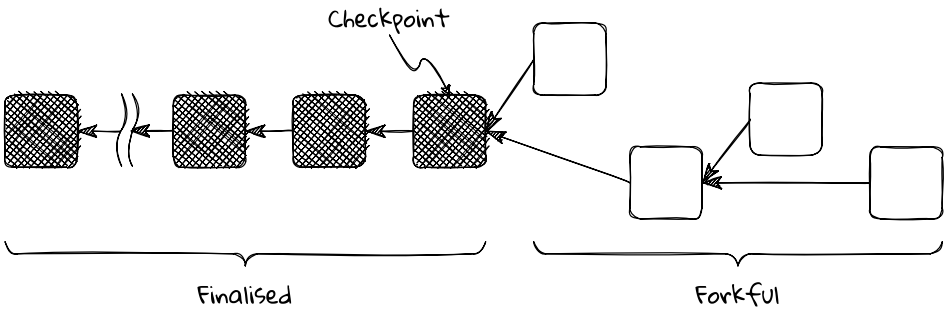
Finality in Ethereum 2 is "economic finality". It is theoretically possible for the protocol to finalise two conflicting checkpoints, that is, two contradictory views of the chain's history. However, it is possible only at enormous and quantifiable cost. For all but the most extreme attack or failure scenarios, final means final.

The next section, on Casper FFG, dives into the detail of how this finality mechanism works.

---

in a reliable asynchronous network (that is, with no bound on how long messages can take to be received), just one faulty node can prevent the system from coming to consensus. That is, even this unpartitioned system cannot be both live and safe. Gilbert and Lynch's paper discusses the FLP theorem in section 3.2.

[5] At the time of writing, at least one exchange requires 40000 confirmations for deposits from the Ethereum Classic network. That means that forty thousand blocks must be built on top of a block containing the deposit transaction before the exchange will process it, which takes about six days. The requirement reflects concern about the vulnerability of ETC's low hash rate proof of work chain to 51% attacks - it is relatively easy for an attacker to revert blocks at will. The reality is that, in the face of a well-crafted 51% attack, no number of confirmations is truly safe.




*The honest nodes have agreed that the checkpoint and all its ancestor blocks are "final" and will never be reverted. There are therefore no forks before the checkpoint. The chain descending from the checkpoint remains liable to forking.*

### See also

It's always worth reading anything that Lamport has had a hand in, and the original paper by Lamport, Shostak, and Pease on The Byzantine Generals Problem contains many insights. While the algorithm they propose is hopelessly inefficient in modern terms, the paper is a good introduction to reasoning about consensus protocols in general. The same is true of Castro and Liskov's seminal paper Practical Byzantine Fault Tolerance which significantly influences the design of Ethereum's Casper FFG protocol. However, you might like to contrast these "classical" approaches with the elegant simplicity of proof of work, as devised by Satoshi Nakamoto and described in the Bitcoin white paper. If proof of work has just one thing in its favour, it is its simplicity.

We've referred above to Gilbert and Lynch's 2012 paper, Perspectives on the CAP Theorem. It is a very readable exploration of the concepts of consistency and availability (or safety and liveness in our context).

The Eth2 beacon chain underwent a seven block reorg in May 2022 due to differences between client implementations of the fork choice rule. These differences were known at the time and thought to be harmless. That proved to be not so. Barnabé Monnot's write up of the incident is very instructive.

Vitalik's blog post On Settlement Finality provides a deeper and more nuanced exploration of the concept of finality.

Our ideal for the systems we are building is that they are *politically* decentralised (for permissionlessness and censorship resistance), *architecturally* decentralised (for resilience, with no single point of failure), but *logically* centralised (so that they give consistent results). These design criteria strongly influence how we build our consensus protocols. Vitalik explores these issues in his article, The Meaning of Decentralization.

## Casper FFG

TODO

## LMD Ghost

TODO

## Gasper

TODO

## Weak Subjectivity

TODO

### Issues

TODO

## The Progress of a Slot

### Introduction

TODO

### Proposing

TODO

### Attesting

TODO

### Aggregating

TODO

### Sync Committee Participation

TODO

# The Progress of an Epoch

## Introduction

TODO

## Applying Rewards and Penalties

TODO

## Justification and Finalisation

TODO

## Other State Updates

TODO

# Validator Lifecycle

## Introduction

TODO

# Deposit Handling

## Introduction

TODO

## The Deposit Contract

TODO

## Deposit Receipts

TODO

## Eth1 Voting and Follow Distance

TODO

## Merkle Proofs

TODO

## Deposit Processing

TODO

## Withdrawal Credentials

TODO

# The Incentive Layer

| First cut | ✓ | Revision | TODO |
|---|---|---|---|

## Carrots and Sticks and Sudden Death

Permissionless blockchains are cryptoeconomic systems: cryptography enforces correct behaviour where possible; economics incentivises correct behaviour where it cannot be enforced. The correct behaviours we're looking for roughly correspond to availability and security. We want the chain to keep making progress, and we want the chain to give reliable, non-contradictory results under all reasonable circumstances.

This chapter describes the economic tools the beacon chain uses to incentivise its participants; the cryptography side is covered elsewhere. Broadly speaking, the tools available to help us meet these goals are (1) rewards for behaviour that helps the protocol, (2) penalties for behaviour that hinders the protocol, and (3) punishments for behaviour that looks like an attack on the protocol.

One of the few attractive aspects of Proof of Work is the simplicity of its economic model. Miners receive block rewards for creating blocks that get included on chain, and receive fees for including transactions in their blocks. The block rewards come from newly created coins (issuance), and transaction fees are from previously issued coins. There are no explicit in-protocol penalties or punishments. Combined with the "heaviest chain" fork choice rule, this simple model has proved to be incredibly robust. Ethereum 1 added a little complexity with uncle rewards for miners and the EIP-1559 fee burning mechanism, but it remains fundamentally simple and fairly easy to reason about.

By contrast, the Ethereum 2.0 Proof of Stake protocol employs an array of different economic incentives. We will break things down into the following elements over the next sections.

1. The most fundamental economic component is the stake itself.
2. Within the protocol, the stake is represented in validator balances, in particular a quantity called the "effective balance" that is the actual measure of the influence a particular validator has on the protocol.
3. Similarly to proof of work, the protocol issues new coins to provide the incentives we are discussing. We'll look at this in the section on issuance.
4. An array of rewards is used to incentivise desirable behaviours such as publishing beacon blocks and timely attestations.
5. Penalties are used to disincentivise undesirable behaviours such as failing to make attestations, or making late or incorrect attestations.
6. The inactivity leak is a special regime that the beacon chain may enter in which rewards and penalties are modified to much more heavily penalise non-participation.
7. Slashings are punishments for breaking the protocol rules in very specific ways that look like attacks.
8. Finally, we close with a note on how aspects of these incentives combine to make diversity of deployment of beacon chain infrastructure the safest strategy.

### See also

Vlad Zamfir's memoirs on the development of the Casper Protocol are not only a great read, but a good introduction to the challenges of designing a proof of stake protocol. They discuss the background to many of the design decisions that led, eventually, to the protocol we see today. Part 1, Part 2, Part 3, Part 4, Part 5.

Much of the material in the following sections is also covered in the more recent report by Umberto Natale of Chorus One, Analysing Ethereum Cryptoeconomics: the validator's perspective.

## Staking

- The stake in proof of stake provides three things: an anti-Sybil mechanism, an accountability mechanism, and an incentive alignment mechanism.
- The 32 ETH stake size is a trade-off between network overhead, number of validators, and time to finality.
- Combined with the Casper FFG rules, stakes provide economic finality: a quantifiable measure of the security of the chain.

### Introduction

A stake is the deposit that a full participant of the Ethereum 2 protocol must lock up. The stake is lodged permanently in the deposit contract on the Ethereum chain, and reflected in a balance in the validator's record on the beacon chain. The stake entitles a validator to propose blocks, to attest to blocks and checkpoints, and to participate in sync committees, all in return for rewards that accrue to its beacon chain balance.

In Ethereum 2 the stake has three key roles.

First, the stake is an anti-Sybil mechanism. Ethereum 2 is a permissionless system that anyone can participate in. Permissionless systems must find a way to allocate influence among their participants. There must be some cost to creating an identity in the protocol, otherwise individuals could cheaply create vast numbers of duplicate identities and overwhelm the chain. In Proof of Work chains a participant's influence is proportional to its hash power, a limited resource[6]. In Proof of Stake chains participants must stake some of the chain's coin, which is again a limited resource. The influence of each staker in the protocol is proportional to the stake that they lock up.

Second, the stake provides accountability. There is a direct cost to acting in a harmful way in Ethereum 2. Specific types of harmful behaviour can be uniquely attributed to the stakers that performed them, and their stakes can be reduced or taken away entirely in a process called slashing. This allows us to quantify the economic security of the protocol in terms of what it would cost an attacker to do something harmful.

Third, the stake aligns incentives. Stakers necessarily own some of what they are guarding, and are incentivised to guard it well.

### Stake size

The size of the stake in Ethereum 2 is 32 ETH per validator.

This value is a compromise. It tries to be as small as possible to allow wide participation, while remaining large enough that we don't end up with too many validators. In short, if we reduced the stake, we would potentially be forcing stakers to run more expensive hardware on higher bandwidth networks, thus increasing the forces of centralisation.

The main practical constraint on the number of validators in a monolithic[7] L1 blockchain is the messaging overhead required to achieve finality. Like other PBFT-style consensus algorithms, Casper FFG requires

[6] In the Bitcoin white paper, Satoshi wrote that, "Proof-of-work is essentially one-CPU-one-vote", although ASICs and mining farms have long subverted this. Proof of Stake is one-stake-one-vote.

[7] A monolithic blockchain is one in which all nodes process all information, be it transactions or consensus-related. Pretty much all blockchains to date, including Ethereum, have been monolithic. One way to escape the scalability trilemma is to go "modular".

- More on the general scalability trilemma: Why sharding is great by Vitalik.
- More on modularity: Modular Blockchains: A Deep Dive by Alec Chen of Volt Capital.

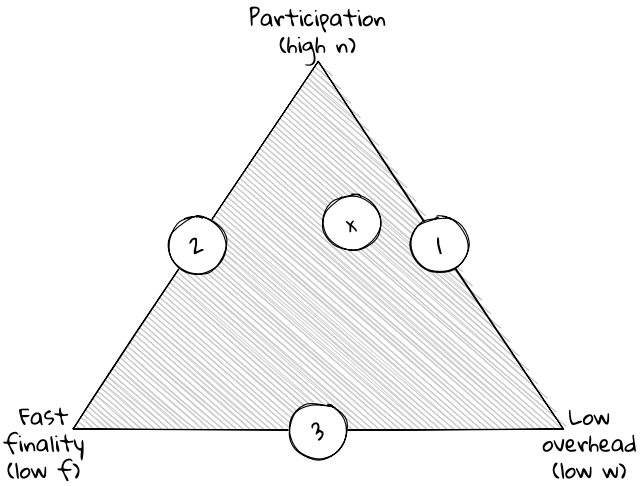
two rounds of all-to-all communication to achieve finality. That is, for all nodes to agree on a block that will never be reverted.

Following Vitalik's notation, if we can tolerate a network overhead of $\omega$ messages per second, and we want a time to finality of $f$, then we can have participation from at most $n$ validators, where

$$n \leq \frac{\omega f}{2}$$

We would like to keep $\omega$ small to allow the broadest possible participation by validators, including those on slower networks. And we would like $f$ to be as short as possible since a shorter time to finality is much more useful than a longer time[8]. Taken together, these requirements imply a cap on $n$, the total number of validators.

This is a classic scalability trilemma. Personally, I don't find these pictures of triangles very intuitive, but they have become the canonical way to represent the trade-offs.




*A version of the scalability trilemma: pick any two.*

1. Our ideal might be to have high participation (large $n$) with low overhead (low $\omega$) – lots of stakers on low-spec machines –, but finality would take a long time since message exchange would be slow.
2. We could have very fast finality and high participation, but would need to mandate that stakers run high spec machines on high bandwidth networks in order to participate.
3. Or we could have fast finality on reasonably modest machines by severely limiting the number of participants.

It's not clear exactly how to place Ethereum 2 on such a diagram, but we definitely favour participation over time to finality: maybe "x" marks the spot. One complexity is that participation and overhead are not entirely independent: we could decrease the stake to encourage participation, but that would increase the hardware and networking requirements (the overhead), which will tend to reduce the number of people able or willing to participate.[9]

To put this in concrete terms, the hard limit on the number of validators is the total Ether supply divided by the stake size. With a 32 ETH stake, that's about 3.6 million validators today, which is consistent with a time to finality of 768 seconds (two epochs), and a message overhead of 9375 messages

[8] In an unfinished paper Vitalik attempts to quantify the "protocol utility" for different times to finality.

> ...a blockchain with some finality time $f$ has utility roughly $-\log(f)$, or in other words increasing the finality time of a blockchain by a constant factor causes a constant loss of utility. The utility difference between 1 minute and 2 minute finality is the same as the utility difference between 1 hour and 2 hour finality.

He goes on to make a justification for this (p.10).

[9] Exercise for the reader: try placing some of the other monolithic L1 blockchains within the trade-off space.

per second[10]. That's a substantial number of messages per second to handle. However, we don't ever expect *all* Ether to be staked, perhaps around 10-20%. In addition, due to the use of BLS aggregate signatures, messages are highly compressed to an asymptotic 1-bit per validator.

Given the capacity of current p2p networks, 32 ETH per stake is about as low as we can go while delivering finality in two epochs. Anecdotally, my staking node continually consumes about 3.5mb/s in up and down bandwidth. That's about 30% of my upstream bandwidth on residential ADSL. If the protocol were more any chatty it would rule out home staking for many.

An alternative approach might be to cap the number of validators active at any one time to put an upper bound on the number of messages exchanged. With something like that in place, we could explore reducing the stake below 32 ETH, allowing many more validators to participate, but each participating only on a part-time basis.

Note that this analysis overlooks the distinction between nodes (which actually have to handle the messages) and validators (a large number of which can be hosted by a single node). A design goal of the Ethereum 2 protocol is to minimise any economies of scale, putting the solo-staker on as equal as possible footing with staking pools. Thus we ought to be careful to apply our analyses to the most distributed case, that of one-validator per node.

Fun fact: the original hybrid Casper FFG PoS proposal (EIP-1011) called for a minimum deposit size of 1500 ETH as the system design could handle up to around 900 active validators. While 32 ETH now represents a great deal of money for most people, decentralised staking pools that can take less than 32 ETH are now becoming available.

### Economic finality

The requirement for validators to lock up stakes, and the introduction of slashing conditions allows us to quantify the security of the beacon chain in some sense.

The main attack we wish to prevent is one that rewrites the history of the chain. The cost of such an attack parameterises the security of the chain. In proof of work, this is the cost of acquiring an overwhelming (51%) of hash power for a period of time. Interestingly, a successful 51% attack in proof of work costs essentially nothing, since the attacker claims all of the block rewards on the rewritten chain.

In Ethereum's proof of stake protocol we can measure security in terms of *economic finality*. That is, if an attacker wished to revert a finalised block on the chain, what would be the cost?

This turns out to be easy to quantify. To quote Vitalik's Parametrizing Casper,

> State $H_1$ is economically finalized if enough validators sign a message attesting to $H_1$, with the property that if both $H_1$ and a conflicting $H_2$ are finalized, then there is evidence that can be used to prove that at least $\frac{1}{3}$ of validators were malicious and therefore destroy their entire deposits.

Ethereum's proof of stake protocol has this property. In order to finalise a checkpoint ($H_1$), two-thirds of the validators must have attested to it. To finalise a conflicting checkpoint ($H_2$) requires two-thirds of validators to attest to that as well. Thus, at least one-third of validators must have attested to both checkpoints. Since individual validators sign their attestations, this is both detectable and attributable: it's easy to submit the evidence on-chain that those validators contradicted themselves, and they can be punished by the protocol.

In the Altair implementation, if one-third of validators were to be slashed simultaneously, they would have around two-thirds of their stake burned (20 ETH each). The plan is to increase this later to their full stake (32 ETH each). At that point with, say, nine million Ether staked, the cost of reverting a finalised block would be three million of the attackers' Ether being permanently burned and the attackers being expelled from the network.

It is obligatory at this point to quote (or paraphrase) Vlad Zamfir: comparing proof of stake to proof of work, "it's as though your ASIC farm burned down if you participated in a 51% attack".

For more on the mechanics of economic finality, see below under Slashing, and for more on the rationale and justification, see the section on Casper FFG. [TODO: link to Casper FFG when written.]

[10] Vitalik's estimate of 5461 is too low since he omits the factor of two in the calculation.

**See also**

- Parametrizing Casper: the decentralization/finality time/overhead tradeoff presents some early reasoning about the trade-offs for different stake sizes. Things have moved on somewhat since then, most notably with the advent of BLS aggregate signatures.
- Why 32 ETH validator sizes? from Vitalik's Serenity Design Rationale.

Vitalik's discussion document around achieving single slot finality looks at the participation/overhead/finality trade-off space from a different perspective.

## Balances

- Each validator maintains an *effective balance* in addition to its actual balance.
- The validator's influence in the protocol is proportional to its effective balance, as are its rewards and penalties.
- The effective balance tracks the validator's actual balance, but is designed to change much more rarely. This is an optimisation.
- A validator's effective balance is capped at 32 ETH.

### Introduction

The beacon chain maintains two separate records of each validator's balance: its actual balance and its effective balance.

A validator's actual balance is straightforward. It is the sum of any deposits made for it via the deposit contract, plus accrued beacon chain rewards, minus accrued penalties. Withdrawals are not yet possible, but will be subtracted from this balance when available. The actual balance is rapidly changing, being updated at least once per epoch for all active validators, and every slot for sync committee participants. It is also fine-grained: units of the actual balance are Gwei, that is, $10^{-9}$ ETH.

A validator's effective balance is derived from its actual balance in such a way that it changes much more slowly. To achieve this, the units of effective balance are whole Ether (see `EFFECTIVE_BALANCE_INCREMENT`), and changes to the effective balance are subject to hysteresis.

Using the effective balance achieves two goals, one to do with economics, the other purely engineering.

### Economic aspects of effective balance

The effective balance was first introduced to represent the "maximum balance at risk" for a validator, capped at 32 ETH. A validator's actual balance could be much higher, for example if a double deposit had been accidentally made a validator would have an actual balance of 64 ETH but an effective balance of only 32 ETH. We could envisage a protocol in which each validator has influence proportional to its uncapped actual balance, but that would complicate committee membership among other things. Instead we cap the effective balance and require stakers to deposit for more validators if they wish to stake more.

The scope of effective balance quickly grew, and now it completely represents the weight of a validator in the consensus protocol.

All of the following consensus-related matters are proportional to the effective balance of a validator:

- the probability of being selected as the beacon block proposer;
- the validator's weight in the LMD-GHOST fork choice rule;

- the validator's weight in the justification and finalisation calculations calculations; and
- the probability of being included in a sync committee.

Correspondingly, the following rewards, penalties, and punishments are also weighted by effective balance:

- the base reward for a validator, in terms of which the attestation rewards and penalties are calculated;
- the inactivity penalties applied to a validator as a consequence of an inactivity leak; and
- both the initial slashing penalty and the correlated slashing penalty.

However, the block proposer reward is not scaled in proportion to the proposer's effective balance. Since a validator's probability of being selected to propose is proportional to its effective balance, the reward scaling with effective balance is already taken care of. For the same reason sync committee rewards are not proportional to the participants' effective balances either.

#### Engineering aspects of effective balance

We could achieve all of the above simply by using validators' actual balances as their weights, capped at 32 ETH. However, we can gain significant performance benefits by basing everything on effective balances instead.

For one thing, effective balances are updated only once per epoch, which means that we need only calculate things like the base reward per increment once and we can cache the result for the whole epoch, irrespective of any changes in actual balances.

But the main feature of effective balances is that they are designed to change much more rarely than that. This is achieved by making them very granular, and by applying hysteresis to any updates.

One of the big performance challenges in calculating the beacon chain state transition is generating the hash tree root of the entire state. The Merkleization process allows parts of the state that have not been changed to be cached, providing a significant performance boost.

The list of validator records in the state is a large data structure. Were we to store the validators' actual balances within those records they would be frequently changing and the whole data structure would need to be re-hashed at least once per epoch.

The first approach to addressing this simply moved the validators' balances out of the validator records into a dedicated list in the state. This reduces the amount of re-hashing required as the whole validator list does not need to be re-hashed when only the validators' balances change.

However, that leads to a performance issue elsewhere. Light clients needing information on validators' balances would now need to acquire data from two different parts of the state – both the validator record and the validator balance list. This requires two Merkle proofs rather than one, significantly increasing their bandwidth costs.

A way round this is to store a slowly changing version of the balances in the validators' records – meaning that they need to be re-hashed infrequently – and to store the fast-changing actual balances in a separate list, a much smaller structure to re-hash.

From the notes for an early attempt at a kind of effective balance implementation:

> [Effective balances are an] "approximate balance" that can be used by light clients in the `validator_registry`, reducing the number of Merkle branches per validator they need to download from 3 to 2 (actually often from ~2.01 to ~1.01, because when fetching a committee the Merkle branches in active_index_roots are mostly shared), achieving a very significant decrease in light client bandwidth costs

The point is that light clients will not need to access the list of actual balances that is stored separately in state, only the validator records they were downloading anyway.

In summary, adding effective balances to validators' records allows us to achieve two performance goals simultaneously: avoiding the workload of frequently re-hashing the validator list in the state while not increasing the workload of light clients.

### Increments

Although effective balances are denominated in Gwei they can only be whole multiples of `EFFECTIVE_BALANCE_INCREMENT`, which is 1 ETH ($10^9$ Gwei). Actual balances can be any number of Gwei.

This multiple is known in the spec as an "increment" and shows up in places like calculating the base reward, and other rewards and penalties calculations. Being a handy 1 ETH, it's easy to mentally substitute "Ether" for "increment" to gain some intuition.

It would probably be cleaner to store effective balance in terms of increments instead of Gwei. It would certainly reduce the amount of dividing and multiplying by `EFFECTIVE_BALANCE_INCREMENT` that goes on, and the associated danger of arithmetic overflows. But the current version evolved over time, and it would be intrusive and risky to go back and change things now.

### Hysteresis

Effective balances are guaranteed to vary much more slowly than actual balances by adding hysteresis to their calculation.

In our context, hysteresis means that if the effective balance is 31 ETH, the actual balance must rise to 32.25 ETH to trigger an effective balance update to 32 ETH. Similarly, if the effective balance is 31 ETH, then the actual balance must fall to 30.75 ETH to trigger an effective balance update to 30 ETH.

The following chart illustrates the behaviour.

- The actual balance and the effective balance both start at 32 ETH.
- Initially the actual balance rises. Effective balance is capped at 32 ETH, so it does not get updated.
- Only when the actual balance falls below 31.75 ETH does the effective balance get reduced to 31 ETH.
- Although the actual balance rises and oscillates around 32 ETH, no effective balance update is triggered and it remains at 31 ETH.
- Eventually the actual balance rises above 32.25 ETH, and the effective balance is updated to 32 ETH.
- Despite the actual balance falling again, it does not fall below 31.75 ETH, so the effective balance remains at 32 ETH.

The hysteresis levels are controlled by the hysteresis parameters in the spec:

| Name | Value |
|---|---|
| `HYSTERESIS_QUOTIENT` | `uint64(4)` |
| `HYSTERESIS_DOWNWARD_MULTIPLIER` | `uint64(1)` |
| `HYSTERESIS_UPWARD_MULTIPLIER` | `uint64(5)` |

These are applied at the end of each epoch during effective balance updates. Every validator in the state (whether active or not) has its effective balance updated as follows:

- If actual balance is less than effective balance minus 0.25 ( `= HYSTERESIS_DOWNWARD_MULTIPLIER / HYSTERESIS_QUOTIENT`) increments (ETH), then reduce the effective balance by an increment.
- If actual balance is more than effective balance plus 1.25 ( `= HYSTERESIS_UPWARD_MULTIPLIER / HYSTERESIS_QUOTIENT`) increments (ETH), then increase the effective balance by an increment.

The effect of the hysteresis is that the effective balance cannot change more often than it takes for a validator's actual balance to change by 0.5 ETH, which would normally take several weeks or months.

Historical note: the initial implementation of hysteresis effectively had `QUOTIENT = 2`, `DOWNWARD_MULTIPLIER = 0`, and `UPWARD_MULTIPLIER = 3`. This meant that a validator starting with 32 ETH actual balance but suffering a minor initial outage would immediately drop to 31 ETH effective balance. To get back to 32 ETH effective balance it would need to achieve a 32.5 ETH actual balance, and meanwhile the validator's

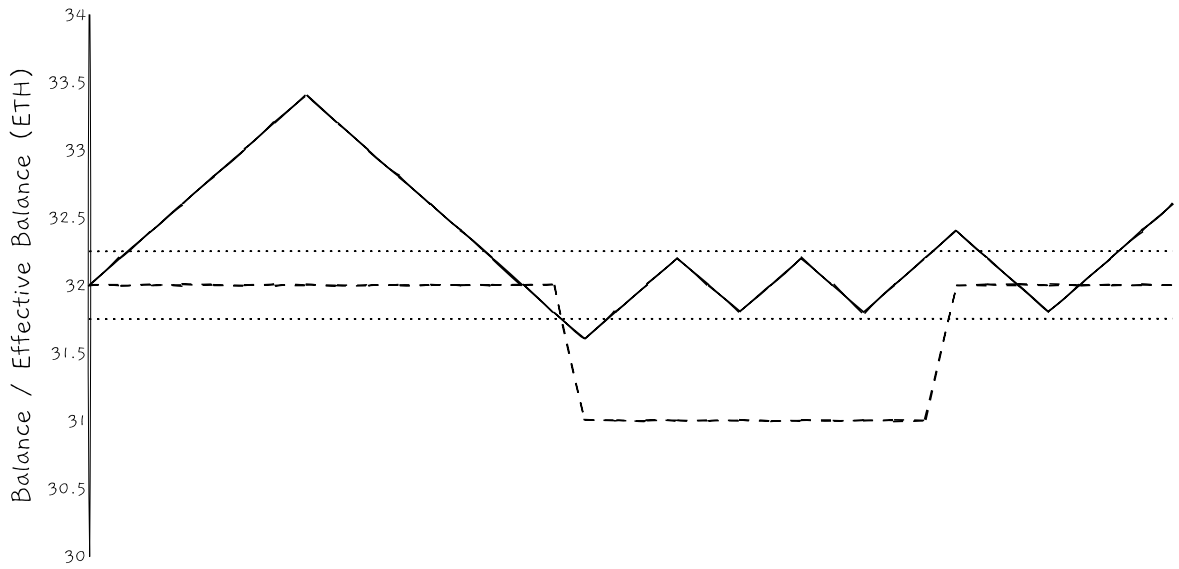



*Illustration of the relationship between the actual balance (solid line) and the effective balance (dashed line) of a validator. The dotted lines are the thresholds at which the effective balance gets updated - the hysteresis.*

rewards would be 3.1% lower due to the reduced effective balance. This seemed unfair, and incentivised stakers to "over-deposit" Ether to avoid the risk of an initial effective balance drop. Hence the change to the current parameters.

#### See also

From the spec:

- The presets that constrain the effective balance, `MAX_EFFECTIVE_BALANCE` and `EFFECTIVE_BALANCE_INCREMENT`.
- The parameters that control the hysteresis.
- The function `process_effective_balance_updates()` for the actual calculation and application of hysteresis.
- `Validator` objects store the effective balances. The registry in the beacon state contains the list of validators alongside a separate list of the actual balances.

## Issuance

- Issuance is the amount of new Ether created by the protocol in order to incentivise its participants.
- An ideally running beacon chain issues a set amount of Ether per epoch, which is a multiple of the base reward per increment.
- Total issuance is proportional to the square root of the number of validators. This is not a completely arbitrary choice.

### Introduction

There are three views we can take of the rewards given to validators to incentivise their correct participation in the protocol.

First, there is "issuance", which is the overall amount of new Ether generated by the protocol to pay rewards. Second there is the expected reward a validator might earn over the long run. And, third, there is the actual reward that any particular validator earns.

In this section we will look at issuance, and in the next we'll look at rewards. There is a strong relationship between these, though, so the separation is not totally clean.

First we must define the fundamental unit of reward, which is the "base reward per increment".

### The base reward per increment

All rewards are calculated in terms of a "base reward per increment". This is in turn calculated as

```
Gwei(EFFECTIVE_BALANCE_INCREMENT * BASE_REWARD_FACTOR //
                                    ↪ integer_squareroot(get_total_active_balance(state)))
```

We will call the base reward per increment $b$ for brevity. An increment is one unit of effective balance, which is 1 ETH (`EFFECTIVE_BALANCE_INCREMENT`), so active validators have up to 32 increments.

The `BASE_REWARD_FACTOR` is the big knob that we could turn if we wished to change the issuance rate of Ether on the beacon chain. So far it's always been set at 64 which results in the issuance graph we see below. This seems to be working very well and there are no plans to change it.

### Rewards come from issuance

Issuance is the amount of new Ether created by the protocol in order to incentivise its participants. The net issuance, after accounting for penalties, burned transaction fees and so forth is sometimes referred to as inflation, or supply growth.

The Eth1 chain issues new Ether in the form of block and uncle rewards. Since the London upgrade this issuance has been offset in part, or even at times exceeded by the burning of transaction base fees due to EIP-1559.

During the current Altair period, issuance on the beacon chain is additional to that on the Eth1 chain, but is much smaller (around 10% as much). After The Merge, there will no longer be any block or uncle rewards on the Eth1 chain. But the base fee burn will remain, and it is very likely that the net issuance will become negative – more Ether will be destroyed than created[11] – at least in the short to medium term. In the longer term, Anders Elowsson argues that there will be a circulating supply equilibrium arising from Ether issuance by proof of stake and Ether destruction due to EIP-1559.

In the following we will be assuming that the beacon chain is running optimally, that is, with all validators performing their duties perfectly. In reality this is impossible to achieve on a permissionless, globally distributed, peer-to-peer network, although the beacon chain has been performing within a few percent of optimally for most of its history. In any case, actual validator rewards and net issuance will certainly be a little or a lot lower, depending on participation rates in the network.

### Overall issuance

Under the ideal conditions we are assuming, the beacon chain is designed to issue a total of exactly $Tb$ Gwei in rewards per epoch. Here, $T$ is the total number of increments held by active validators, or in other words the total of all their effective balances in Ether. This is the maximum issuance – the maximum amount of new Ether – that the beacon chain can generate. If all $N$ validators have the maximum 32 ETH effective balance, then this works out to be $32Nb$ Gwei per epoch in total.

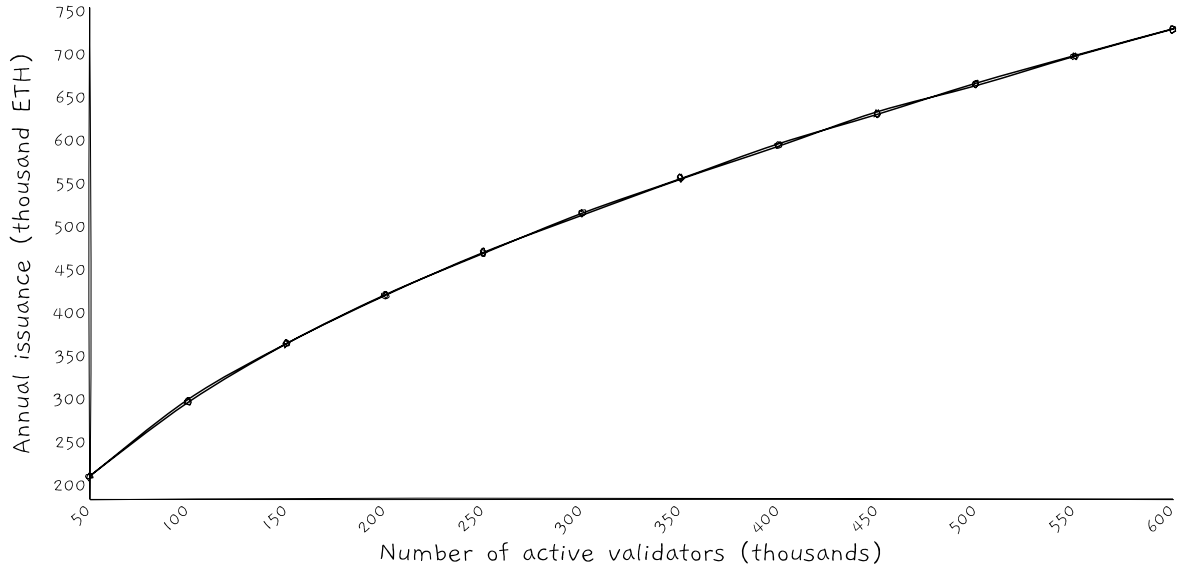
With $365.25 \times 225 = 82181.25$ epochs per year, and `BASE_REWARD_FACTOR` $= 64$,

[11] You can see Ethereum's current issuance and play with various scenarios at ultrasound.money.

$$\begin{aligned}\text{Max issuance per year} &= 82181.25 \times \frac{32 \times 64 \times N}{\sqrt{32 \times 10^9 \times N}}\text{ETH} \\ &= 940.87\sqrt{N}\end{aligned}$$

With 300,000 validators this equates to 515,333 ETH per year, plus change. For comparison, the Eth1 block and uncle rewards currently amount to almost five million ETH per year.

We can graph the maximum issuance as a function of the number of validators. It's just a scaled square root curve.




*Maximum annual protocol issuance on the beacon chain as a function of the number of active validators.*

### Validator rewards

The goal is to distribute these rewards evenly among validators (continuing to assume that things are running optimally), so that, on a long term average, each validator $i$ earns $n_i b$ Gwei per epoch, where $n_i$ is the number of increments it possesses, equivalently its effective balance in Ether. In these terms $T = \sum_{i=0}^{N-1} n_i$.

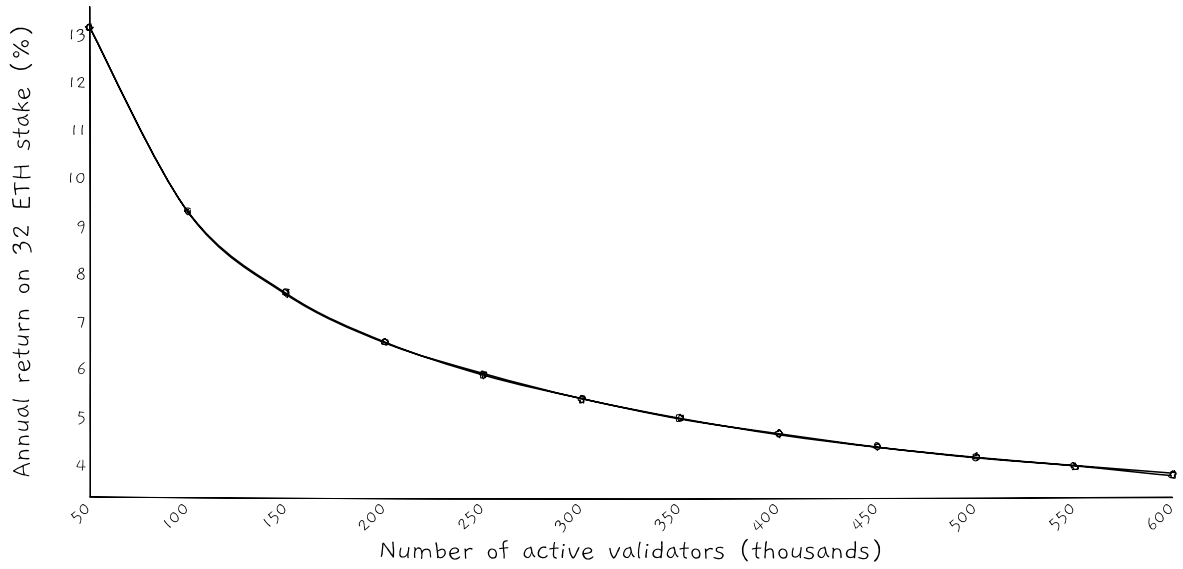
Thus, a well-performing validator with a 32 ETH effective balance can expect to earn a long-term average of $32b$ Gwei per epoch. Of course, $b$ changes over time as the total active balance changes, but absent a mass slashing event that change will be slow.

Similarly to the issuance calculation, we can calculate the expected annual percentage reward for a validator due to participating in the beacon chain protocol:

$$\begin{aligned}\text{APR} &= 100 \times 82181.25 \times \frac{64}{\sqrt{32 \times 10^9 \times N}}\% \\ &= \frac{2940.21}{\sqrt{N}}\%\end{aligned}$$

For example, with 300,000 validators participating, this amounts to an expected return of 5.37% on a validator's effective balance.

Graphing this give us an inverse square root curve.




*The expected annual percentage rewards for stakers as a function of the number of active validators.*

**Inverse square root scaling**

The choice to scale the per-validator expected reward with $\frac{1}{\sqrt{N}}$ is not obvious, and we can imagine different scenarios.

If we model the per-validator reward as $r \propto N^{-p}$, then some options are as follows.

1. $p = 0$: each validator earns a constant return regardless of the total number of validators. Issuance is proportional to $N$.
2. $p = \frac{1}{2}$: issuance scales like $\sqrt{N}$, the formula we are using.
3. $p = 1$: each validator's expected reward is inversely proportional to the total number of validators. Issuance is independent of the total number of validators.

Adopting a concave function is attractive as it allows an equilibrium number of validators to be discovered without constantly fiddling with parameters. Ideally, if more validators join, we want the per-validator reward to decrease to disincentivise further joiners; if validators drop out we want the per-validator reward to increase to encourage new joiners. Eventually, an equilibrium number of validators will be found that balances the staking reward against the perceived risk and opportunity cost of staking. Assuming that the protocol is not overly sensitive to the total number of validators, this seems to be a nice feature to have.

That would rule out the first, $p = 0$, option. The risk with $p = 0$ is that, if the reward rate is set lower than the perceived risk, then all rational validators will exit. If we set it too high, then we end up paying for more security than we need (too many over-incentivised validators). Frequent manual tuning via hard-forks could be required to adjust the rate.

The arguments for selecting $p = \frac{1}{2}$ over $p = 1$ are quite subtle and relate to discouragement attacks. With $p \neq 0$, a set of validators may act against other validators by censoring them, or performing other types of denial of service, in order to persuade them to exit the system, thus increasing the rewards for themselves. Subject to various assumptions and models, we find that we require $p \leq \frac{1}{2}$ for certain kinds of attack to be profitable. Essentially, we don't want to increase rewards too much for validators that succeed in making other validators exit the beacon chain.

Note that after the Merge, validators' income will include a significant component from transaction tips and MEV, which will have the effect of pushing $p$ closer to 1, and much of this reasoning will become moot. Discouragement attacks in that regime are an unsolved problem.

### See also

For more background to the $\frac{1}{\sqrt{N}}$ reward curve, see

- Casper: The Fixed Income Approach,
- Vitalik's Serenity Design Rationale, and
- the Discouragement Attacks paper.

Anders Elowsson's work on Ethereum's circulating supply equilibrium and minimum viable issuance takes a deeper look at the relationship between staking issuance and total Ether supply. See his post and comments on Ethresear.ch, and ETHconomics presentation at Devconnect 2022.

## Rewards

- Validators receive rewards for making attestations according to their view of the chain, proposing blocks, and participating in sync committees in varying proportions.
- Votes that make up attestations must be both correct and timely in order to be rewarded.
- The proposer's reward is a fixed proportion (1/7) of the total reward for all the duties it is including in its block.
- A validator's expected long-term reward is $nb$ per epoch (number of increments times the base reward per increment), but there is significant variance around that due to the randomness of proposer and sync committee assignments.
- Rewards are scaled both with a validator's effective balance and with the total participation rate of the validator set.
- The need to defend against discouragement attacks has shaped various aspects of the protocol.

### Introduction

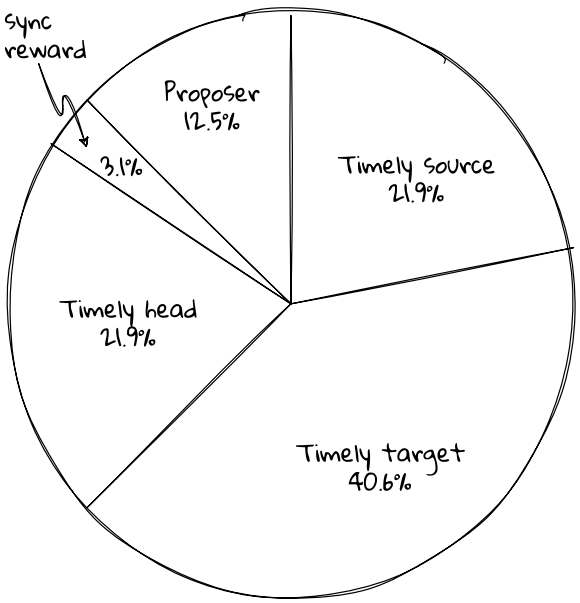
In this section we will consider only rewards. We'll cover penalties in the next section.

The beacon chain protocol incentivises each validator to behave well by providing rewards for three activities as follows.

1. Attesting to its view of the chain as part of the consensus protocol:
    - voting for a source checkpoint for Casper FFG;
    - voting for a target checkpoint for Casper FFG; and
    - voting for a chain head block for LMD-GHOST.
2. Proposing beacon chain blocks.
3. Signing off on blocks in the sync committees that support light clients.

The first of these, making attestations, happens regularly every epoch and accounts for the majority a validator's total expected reward.

However, validators are selected at random to propose blocks or participate in sync committees, so there is a natural variance to the latter two rewards. Over the long run, the expected proportion of rewards earned for each activity breaks down as per the following chart.




*The proportion of a validator's total reward derived from each activity.*

These proportions are set by the incentivisation weights in the spec. For convenience, I've assigned a symbol to each weight in the last column.

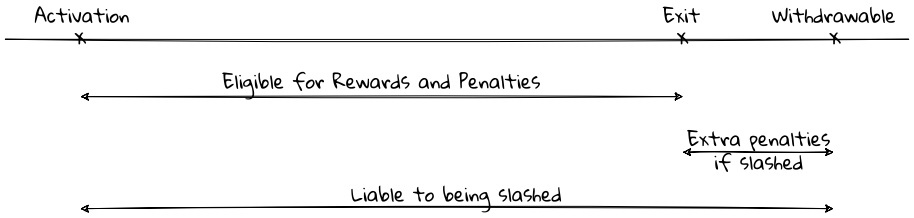
| Name | Value | Percentage | Symbol |
|---|---|---|---|
| `TIMELY_SOURCE_WEIGHT` | `uint64(14)` | 21.9% | $W_s$ |
| `TIMELY_TARGET_WEIGHT` | `uint64(26)` | 40.6% | $W_t$ |
| `TIMELY_HEAD_WEIGHT` | `uint64(14)` | 21.9% | $W_h$ |
| `SYNC_REWARD_WEIGHT` | `uint64(2)` | 3.1% | $W_y$ |
| `PROPOSER_WEIGHT` | `uint64(8)` | 12.5% | $W_p$ |
| `WEIGHT_DENOMINATOR` | `uint64(64)` | 100% | $W_\Sigma$ |

One further reward is available to block proposers for reporting violations of the slashing rules, but this ought to be very rare and we will ignore it in this section (see Slashing for more on this).

Rewards are newly created Ether that is simply added to validators' balances on the beacon chain.

### Eligibility for rewards

There are three relevant milestones in a validator's lifecycle: its activation epoch, its exit epoch, and its withdrawable epoch. Eligibility for rewards, penalties and slashing vary based on these.




*Timeline of the eligibility of validators for rewards*

Validators may receive rewards only between their activation and exit epochs. Note that, after submitting a voluntary exit, there may be a delay while the validator moves through the exit queue until its exit epoch is passed. The validator is expected to participate as usual during this period.

Similarly, validators receive penalties only between their activation and exit epochs. The exception to this is slashed validators. As a special case, slashed validators continue to receive penalties until they reach their withdrawable epoch, which may be long after their exit epoch.

All unslashed validators that are between their activation epoch and their withdrawable epoch are liable to being slashed.

For the random elements – block proposals and sync committee participation – the scaling is achieved implicitly by modifying the probability that a validator is selected for duty to be proportional to $\frac{n}{T}$, where $T$ is the total number of increments of the active validator set. So, if your effective balance is 24 ETH, then you are 25% less likely to be selected to propose a block or join a sync committee than a validator with 32 ETH. See `compute_proposer_index()` and `get_next_sync_committee_indices()` for the details.

### Attestation rewards

The largest part, 84.4%, of validators' rewards come from making attestations. Although committee and slot assignments for attesting are randomised, every active validator will be selected to make exactly one attestation each epoch.

Attestations receive rewards only if they are included in beacon chain blocks. An attestation contains three votes. Each vote is eligible for a reward subject to conditions.

| Validity | Timeliness | Reward |
|---|---|---|
| Correct source | Within 5 slots | $\frac{W_s}{W_\Sigma} nb$ |
| Correct source and target | Within 32 slots | $\frac{W_t}{W_\Sigma} nb$ |
| Correct source, target and head | Within 1 slot | $\frac{W_h}{W_\Sigma} nb$ |

These are cumulative, so the maximum attestation reward per epoch (for getting all three votes correct and getting the attestation included the next block) is $\frac{W_s+W_t+W_h}{W_\Sigma} nb$, or $0.84375nb$.

The full matrix of possible weights for an attestation reward is as follows. In each case we need to multiply by $\frac{nb}{W_\Sigma}$ to get the actual reward.

| Timeliness | 1 slot | <= 5 slots | <= 32 slots | > 32 Slots (missing) |
|---|---|---|---|---|
| Wrong source | 0 | 0 | 0 | 0 |
| Correct source | $W_s$ | $W_s$ | 0 | 0 |
| Correct source and target | $W_s + W_t$ | $W_s + W_t$ | $W_t$ | 0 |
| Correct source, target and head | $W_s + W_t + W_h$ | $W_s + W_t$ | $W_t$ | 0 |

But this is not the whole picture: we will also need to account for penalties for incorrect or late attestations.

The maximum total issuance per epoch across all validators is

$$I_A = \frac{W_s + W_t + W_h}{W_\Sigma} Tb$$

where, once again, $T$ is the total number of increments of active validators (the sum of their effective balances in ETH terms).

### Correctness

"Correct" in the above means that the attestation agrees with the view of the blockchain that the current block proposer has. If the attesting validator votes for different checkpoints or head blocks then it is on a different fork and that vote is not useful to us. For instance, if the source checkpoint vote is different from what we as proposer think it ought to be, then our view of the chain's history is fundamentally different from the attester's, and so we must ignore their attestation. The attestation will instead receive

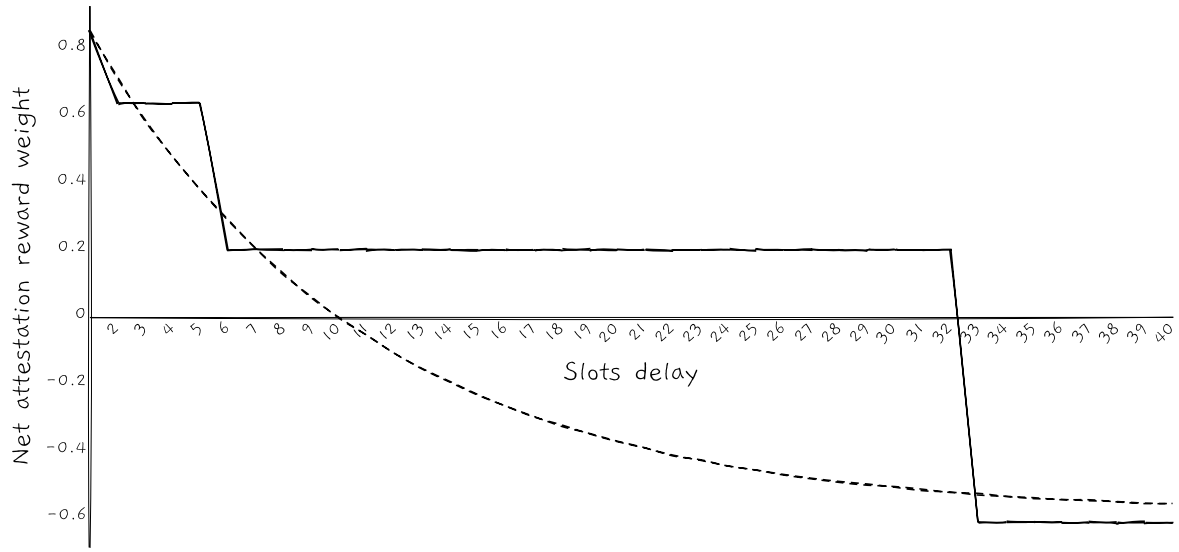
rewards in blocks on the other fork, and eventually one fork or the other fork will win. To disincentivise attacks it is important that only participants in the winning chain receive rewards.

### Timeliness

One of the changes brought in with Altair was a tightening of the timeliness requirements for attestations. Previously, there were rewards for correctness and a separate reward for timely inclusion that declined as $\frac{1}{d}$, where $d$ was the inclusion distance in slots, up to a maximum of 32 slots. This led to oddities, like it being worth waiting slightly longer to make sure to get the head vote correct since that was worth more than any loss due to lateness of inclusion, even though a late head vote is pretty much useless.

The new timeliness reward better reflect the relative importance of the votes. A head vote that is older than one slot is not useful, so it gets no reward, Target votes are always useful, but we only want to track attestations pertaining to the current and previous epochs, so we ignore them if they are older than 32 slots.

The choice of distance for including the source vote is interesting. It is chosen to be $\lfloor\sqrt{\texttt{SLOTS\_PER\_EPOCH}}\rfloor = \lfloor\sqrt{32}\rfloor = 5$, which is the geometric mean of 1 and 32, the head and target values. It's a somewhat arbitrary choice, but is intended to put a fully correct attestation on an exponentially decreasing curve with respect to timeliness: each step down in (net) reward happens after an exponentially increasing number of slots.[13]




*It is plausible that setting the inclusion distance for correct source to 5 gives a kind of exponential reduction in reward with time. This graph shows the net reward (reward + penalty) for a completely correct attestation as it gets older plotted against an exponential curve for comparison.*

### Remarks

Note that the attester does not have full control over whether it receives rewards or not. An attester

[13]This is taken from a conversation on the Ethereum R&D Discord server:

> vbuterin:
> The rationale for the number 5 is just that 5 is geometrically halfway in between 1 and 32
> And so we get the closest that makes sense to a smooth curve in terms of rewarding earlier inclusion
> …
> ah I mean on an exponential curve, not quadratic
> To me exponential feels more logical
> What's a bigger improvement in quality, 4 slot delay vs 6 slot delay, or 20 slot delay vs 23 slot delay?

may behave perfectly, but if the next block is skipped because the proposer is offline, then it will not receive the correct head block reward. Or if the next proposer happens to be on a minority fork, the attester will again forgo rewards. Or if the next proposer's block is late and gets orphaned - subsequent proposers are supposed to pick up the orphaned attestations, but there can be considerable delays if block space is tight. There are countless failure modes outside the attester's control.

It often perplexes stakers when, to all intents and purposes, their validators seem to be working perfectly but they still miss out on rewards or receive penalties. But this is the nature of permissionless, global, peer-to-peer networks. It is a testament to the quality of the protocol and the various client implementations that missed rewards have been surprisingly rare on the beacon chain so far.

### Proposer rewards for attestations

If the attestations in a block are worth a total of $R$ in rewards to the attesters, then the proposer that includes the attestations in a block receives a reward of

$$R_{A_P} = \frac{W_p}{W_\Sigma - W_p} R$$

Thus, over an epoch, the maximum total issuance due to proposer rewards in respect of attestations is

$$I_{A_P} = \frac{W_p}{W_\Sigma - W_p} I_A$$

with $I_A$ being the maximum issuance to attesters per epoch, as above.

Thus, a proposer is strongly incentivised to include high value attestations, which basically means including them quickly, and including well-packed, as correct as possible aggregates.

### Sync committee rewards

Once every 256 epochs (27.3 hours), 512 validators are selected to participate in the sync committee. For any given validator this will happen rarely; with 300,000 validators, the expected interval between being chosen for duty is around 22 months. However, during the 27 hour period of participation the rewards are relatively very large.

Sync committee participants receive a reward every slot that they correctly perform their duties. With 512 members in the committee, and 32 slots per epoch, the reward per validator per slot for correct participation is

$$R_Y = \frac{W_y}{32 \times 512 \times W_\Sigma} Tb$$

The $T$ here is the total increments of the whole active validator set, so this is a large number. The per-epoch per-validator reward is 32 times this.

The maximum issuance per epoch to sync committee members is then

$$I_Y = \frac{W_y}{W_\Sigma} Tb$$

### Proposer rewards for sync committees

As with attestations, the block proposer that includes the sync committee's output receives a reward proportional to the reward of the whole committee:

$$R_{Y_P} = 512 \frac{W_p}{W_\Sigma - W_p} R_Y$$

So the maximum issuance per epoch to sync committee proposers is

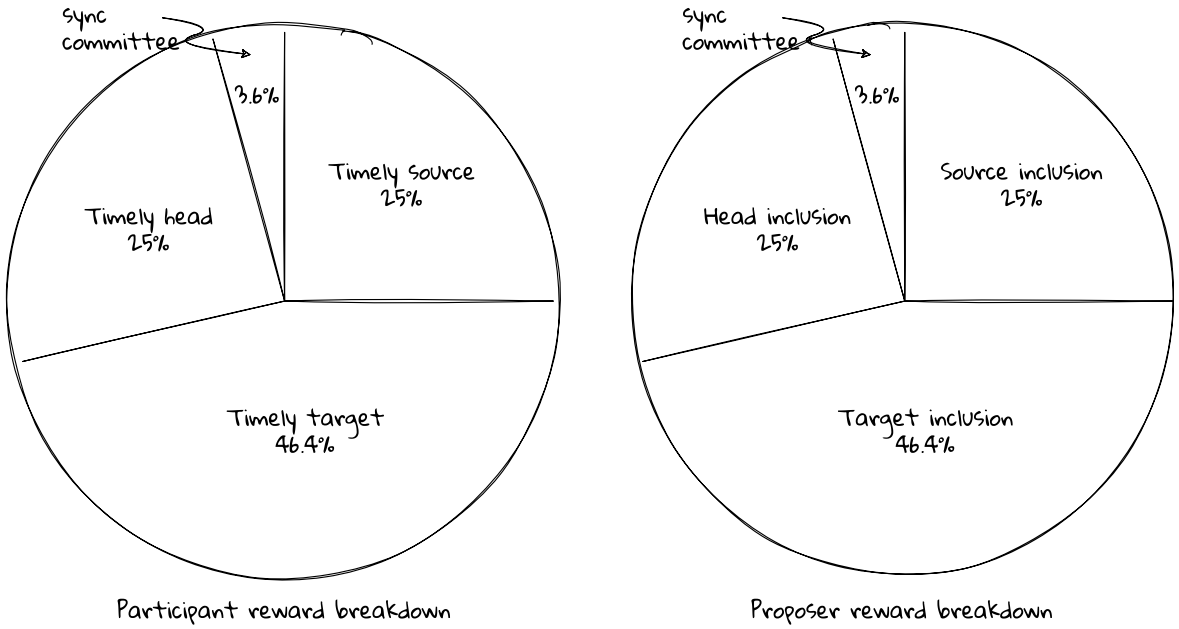
$$I_{Y_P} = \frac{W_p}{W_\Sigma - W_p} I_Y$$

**Remarks on proposer rewards**

You'll note that, for both attestations and sync committees, the proposer reward for including them in a block is a fixed fraction of the validator reward. If $R$ is the validator reward for a duty, then the proposer reward is $\frac{W_p}{W_\Sigma - W_p} R$. In Vitalik's words, "The proposer reward for a duty is the attester reward for that duty, multiplied by the *proposer reward as a fraction of everything but the proposer reward*" (emphasis his).

This factor works out to be $\frac{8}{56} = \frac{1}{7}$ which means that $\frac{7}{8}$ of rewards go to validators performing duties and $\frac{1}{8}$ to the proposers including the evidence in blocks.

In the following charts, I have separated out the validator rewards from the proposer rewards, and we can see that they have exactly the same division among the duties. The chart on the right should probably be one seventh of the size of the one on the left for true accuracy.




*On the left, the breakdown of expected rewards for validators for performing duties. On the right, the breakdown of rewards for proposers for including evidence of those duties.*

This equivalence ensures that the interests of attesters and proposers are aligned.

**Total issuance**

To check that the calculations above are consistent with our claim that the maximum issuance by the beacon chain per epoch is $Tb$ Gwei, let us sum up the issuance due to the four rewards: attester rewards, proposer rewards in respect of attestation inclusion, sync committee rewards, and proposer rewards in respect of sync committee inclusion. The total maximum issuance per epoch is

$$\begin{aligned} I &= I_A + I_{A_P} + I_Y + I_{Y_P} \\ &= \left(1 + \frac{W_p}{W_\Sigma - W_p}\right)(I_A + I_Y) \\ &= \left(1 + \frac{W_p}{W_\Sigma - W_p}\right)\left(\frac{W_s + W_t + W_h + W_y}{W_\Sigma}\right) Tb \\ &= \left(\frac{W_\Sigma}{W_\Sigma - W_p}\right)\left(\frac{W_\Sigma - W_p}{W_\Sigma}\right) Tb \\ &= Tb \end{aligned}$$

as expected.

**Rewards in numbers**

The following calculations are based on 300 thousand active validators, all performing perfectly and all with 32 ETH of effective balance.

- Base reward per increment
  - $b = 653$ Gwei
- Value of a single attestation
  - $R_A = \frac{14+26+14}{64} 32b = 17{,}631$ Gwei
- Value of a single sync committee contribution
  - $R_Y = \frac{2}{32\times 512\times 64} 300{,}000 \times 32b = 11{,}957$ Gwei
- Value of a block proposal due to attestations
  - $R_{A_P} = \frac{300{,}000}{32} \frac{8}{64-8} R_A = 23{,}612{,}946$ Gwei
  - Note: this can actually be higher if the chain is not performing perfectly, as after a skip slot the proposer can include high value attestations from the missed slot.
- Value of a block proposal due to sync committee contributions
  - $R_{Y_P} = 512 \frac{8}{64-8} R_Y = 874{,}569$ Gwei

Putting it all together, the total available reward per epoch across all validators is $300{,}000 R_A + 32(512 R_Y + R_{A_P} + R_{Y_P}) = 6{,}268{,}800{,}000$ Gwei (to 5 significant figures)

Finally, as a check-sum, $Tb = 300{,}000 \times 32b = 6{,}268{,}800{,}000$ Gwei $= 6.268$ ETH.

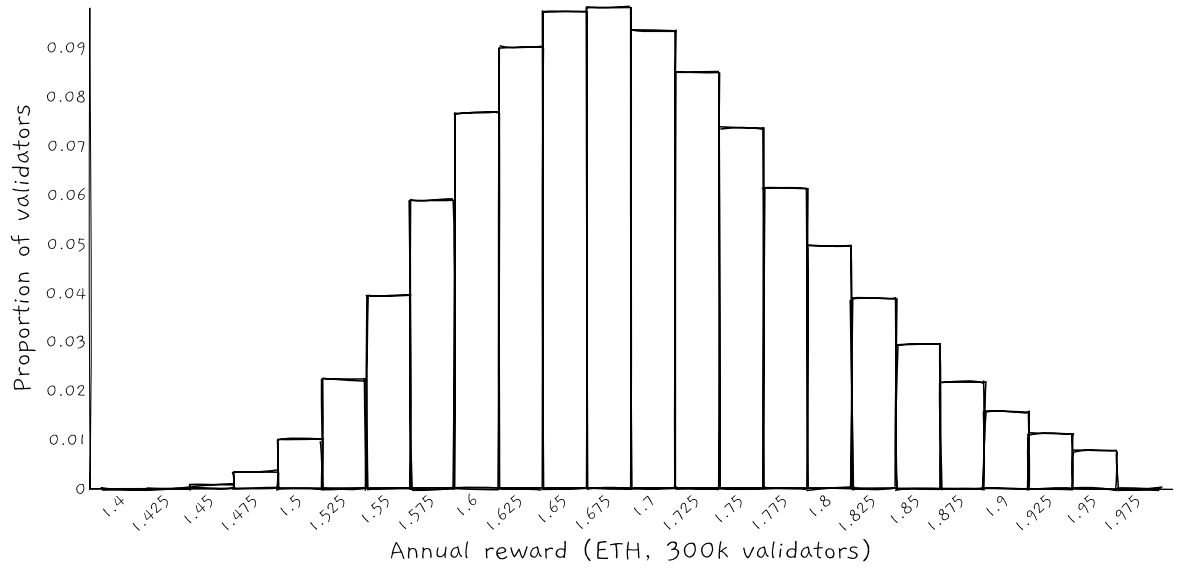
**Individual validator rewards vary**

Actual individual validator returns, even on an optimally running beacon chain, will vary above and below the expected amounts, since block proposals and sync committee duties are assigned randomly. This leads to variance in the rewards, with some validators earning more and some earning less. Nonetheless, an average validator over a long period can expect to earn a return in line with $nb$ per epoch.

The following chart shows the expected distribution of rewards for 300,000 validators, all participating perfectly, each with 32 ETH of effective balance. The mean reward is 1.7177 ETH/year (the 5.37% number from earlier), and the median 1.7188 ETH/year, but there is a large standard deviation of 0.1025 due to the randomness of being selected to propose blocks or participate in sync committees. In fact, ten percent of validators will earn less than 1.596 ETH in rewards over the year, and 10% more than 1.866 ETH, due solely to randomness in assigning duties.

A few remarks on this.

First, the Altair upgrade did not change the expected reward per validator, but it did change the variance considerably. This is due to an increase in the block reward of a factor of four and the introduction of sync committees, with a corresponding reduction in attestation rewards. Since block proposals and




*Distribution of rewards for 300,000 validators with 32 ETH staked.*

sync committee participation are randomly assigned, while attestation rewards are steady, Altair greatly increased the variance in actual rewards. For an analysis of the change, see Pintail's article.

Second, there are further sources of variation that the above analysis doesn't account for. For example, if my validator proposes a block right after a skipped slot, in which there was no block, then my block proposal could be worth up to 71.4% more than a normal block proposal. This is because I get to include attestations from the skipped slot as well as from my own slot, and benefit from the extra source and target votes (but not the extra head votes, which will be too late, or the extra sync committee inclusion).

Third (and most significantly), post-Merge, validators will additionally receive the transaction tips from execution blocks, and potentially MEV-related income as well. These will substantially increase the percentage earnings, and variance in earnings, for stakers, but will not affect overall issuance on the beacon chain since they come from recycled Ether rather than new issuance.

### Rewards scale with participation

One surprising aspect of attestation rewards not so far mentioned is that they are scaled in proportion to participation. That is, for each duty (source, target, head vote) the attester's reward is scaled by the proportion of the total stake that made the same vote.

For example, if I made a correct head vote, and validators with 75% of the total effective balance increments made the same head vote, then I would receive $0.75 \times \frac{W_h}{W_\Sigma} nb$ reward for that vote.

A hand-wavy reason for this is that this scaling makes it to my advantage to help other validators get their attestations included. Several aspects of the protocol are not explicitly incentivised yet are somewhat expensive, such as forwarding gossip messages and attestation aggregation duty. This scaling provides me with an implicit reward for helping out other validators by providing these services: if they perform better, then I perform better.

For a more quantitative analysis, see on discouragement attacks below.

One interesting side-effect of this is that, if participation drops by 10% (due to 10% of validators being offline, say), then total issuance of rewards due to attestations will fall by 19%, in addition to a further reduction from penalties.

We can calculate the participation rate at which net issuance due to attestations turns negative. With a participation rate $p$, the reward for a fully correct attestations is $0.844nbp$, and the penalty for a missed attestation is $0.625Tb$. This gives us a net issuance of $p^2(0.844Tb) - (1-p)(0.625Tb)$. The positive root of this is $p = 56.7\%$. But since this is below the 2/3 participation rate for finalisation, the inactivity

leak will kick-in before we reach this level and completely change the reward and penalty profile, so the calculation is of theoretical interest only.

Note that the proposer reward is not scaled like this – proposers are already well incentivised to include all relevant attestations – and neither are sync committee rewards. Penalties do not scale with participation, either.

#### Discouragement attacks

Quoting from Vitalik's Discouragement Attacks paper,

> A discouragement attack consists of an attacker acting maliciously inside a consensus mechanism in order to reduce other validators' revenue, even at some cost to themselves, in order to encourage the victims to drop out of the mechanism.

Attackers might do this to gain more rewards with fewer participants in the system. Or they might do it as preparation for an attack on the chain: by reducing the number of validators they decrease their own cost of attack.

The paper goes into some quantitative analysis of different kinds of discouragement attacks. I would _en_courage you to read it and think through these things. As per the conclusion:

> In general, this is still an active area of research, and more research on counter-strategies is desired.

Some parts of the beacon chain design that have already been influenced by a desire to avoid discouragement attacks are:

- the inverse square root scaling of validator rewards;
- the scaling of rewards with participation;
- the zeroing of attestation rewards during an inactivity leak; and
- rate limiting of validator exits, which means that an attacker needs to sustain an attack for longer, at greater cost to achieve the same ends.

#### See also

The detailed rewards calculations are defined in the spec in these functions:

- Validator rewards for attestations are calculated in `get_flag_index_deltas()` as part of epoch processing.
- Proposer rewards for attestations are calculated in `process_attestation()` as part of block processing.
- Both validator and proposer rewards for sync committee participation are calculated in `process_sync_aggregate()` as part of block processing.

The discussion of the variance of rewards is based on Pintail's analysis of Altair. The code I used to generate the stats and the chart are based on the code in that article.

Discouragement attacks are analysed in a paper by Vitalik.

## Penalties

- Validators that do not fulfil their assigned duties are penalised by losing small amounts of stake.
- Receiving a penalty is not the same as being slashed!
- Break-even uptime for a validator is around 43%.

### Introduction

Incentivisation of validators on the beacon chain is a combination of carrot and stick. Validators are rewarded for contributing to the chain's security, and penalised for failing to contribute. As we shall see, penalties are quite mild. Nonetheless they provide good motivation for stakers to ensure that their validator deployments are running well.

It's common to hear of the penalties for being offline being referred to as "getting slashed". This is incorrect. Being slashed is a severe punishment for very specific misbehaviours, and results in the validator being ejected from the protocol in addition to some or all of its stake being removed.

Penalties are subtracted from validators' balances on the beacon chain and effectively burned, so they reduce the net issuance of the beacon chain.

### Attestation penalties

Attestations are penalised for being missing, late, or incorrect. We'll lump these together as "missed" for conciseness.

Attesters are penalised for missed Casper FFG votes, that is, missed source or target votes. But there is no penalty for a missed head vote. If a source vote is incorrect, then the target vote is missed; if the source or target vote is incorrect then the head vote is missed.

Let's update our rewards matrix to give the full picture of penalties and rewards for attestations. Recall that this shows the weights; we need to multiply by $\frac{nb}{W_\Sigma}$ to get the actual reward.

| Timeliness | 1 slot | <= 5 slots | <= 32 slots | > 32 Slots (missing) |
|---|---|---|---|---|
| Wrong source | $-W_s - W_t$ | $-W_s - W_t$ | $-W_s - W_t$ | $-W_s - W_t$ |
| Correct source only | $W_s - W_t$ | $W_s - W_t$ | $-W_s - W_t$ | $-W_s - W_t$ |
| Correct source and target only | $W_s + W_t$ | $W_s + W_t$ | $-W_s + W_t$ | $-W_s - W_t$ |
| Correct source, target and head | $W_s + W_t + W_h$ | $W_s + W_t$ | $-W_s + W_t$ | $-W_s - W_t$ |

For more intuition, we can put in the numbers, $W_s = 14$, $W_t = 26$, $W_h = 14$, and normalise with $W_\Sigma = 64$:

| Timeliness | 1 slot | <= 5 slots | <= 32 slots | > 32 Slots (missing) |
|---|---|---|---|---|
| Wrong source | $-0.625$ | $-0.625$ | $-0.625$ | $-0.625$ |
| Correct source only | $-0.188$ | $-0.188$ | $-0.625$ | $-0.625$ |
| Correct source and target only | $+0.625$ | $+0.625$ | $+0.188$ | $-0.625$ |
| Correct source, target and head | $+0.844$ | $+0.625$ | $+0.188$ | $-0.625$ |

### Break-even uptime

Stakers sometimes worry that downtime will be very expensive. To examine this, we can estimate the break-even uptime. We'll ignore sync committee participation since that is so rare, so only attestations are relevant for the calculation.

We'll assume that, when online, the validator's performance is perfect, and that the rest of the validators are performing well (both of which are pretty good approximations to the beacon chain's actual

performance over its first year).

If $p$ is the proportion of time the validator is online, then its net income is, $0.844p - 0.625(1 - p) = 1.469p - 0.625$. This is positive for $p > 42.5\%$. So, if your validator is online more than 42.5% of the time, you will be earning a positive return.

A useful rule of thumb is that it takes about a day of uptime to recover from a day of downtime.

#### Sync committee penalties

The small group of validators currently on sync committee duty receive a reward in each slot that they sign off on the correct head block (correct from the proposer's point of view).

Validators that don't participate (sign the wrong head block or don't show up at all) receive a penalty exactly equal to the reward they would have earned for being correct. And the block proposer receives nothing for the missing contribution.

Historical note: Since sync committee participation is rare for any given validator, and since rewards are significant, there were concerns with earlier designs that the resulting variance in rewards for validators would be quite unfair. Small stakers might prefer to join staking pools rather than solo stake in order to smooth out the variance, similarly to how proof of work mining pools have sprung up.

One suggested approach to reducing the variance was not to reward sync committee participation at all, but rather to raise overall reward levels for everyone and to penalise the sync committee validators if they did not participate. Ultimately the approach adopted was to reduce the length of sync committees (meaning lower rewards, but more often), reduce the proportion of total reward for participation, and introduce a penalty for non-participation – kind of half-way to the other proposal.

The main reasons[14] for not adopting the former proposal, although it is elegant, seem to be around the psychology of being explicitly penalised but never explicitly rewarded. The penalty for not participating in a sync committee would be substantially bigger than the attestation reward over an epoch. In addition, participation is not entirely in the validator's own hands: it depends on the next block proposer being on the right fork. There were also concerns about changing the clean relationship between proposer rewards and the value of the duties they include in blocks.

#### Remarks on penalties

There are no explicit penalties related to block proposers.

In particular, there is no explicit penalty for failing to include deposits from the Eth1 chain, nor any direct incentive for including them. However, if a block proposer does not include deposits that the rest of the network knows about, then its block is invalid. This provides a powerful incentive to include outstanding deposits.

Also note that penalties are not scaled with participation as rewards are.

#### See also

The detailed penalty calculations are defined in the spec in these functions:

- Penalties for missed attestations are calculated in `get_flag_index_deltas()` as part of epoch processing.
- Penalties for missed sync committee participation are calculated in `process_sync_aggregate()` as part of block processing.

### Inactivity leak

[14] The quite interesting discussion remains on the Ethereum R&D Discord.

- When the beacon chain is not finalising it enters a special "inactivity leak" mode.
- Attesters receive no rewards. Non-participating validators receive increasingly large penalties based on their track records.
- This is designed to restore finality in the event of the permanent failure of large numbers of validators.

### Introduction

If the beacon chain hasn't finalised a checkpoint for longer than `MIN_EPOCHS_TO_INACTIVITY_PENALTY` (4) epochs, then it enters "inactivity leak" mode.

The inactivity leak is a kind of emergency state in which rewards and penalties are modified as follows.

- Attesters receive no attestation rewards while attestation penalties are unchanged.
- Any validators deemed inactive have their inactivity scores raised, leading to an additional inactivity penalty that potentially grows quadratically with time. This is the inactivity leak, sometimes known as the quadratic leak.
- Proposer and sync committee rewards are unchanged.

The idea for the inactivity leak was proposed in the original Casper FFG paper. The problem it addresses is that of how to recover finality (liveness, in some sense) in the event that over one-third of validators goes offline. Finality requires a majority vote from validators representing 2/3 of the total stake.

The mechanism works as follows. When loss of finality is detected the inactivity leak gradually reduces the stakes of validators who are not making attestations until, eventually, the participating validators control 2/3 of the remaining stake. They can then begin to finalise checkpoints once again.

This inactivity penalty mechanism is designed to protect the chain long-term in the face of catastrophic events (sometimes referred to as the ability to survive World War III). The result might be that the beacon chain could permanently split into two independent chains either side of a network partition, and this is assumed to be a reasonable outcome for any problem that can't be fixed in a few weeks. In this sense, the beacon chain formally prioritises availability over consistency. (You can't have both.)

In any case, it provides a powerful incentive for stakers to fix any issues they have and to get back online.

The reason why no validators receive attestation rewards during an inactivity leak is once again due to the possibility of discouragement attacks. An attacker might deliberately drive the beacon chain into an inactivity leak, perhaps by a combination of censorship and denial of service attack on other validators. This would cause the non-participants to suffer the leak, while the attacker continues to attest normally. We need to increase the cost to the attacker in this scenario, which we do by not rewarding attestations at all during an inactivity leak.

As with penalties, the amounts subtracted from validators' beacon chain accounts due to the inactivity leak are effectively burned, reducing the overall net issuance of the beacon chain.

### Mathematics

Let's study the effect of the leak on a single validator's balance, assuming that during the period of the inactivity leak (non-finalisation) the validator is completely offline.

At each epoch, the offline validator will be penalised an amount proportional to $tB/\alpha$, where $t$ is the number of epochs since the chain last finalised, $B$ is the validator's effective balance, and $\alpha$ is the `INACTIVITY_PENALTY_QUOTIENT`.

The effective balance $B$ will remain constant for a while, by design, during which time the total amount of the penalty after $t$ epochs would be $t(t+1)B/2\alpha$: the famous "quadratic leak". If $B$ were continuously

variable, the penalty would satisfy $\frac{dB}{dt} = -\frac{tB}{\alpha}$, which can be solved to give the exponential $B(t) = B_0 e^{-t^2/2\alpha}$. The actual behaviour is somewhere between these two since the effective balance decreases in a step-wise fashion.

In the continuous case, the `INACTIVITY_PENALTY_QUOTIENT`, $\alpha$, is the square of the time it takes to reduce the balance of a non-participating validator to $1/\sqrt{e}$, or around 60.7% of its initial value. With the value of `INACTIVITY_PENALTY_QUOTIENT` at `3 * 2**24`, this equates to around seven thousand epochs, or 31.5 days.

For Phase 0 of the beacon chain, the value of `INACTIVITY_PENALTY_QUOTIENT` was increased by a factor of four from $2^{24}$ to $2^{26}$, so that validators would be penalised less severely if there were non-finalisation due to implementation problems in the early days. As it happens, there were no instances of non-finalisation during the whole eleven months of Phase 0 of the beacon chain.

The value was decreased by one quarter in the Altair upgrade from $2^{26}$ to $3 \times 2^{24}$ as a step towards eventually setting it to its final value. Decreasing the inactivity penalty quotient speeds up recovery of finalisation in the event of an inactivity leak.

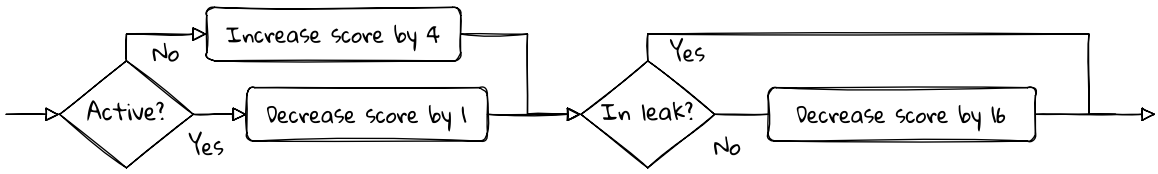
### Inactivity scores

During Phase 0, the inactivity penalty was an increasing global amount applied to all validators that did not participate in an epoch, regardless of their individual track records of participation. So a validator that was able to participate for a significant fraction of the time could still be quite severely penalised due to the growth of the inactivity penalty. Vitalik gives a simplified example: "if fully [off]line validators get leaked and lose 40% of their balance, someone who has been trying hard to stay online and succeeds at 90% of their duties would still lose 4% of their balance. Arguably this is unfair." We found during the Medalla testnet incident that keeping a validator online when all around you is chaos is not easy. We don't want to punish stakers who are honestly doing their best.

To improve this, the Altair upgrade introduced individual validator inactivity scores that are stored in the state. The scores are updated each epoch as follows.

- Every epoch, irrespective of the inactivity leak,
  - decrease the score by one when the validator makes a correct timely target vote, and
  - increase the score by `INACTIVITY_SCORE_BIAS` (four) otherwise.
- When *not* in an inactivity leak,
  - decrease every validator's score by `INACTIVITY_SCORE_RECOVERY_RATE` (sixteen).

Graphically, the flow-chart looks like this.




*How each validator's inactivity score is updated. The happy flow is right through the middle.*

Note that there is a floor of zero on the score.

When not in an inactivity leak validators' inactivity scores are reduced by `INACTIVITY_SCORE_RECOVERY_RATE + 1` per epoch when they make a timely target vote, and by `INACTIVITY_SCORE_RECOVERY_RATE - INACTIVITY_SCORE_BIAS` when they don't. So, even for non-performing validators, scores decrease outside a leak.

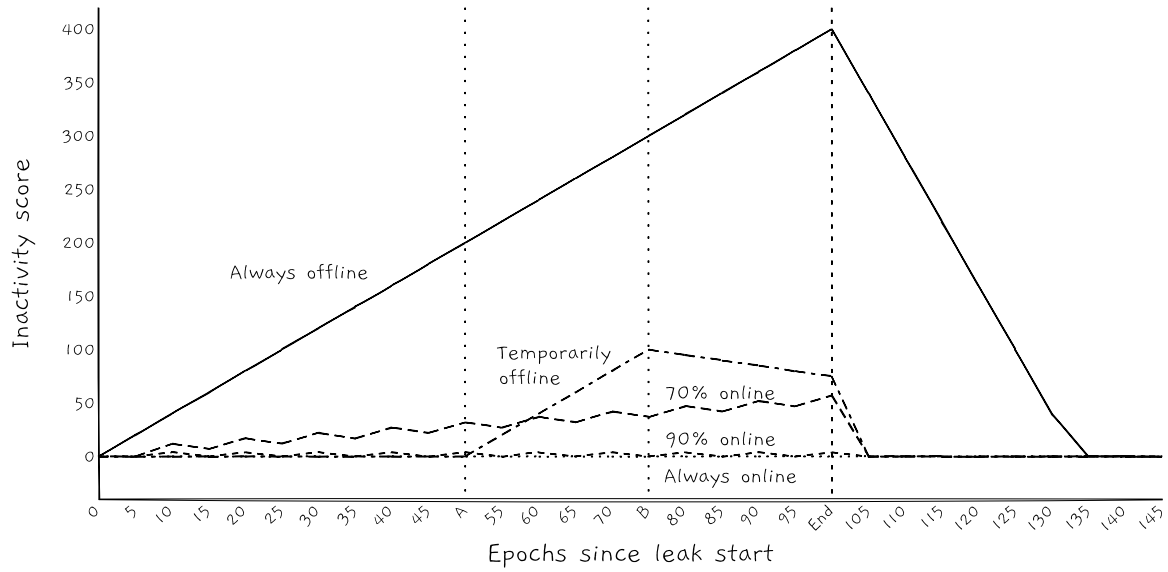
When in a leak, if $p$ is the participation rate between 0 and 1, and $\lambda$ is `INACTIVITY_SCORE_BIAS`, then the expected score after $N$ epochs is $\max(0, N((1-p)\lambda - p))$. For $\lambda = 4$ this is $\max(0, N(4-5p))$. So a

validator that is participating 80% of the time or more can maintain a score that is bounded near zero. With less than 80% average participation, its score will increase unboundedly.

This is nice because, if many validators are able to participate intermittently, it indicates that whatever event has befallen the chain is potentially recoverable, unlike a permanent network partition, or a super-majority network fork, for example. The inactivity leak is intended to bring finality to irrecoverable situations, so prolonging the time to finality if it's recoverable is likely a good thing.

The following graph illustrates some scenarios. We have an inactivity leak that starts at zero, and ends after 100 epochs, after which finality is recovered and we are no longer in the leak. There are five validators. Working up from the lowest line, they are:

1. Always online: correctly registering a timely target vote in every epoch. The inactivity score remains at zero.
2. 90% online: the inactivity score remains bounded near zero. From the analysis above, it is expected that anything better than 80% online will bound the score near zero.
3. 70% online: the inactivity score grows slowly over time.
4. Generally online, but offline between epochs 50 and 75: the inactivity score is zero during the initial online period; grows linearly and fairly rapidly while offline during the leak; declines slowly when back online during the leak; and declines rapidly once the leak is over.
5. Always offline: the inactivity score increases rapidly during the leak, and declines even more rapidly once the leak is over.




*The inactivity scores of five different validator personas in an inactivity leak that starts at zero and ends at epoch 100 (labelled "End" and shown with a dashed line). The dotted lines labelled "A" and "B" mark the start and end of the offline period for the fourth validator.*

### Inactivity penalties

The inactivity penalty is applied to all validators at every epoch based on their individual inactivity scores, irrespective of whether a leak is in progress or not. When there is no leak, the scores return to zero (rapidly for active validators, less rapidly for inactive ones), so most of the time this is a no-op.

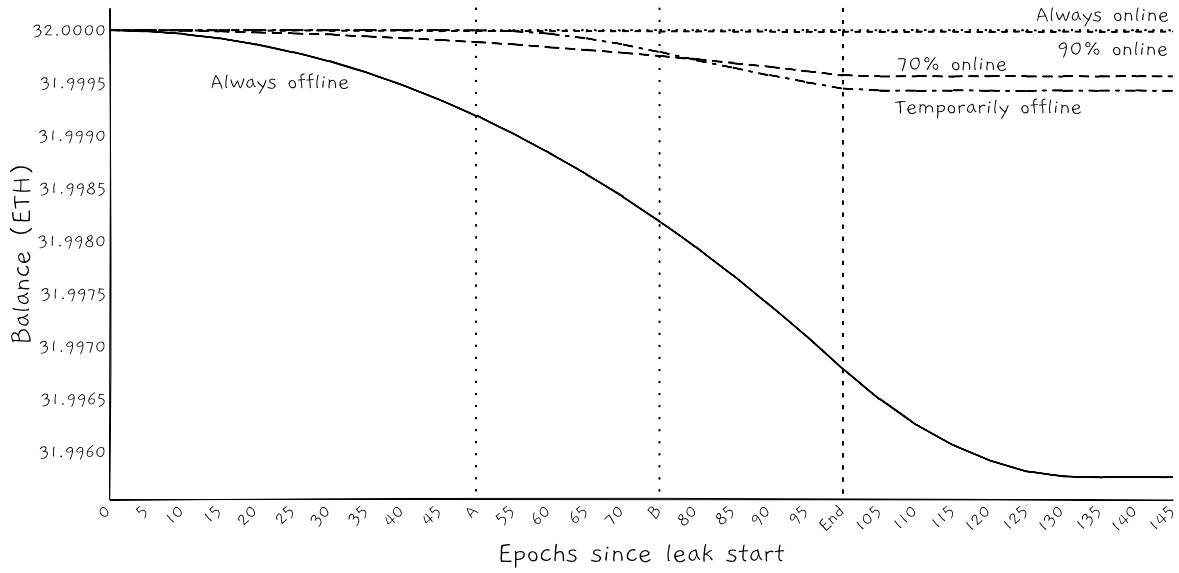
The penalty for validator $i$ is calculated as

$$s_i B_i / (\texttt{INACTIVITY\_SCORE\_BIAS} \times \texttt{INACTIVITY\_PENALTY\_QUOTIENT\_ALTAIR}) = \frac{s_i B_i}{4 \times 50{,}331{,}648}$$

where $s_i$ is the validator's inactivity score, and $B_i$ is the validator's effective balance.

This penalty is applied at each epoch, so (for constant $B_i$) the total penalty is proportional to the area under the curve of the inactivity score, above. With the same five validator persona's we can quantify the penalties in the following graph.

1. Always online: no penalty due to the leak.
2. 90% online: negligible penalty due to the leak.
3. 70% online: the total penalty grows quadratically but slowly during the leak, and rapidly stops after the leak ends.
4. Generally online, but offline between epochs 50 and 75: a growing penalty during the leak, that rapidly stops when the leak ends.
5. Always offline: we can clearly see the quadratic nature of the penalty in the initial parabolic shape of the curve. After the end of the leak it takes around 35 epochs for the penalties to return to zero.




*The balance retained by each of the five validator personas after the inactivity leak penalty has been applied. The scenario is identical to the chart above.*

We can see that the new scoring system means that some validators will continue to be penalised due to the leak even after finalisation starts again. This is intentional. When the leak causes the beacon chain to finalise, at that point we have just two-thirds of the stake online. If we immediately stop the leak (as we used to), then the amount of stake online would remain close to two-thirds and the chain would be vulnerable to flipping in and out of finality as small numbers of validators come and go. We saw this behaviour on some of the testnets prior to launch. Continuing the leak after finalisation serves to increase the balances of participating validators to greater than two-thirds, providing a buffer that should mitigate such behaviour.

### See also

From the spec:

- Inactivity scores are updated during epoch processing in `process_inactivity_updates()`.

- Inactivity penalties are calculated in `def_get_inactivity_penalty_deltas()`.

For the original description of the mechanics of the inactivity leak, see the Casper paper, section 4.2.

## Slashing

- Validators are slashed for breaking very specific protocol rules that could be part of an attack on the chain.
- Slashed validators are exited from the beacon chain and receive three types of penalty.
- Correlated penalties mean that punishment is light for isolated incidents, but severe when many validators are slashed in a short time period.
- Block proposers receive rewards for reporting evidence of slashable offences.

### Introduction

Slashing occurs when validators break very specific protocol rules when submitting attestations or block proposals which could constitute attacks on the chain. Getting slashed means losing a potentially significant amount of stake and being ejected from the protocol. It is more "punishment" than "penalty". The good news is that stakers can take simple precautions to protect against ever being slashed.

Validators' stakes can be slashed for two distinct behaviours:

1. as attesters, for breaking the Casper commandments, the two rules on voting for source and target checkpoints; and
2. as proposers, for proposing two different blocks at the same height (equivocation).

The slashing of misbehaving attesters is what underpins Ethereum 2.0's economic finality guarantee by enforcing the Casper FFG protocol rules.

Proposer slashing, however, is not part of the Casper FFG protocol, and is not directly related to economic finality. It punishes a proposer that spams the block tree with multiple blocks that could partition the network, for example in a balancing attack.

As with penalties, the amounts removed from validators' beacon chain accounts due to slashing are effectively burned, reducing the overall net issuance of the beacon chain.

### The cost of being slashed

When it comes to the punishment for being slashed it does not matter which rule was broken. All slashings are dealt with in the same way.

### The initial penalty

Slashing is triggered by the evidence of the offence being included in a beacon chain block. Once the evidence is confirmed by the network, the offending validator (or validators) is slashed.

The offender immediately has one sixty-fourth (`MIN_SLASHING_PENALTY_QUOTIENT_ALTAIR`) of its effective balance deducted from its actual balance. This is a maximum of 0.5 ETH due to the cap on effective balance.

This initial penalty was introduced to make it somewhat costly for validators to self-slash for any reason.

Along with the initial penalty, the validator is queued for exit, and has its withdrawability epoch set to around 36 days (`EPOCHS_PER_SLASHINGS_VECTOR`, which is 8192 epochs) in the future.

During Phase 0, this initial penalty was $\frac{1}{128}$ of the offender's effective balance. It is expected to be raised to its full value of $\frac{1}{32}$ of the effective balance, a maximum of 1 ETH, as part of The Merge.

### The correlation penalty

At the half way point of its withdrawability period (18 days after being slashed) the slashed validator is due to receive a second penalty.

This second penalty is based on the total amount of stake slashed during the 18 days before and after our validator was slashed. The idea is to scale the punishment so that a one-off event posing little threat to the chain is only lightly punished, while a mass slashing event that might be the result of an attempt to finalise conflicting blocks is punished to the maximum extent possible.

To be able to calculate this, the beacon chain maintains a record of the effective balances of all validators that were slashed during the most recent 8192 epochs (about 36 days).

The correlated penalty is calculated as follows.

1. Compute the sum of the effective balances (as they were when the validators were slashed) of all validators that were slashed in the previous 36 days. That is, for the 18 days preceding and the 18 days following our validator's slashing.
2. Multiply this sum by `PROPORTIONAL_SLASHING_MULTIPLIER_ALTAIR`, but cap the result at `total_balance`, the total active balance of all validators.
3. Multiply the slashed validator's effective balance by the result of #2 and then divide by the `total_balance`. This results in an amount between zero and the full effective balance of the slashed validator. That amount is subtracted from its actual balance as the penalty. Note that the effective balance could exceed the actual balance in odd corner cases, but `decrease_balance()` ensures the balance does not go negative.

The slashing multiplier in Altair is set to 2. With $S$ being the sum of increments in the list of slashed validators over the last 36 days, $B$ my effective balance, and $T$ the total increments, the calculation looks as follows.

$$\text{Correlation penalty} = \min(B, \frac{2SB}{T})$$

Interestingly, due to the way the integer arithmetic is constructed in the implementation the result of this calculation will be zero if $2SB < T$. Effectively, the penalty is rounded down to the nearest whole amount of Ether. As a consequence, when there are few slashings there is no extra correlated slashing penalty at all, which is probably a good thing.

The intention is to raise the proportional slashing multiplier from 2 to 3 around The Merge, three being its "correct" cryptoeconomic value. To successfully finalise conflicting blocks at least one third of validators need to break attestation slashing rules. With the multiplier set to 3, that third would lose their entire stakes through this mechanism, which is optimal for security.

### Other penalties

Validators that exit normally (by sending a voluntary exit message) are expected to participate only until their exit epoch, which is normally only a couple of epochs later.

A validator that is slashed continues to receive attestation penalties until its withdrawable epoch, which is set to 8192 epochs (36 days) after the slashing, and they are unable to receive any attestation rewards during this time. They are also subject for this entire period to any inactivity leak that might be in operation. Whatever the slashed validator does, it is penalised exactly as if it is failing to participate.[15]

[15] It's not clear to me why we have such a large hang-over from being slashed during which validators continue to receive penalties. It seems like kicking a man when he's down, especially since slashed validators are locked in for twice as long as needed to calculate the correlation penalty. Vitalik says that this measure "is included to prevent self-slashing from being a way to escape inactivity leaks." But validators don't need to self-slash to avoid this; they could just make a normal voluntary exit.

So, in addition to the initial slashing penalty and the correlation penalty, there is a further penalty of up to $8192\frac{14+26}{64}32b$ = 106,987,520 Gwei = 0.107 ETH, based on 300k validators, assuming that the chain is not in an inactivity leak. And (much) more if it is.

Slashed validators are eligible to be selected to propose blocks until they reach their exit epoch, but those blocks will be considered invalid, so there is no proposer reward available to them. This is in preference to immediately recomputing the duties assignments which would break the lookahead guarantees they have. (The proposer selection algorithm could easily be modified to skip slashed validators, but that is not how it is implemented currently.)

In an interesting edge case, however, slashed validators are eligible to be selected for sync committee duty until they reach their exit epoch and to receive the rewards for sync committee participation. Though the odds of this happening, absent a mass slashing event, are pretty tiny.

### The value of reporting a slashing

In order for the beacon chain to verify slashings and take action against the offender, the evidence needs to be included in a beacon block. To incentivise validators to make the effort there is a specific reward for the proposer of a block that includes slashings.

### The proposer reward

At the point of the initial slashing report being included in a block, the proposer of the block receives a reward of `validator.effective_balance` / `WHISTLEBLOWER_REWARD_QUOTIENT`, which is $B/512$ if $B$ is the effective balance of the validator being slashed.

A report of a proposer slashing violation can slash only one validator, but a report of an attestation slashing violation can simultaneously slash up to an entire committee, which might be hundreds of validators. This could be extremely lucrative for the proposer including the reports. A single block can contain up to 16 proposer slashing reports and up to 2 attester slashing reports.

Note that no new issuance is required to pay for this reward. The proposer reward is much less than the initial slashing applied to the validator, so the net issuance due to a slashing event is always negative.

### The whistleblower reward

In the code implementing the reward for reporting slashing evidence there is provision for a "whistleblower reward", with the whistleblower receiving $\frac{7}{8}$ of the above reward and the proposer $\frac{1}{8}$.

The idea is to incentivise nodes that search for and discover evidence of slashable behaviour, which can be an intensive process.

However, this functionality is not currently used on the beacon chain, and the proposer receives both the whistleblower reward and the proposer reward, as above. The challenge is that it is too easy for a proposer just to steal a slashing report, so there's no point incentivising them separately. It's not an ideal situation, but so far there seem to be sufficient altruistic slashing detectors running on the beacon chain for slashings to be reported swiftly. There only needs to be one in practice.

This functionality may become useful in future upgrades.

### See also

From the spec:

- The initial slashing penalty and proposer reward are applied in `slash_validator()` during block processing.
- The correlation slashing penalty is applied in `process_slashings()` during epoch processing.

In the Serenity Design Rationale Vitalik gives some further background on why Ethereum 2.0 includes proposer slashing. It is specifically intended to discourage stakers from simultaneously running primary and backup nodes.

## Diversity

- Beacon chain incentives strongly encourage diversity among client deployments, hosting infrastructure, and staking pools.
- Lack of diversity puts at risk both the chain in general and all those running the majority client.
- The greater the share of validators hosted by a single client implementation the greater the risk.
- The beacon chain is at its most robust and fault-tolerant when no single client type manages more than one-third (33%) of validators.

### Diversity makes us all stronger

Just as diversity in biological ecosystems makes them more resilient, and monocultures make them very fragile – yes, I've been watching David Attenborough –, so it is with Ethereum staking.

It is not unintentional that both the inactivity leak and the slashing correlation penalty provide a strong encouragement to diversify the network as much as possible.

For example, the inactivity leak is much more likely to occur on a network in which a single client implementation runs over 33% of validators, or a single staking operator controls over 33% of validators, or over 33% of validators are deployed to the same hosting infrastructure. All these scenarios constitute single points of failure that could prevent the beacon chain from finalising and lead to a leak that penalises those running the majority (offline) client most harshly.

### Scenarios

Let's consider some scenarios. For the sake of this exercise you are running the beacon chain client X. In each scenario you and others using client X host validators managing a certain fraction of the total stake. We will consider what happens if client X has a bug that takes it down. It might be a consensus bug or another kind of bug that takes the client off the network: we saw examples of both of these on the pre-launch testnets.

### 1. Client X has less than one-third of the stake

When a client managing less than one-third of the total stake goes down, the consequences are minimal. The beacon chain can continue to finalise as normal. Users of client X will suffer only the normal offline penalties until the bug is fixed, though rewards will be lower across the board for the other validators. But this is not catastrophic and there is time to recover without a panic, either by fixing the bug or swapping to a different client.

*The beacon chain is at its most robust and fault-tolerant when no single client type manages more than one-third (33%) of validators.*

### 2. Client X has more than one-third of the stake

If client X goes down while managing more than one-third of the total stake, then the beacon chain will be unable to finalise and will enter the inactivity leak.

In this situation no validators will receive rewards for attesting. Users of non-X clients will not lose stake, but users of client X will suffer much bigger losses than usual, due to the quadratically increasing inactivity leak. There is strong time pressure to get the issue with client X resolved either by fixing the bug or swapping to a different client.

### 3. Client X has around half of the stake

The situation becomes potentially much worse when X hosts around half of the validators. If X were to have a consensus bug, but otherwise keep running, the beacon chain would split into two similarly sized chains. Each chain would see half its validators missing and start leaking out the stakes of those validators. Within three to four weeks each chain would have leaked out enough of the stake of the missing validators that the present validators would control two-thirds of the remaining stake, meaning that the chains could each finalise separately. It would be extremely difficult – effectively impossible – to reunite these chains ever again since they would contain conflicting finalised checkpoints. The beacon chain would be permanently partitioned.

Hopefully, 3-4 weeks is sufficient time for client X to fix its bug or for users of X to migrate to other clients. Meanwhile users of X are suffering large inactivity penalties on the correct chain as per scenario 2.

### 4. Client X approaches or exceeds two-thirds of the stake

A scenario in which a single client approaches[16] hosting two-thirds (66%) of the validators is potentially catastrophic. A consensus bug in that client would very quickly – possibly within 13 minutes – finalise a broken version of the chain with no chance to intervene.

That would leave the Ethereum community with a horrible dilemma.

One possible response would be to modify the other clients (and the specification) to reproduce the bug and allow them to join X's chain. The feasibility of this depends on the nature of the consensus bug. For a trivial bug it might be possible, but it would be very unfair to the non-X clients since they would suffer penalties despite having acted perfectly correctly. In any case, many types of consensus bug would make this infeasible: one way or another X's chain is broken and now incompatible with the entirety of the rest of the ecosystem.

The correct – but nuclear – option is to fix the bug in client X. Unfortunately, however, there would be no way for the stakers on the incorrect X chain to rejoin the correct chain. Any that tried to do so would be slashed, having previously finalised a checkpoint on the incorrect chain. The only reasonable strategy for (former) users of client X would be to stop validating and voluntarily exit their stakes. Exiting could take a long time due to the queuing mechanism, resulting in large penalties from the inactivity leak. Many of the affected stakers are likely to try to start validating again and would surely be slashed.

There are no good outcomes here, which is why it is critical that we never have a client with a two-thirds or more supermajority.[17]

### Slashing

As for slashing, once again running a majority client could be act of self-harm. In the unlikely event that a client implementation has a bug that leads to its validators becoming slashed en-masse, the correlated slashing penalties would be much more severe than if the same thing happened to those running a minority client.

### Another view

Danny Ryan has presented a slightly different angle on client diversity that's insightful:

> If a single client:
>
> - Does not exceed 66.6%, a fault/bug in a single client cannot be finalized.
> - Does not exceed 50%, a fault/bug in a single client's fork choice cannot dominate the head of the chain.
> - Does not exceed 33.3%, a fault/bug in a single client cannot disrupt finality.

[16] If the share is less than 67% the incorrect chain won't finalise immediately, but very soon the inactivity leak will raise the proportion above 67% on that chain and it will then finalise.

[17] As of 2022-01-12, the Prysm client appeared to have 68.1% of the validators.

### Epilogue

Let me emphasise that *these scenarios are far from theoretical*. It is of existential importance to the Ethereum network that stakers pay attention to the distribution of client software and avoid adding to the share of the majority client.

It is instructive to revisit the major incident that occurred on the Medalla testnet, in which an issue in the majority client caused a high degree of chaos and led to large numbers of slashings. Had that client managed a smaller proportion of the network, the consequences for everybody would have been much less severe.

### See also

- Run the majority client at your own peril! by Dankrad Feist.
- What Happens If Beacon Chain Consensus Fails? by Adrian Sutton.

# The Building Blocks

## Introduction

In this chapter we will explore some of the fundamental innovations that make the Ethereum 2 protocol practical, the building blocks from which the higher level protocol is constructed.

None of the building blocks is absolutely brand new – they all depend to a degree on existing technologies – but in each case some aspect of the application to Eth2 is novel. The Ethereum Foundation R&D team deserves huge credit for the research and insights behind these advances.

Be alert, as you read, to the trade-offs that underpin these design choices. The gateway to deep understanding is always in the trade-offs.

Some of the trade-offs are quite interesting. For example, neither the shuffling algorithm nor the state root calculation algorithm are the most efficient that we could have chosen, at least in terms of pure speed. In both cases we preferred algorithms that enable a light client ecosystem over algorithms that might be more performant for full nodes.

The building blocks I've grouped together in this chapter are those that are part of the protocol specification itself. Client implementations often employ other optimisations that are not part of the specification. We'll consider some of those later in the Implementation chapter.

These are the topics that I've picked out for special attention.

- BLS Signatures precipitated the total redesign of Ethereum's proof of stake protocol, and underpin the scale and ambition of Ethereum 2.
- Randomness is a vital aspect of security, but difficult to generate in a deterministic system. The beacon chain accomplishes it with BLS signatures.
- Shuffling is uses randomness to populate committees. But, for the sake of light clients, we use an "oblivious" shuffle rather than the standard Fisher–Yates.
- Committees distribute the workload of the beacon chain.
- Aggregator Selection secretly selects small subsets of committees to do the work of aggregating attestations.
- SSZ: Simple Serialize is a novel serialisation technique that appears everywhere in the protocol. It embodies elegance and efficiency.
- Hash Tree Roots and Merkleization are applications of SSZ. Among other things, they make light clients practical.
- Generalised indices and Merkle proofs (TODO).
- Sync Committees (TODO).

## BLS Signatures

| | | | |
|---|---|---|---|
| First cut | ✓ | Revision | TODO |

- Proof of stake protocols use digital signatures to identify their participants and hold them accountable.
- BLS signatures can be aggregated together, making them efficient to verify at large scale.
- Signature aggregation allows the beacon chain to scale to hundreds of thousands of validators.

- Ethereum transaction signatures on the execution (Eth1) layer remain as-is.

### Digital signatures

Digital signatures are heavily used in blockchain technology. A digital signature is applied to a message to ensure two things: (1) that the message has not been tampered with in any way; and (2) that the sender of the message is who it claims to be. Digital signatures are not new, and really developed during the 1980s as a result of the invention of asymmetric cryptography. However, more recent developments involving elliptic curve, pairing-based cryptography have heavily influenced the design of Ethereum 2.

Every time you send an Ethereum transaction you are using a digital signature; all Ethereum users are familiar with the signing work flow. But that's at the transaction level. At the consensus protocol level digital signatures are not used at all in Ethereum 1 – Under proof of work, a block just needs to have a correct `mixHash` proving that it was correctly mined, nobody cares who actually mined the block, so no signature is needed.

In Ethereum 2, however, validators have identities and are accountable for their actions. In order to enforce the Casper FFG rules, and in order to be able to count votes for the LMD GHOST fork choice, we need to be able to uniquely identify the validators making individual attestations and blocks.

### Digital signature usage

The primary function of a digital signature is to irrevocably link the sender of a message with the contents of the message. This can be used, for example, to prove with certainty that a validator has published conflicting votes and thus is subject to being slashed.

The ability to tie messages to validators is also useful outside the protocol. For example, in the gossip layer, signatures are validated by nodes before they are forwarded as an anti-spam mechanism.

Alongside their usual function of identifying message senders, digital signatures have a couple of fairly novel uses within the Ethereum 2 protocol. They are used when contributing randomness to the RANDAO, and they are used to select subsets of committees for aggregation duty. We will discuss those usages in their respective sections and focus on the signing of protocol messages in this section.

### Background

One of the characteristics of proof of stake protocols is the sheer number of protocol messages that need to be handled. With 300,000 active validators, the current beacon chain design calls for over 780 attestations per second to be gossiped globally. That's a sustained average, there are much higher bursts in practice. Not only do these messages need to travel over the network, but each individual digital signature needs to be verified by every node, which is a CPU-intensive operation. Not to mention having to store all those signed messages in the block history. These challenging requirements have typically limited the validator numbers in proof of stake or proof of authority networks. Pure PBFT-based consensus protocols tend to have validator sets that number in the dozens rather than the thousands.

The prevailing work-in-progress design in early 2018 for Ethereum's (partial) move to proof of stake, EIP-1011, estimated that the protocol could handle a maximum of around 900 validators due to this message overhead, and accordingly set a hefty stake size of 1500 ETH per validator.

The turning point came in May 2018 with the publication by Justin Drake of an article on the Ethresear.ch forum titled Pragmatic signature aggregation with BLS. The article proposed using a new signature scheme that is able to *aggregate* many digital signatures into one while preserving the individual accountability of each validator that signed. Aggregation provides a way to dramatically reduce the number of individual messages that must be gossiped around the network and the cost of verifying the integrity of those messages, and thus enables scaling to hundreds of thousands of participants.[18]

[18] To give credit where it is due, the Dfinity blockchain researchers had published a white paper a few months earlier proposing the use of BLS signatures in a threshold scheme. However, their use of threshold signatures makes the chain vulnerable to liveness failures, and also requires a tricky distributed key generation protocol. Ethereum's aggregation-

This signature aggregation capability was the main breakthrough that prompted us to abandon the EIP-1011 on-chain PoS management mechanism entirely and move to the "beacon chain" model that we have today[19].

### BLS Digital Signatures

Digital signatures in the blockchain world are usually based on elliptic curve groups. For signing users' transactions, Ethereum uses ECDSA signatures with the secp256k1 elliptic curve. However, the beacon chain protocol uses BLS signatures with the BLS12-381 elliptic curve[20]. Although similar in usage, ECDSA and BLS signatures are mathematically quite different, with the latter relying on a special property of certain elliptic curves called "pairing". Although ECDSA signatures are much faster than BLS signatures, it is the pairing property of BLS signatures that allows us to aggregate signatures, thus making the whole consensus protocol practical.

Several other blockchain protocols have adopted or will adopt BLS signatures over the BLS12-381 curve, and throughout our implementation in Eth2 we have been mindful to follow whatever standards exist, and to participate in the defining of those standards where possible. This both helps interoperability and supports the development of common libraries and tooling.

The high-level workflow for creating and verifying a BLS signature is relatively straightforward. In the sections that follow I'll describe how it all works with some words, some pictures, and some maths. Feel free to skip the maths if you wish, it's not compulsory and there's no test at the end. Though it is rather elegant.

### Components

There are four component pieces of data within the BLS digital signature process.

1. The *secret key*. Every entity acting within the protocol (that is, a validator in the context of Eth2) has a secret key, sometimes called a private key. The secret key is used to sign messages and must be kept secret, as its name suggests.
2. The *public key*. The public key is uniquely derived from the secret key, but the secret key cannot be reverse engineered from it (without impossibly huge amounts of work). A validator's public key represents its identity within the protocol, and is known to everybody.
3. The *message*. We'll look later at the kinds of messages used in the Eth2 protocol and how they are constructed. For now, the message is just a string of bytes.
4. The *signature*, which is the output of the signing process. The signature is created by combining the message with the secret key. Given a message, a signature for that message, and a public key, we can verify that the validator with that public key signed exactly that message. In other words, no-one else could have signed that message, and the message has not been changed since signing.

More mathematically, things look like this. We use two subgroups of the BLS12-381 elliptic curve: $G_1$ defined over a base field $F_q$, and $G_2$ defined over the field extension $F_{q^2}$. The order of both the subgroups is $r$, a 77 digit prime number. The (arbitrarily chosen) generator of $G_1$ is $g_1$, and of $G_2$, $g_2$.

1. The secret key, $sk$, is a number between 1 and $r$ (technically the range includes 1, but not $r$. However, very small values of $sk$ would be hopelessly insecure).
2. The public key, $pk$, is $[sk]g_1$ where the square brackets represent scalar multiplication of the elliptic curve group point. The public key is thus a member of the $G_1$ group.
3. The message, $m$ is a sequence of bytes. During the signing process this will be mapped to some point $H(m)$ that is a member of the $G_2$ group.

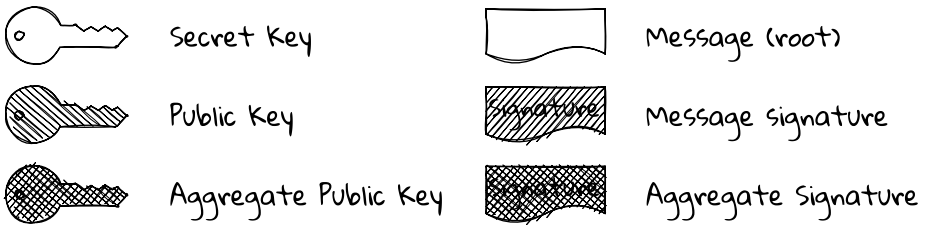
---

based approach has neither of these issues. Nonetheless, the name "beacon chain" that we still use today derives from Dfinity's "randomness beacon" described in that paper.

[19] The last significant update to EIP-1011 was made on the 16th of May, 2018. Justin Drake's post on signature aggregation was made just two weeks later.

[20] There is a curious naming collision here. The BLS trio of "BLS signatures" are Boneh, Lynn, and Shacham, whereas those of the "BLS12-381" elliptic curve are Barreto, Lynn, and Scott. Ben Lynn is the only common name between the two.

4. The signature, $\sigma$, is also a member of the $G_2$ group, namely $[sk]H(m)$.




*The key to the keys. This is how we will depict the various components in the diagrams below. Variants of the same object are hatched differently. The secret key is mathematically a scalar; public keys are $G_1$ group members; message roots are mapped to $G_2$ group members; and signatures are $G_2$ group members.*

### Key pairs

A key pair is a secret key along with its public key. Together these irrefutably link each validator with its actions.

Every validator on the beacon chain has at least one key pair, the "signing key" that is used in daily operations (making attestations, producing blocks, etc.). Depending on which version of withdrawal credentials the validator is using, it may also have a second BLS key pair, the "withdrawal key", that is kept offline.

The secret key is supposed to be uniformly randomly generated in the range $[1, r)$. EIP-2333 defines a standard way to do this based on the `KeyGen` method of the draft IETF BLS signatures standard. It's not compulsory to use this method – no-one will ever know if you don't – but you'd be ill advised not to. In practice, many stakers generate their keys with the `eth2.0-deposit-cli` tool created by the Ethereum Foundation. Operationally, key pairs are often stored in password-protected EIP-2335 keystore files.

The secret key, $sk$ is a 32 byte unsigned integer. The public key, $pk$, is a point on the $G_1$ curve, which is represented in-protocol in its compressed serialised form as a string of 48 bytes.


*A validator randomly generates its secret key. Its public key is then derived from that.*

### Signing

In the beacon chain protocol the only messages that get signed are hash tree roots of objects: their so-called signing roots, which are 32 byte strings. The `compute_signing_root()` function always combines the hash tree root of an object with a "domain" as described below.

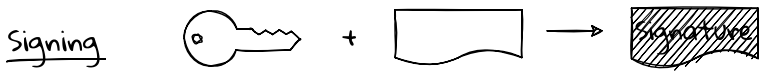
Once we have the signing root it needs to be mapped onto an elliptic curve point in the $G_2$ group. If the message's signing root is $m$, then the point is $H(m)$ where $H()$ is a function that maps bytes to $G_2$. This mapping is hard to do well and an entire draft standard exists to define the process. Thankfully, we can ignore the details completely and leave them to our cryptographic libraries[21].

Now that we have $H(m)$, the signing process itself is simple, being just a scalar multiplication of the $G_2$ point by the secret key:

[21] Unless you have to implement the thing, as I ended up doing in Java.

$$\sigma = [sk]H(m)$$

Evidently the signature $\sigma$ is also a member of the $G_2$ group, and it serialises to a 96 byte string in compressed form.




*A validator applies its secret key to a message to generate a unique digital signature.*

### Verifying

To verify a signature we need to know the public key of the validator that signed it. Every validator's public key is stored in the beacon state and can be simply looked up via the validator's index which, by design, is always available by some means whenever it's required.

Signature verification can be treated as a black-box: we send the message, the public key, and the signature to the verifier; if after some cryptographic magic the signature matches the public key and the message then we declare it valid. Otherwise, either the signature is corrupt, the incorrect secret key was used, or the message is not what was signed.

More formally, signatures are verified using elliptic curve pairings.

With respect to the curve BLS12-381, a pairing simply takes a point $P \in G_1$, and a point $Q \in G_2$ and outputs a point from a group $G_T \subset F_{q^{12}}$. That is, for a pairing $e$, $e : G_1 \times G_2 \to G_T$.

Pairings are usually denoted $e(P, Q)$ and have very special properties. In particular, with $P$ and $S$ in $G_1$ and $Q$ and $R$ in $G_2$,

- $e(P, Q + R) = e(P, Q) \cdot e(P, R)$, and
- $e(P + S, R) = e(P, R) \cdot e(S, R)$.

(Conventionally $G_1$ and $G_2$ are written as additive groups, and $G_T$ as multiplicative, so the $\cdot$ operator is point multiplication in $G_T$.)

From this, we can deduce that all of the following identities hold:

$$e([a]P, [b]Q) = e(P, [b]Q)^a = e(P, Q)^{ab} = e(P, [a]Q)^b = e([b]P, [a]Q)$$

Armed with our pairing, verifying a signature is straightforward. The signature is valid if and only if

$$e(g_1, \sigma) = e(pk, H(m))$$

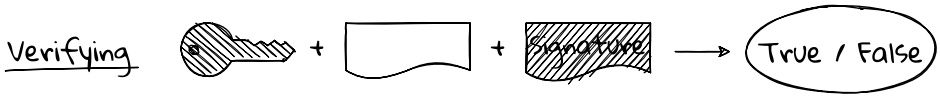
That is, given the message $m$, the public key $pk$, the signature $\sigma$, and the fixed public value $g_1$ (the generator of the $G_1$ group), we can verify that the message was signed by the secret key $sk$.

This identity comes directly from the properties of pairings described above.

$$e(pk, H(m)) = e([sk]g_1, H(m)) = e(g_1, H(m))^{(sk)} = e(g_1, [sk]H(m)) = e(g_1, \sigma)$$

Note that elliptic curves supporting such a pairing function are very rare. Such curves can be constructed, as BLS12-381 was, but general elliptic curves such as the more commonly used secp256k1 curve do not support pairings and cannot be used for BLS signatures.

The verification will return `True` if and only if the signature corresponds both to the public key (that is, the signature and the public key were both generated from the same secret key) and to the message (that is, the message is identical to the one that was signed originally). Otherwise it will return `False`.




*To verify that a particular validator signed a particular message we use the validator's public key, the original message, and the signature. The verification operation outputs true if the signature is correct and false otherwise.*

### Aggregation

So far we've looked at the basic set up of BLS signatures. In functional terms, what we've seen is very similar to any other digital signature scheme. Where the magic happens is in *aggregation*.

Aggregation means that multiple signatures over the same message – potentially thousands of signatures – can be checked with a single verification operation. Furthermore, the aggregate signature has the same size as a regular signature, 96 bytes. This is a massive gain in scalability, and it is this gain that fundamentally makes the Ethereum 2 consensus protocol viable.

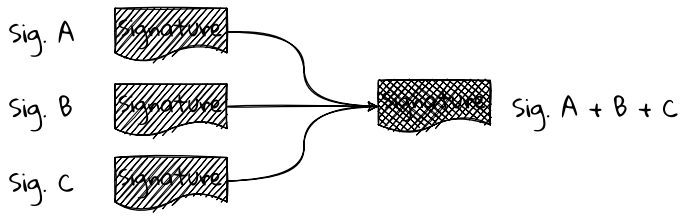
How does this work? Recall that public keys and signatures are elliptic curve points. Because of the bilinearity property of the pairing function, $e()$, it turns out that we can form linear combinations of public keys and signatures over the same message, and verification still works as expected.

This statement is a little opaque; let's go step by step.

### Aggregating signatures

In all of the following we will only consider aggregation of signatures over the same message[22].

The process is conceptually very simple: we simply "add up" the signatures. The exact operations are not like the normal addition of numbers that we are familiar with, but the operation is completely analogous. Addition of points on the elliptic curve is the group operation for the $G_2$ group, and each signature is a point in this group, thus the result is also a point in the group. An aggregated signature is mathematically indistinguishable from a non-aggregated signature, and has the same 96 byte size.




*Aggregation of signatures is simply group addition in the $G_2$ group.*

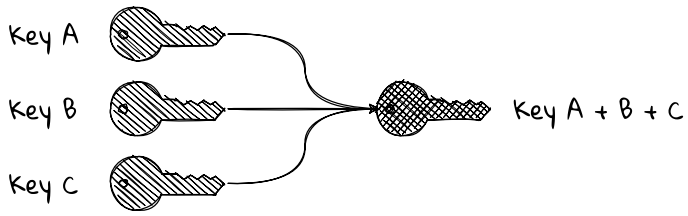
### Aggregating public keys

To verify an aggregate signature, we need an aggregate public key. As long as we know exactly which validators signed the original message, this is equally easy to construct. Once again we simply "add up" the public keys of the signers. This time the addition is the group operation of the $G_1$ elliptic curve group, and the result will also be a member of the $G_1$ group, so it is mathematically indistinguishable from a non-aggregated public key, and has the same 48 byte size.

### Verifying aggregate signatures

Since aggregate signatures are indistinguishable from normal signatures, and aggregate public keys

[22] A note on terminology. The original paper describing this scheme uses the term "multi-signature" when combining signatures over the same message, and "aggregate signature" when combining signatures over distinct messages. In Eth2 we only do the former, and just call it aggregation.




*Aggregation of public keys is simply group addition in the $G_1$ group.*

are indistinguishable from normal public keys, we can simply feed them into our normal verification algorithm.


*Verification of an aggregate signature is identical to verification of a normal signature as long as we use the corresponding aggregate public key.*

This miracle is due to the bilinearity of the pairing operation. With an aggregate signature $\sigma_{agg}$ and a corresponding aggregate public key $pk_{agg}$, and common message $m$, we have the following identity, which is exactly the same as the verification identity for a single signature and public key.

$$\begin{aligned} e(pk_{agg}, H(m)) &= e(pk_1 + pk_2 + \cdots + pk_n, H(m)) \\ &= e([sk_1 + sk_2 + \cdots + sk_n]g_1, H(m)) \\ &= e(g_1, H(m))^{(sk_1+sk_2+\cdots+sk_n)} \\ &= e(g_1, [sk_1 + sk_2 + \cdots + sk_n]H(m)) \\ &= e(g_1, \sigma_1 + \sigma_2 + \cdots + \sigma_n) \\ &= e(g_1, \sigma_{agg}) \end{aligned}$$

**Benefits of aggregation**

Verification of a BLS signature is expensive (resource intensive) compared with verification of an ECDSA signature, more than an order of magnitude slower due to the pairing operation, so what benefits do we gain?

The benefits accrue when we are able to aggregate significant numbers of signatures. This is exactly what we have with beacon chain attestation committees. Ideally, all the validators in the committee sign-off on the same attestation data, so all their signatures can be aggregated. In practice, there might be differences of opinion about the chain state between committee members resulting in two or three different attestations, but even so there will be many fewer aggregates compared with the total number of committee members.

**Speed benefits**

To a first approximation, then, we can verify all of the attestations of a whole committee – potentially hundreds – with a single signature verification operation.

This is a first approximation because we also need to account for aggregating the public keys and the signatures. But these aggregation operations involve only point additions in their respective elliptic curve groups, which are very cheap compared with the verification.

In summary:

- We can verify a single signature with two pairings.

- We can naively verify $N$ signatures with $2N$ pairings.
- Or we can verify $N$ signatures via aggregation with just two pairings, $N-1$ additions in $G_1$, and $N-1$ additions in $G_2$. Each elliptic curve point additions is much, much cheaper than a pairing.

### Space benefits

There is also a huge space saving when we aggregate signatures.

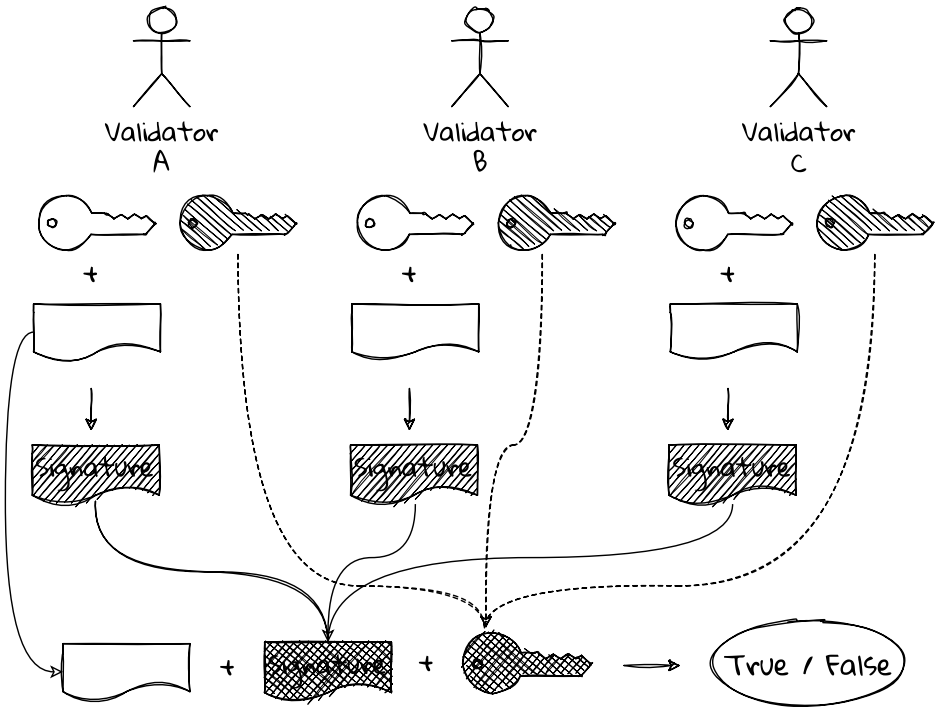
An aggregate signature has 96 bytes as all BLS signatures do. So, to a first approximation, an aggregate of $N$ signatures occupies $\frac{1}{N}$ the space of the unaggregated signatures.

Again, this is only a first approximation. The subtlety here is that, in order to construct the corresponding aggregate public key, we somehow need to keep track of which validators signed the message. We cannot assume that the whole committee participated, and we need to be careful not to include any validator more than once.

If we know in advance who the members of the committee are and how they are ordered then this tracking can be done at the marginal cost of one bit per validator: true if the validator contributed to the aggregate, false if it did not.

### The full picture

This diagram illustrates the full flow from signing, through aggregating, to verifying. There are three validators in this case, although there could be many more, and each is signing the same message contents. Each validator has its own unique secret key and public key pair. The workflow is entirely non-interactive, and any of the actions before the verification can happen independently. Even the aggregation can be done incrementally.




*The end-to-end aggregate signature workflow. Verifying the single aggregate signature is much faster than verifying the original signatures separately.*

### Aggregation examples

Two useful examples of how aggregate signatures are used in practice are in aggregate attestations and in sync committee aggregates.

### Aggregate attestations

Aggregate attestations are a very compact way store and prove which validators made a particular attestation.

Within each beacon chain committee at each slot, individual validators attest to their view of the chain, as described in the validator spec.

An `Attestation` object looks like this:

```
class Attestation(Container):
    aggregation_bits: Bitlist[MAX_VALIDATORS_PER_COMMITTEE]
    data: AttestationData
    signature: BLSSignature
```

When making its attestation, the validator sets a single bit in the `aggregation_bits` field to indicate which member of the committee it is. That is sufficient, in conjunction with the slot number and the committee index, to uniquely identify the attesting validator in the global validator set.

The `signature` field is the validator's signature over the `AttestationData` in the `data` field.

This attestation will later be aggregated with other attestations from the committee that contain identical `data`. An attestation is added to an aggregate by copying over its bit from the `aggregation_bits` field and adding (in the sense of elliptic curve addition) its signature to the `signature` field. Aggregate attestations can be aggregated together in the same way, but only if their `aggregation_bits` lists are disjoint: we must not include a validator more than once. (In principle we could include individual validators multiple times, but then we'd need more than a single bit to track how many times, and the redundancy is not useful.)

This aggregate attestation will be gossiped around the network and eventually included in a block. At each step the aggregate signature will be verified.

To verify the signature, a node needs to reconstruct the list of validators in the committee, which it can do from the information in the `AttestationData`:

```
class AttestationData(Container):
    slot: Slot
    index: CommitteeIndex
    beacon_block_root: Root
...
```

Given the reconstructed list of committee members, the validating node filters the list according to which `aggregation_bits` are set in the attestation. Now it has the indices of all the validators that contributed to this attestation. The node retrieves the public keys of those validators from the beacon state and aggregates those keys together (by elliptic curve addition).

Finally, the aggregate signature, the aggregate public key, and the signing root of the `data` are fed into the standard BLS signature verification function. If all is well this will return `True`, else the aggregate attestation is invalid.

### Sync aggregates

`SyncAggregate`s are produced by a sync committee of 512 members.

```
class SyncAggregate(Container):
    sync_committee_bits: Bitvector[SYNC_COMMITTEE_SIZE]
    sync_committee_signature: BLSSignature
```

The current members of the `SyncCommittee` are stored in the beacon state in the following form:

```
class SyncCommittee(Container):
    pubkeys: Vector[BLSPubkey, SYNC_COMMITTEE_SIZE]
    aggregate_pubkey: BLSPubkey
```

Production and aggregation of sync committee messages differs slightly from attestations, but is sufficiently similar that I'll skip over it here.

The main points of interest are that the `SyncCommittee` object contains the actual public keys of all the members (possibly with duplicates), rather than validator indices. It also contains a pre-computed `aggregate_pubkey` field that is the aggregate of all the public keys in the committee.

The idea of this is to reduce the computation load for light clients, who will be the ones needing to verify the `SyncAggregate` signatures. Sync committees are expected to have high participation, with, say, 90% of the validators contributing. To verify the aggregate signature we need to aggregate the public keys of all the contributors. Starting from an empty set, that would mean 461 elliptic curve point additions (90% of 512). However, if we start from the *full* set, `aggregate_pubkey`, then we can achieve the same thing by *subtracting* the 10% that did not participate. That's 51 elliptic curve subtractions (which have the same cost as additions) and nine times less work.

### Various topics

#### Domain separation and forks

Every signature that's used in the Eth2 protocol has a `domain` value mixed into the message before signing. This is taken care of by the `compute_signing_root()` function which both calculates the SSZ hash tree root of the object to be signed and mixes in the given domain.

```
def compute_signing_root(ssz_object: SSZObject, domain: Domain) -> Root:
    return hash_tree_root(SigningData(
        object_root=hash_tree_root(ssz_object),
        domain=domain,
    ))
```

The domain, in turn, is calculated by the `compute_domain()` function which combines one of ten domain types with a mash-up of the fork version and the genesis validators root.

Each of the extra quantities that's rolled into the message has a specific purpose.

- The domain type ensures that signatures made for one purpose cannot be re-used for a different purpose. Objects of different SSZ types are not guaranteed to have unique hash tree roots, and we would rather like to be able to tell the difference between them. The ten domain types are all the different ways signatures are used in the protocol.
- The genesis validators root uniquely identifies this particular beacon chain, distinguishing it from any other testnet or alternative chain. This ensures that signatures from different chains are always incompatible.
- The fork version identifies deliberate consensus upgrades to the beacon chain. Mixing the fork version into the message ensures that messages from validators that have not upgraded are invalid. They are out of consensus and have no information that is useful to us, so this provides a convenient way to ignore their messages. Alternatively, a validator may wish to operate on both sides of a contentious fork, and the fork version provides a way for them to do so safely.

The sole exception to the mixing-in of the fork version is signatures on deposits. Deposits are always valid, however the beacon chain gets upgraded.

#### Choice of groups

BLS signatures are based on two elliptic curve groups, $G_1$ and $G_2$. Elements of $G_1$ are small (48 bytes when serialised), and their group arithmetic is faster; elements of $G_2$ are large (96 bytes when serialised) and their group arithmetic is slower, perhaps three times slower.

We can choose to use either group for public keys, as long as we use the other group for signatures: the pairing function doesn't care; everything still works if we swap the groups over. The original paper describing BLS aggregate signatures has public keys in $G_2$ and signatures in $G_1$, while for Ethereum 2 we made the opposite choice.

The main reason for this is that we want public key aggregation to be as fast as possible. Signatures are verified much more often than they are aggregated – by far the main load on beacon chain clients currently is signature verification – and verification requires public key aggregation. So we choose to have our public keys in the faster $G_1$ group. This also has the benefit of reducing the size of the beacon

state, since public keys are stored in validator records. If we were to use the $G_2$ group for public keys, the beacon state would be about 35% larger.

The trade-off is that protocol messages and beacon chain blocks are larger due to the larger signature size.

Fundamentally, verification of aggregate signatures is an "on-chain" activity that we wish to be as light as possible, and signature aggregation is "off-chain" so can be more heavyweight.

### Proof of possession

There is a possible attack on the BLS signature scheme that we wish to avoid, the "rogue public key" attack.

Say your public key is $pk_1$, and I have a secret key, $sk_2$. But instead of publishing my true public key, I publish $pk_2' = [sk_2]g_1 - pk_1$ (that is, my real public key plus the inverse of yours). I can sign a message $H(m)$ with my secret key to make $\sigma = [sk_2]H(m)$. I then publish this claiming that it is an aggregate signature that both you and I have signed.

Now, when verifying with my rogue public key and your actual public key, the claim checks out: it looks like you signed the message when you didn't: $e(g1, \sigma) = e(g_1, [sk_2]H(m)) = e([sk_2]g_1, H(m)) = e(pk_1 + pk_2', H(m))$.

One relatively simple defence against this – the one we are using in Ethereum 2 – is to force validators to register a "proof of possession" of the secret key corresponding to their claimed public key. You see, the attacker doesn't have and cannot calculate the $sk_2'$ corresponding to $pk_2'$. The proof of possession can be done simply by getting all validators to sign their public keys on registration, that is, when they deposit their stakes in the deposit contract. If the actual signature validates with the claimed public key then all is well.

### Threshold signatures

In addition to aggregation, the BLS scheme also supports threshold signatures. This is where a secret key is divided between $N$ validators. For a predefined value of $M \leq N$, if $M$ of the validators sign a message then a single joint public key of all the validators can be used to verify the signature.

Threshold signatures are not currently used within the core Ethereum 2 protocol. However they are useful at an infrastructure level. For example, for security and resilience it might be desirable to split a validator's secret key between multiple locations. If an attacker acquires fewer than $M$ shares then the key still remains secure; if up to $N - M$ keystores are unavailable, the validator can still sign correctly. An operational example of this is Attestant's Dirk key manager.

Threshold signatures also find a place in Distributed Validator Technology, which I will write about in a different chapter.

### Batch verification

The bilinearity of the pairing function allows for some pretty funky optimisations. For example, Vitalik has formulated a method for verifying a batch of signatures simultaneously – such as all the signatures contained in a block – that significantly reduces the number of pairing operations required. Since this technique constitutes a client-side optimisation rather than being a fundamental part of the protocol, I shall describe it properly in the Implementation chapter.

### Quantum security

The security (unforgeability) of BLS signatures relies, among other things, on the hardness of something called the elliptic curve discrete logarithm problem (ECDLP)[23]. Basically, given the public key $[sk]g_1$ it is computationally infeasible to work out what the secret key $sk$ is.

The ECDLP is believed to be vulnerable to attack by quantum computers, thus our signature scheme may have a limited shelf-life.

[23] It's puzzling to me that this is called the discrete logarithm problem when we write groups additively, rather than the discrete division problem. But it's far from being the most confusing thing about elliptic curves.

Quantum-resistant alternatives such as zkSTARKs are known, but currently not as practical as the BLS scheme. The expectation is that, at some point, we will migrate to such a scheme as a drop-in replacement for BLS signatures.

In case someone overnight unveils a sufficiently capable quantum computer, EIP-2333 (which is a standard for BLS key generation in Ethereum) describes a way to generate a hierarchy of Lamport signatures. Lamport signatures are believed to be quantum secure, but come with their own limitations. In principle we could make an emergency switch over to these to tide us over while implementing STARKs. But this would be extremely challenging in practice.

### BLS library functions

As a reference, the following are the BLS library functions used in the Ethereum 2 specification. They are named for and defined by the BLS Signature Standard. Function names link to the definitions in the standard. Since we use the proof of possession scheme defined in the standard, our `Sign`, `Verify`, and `AggregateVerify` functions correspond to `CoreSign`, `CoreVerify`, and `CoreAggregateVerify` respectively.

- `def Sign(privkey: int, message: Bytes) -> BLSSignature`
  - Sign a message with the validator's secret (private) key.
- `def Verify(pubkey: BLSPubkey, message: Bytes, signature: BLSSignature) -> bool`
  - Verify a signature given the public key and the message.
- `def Aggregate(signatures: Sequence[BLSSignature]) -> BLSSignature`
  - Aggregate a list of signatures.
- `def FastAggregateVerify(pubkeys: Sequence[BLSPubkey], message: Bytes, signature: BLSSignature) - bool`
  - Verify an aggregate signature given the message and the list of public keys corresponding to the validators that contributed to the aggregate signature.
- `def AggregateVerify(pubkeys: Sequence[BLSPubkey], messages: Sequence[Bytes], signature: BLSSignature) -> bool`
  - This is not used in the current spec but appears in the future Proof of Custody spec. It takes $n$ messages signed by $n$ validators and verifies their aggregate signature. The mathematics is similar to that above, but requires $n + 1$ pairing operations rather than just two. But this is better than the $2n$ pairings that would be required to verify the unaggregated signatures.
- `def KeyValidate(pubkey: BLSPubkey) -> bool`
  - Checks that a public key is valid. That is, it lies on the elliptic curve, it is not the group's identity point (corresponding to the zero secret key), and it is a member of the $G_1$ subgroup of the curve. All these checks are important to avoid certain attacks. The group membership check is quite expensive but only ever needs to be done once per public key stored in the beacon state.

The Eth2 spec also defines two further BLS utility functions, `eth_aggregate_pubkeys()` and `eth_fast_aggregate_verify()` that I describe in the annotated spec.

### See also

The main standards that we strive to follow are the following IETF drafts:

- BLS Signatures
- Hashing to Elliptic Curves
- Pairing-Friendly Curves

Compact Multi-Signatures for Smaller Blockchains (Boneh, Drijvers, Neven) is the original paper that described efficient BLS multi-signatures. And Pragmatic signature aggregation with BLS is Justin Drake's proposal to use these signatures in an Ethereum 2 context.

For a gentle(ish) introduction to pairings, Vitalik's Exploring Elliptic Curve Pairings is very good. If you are looking for a very deep rabbit hole to explore, Pairings for Beginners by Craig Costello is amazing.

I've written a lengthy homage to the BLS12-381 elliptic curve that also covers some BLS signature topics.

Three EIPs are intended to govern the generation and storage of keys in practice:

- EIP-2333 provides a method for deriving a tree-hierarchy of BLS12-381 keys based on an entropy seed.
- EIP-2334 defines a deterministic account hierarchy for specifying the purpose of keys.
- EIP-2335 specifies a standard keystore format for storage and interchange of BLS12-381 keys.

There are several implementations of pairings on the BLS12-381 curve around, which can be used to implement the BLS signature scheme we use:

- The Blst library is the most commonly used by Eth2 client implementers.
- The noble-bls12-381 library is better documented and may be more enjoyable if you want to try playing around with these things.

## Randomness

- Assigning beacon chain duties unpredictably is an important defence against some attacks.
- The beacon chain maintains a RANDAO to accumulate randomness.
- Duties such as proposing blocks, committee assignments, and sync committee participation are assigned based on the RANDAO, with a limited lookahead period.
- Block proposers verifiably contribute randomness to the RANDAO via BLS signatures over the epoch number.
- Validators are able to bias the RANDAO to a small extent but this is not significant problem in practice.

### Introduction

An element of randomness is an important part of a permissionless blockchain protocol, both for security and for fairness.

A protocol that is fully predictable could work well in a benign environment. But we must assume that our protocols will come under attack, and predictability provides attackers with opportunities - just as the bad guys in crime thrillers often take advantage of their victims' predictable routines.

An attacker with advance knowledge of which validators will be active in different roles has a significant foothold for mounting an attack. For example, to selectively mount denial of service attacks against future proposers, or to bribe members of a particular committee, or to register especially advantageous validator numbers for themselves allowing them to take over a future committee, or simply to censor transactions.[24]

To quote from a paper by Brown-Cohen et al,[25]

[24] For a cute illustration of the perils of insufficient unpredictability, see Issue 1446 on the specs repo: Manipulating deposit contract to gain an early majority. Hat-tip to Paul Hauner.

[25] Formal Barriers to Longest-Chain Proof-of-Stake Protocols, Jonah Brown-Cohen, Arvind Narayanan, Christos-Alexandros Psomas, and S. Matthew Weinberg (2018). Quotation is from section 3.1.

> Intuitively, it is good for protocols to be unpredictable in the sense that miners do not learn that they are eligible to mine a block until shortly before it is due to be mined. Many attacks, such as double-spending, or selfish-mining, can become much more profitable if miners know in advance when they become eligible to mine.

Unpredictability, arising from randomness, is an excellent first line of defence against many attacks.

Unpredictability in Proof of Work comes from the process used to mine blocks. A block is valid only if it satisfies a certain condition, and the only way to satisfy that condition is through trial and error. Miners make a random guess, test it, and try again if it's not correct - this is the "work" in Proof of Work. Only if the guess is correct is the block valid and the miner gets to extend the chain. As I write, the difficulty of the Ethereum PoW chain is around 12.5 Peta hashes. That means that mining an Ethereum block requires $1.25 \times 10^{16}$ guesses on average. This is similar to the odds of rolling 21 dice until they all come up six on the same roll. It is fabulously unlikely, yet somewhere on the Ethereum network somebody manages to do it every 13 seconds or so. Since the process is uniform – nobody is better at guessing (rolling dice) than anyone else – it provides fairness. Every Giga hash per second is equivalent to every other Giga hash per second (although there are other sources of unfairness in Proof of Work). And since guessing is random it provides unpredictability, which mitigates the attacks mentioned above.

Randomness[26] in Ethereum's Proof of Stake protocol is used to bring unpredictability to the selection of block proposers, and to the membership of the committees that attest to blocks and sign sync data.

In this section we will look at the way that randomness is introduced into the beacon chain, some of the ways in which it is used, and finally some of the issues with the current scheme.

### The RANDAO

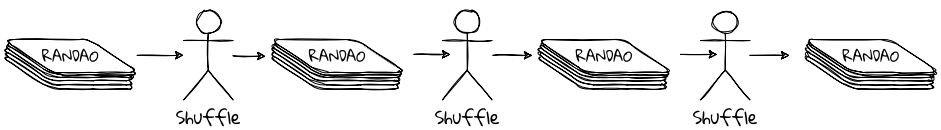
The beacon chain design has always used a RANDAO[27] mechanism to provide its in-protocol randomness. A RANDAO is simply an accumulator that incrementally gathers randomness from contributors. So, with each block, the proposer mixes in a random contribution to the existing RANDAO value.

To unpack that a little, the beacon chain maintains a RANDAO value. Every block included in the chain contains a verifiable random value provided by the validator that proposed it, its `randao_reveal`. As each block is processed the beacon chain's RANDAO value is mixed with the `randao_reveal` from the block. Thus, over time, the RANDAO accumulates randomness from all the block proposers.

If $R_n$ is the RANDAO value after $n$ contributions, and $r_n$ is the $n$th `randao_reveal`, then the following holds. Here we are mixing in the new contribution using the `xor` function, $\oplus$. Alternatives might be to use a sum or a hash, but `xor` is simple and has useful properties.

$$R_n = r_n \oplus R_{n-1}$$

We can think of a RANDAO as being like a deck of cards that's passed round the table, each person shuffling it in turn: the deck gets repeatedly re-randomised. Even if one contributor's randomness is weak, the cumulative result has a high level of entropy.




*We can imagine the RANDAO as a deck of cards that accumulates randomness over time as each participant shuffles the deck in turn.*

---

[26] I'm not going to distinguish the niceties of randomness and pseudo-randomness in this section. We are actually using pseudo-randomness seeded with (presumed) genuine randomness. It must be the case as it is impossible to come to consensus on genuine randomness. However, I will just call it "randomness" throughout.

[27] I'm not certain where the name RANDAO comes from, but it's modelled as a DAO (decentralised autonomous organisation) that deals in randomness. The Ethereum randao project from 2016 may be the origin of the name.

Current and past RANDAO values are stored in the beacon state in the `randao_mixes` field. The current value is updated by `process_randao` with every block that the beacon chain processes. If there is no block in a slot then the RANDAO is not updated. In addition to the RANDAO's current value, `EPOCHS_PER_HISTORICAL_VECTOR` (minus one) past values of the RANDAO at the ends of epochs are also stored in the state. These can be used to recalculate past committee assignments, which allows historical attestations to be slashed even months later.

### Source of randomness

With every block the proposer includes a field `randao_reveal` that is its contribution to be mixed in to the RANDAO.

This contribution needs to satisfy two properties: it should be unpredictable by any other node, yet it should be verifiable by all nodes.

"Verifiable" means that, although random (read pseudo-random), the RANDAO contribution value must not be arbitrary. The proposer must not be able to pick and choose its contribution, otherwise it will just choose a value that advantages itself in some way. There must be a single valid contribution that the proposer can make in any given block, and all the other nodes must be able to verify that contribution.

### The old: hash onions

Early ideas for verifiable randomness had each validator pre-committing to a "hash onion". Before joining the beacon chain a validator would generate a random number. When registering its initial deposit the validator would include the result of repeatedly cryptographically hashing that number a large number (thousands) of times as a commitment. Then when proposing a block the `randao_reveal` would be the pre-image of that commitment: one layer would be "peeled off the onion". Since a cryptographic hash is not invertible, only the proposer could calculate this value, but it's easily verifiable by everyone. Then the reveal gets stored as the new commitment and so on. This scheme is viable, but has complexities and edge cases – for example if a proposer's block gets orphaned everybody (except the beacon chain) can now see the reveal – that make it clunky.

### The new: BLS signatures

A natural alternative to the hash onion became available when we moved to using BLS signatures in the protocol. With the BLS scheme every validator already has a closely guarded random value: the secret key that it signs blocks and attestations with. As far as anyone knows the signatures produced are uniformly random. Therefore, BLS signatures generated by block proposers are an excellent source of the randomness that we need for updating the RANDAO.

Using signatures rather than the hash onion both simplifies the protocol and provides for multi-party (distributed) validators. The aggregation properties of BLS signatures allow signatures from multiple validators to be combined as a threshold signature so that they can effectively act as a single validator. This valuable property would be very difficult with the hash onion approach.

For these reasons we now use a BLS signature as the entropy contribution to the RANDAO, that is, the `randao_reveal`.

### Where does the entropy come from?

Evidently the predominant source of randomness in the Ethereum 2 protocol is the secret keys of the validators. If every validator key is generated uniformly randomly and independently then each contributes 256 bits of entropy to the overall pool. However, keys are sometimes not independently generated[28]. EIP-2333 provides a way to derive multiple validator keys from a single entropy seed, and large stakers are likely to have done this. Thus, the total entropy from $N$ validator keys will be less than $N \times 256$ bits, but we don't know how much less.

Some other sources of entropy for the RANDAO are noted in EIP-4399.

[28] I am indebted to Vasiliy Shapovalov for reminding me of this.

- Missed or orphaned block proposals directly affect the RANDAO's output. Network conditions, node faults, or maintenance downtime can all lead to missed block proposals that have a degree of randomness.
- The total number of active validators in an epoch affects the selection of proposers which in turn affects participation in the RANDAO. Thus, deposits and exits (both voluntary and forced) contribute entropy.
- A validator's effective balance affects its likelihood of being selected to propose a block. Thus, changes in effective balances (perhaps due to one or more validators being offline for a period of time) add entropy.

### Updating the RANDAO

When a validator proposes a block, it includes a field `randao_reveal` which has `BLSSignature` type. This is the proposer's signature over the epoch number, using it's normal signing secret key.

The `randao_reveal` is computed by the proposer as follows, the `privkey` input being the validator's random secret key.

```
def get_epoch_signature(state: BeaconState, block: BeaconBlock, privkey: int) -> BLSSignature:
    domain = get_domain(state, DOMAIN_RANDAO, compute_epoch_at_slot(block.slot))
    signing_root = compute_signing_root(compute_epoch_at_slot(block.slot), domain)
    return bls.Sign(privkey, signing_root)
```

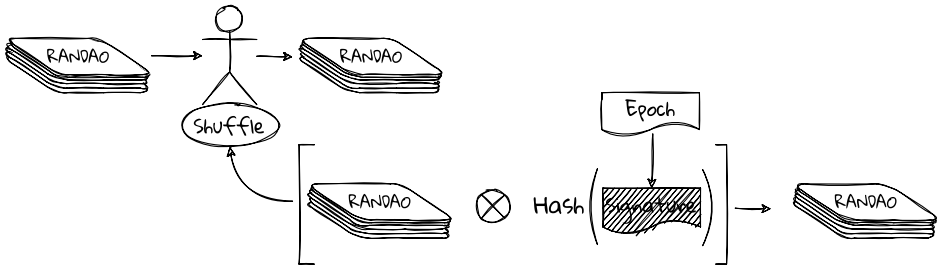
When a block is processed, the `randao_reveal` is mixed into the RANDAO like this:

```
def process_randao(state: BeaconState, body: BeaconBlockBody) -> None:
    epoch = get_current_epoch(state)
    # Verify RANDAO reveal
    proposer = state.validators[get_beacon_proposer_index(state)]
    signing_root = compute_signing_root(epoch, get_domain(state, DOMAIN_RANDAO))
    assert bls.Verify(proposer.pubkey, signing_root, body.randao_reveal)
    # Mix in RANDAO reveal
    mix = xor(get_randao_mix(state, epoch), hash(body.randao_reveal))
    state.randao_mixes[epoch % EPOCHS_PER_HISTORICAL_VECTOR] = mix
```

Two things are going on in the processing of the `randao_reveal` signature.

First, the signature is verified using the proposer's public key before being mixed in. This means that the proposer has almost no choice about what it contributes to the RANDAO: it either contributes a single verifiable value – the correct signature over the epoch number – or it withholds its block and contributes nothing. (Equivalently, a block with an incorrect reveal is invalid.)

Second, the hash of the signature is mixed in to the beacon state's RANDAO using `xor`. The combination of using the epoch number as the signed quantity and using `xor` to mix it in leads to a subtle, albeit tiny, improvement in attack-resistance of the RANDAO.




*What's really happening when the RANDAO is shuffled. The signature over the epoch number is the RANDAO reveal that the proposer includes in its block. This is hashed then mixed in to the existing RANDAO with an* `xor` *operation.*

Justin Drake explains in his notes:

> Using `xor` in `process_randao` is (slightly) more secure than using `hash`. To illustrate why, imagine an attacker can grind randomness in the current epoch such that two of his validators are the last proposers, in a different order, in two resulting samplings of the next epochs. The commutativity of `xor` makes those two samplings equivalent, hence reducing the attacker's grinding opportunity for the next epoch versus `hash` (which is not commutative). The strict security improvement may simplify the derivation of RANDAO security formal lower bounds.

We will see shortly that it can be advantageous to an attacker to have control of the last slots of an epoch. Justin's point is that, under the current scheme, the attacker having validators $V_0, V_1$ in the two last slots of an epoch is equivalent to it having $V_1, V_0$ with respect to the `randao_reveals`. This fractionally reduces an attackers choices when it comes to influencing the RANDAO. If we used `hash` rather than `xor`, or if we signed over the slot number rather than the epoch number, these orderings would result in different outcomes from each other, giving an attacker more choice and therefore more power.

### Lookahead

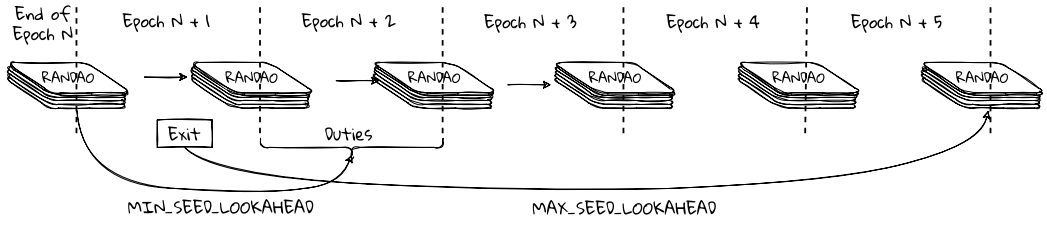
We started this section with a discussion of unpredictability. Ideally, it should not be possible to predict the duties for any block proposer or committee member until the moment they become active. However, in practice, proposers and committee members need a little advance notice of their duties to allow them to join the right p2p network subnets and do whatever other preparation they need to do.

The RANDAO seed at the end of epoch $N$ is used to compute validator duties for the whole of epoch $N + 2$. This interval is controlled by `MIN_SEED_LOOKAHEAD` via the `get_seed()` function. Thus validators have at least one full epoch to prepare themselves for any duties, but no more than two.

Under normal circumstances, then, an attacker is not able to predict the duty assignments more than two epochs in advance. However, if an attacker has a large proportion of the stake or is, for example, able to mount a DoS attack against block proposers for a while, then it might be possible for the attacker to predict the output of the RANDAO further ahead than `MIN_SEED_LOOKAHEAD` would normally allow. The attacker might then use this foreknowledge to strategically exit validators or make deposits[29] in order to gain control of a committee, or a large number of block proposal slots.

It's certainly not an easy attack. Nonetheless it's easy to defend against, so we might as well do so.

To prevent this, we assume a maximum feasible lookahead that an attacker might achieve, `MAX_SEED_LOOKAHEAD` and delay all activations and exits by this amount, which allows time for new randomness to come in via block proposals from honest validators, making irrelevant any manipulation by the entering or exiting validators. With `MAX_SEED_LOOKAHEAD` set to 4, if only 10% of validators are online and honest, then the chance that an attacker can succeed in forecasting the seed beyond (`MAX_SEED_LOOKAHEAD - MIN_SEED_LOOKAHEAD`) = 3 epochs is $0.9^{3\times32}$, which is about 1 in 25,000.




*The RANDAO value at the end of epoch $N$ is used to set duties for epoch $N+2$, which is controlled by `MIN_SEED_LOOKAHEAD`. A validator exiting in epoch $N+1$ remains active until at least the end of epoch $N+5$ (depending on the exit queue). This is controlled by `MAX_SEED_LOOKAHEAD`.*

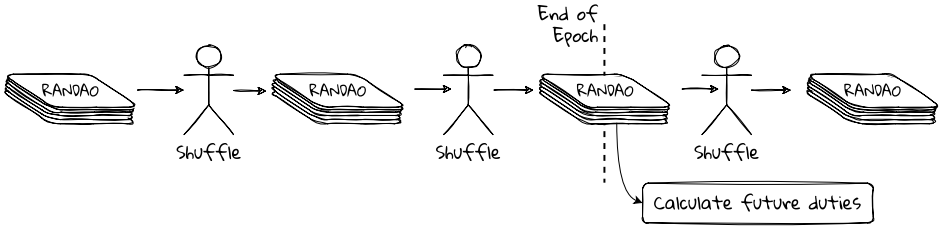
[29] In the current protocol you'd need to predict the RANDAO for around 16 hours ahead for deposits to be useful in manipulating it, due to `ETH1_FOLLOW_DISTANCE` and `EPOCHS_PER_ETH1_VOTING_PERIOD`. However, at some point post-Merge, it may become possible to onboard deposits more-or-less immediately.

### Single Secret Leader Election

As currently implemented, both the minimum and maximum lookaheads smell a little of engineering hackery. In a perfect design only the block proposer would know ahead of time that it has been chosen to propose in that slot. Once its block is revealed then the rest of the network would be able to verify that, yes, this was indeed the chosen proposer. This feature is called Single Secret Leader Election. We do not yet have it in the Ethereum protocol, and I shall write about it elsewhere. Meanwhile, some good progress is being made towards making it practical.

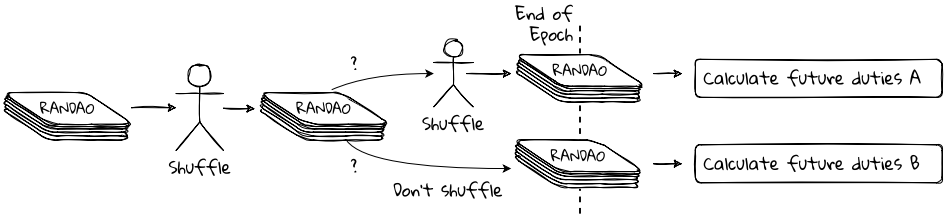
### RANDAO biasability

The RANDAO value for an epoch is set at the end of the previous epoch, and duty assignments for the entire epoch (proposals and committee memberships) depend on that value. (Actually – due to `MIN_SEED_LOOKAHEAD` – on the RANDAO value at the end of the last-but-one epoch, but we'll overlook that in what follows.)




*Future duty assignments for validators – block proposers, committee members, sync committee duty – are calculated based on the state of the RANDAO at the end of each epoch.*

Thus, when a validator happens to be assigned to propose a block in the last slot of an epoch, it gains a small amount of control over the assignments for the next epoch. This is because it can choose to reveal its block, which mixes in its RANDAO reveal, or it can choose (at a cost) to withhold its block and keep the existing RANDAO value, knowing that there will be no subsequent RANDAO change before duties are calculated. In this way, a validator is able to exert a little influence over the proposer and committee assignments in the next epoch. This is called "one bit of influence" over the RANDAO as the validator has a choice of two outcomes.




*The last proposer in an epoch has a choice. It can propose its block as usual, updating the RANDAO, resulting in a set of duty assignments A. Or it can withhold its block, leaving the RANDAO as-is, resulting in a set of duty assignments B. If outcome B gives the owner of the validator sufficient advantage to compensate for having missed a proposal, then it is an opportunity to "cheat".*

If an attacker gets a string of proposals at the end of an epoch then it has more power. Having $k$ consecutive proposals at the end of an epoch gives the attacker $2^k$ choices for the ultimate value of the RANDAO that will be used to compute future validator duties. In this scenario the attacker has "$k$ bits of influence" over the RANDAO.

### Biasability analyses

This section is fully optional. I got a bit carried away with the maths and it's fine to skip to the next section.

To make discussion of RANDAO biasability more concrete I shall try to quantify what it means in practice with a couple of examples. In each case the entity "cheating" or "attacking" has control over a proportion of the stake $r$, either directly or through some sort of collusion, and we will assume that the remaining validators are all acting independently and correctly. We will also assume, of course, that individual `randao_reveal`s are uniformly random.

In the first example, I will try to gain control of the RANDAO by permanently acquiring proposals in the last slots of an epoch. In the second example I will try to improve my expected number of block proposals by biasing the RANDAO when I get the opportunity to do so. In both cases I will be selectively making and withholding proposals having computed the best outcome: a process of "grinding" the RANDAO.

These examples are intended only as illustrations. They are not academic studies, and there are lots of loose ends. It's very likely I've messed something up: probability is *hard*. I'd be very interested if anyone wanted to make them more rigorous and complete. Some related work, more simulation based, was previously done by Runtime Verification.

### RANDAO takeover

If I control a proportion $r$ of the total stake, how much can I boost my influence over the protocol by manipulating the RANDAO?

The ability to influence the RANDAO depends on controlling a consecutive string of block proposals at the end of an epoch. We shall call this property "having a tail", and the tail will have a length $k$ from 0 to a maximum of 32, an entire epoch.

Our question can be framed like this: if I have a tail of length $k$ in one epoch, what is my expected length of tail in the next epoch? With a tail of length $k$ I have $2^k$ opportunities to reshuffle the RANDAO by selectively making or withholding block proposals. Can I grind through the possibilities to increase my tail length next time, and eventually take over the whole epoch?

In the absence of any manipulation, my probability of having a tail of length exactly $k$ in any given epoch is $(1-r)r^k$ for $k < 32$, and $r^{32}$ when $k = 32$. This is the chance that I make $k$ proposals in the tail positions preceded by a proposal that I did not make.

$$q_k = \begin{cases} (1-r)r^k & 0 \le k < 32 \\ r^k & k = 32 \end{cases}$$

So the expected tail length for someone controlling a proportion $r$ of the stake is,

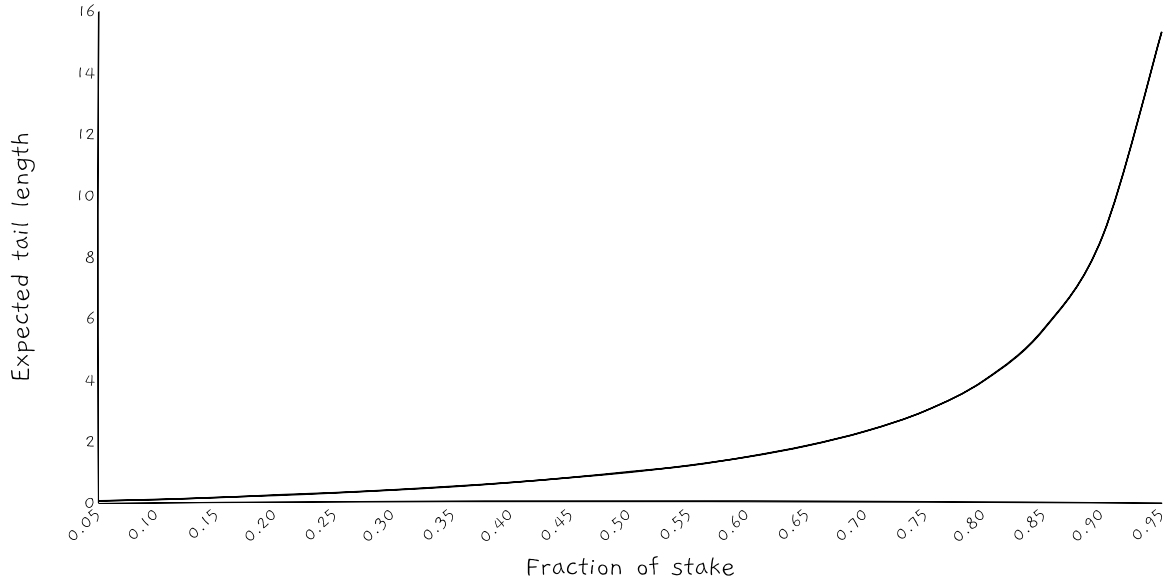
$$E(r) = \sum_{n=1}^{32} nq_n = \sum_{n=1}^{31} n(1-r)r^n + 32r^{32}$$

Now we will calculate $E^{(k)}(r)$, the expected length of tail I can achieve in the next epoch by using my previous tail of length $k$ to grind the options.

Consider the case where I have a tail of length $k = 1$ in some epoch. This gives me two options: I can publish my RANDAO contribution or I can withhold my RANDAO contribution (by withholding my block). My strategy is to choose the longest tail for the next epoch that I can gain via either of these options.

The probability, $p_j^{(1)}$, of gaining a tail of exactly length $j$ as a result of having a tail of length 1 is,

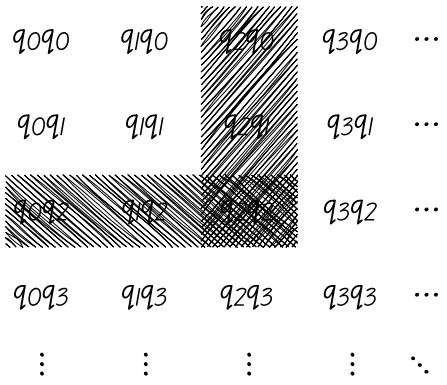
$$p_j^{(1)} = 2\sum_{i=0}^{j-1} q_j q_i + q_j q_j = q_j \left( 2\sum_{i=0}^{j-1} q_i + q_j \right)$$




*The bottom axis is r, and the side axis is my expected proposals tail length $E(r)$ assuming no RANDAO manipulation.*

We can think about this as follows. With $k = 1$ we get two attempts, therefore $q$ appears twice in each product. To calculate $p_j^{(1)}$ we need the sum over the all the combinations of the probability of getting a tail of length exactly $j$ (that is, $q_j$) multiplied by the probability of getting a tail of $j$ or less (that is, not getting a tail longer than $j$, otherwise we would have chosen that length instead of $j$).

Visually, calculating $p_2^{(1)}$ looks like the sum of the values in the shaded area of the next diagram.




*The probability that we get a maximum tail length of exactly two with two attempts is the sum of the terms in the shaded areas. Despite the overlap, each term is included only once.*

This example with tail length $k = 1$ results in a two-dimensional square since we have two possibilities to try. One way to calculate $p_j^{(1)}$ is to take the difference between the sum of all the products in the square side $j + 1$ and the sum of all the products in the square side $j$.

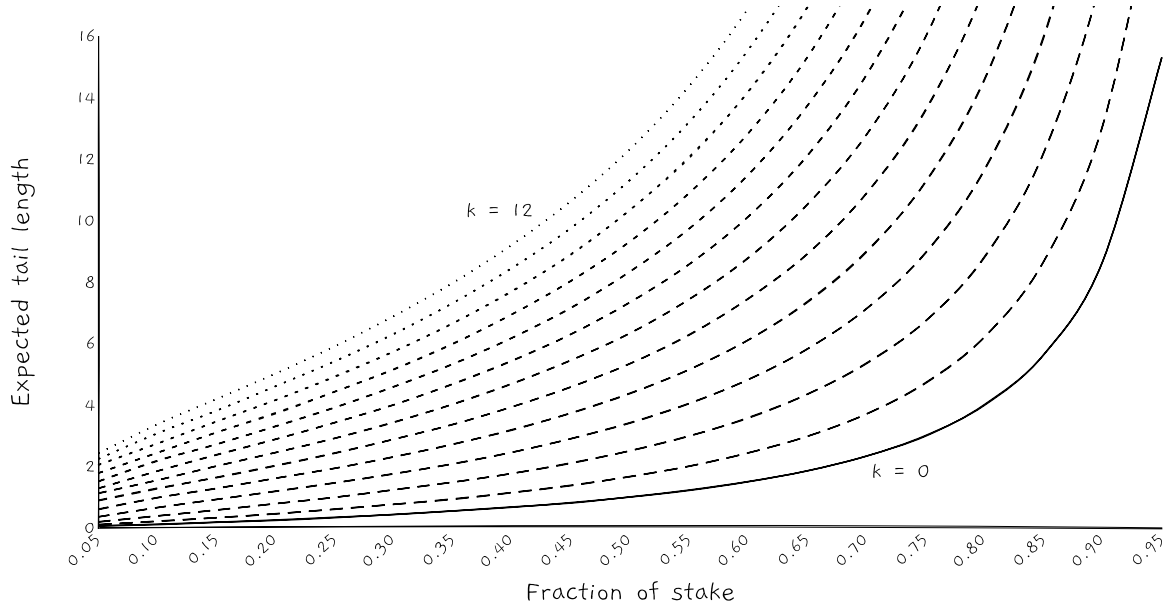
Thinking of it like this helps us to generalise to the cases when $k > 1$. In those cases we are dealing with a hyper-cube of dimension $2^k$; each element is the product of $2^k$ values of $q$. To calculate $p_j^{(k)}$ we can find the difference between the sum of all the products in the $2^k$-dimensional cube side $j + 1$ and the sum of all the products in the $2^k$-dimensional cube side $j$. This is tedious to write down and involves a

mind-boggling number of calculations even for quite small $k$, but see my example code for an efficient a way to calculate it.

Now, finally, we can calculate the expected tail length in the next epoch given that we have a tail of length $k$ in this epoch.

$$E^{(k)}(r) = \sum_{n=1}^{32} np_n^{(k)}$$

Graphing this for various values of $k$ we get the following. Note that the solid, $k = 0$, line is the same as $E(r)$ above - the expected tail with no manipulation. That is, $E^{(0)}(r) = E(r)$ as you'd expect.




*The bottom axis is $r$, and the side axis is my subsequent expected proposals tail length, $E^{(k)}(r)$ given various values of tail length $k$ that I can play with. Note that $E^{(0)}(r) = E(r)$ from the graph above.*

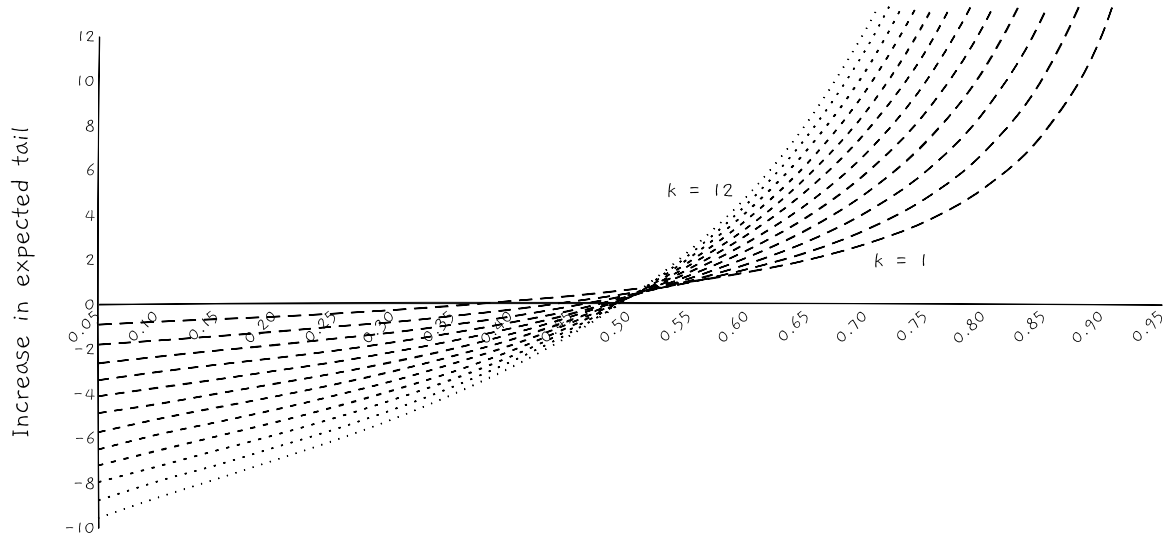
We see that, if I end up with any length of tail in an epoch, I can always grind my RANDAO contributions to improve my expected length of tail in the next epoch when compared with not grinding the RANDAO. And the longer the tail I have, the better the tail I can expect to have in the next epoch. These results are not surprising.

The important question is, under what circumstances can I use this ability in order to indefinitely increase my expected tail length, so that I can eventually gain full control of the RANDAO?

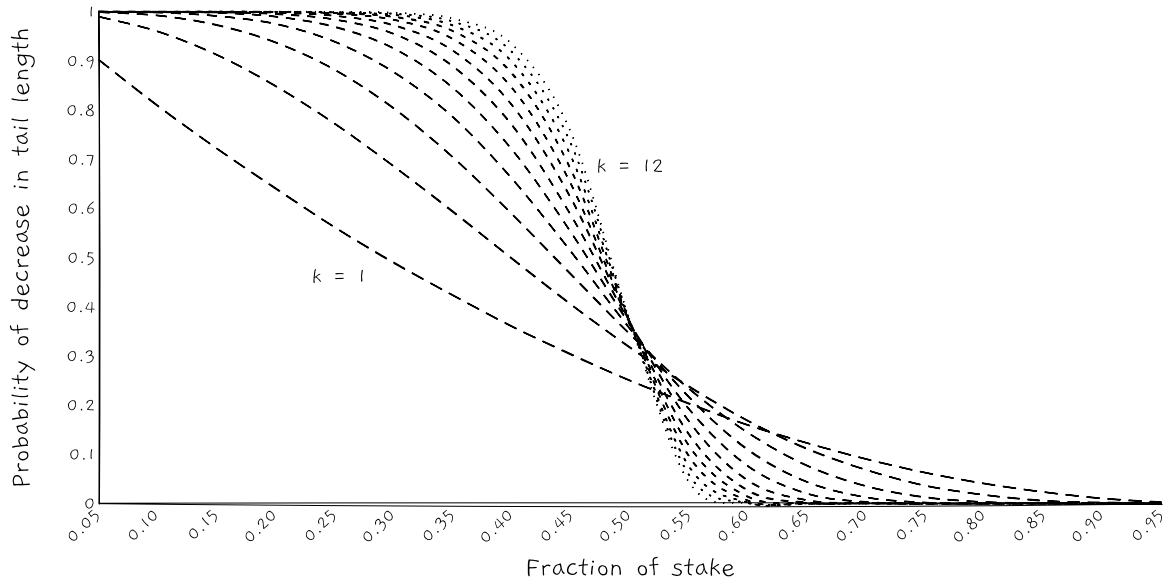
To investigate this, consider the following graph. Here, for each $k$ line we have plotted $E^{(k)}(r) - k$. This allows us to see whether our expected tail in the next epoch is greater or less than our current tail. If $E^{(k)}(r) - k$ is negative then I can expect to have fewer proposals in the next epoch than I have in this one.

We can see that for $r$ less than around 0.5, especially as $k$ grows, we expect our tail length to shrink rather than grow, despite our best RANDAO grinding efforts. However, for $r$ greater than 0.5, we expect our tail length to grow as a result of our grinding, whatever tail length we start with.

For completeness, we shouldn't only look at expectations, but also at probabilities. The following graph shows the probability that if I have a tail of length $k$ then I will have a tail of length less than $k$ in the next epoch. As $k$ increases you can see that a step function is forming: for a proportion of stake less than about 50% it becomes practically certain that my tail will decrease in length from one epoch to the next despite my best efforts to grow it; conversely, for a proportion of stake greater than a little over 50% it becomes practically certain that I can maintain or grow my tail of block proposals.




*The bottom axis is $r$, and the side axis is my subsequent expected proposals tail length minus my current tail length, $E^{(k)}(r) - k$ for various values of $k$.*




*The bottom axis is $r$, and the side axis is the probability that my best tail length in the next epoch is less than my current tail length for various values of tail length $k$.*

### Discussion of RANDAO takeover

What can we conclude from this? If I control less than about half the stake, then I cannot expect to be able to climb the ladder of increasing tail length: with high probability the length of tail I have will decrease rather than increase. Whereas, if I have more than half the stake, my expected length of tail increases each epoch, so I am likely to be able to eventually take over the RANDAO completely. With high enough $r$, the $2^k$ options I have for grinding the RANDAO overwhelm the probability of losing tail proposals. For large values of $k$ it will not be practical to grind through all of these options, but we need to arrive at only one good combination in order to succeed so we might not need to do the full calculation.

The good news is that, if attackers control more than half the stake, they have more interesting attacks available, such as taking over the LMD fork choice rule. So we generally assume in the protocol that any attacker has less than half the stake, in which case the RANDAO takeover attack appears to be infeasible.

As a final observation, we have ignored cases where two or more of the tail proposals are from the same validator. As discussed above, these proposals would each result in the same RANDAO contribution and reduce my grinding options. However, with a large number of validators in the system this is a reasonable approximation to make.

Code for calculating the length of tail with cheating

Here is the code for generating the data for the graphs above. The length of tail goes up to $k = 12$. Feel free to increase that, although it gets quite compute intensive. Twelve is enough to see the general picture.

```
def prob_tail_eq(r, k):
    return (1 - r) * r**k if k < N else r**k

# The sum of the products of all the q_i in the hypercube of side j and dim k
# Recursive is cooler, but written iteratively so that python doesn't run out of stack
def hyper(q, j, k):
    h = 1
    for n in range(1, k + 1):
        h = sum([q[i] * h for i in range(j)])
    return h

# Smoke test.
assert abs(hyper([0.9, 0.09, 0.009, 0.0009, 0.00009, 0.00001], 6, 32) - 1.0) < 1e-12

N    = 32 # The number of slots per epoch
KMAX = 12 # The maximum length of prior tail we will consider
NINT = 20 # The number of intervals of r between 0 and 1 to generate

expected = [[] for i in range(KMAX + 1)]
prob_dec = [[] for i in range(KMAX + 1)]
rs = [i / NINT for i in range(1, NINT)]
for r in rs:
    # q[j] = the probability of having a tail of exactly j in one attempt
    q = [prob_tail_eq(r, j) for j in range(N + 1)]
    for k in range(KMAX + 1):
        h = [hyper(q, j, 2**k) for j in range(N + 2)]
        # p[j] = the probability that with a tail of k I can achieve a tail of j in the next epoch
        p = [h[j + 1] - h[j] for j in range(N + 1)]
        # The expected length of tail in the next epoch given r and k
        expected[k].append(sum([j * p[j] for j in range(N + 1)]))
        # The probability of a decrease in tail length to < k
        prob_dec[k].append(h[k])
print(rs)
print(expected)
print(prob_dec)
```

**Block proposals boost**

For the second worked example I will try to improve the overall number of proposals that I get among my validators. Unlike in the first example, I will not be trying to maximise my advantage at any cost. I will only manipulate the RANDAO when I can do so without any net cost to myself.

Once again, I control a proportion $r$ of the stake. I will only be considering tails of length zero or of length one - going beyond that gets quite messy, and my intuition is that for values of $r$ less than a half or so it will make little difference.

Let $q_j$ be my probability of getting exactly $j$ proposals in an epoch without any manipulation of the RANDAO (different from the $q$ in the first example, but related):

$$q_j = r^j(1-r)^{32-j}\binom{32}{j}$$

My expected number of proposals per epoch when acting honestly is simple to compute,

$$E = \sum_{n=1}^{32} nq_n = 32r$$

Now I will try to bias the RANDAO to give myself more proposals whenever I have the last slot of an epoch, which will happen with probability $r$. Doing this, my expected number of proposals in the next epoch is as follows. The prime is to show that I am trying to maximise my advantage (cheat), and the subscript is to show that we are looking one epoch ahead.

$$E'_1 = \sum_{n=1}^{32} n((1-r)q_n + rp_n)$$

Unpacking this, the first term in the addition is the probability, $1-r$, that I did not have the last slot in the previous epoch (so I cannot do any biasing) combined with the usual probability $q_n$ of having $n$ proposals in an epoch.

The second term is the probability, $r$, that I *did* have the last slot in the previous epoch combined with the probability $p_n$ that I get either $n$ proposals by proposing my block, or $n+1$ proposals by withholding my block. We need the plus one to make up for the block I would be withholding at the end of the previous epoch in order to get this outcome.

$$p_j = \begin{cases} \sum_{i=0}^{j} q_i(q_j + q_{j+1}) & 0 \le j < 32 \\ \sum_{i=0}^{j} q_i q_j & j = 32 \end{cases}$$

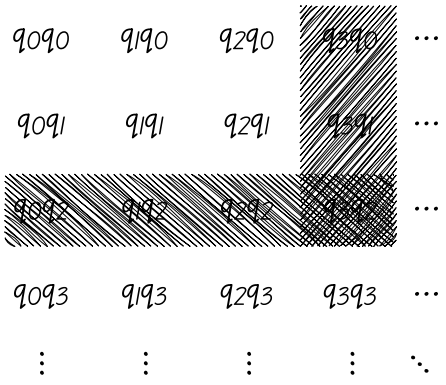
As before, we can illustrate this by considering the matrix of probabilities. With a tail of one I have two choices: to propose or to withhold. To achieve a net number of exactly $j$ proposals we are looking for the combinations where either of the following holds.

1. Proposing gives me exactly $j$ proposals and withholding gives no more than $j+1$ (that is, $\sum_{i=0}^{j+1} q_i q_j$). These are the elements in the horizontal bar in the diagram below.
2. Proposing gives me no more than $j$ proposals and withholding gives me exactly $j+1$ (that is, $\sum_{i=0}^{j} q_{j+1} q_i$).[30] These are the elements in the vertical bar in the diagram below.

Note that the $q_{j+1}q_j$ element appears in both outcomes, but must be included only once.

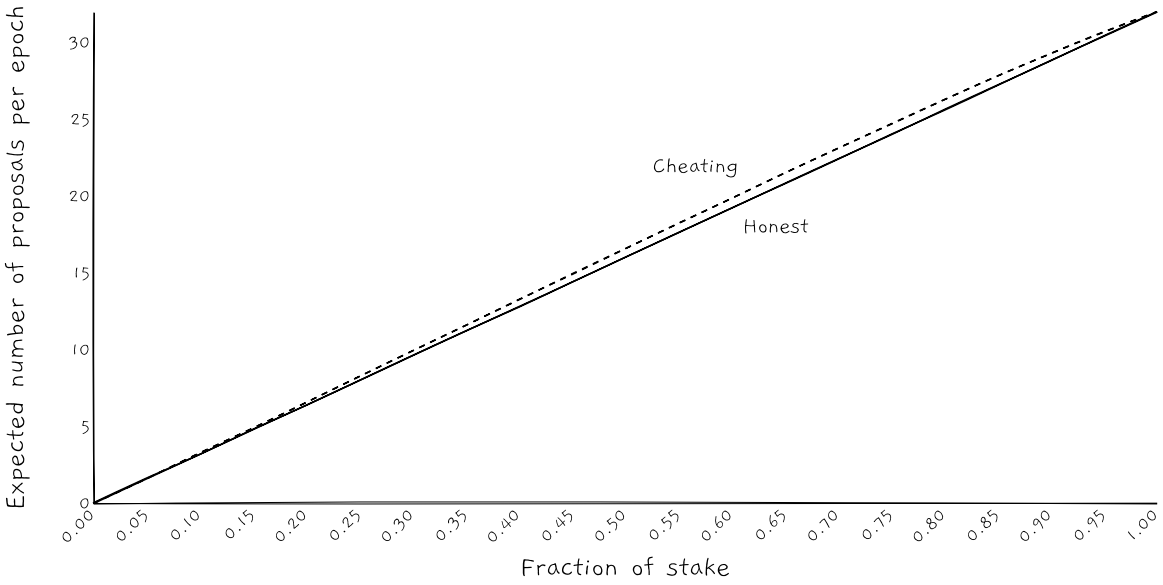
We can iterate this epoch by epoch to calculate the maximum long-term improvement in my expected number of proposals. The probability that I gain the last slot of epoch $N$ is $E'_N/32$.

[30] You can see why I am restricting this example to tails of length just zero or one: I don't want to think about what this looks like in a $2^k$ dimensional space.




*The probability that we get a net number of exactly two proposals with two attempts is the sum of the terms in the shaded areas. Despite the overlap, each term is included only once.*

$$E'_{N+1} = \sum_{n=1}^{32} n \left( (1 - \frac{E'_N}{32}) q_n + \frac{E'_N}{32} p_n \right)$$



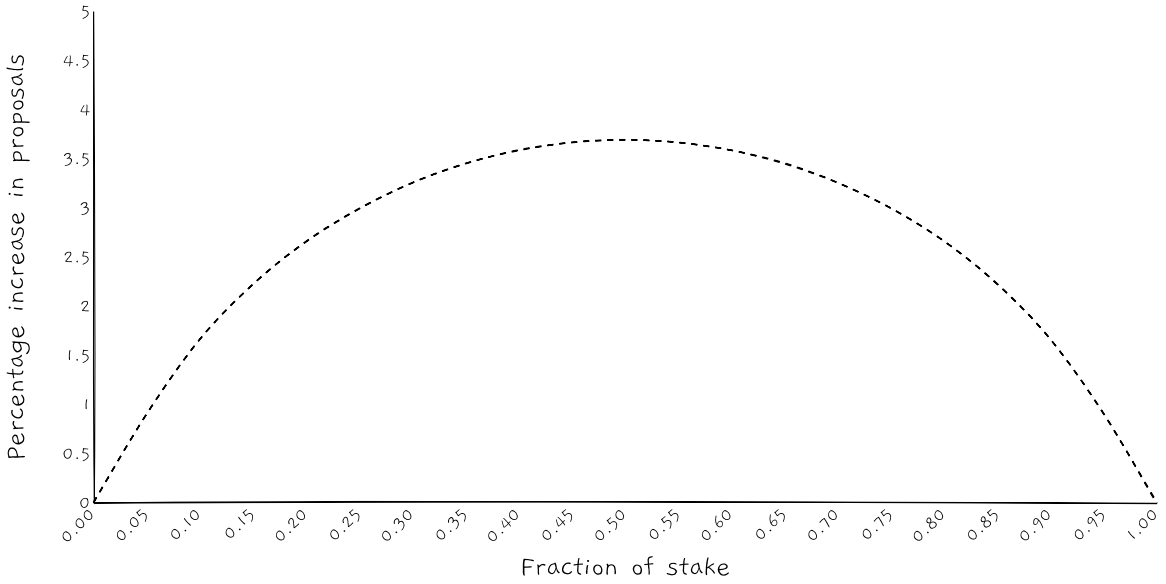

*The solid line is $E$, the expected number of block proposals per epoch for a proportion of the stake that does not seek to bias the RANDAO. The dashed line is $E'$, the long-term expected number of block proposals per epoch for a proportion of the stake that coordinates to bias the RANDAO in its favour.*

The maximum percentage gain in block proposals that I can acquire is shown in the following graph.

Code for calculating the expected number of proposals with cheating

The following Python code calculates $E'_N$ to convergence.

```
def fac(n):
    return n * fac(n - 1) if n else 1

def choose(n, k):
```




*The long-term percentage increase in the expected number of proposals per epoch that can be gained by a proportion of the stake coordinating to bias the RANDAO. An entity with 25% of the stake can gain an extra 2.99% of proposals (8.24 per epoch rather than exactly 8), assuming that the remaining stakers are uncoordinated.*

```
    return fac(n) / fac(k) / fac(n - k)

def prob(n, k, r):
    return r**k * (1 - r)**(n - k) * choose(n, k)

nintervals = 20
for idx in range(1, nintervals + 1):
    r = r0 = idx / nintervals
    q = [prob(32, j, r0) for j in range(33)]

    p = []
    for j in range(33):
        p.append(sum([q[i] * q[j] + (q[j + 1] * q[i] if (j < 32) else 0) for i in range(j + 1)]))

    # Iterate to convergence
    e = 0
    while (e == 0 or abs(e - e_old) > 0.000001):
        e_old = e
        e = sum([i * (q[i] * (1 - r) + p[i] * r) for i in range(33)])
        r = e / 32

    print(r0, r0 * 32, e, 100 * (e / (r0 * 32) - 1))
```

### Discussion of proposals boost

In the above analysis we considered only the effect of using the last slot of an epoch to bias the RANDAO and saw that an entity with any amount of stake can fractionally improve its overall expected number of block proposals, assuming that everyone else is acting honestly.

The expected gain may be higher if we consider using the two last slots, or the $k$ last slots, especially if combined with the previous tail-extension attack. But I expect that for $r$ less than a half or so any further improvement will be very small.

### Verifiable delay functions

We've seen that, although the RANDAO is biasable, it is not so biasable as to break the protocol: for our purposes the randomness is "good enough".

Nonetheless, it is interesting to explore how it might be improved, especially as, with The Merge, the RANDAO contents will be available to Ethereum's smart contract layer. Randomness biasability in a large lottery contract, for example, could be more of a problem than biasability in the consensus layer.

The long-term fix for biasability is to use a verifiable delay function (VDF). A VDF is guaranteed to be slow to compute its output, but that output can be verified quickly. In practice the VDF is a calculation run on a specialised hardware device that is assumed to have a performance within a small factor of the theoretical maximum performance. So, a VDF might output a result in, say, 20 seconds with the assumption that the best that any other device could do is to obtain the result in, say, 5 seconds.

The idea is that RANDAO updates would come from the output of the VDF. A proposer would have to decide whether to commit its `randao_reveal` before it is possible for it to compute the actual contribution: the future output of the VDF. This eliminates any opportunistic biasing of the RANDAO.

Only one VDF needs to be active at any time on the network since it can publish its result for quick verification by all the other nodes.

Although a lot of work has been done on designing and specifying VDFs there is no active plan to implement one in Ethereum at this time.

### See also

Vitalik has some notes on randomness in his Annotated Ethereum 2.0 Spec. His article Validator Ordering and Randomness in PoS summarises some early thinking on the options for random validator selection in proof of stake[31].

On RANDAO biasability, Runtime Verification did an analysis in 2018 that both complements and goes deeper than the sketches I presented in this section. There is both a statistical model and a thorough write-up of their work.

A search for RANDAO on ethresear.ch yields quite a few articles discussing various issues with it, and proposing some solutions (none of which we have adopted).

A good place to start exploring verifiable delay functions is the VDF Alliance site.

## Shuffling

| First cut | ✓ | Revision | TODO |
|---|---|---|---|

- Shuffling is used to randomly assign validators to committees and choose block proposers.
- Ethereum 2 uses a "swap-or-not" shuffle.
- Swap-or-not is an oblivious shuffle: it can be applied to single list elements and subsets.
- This makes it ideal for supporting light clients.

[31] This article seems only to be available now on the Internet Archive. I am grateful to Patrick McCorry for tracking it down.

### Introduction

Shuffling is used to randomly assign validators to committees, both attestation committees and sync committees. It is also used to select the block proposer at each slot.

Although there are pitfalls to be aware of, shuffling is a well understood problem in computer science. The gold standard is probably the Fisher–Yates shuffle. So why aren't we using that for Eth2? In short: light clients.

Other shuffles rely on processing the entire list of elements to find the final ordering. We wish to spare light clients this burden. Ideally, they should deal with only the subsets of lists that they are interested in. Therefore, rather than Fisher–Yates, we are using a construction called a "swap-or-not" shuffle. The swap-or-not shuffle can tell you the destination index (or, conversely, the origin index) of a single list element, so is ideal when dealing with subsets of the whole validator set.

For example, formally committees are assigned by shuffling the full validator list and then taking contiguous slices of the resulting permutation. If I only need to know the members of committee $k$, then this is very inefficient. Instead, I can run the swap-or-not shuffle backwards for only the indices in slice $k$ to find out which of the whole set of validators would be shuffled into $k$. This is much more efficient.

### Swap-or-not Specification

The algorithm for shuffling in the specification deals with only a single index at a time.

```
def compute_shuffled_index(index: uint64, index_count: uint64, seed: Bytes32) -> uint64:
    """
    Return the shuffled index corresponding to ``seed`` (and ``index_count``).
    """
    assert index < index_count

    # Swap or not (https://link.springer.com/content/pdf/10.1007%2F978-3-642-32009-5_1.pdf)
    # See the 'generalized domain' algorithm on page 3
    for current_round in range(SHUFFLE_ROUND_COUNT):
        pivot = bytes_to_uint64(hash(seed + uint_to_bytes(uint8(current_round)))[0:8]) % index_count
        flip = (pivot + index_count - index) % index_count
        position = max(index, flip)
        source = hash(
            seed
            + uint_to_bytes(uint8(current_round))
            + uint_to_bytes(uint32(position // 256))
        )
        byte = uint8(source[(position % 256) // 8])
        bit = (byte >> (position % 8)) % 2
        index = flip if bit else index

    return index
```

An index position in the list to be shuffled, `index`, is provided, along with the total number of indices, `index_count`, and a `seed` value. The output is the index that the initial index gets shuffled to.

The hash functions used to calculate `pivot` and `source` are deterministic, and are used to generate pseudo-random output from the inputs: given the same input, they will generate the same output. So we can see that, for given values of `index`, `index_count`, and `seed`, the routine will always return the same output.

The shuffling proceeds in rounds. In each round, a `pivot` index is pseudo-randomly chosen somewhere in the list, based only on the `seed` value and the round number.

Next, an index `flip` is found, which is `pivot - index`, after accounting for wrap-around due to the modulo function. The important points are that, given `pivot`, every `index` maps to a unique `flip`, and that the calculation is symmetrical, so that `flip` maps to `index`.

- With `index_count = 100`, `pivot = 70`, `index = 45`, we get `flip = 25`.
- With `index_count = 100`, `pivot = 70`, `index = 82`, we get `flip = 88`.

Finally in the round, a decision is made as to whether to keep the index as-is, or to update it to `flip`. This decision is pseudo-randomly made based on the values of `seed`, the round number, and the higher of `index` and `flip`.

Note that basing the swap-or-not decision on the higher of `index` and `flip` brings a symmetry to the algorithm. Whether we are considering the element at `index` or the element at `flip`, the decision as to whether to swap the elements or not will be the same. This is the key to seeing the that full algorithm delivers a shuffling (permutation) of the original set.

The algorithm proceeds with the next iteration based on the updated index.

It may not be immediately obvious, but since we are deterministically calculating `flip` based only on the round number, the shuffle can be run in reverse simply by running from `SHUFFLE_ROUND_COUNT - 1` to `0`. The same swap-or-not decisions will be made in reverse. As described above, this reverse shuffle is perfect for finding which validators ended up in a particular committee.

### A full shuffle

To get an intuition for how this single-index shuffle can deliver a full shuffling of a list of indices, we can consider how the algorithm is typically implemented in clients when shuffling an entire list at once.

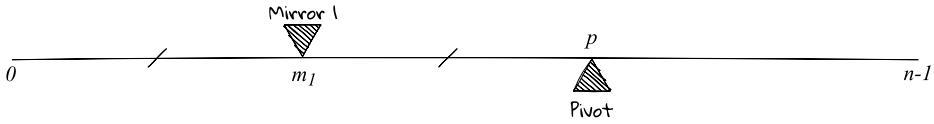
As an optimisation, the loop over the indices to be shuffled is brought inside the loop over rounds. This hugely reduces the amount of hashing required since the pivot is fixed for the round (it does not depend on the index) and the bits of `source` can be reused for 256 consecutive indices, since the hash has a 256-bit output.

For each round, we do the following.

### 1. Choose a pivot and find the first mirror index

First, we pick a pivot index $p$. This is pseudo-randomly chosen, based on the round number and some other seed data. The pivot is fixed for the rest of the round.

With this pivot, we then pick the mirror index $m_1$ halfway between $p$ and 0. That is, $m_1 = p/2$. (We will simplify by ignoring off-by-one rounding issues for the purposes of this explanation.)




*The pivot and the first mirror index.*

### 2. Traverse first mirror to pivot, swapping or not

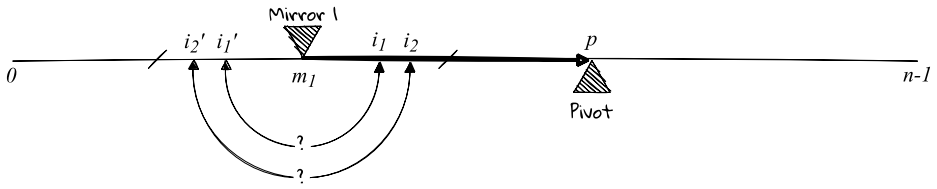
For each index between the mirror index $m_1$ and the pivot index $p$, we decide whether we are going to swap the element or not.

Consider the element at index $i$. If we choose not to swap it, we just move on to consider the next index.

If we do decide to swap, then we exchange the list element at $i$ with that at $i'$, its image in the mirror index. That is, $i$ is swapped with $i' = m_1 - (i - m_1)$, so that $i$ and $i'$ are equidistant from $m_1$. In practice we don't exchange the elements at this point, we just update the indices $i \to i'$, and $i' \to i$.

We make the same swap-or-not decision for each index between $m_1$ and $p$.

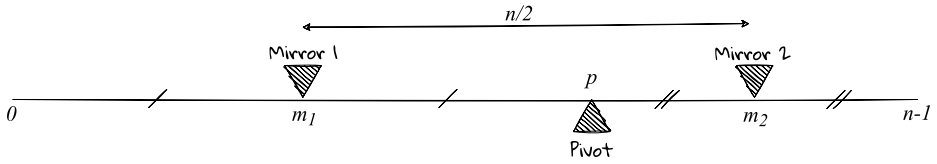
The decision as to whether to swap or not is based on hashing together the random seed, the round number, and some position data. A single bit is extracted from this hash for each index, and the swap is made or not according to whether this bit is one or zero.




*Swapping or not from the first mirror up to the pivot.*

### 3. Calculate the second mirror index

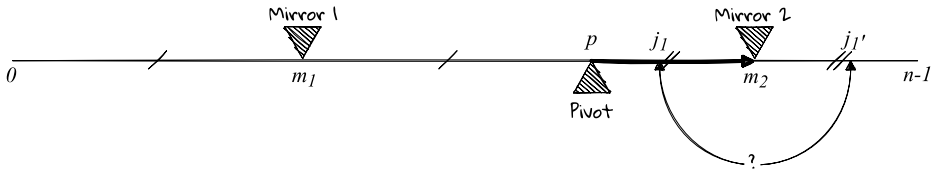
After considering all the indices $i$ from $m_1$ to $p$, mirroring in $m_1$, we now find a second mirror index at $m_2$, which is the point equidistant between $p$ and the end of the list: $m_2 = m_1 + n/2$.




*The second mirror index.*

### 4. Traverse pivot to second mirror, swapping or not

Finally, we repeat the swap-or-not process, considering all the points $j$ from the pivot, $p$ to the second mirror $m_2$. If we choose not to swap, we just move on. If we choose to swap then we exchange the element at $j$ with its image at $j'$ in the mirror index $m_2$. Here, $j' = m_2 + (m_2 - j)$.




*Swapping or not from the pivot to the second mirror.*

### Putting it all together

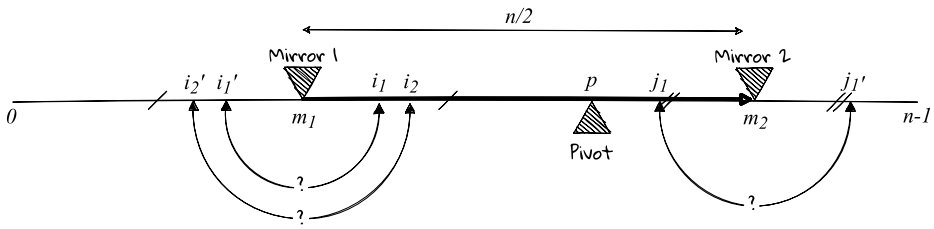
At the end of the round, we have considered all the indices between $m_1$ and $m_2$, which, by construction, is half of the total indices. For each index considered, we have either left the element in place, or swapped the element at a distinct index in the other half. Thus, all of the indices have been considered exactly once for swapping.

The next round begins by incrementing (or decrementing for a reverse shuffle) the round number, which gives us a new pivot index, and off we go again.

### Discussion

### A key insight

When deciding whether to swap or not for each index, the algorithm cleverly bases its decision on the higher of the candidate index or its image in the mirror. That is, $i$ rather than $i'$ (when below the pivot), and $j'$ rather than $j$ (when above the pivot). This means that we have flexibility when running through the indices of the list: we could do 0 to $m_1$ and $p$ to $m_2$ as two separate loops, or do it with a single loop from $m_1$ to $m_2$ as I outlined above. The result will be the same: it doesn't matter if we are considering $i$ or its image $i'$; the decision as to whether to swap or not has the same outcome.




*The whole process running from one mirror to the other in a single round.*

### The number of rounds

In Ethereum 2.0 we do 90 rounds of the algorithm per shuffle, set by the constant `SHUFFLE_ROUND_COUNT` [TODO - link]. The original paper on which this technique is based suggests that $6 \lg N$ rounds is required "to start to see a good bound on CCA-security", where $N$ is the list length. In his annotated spec Vitalik says "Expert cryptographer advice told us ~$4 \log_2 N$ is sufficient for safety". The absolute maximum number of validators in Eth2, thus the maximum size of the list we would ever need to shuffle, is about $2^{22}$ (4.2 million). On Vitalik's estimate that gives us 88 rounds required, on the paper's estimate, 92 rounds (assuming that lg is the natural logarithm). So we are in the right ballpark, especially as we are very, very unlikely to end up with that many active validators.

It might be interesting to make the number of rounds adaptive based on list length. But we don't do that; it's probably an optimisation too far.

Fun fact: when Least Authority audited the beacon chain specification, they initially found bias in the shuffling used for selecting block proposers (see Issue F in their report). This turned out to be due to mistakenly using a configuration that had only 10 rounds of shuffling. When they increased it to the 90 we use for mainnet, the bias no longer appeared.

### (Pseudo) randomness

The algorithm requires that we select a pivot point randomly in each round, and randomly choose whether to swap each element or not in each round.

In Eth2, we deterministically generate the "randomness" from a seed value, such that the same seed will always generate the same shuffling.

The pivot index is generated from eight bytes of a SHA256 hash of the seed concatenated with the round number, so it usually changes each round.

The decision bits used to determine whether or not to swap elements are bits drawn from SHA256 hashes of the seed, the round number, and the index of the element within the list.

### Efficiency

This shuffling algorithm is much slower than Fisher–Yates. That algorithm requires $N$ swaps. Our algorithm will require $90N/4$ swaps on average to shuffle $N$ elements.

We should also consider the generation of pseudo-randomness, which is the most expensive part of the algorithm. Fisher–Yates needs something like $N \log_2 N$ bits of randomness, and we need $90(\log_2 N + N/2)$ bits, which, for the range of $N$ we need in Eth2, is many more bits (about twice as many when $N$ is a million).

### Why swap-or-not?

Why would we use such an inefficient implementation?

### Shuffling single elements

The brilliance is that, if we are interested in only a few indices, we do not need to compute the shuffling of the whole list. In fact, we can apply the algorithm to a single index to find out which index it will be swapped with.

So, if we want to know where the element with index 217 gets shuffled to, we can run the algorithm with only that index; we do not need to shuffle the whole list. Moreover, if we want to know the converse, which element gets shuffled into index 217, we can just run the algorithm backwards for element 217 (backwards means running the round number from high to low rather than low to high).

In summary, we can compute the destination of element $i$ in $O(1)$ operations, and the source of element $i'$ (the inverse operation) also in $O(1)$, not dependent on the length of the list. Shuffles like the Fisher–Yates shuffle do not have this property and cannot work with single indices, they always need to iterate the whole list. The technical term for a shuffle having this property is that it is *oblivious* (to all the other elements in the list).

#### Keeping light clients light

This property is important for light clients. Light clients are observers of the Eth2 beacon and shard chains that do not store the entire state, but do wish to be able to securely access data on the chains. As part of verifying that they have the correct data – that no-one has lied to them – it is necessary to compute the committees that attested to that data. This means shuffling, and we don't want light clients to have to hold and shuffle the entire list of validators. By using the swap-or-not shuffle, light clients need only to consider the small subset of validators that they are interested in, which is vastly more efficient overall.

#### See also

- The initial discussion about the search for a good shuffling algorithm is Issue 323 on the specs repo.
- The winning algorithm was announced in Issue 563.
- The original paper describing the swap-or-not shuffle is Hoang, Morris, and Rogaway, 2012, "An Enciphering Scheme Based on a Card Shuffle". See the "generalized domain" algorithm on page 3.

## Committees

- Committees are subsets of the full set of active validators that are used to distribute the overall workload.
- Beacon committees manage attestations for the consensus protocol; sync committees are discussed elsewhere.
- Having 64 beacon committees at each slot is a relic of previous Eth2 designs.
- Nonetheless, multiple committees per slot allow us to parallelise attestation aggregation.
- Beacon committee membership is random and transient.
- A target minimum committee size of 128 protects them against capture.

#### Introduction

One of the challenges of building a highly scalable consensus protocol is organising the work involved so as not to overwhelm the network or individual nodes.

A goal of the Ethereum 2 Proof of Stake protocol is to achieve economic finality. In the current design (though see below for discussion of single slot finality) this requires us to gather votes from at least two-thirds of the validator set, and we must do this twice: once to justify an epoch, and once again to finalise it.

If the whole validator set were to attest simultaneously, the number of messages on the network would be immense, and the amount of work required of beacon nodes too much for modest hardware. This is

where committees help. The work of attesting is divided among subsets of the validator set (committees) and spread across an epoch (6.4 minutes). Each validator participates in only one of the committees.

The Altair spec has two types of committees, beacon committees and sync committees, each having quite a different function. We will focus on beacon committees in this section, and deal with sync committees in a later section.

The current beacon committee structure was strongly influenced by a previous roadmap that included in-protocol data sharding. That design is now deprecated, yet a remnant of it remains in our 64 beacon committees per slot. These were originally intended to map directly to 64 shards as "crosslink committees" but no longer have that function. Nonetheless, beacon committees still serve a useful purpose in parallelising the aggregation of attestations. Whether 64 remains the right number of committees per slot has not been analysed to my knowledge. The trade-off is that fewer beacon committees would reduce the amount of block space needed for aggregate attestations, but would increase the time needed for aggregators to do their work.

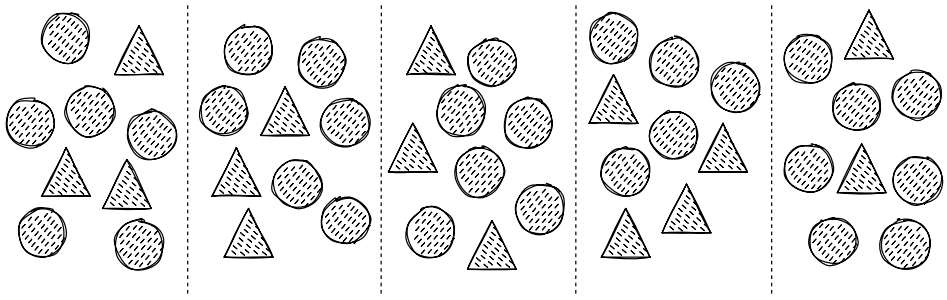
In any case, logically, the 64 committees in a slot now act as a single large committee, all voting on the same information.

### Committee assignments

Beacon committees are convened to vote exactly once and then disbanded immediately - they are completely transient. By contrast, a sync committee lasts for 256 epochs (a little over 27 hours), and votes 8192 times during that period.

During an epoch, every active validator is a member of exactly one beacon committee, so the committees are all disjoint. At the start of the next epoch, all the existing committees are disbanded and the active validator set is divided into a fresh set of committees.

The composition of the committees for an epoch is fully determined at the start of an epoch by (1) the active validator set for that epoch, and (2) the RANDAO seed value at the start of the previous epoch.



*Here we have divided thirty circles and fifteen triangles into five committees at random. The attacking triangles do not have a majority in any committee.*

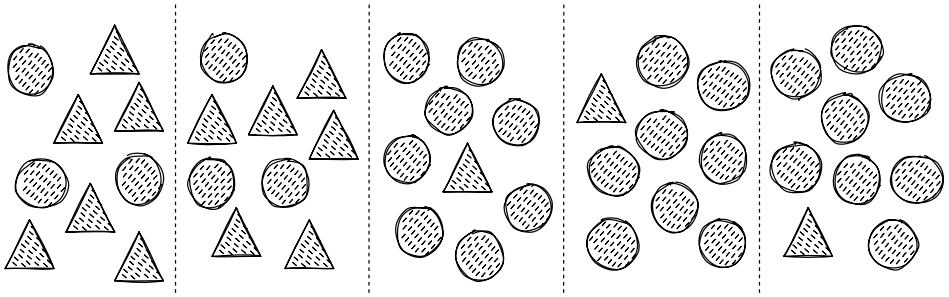
We assign validators to committees randomly in order to defend against a minority attacker being able to capture any single committee. If committee assignments were not random, or were calculable long in advance, then it might be possible for an attacker with a minority of validators to organise them so that they became a supermajority in some committees. They might do this by manipulating the entries and exits of their validators, for example.

The committee sizes used in the Eth2 protocol were chosen to make the takeover of a committee by a minority attacker extremely unlikely. See target committee size, below, for further analysis of this.

### The number of committees

The protocol adjusts the total number of committees in each epoch according to the number of active validators. The goals are,

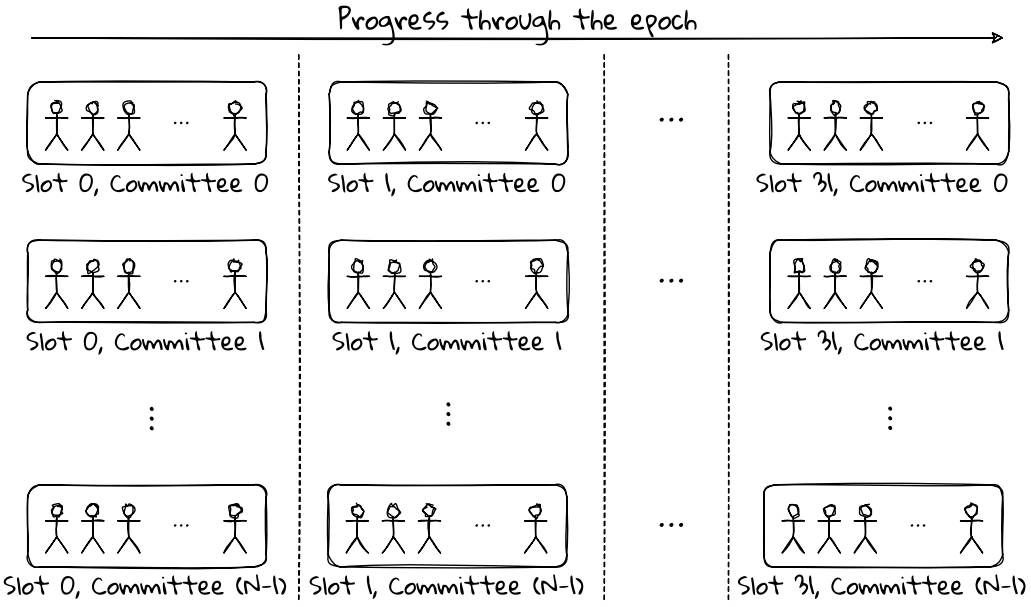
1. to have the same number of committees per slot throughout the epoch (so the number of committees in an epoch is always a multiple of `SLOTS_PER_EPOCH`),



*It would be improbable for the triangles to gain a 2/3 supermajority in a committee purely by chance. But if the attacker could manipulate the assignments then they might gain a supermajority in some committees, such as the first two here.*

2. to have the largest number of committees that ensures that each committee has at least `TARGET_COMMITTEE_SIZE` members, and
3. to have a maximum of `MAX_COMMITTEES_PER_SLOT` committees per slot.

Clearly, the first goal is not achievable if there are fewer than `SLOTS_PER_EPOCH` validators – is a committee a committee if nobody is in it? – and the second goal is not achievable if there are fewer than `SLOTS_PER_EPOCH * TARGET_COMMITTEE_SIZE` (4096) validators. The protocol could hardly be considered secure with fewer than 4096 validators, so this is not a significant issue in practice.




*Every slot in an epoch has the same number of committees, $N$, up to a maximum of `MAX_COMMITTEES_PER_SLOT`. Every active validator in the epoch appears in exactly one committee, thus the committees are all disjoint.*

The number of committees per slot is calculated by the spec function `get_committee_count_per_slot()`. This can be simplified for illustrative purposes, given the number $n$ of active validators in the epoch, as

```
MAX_COMMITTEES_PER_SLOT = 64
SLOTS_PER_EPOCH = 32
TARGET_COMMITTEE_SIZE = 128
```

```
def committees_per_slot(n):
    return max(1, min(MAX_COMMITTEES_PER_SLOT, n // SLOTS_PER_EPOCH // TARGET_COMMITTEE_SIZE))
```

This generates a committee structure that evolves as per the following table as the number of validators grows or shrinks.

| $n$ min | $n$ max | Committees / slot | Members per committee | Min | Max |
|---|---|---|---|---|---|
| 0 | 31 | 1 | Some committees have zero members | 0 | 1 |
| 32 | 4095 | 1 | $\lceil n/32 \rceil$ or $\lfloor n/32 \rfloor$, which is below `TARGET_COMMITTEE_SIZE` | 1 | 128 |
| 4096 | 262 143 | $N = \lfloor n/4096 \rfloor$ | $\lceil n/(32N) \rceil$ or $\lfloor n/(32N) \rfloor$ | 128 | 256 |
| 262 144 | 4 194 304 | 64 | $\lceil n/2048 \rceil$ or $\lfloor n/2048 \rfloor$ | 128 | 2048 |
| 4 194 305 | - | 64 | Things break Note that this can never happen in practice. | - | - |

The numbers at the various thresholds in this table are calculated from the spec constants:

- 32 is `SLOTS_PER_EPOCH`.
- 4096 is `SLOTS_PER_EPOCH * TARGET_COMMITTEE_SIZE`. This is the point at which all the committees achieve their target minimum size.
- 262,144 is `SLOTS_PER_EPOCH * TARGET_COMMITTEE_SIZE * MAX_COMMITTEES_PER_SLOT`. We have reached the maximum number of committees per slot (64). We no longer add new committees as the validator set grows, we just make the committees larger.
- 4,194,304 is `SLOTS_PER_EPOCH * MAX_VALIDATORS_PER_COMMITTEE * MAX_COMMITTEES_PER_SLOT`. There is not enough Ether in existence to allow us to reach this number of active validators. The limit exists in protocol to enable us to specify a maximum size for the `aggregation_bits` SSZ `Bitlist` type in attestations.

### Committee index

Each of the $N$ committees within a slot has a committee index from 0 to $N-1$. I will call this $i$ in what follows and refer to it as the slot-based index. This slot-based index is included in committees' attestations via the `AttestationData` object,

```
class AttestationData(Container):
    slot: Slot
    index: CommitteeIndex
    # LMD GHOST vote
    beacon_block_root: Root
    # FFG vote
    source: Checkpoint
    target: Checkpoint
```

The `slot` and the committee `index` within that slot together uniquely identify a committee, and together with the RANDAO value, its membership.

Since all committees in a slot are voting on exactly the same information (source, target, and head block), the `index` is the only thing that varies between the aggregate attestations produced by the slot's committees (assuming that most of the validators have the same view of the network). This prevents the attestations from the slot's committees being aggregated further, so we will generally end up with $N$ aggregate attestations per slot that we must store in a beacon block.

If it were not for the `index` then all these $N$ aggregate attestations could be further aggregated into a single aggregate attestation, combining the votes from all the validators voting at that slot.

As a thought experiment we can calculate the potential space savings of doing this. Given a committee size of $k$ and $N$ committees per slot, the current space required for $N$ aggregate `Attestation` objects is $N * (229 + \lfloor k/8 \rfloor)$ bytes. If we could remove the committee index from the signed data and combine all of these into a single aggregate `Attestation` the space required would be $221 + \lfloor kN/8 \rfloor$ bytes. So we could save $229N - 221$ bytes per block, which is 14.4KB with the maximum 64 committees. This seems nice to have, but would likely make the committee aggregation process more complex.

There is another index that appears when assigning validators to committees in `compute_committee()`: an epoch-based committee index that I shall call $j$. The indices $i$ and $j$ are related as $i = \mod(j, N)$ and $j = Ns + i$ where $s$ is the slot number in the epoch.

### The size of committees

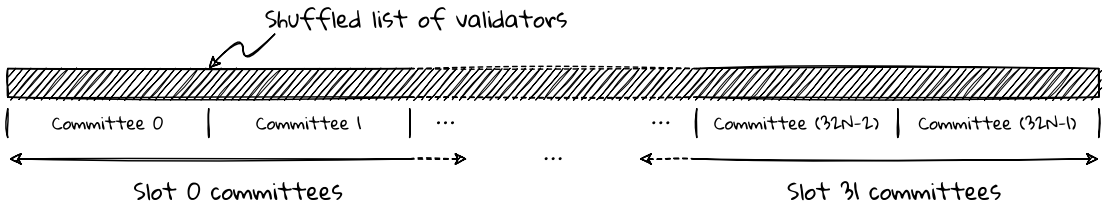
Validators are divided among the committees in an epoch by the `compute_committee()` function.

Given the epoch-based index $j$, `compute_committee()` returns a slice of the full, shuffled validator set as the committee membership. Within the shuffled list, the index of the first validator in the committee is $\lfloor nj/32N \rfloor$, and the index of the last validator in the committee is $\lfloor n(j+1)/32N \rfloor - 1$. So the size of each committee is either $\lfloor n/32N \rfloor$ or $\lceil n/32N \rceil$. In any case, committee sizes differ by at most one.

In simplified form the `compute_committee()` calculation looks like this. `N` is the number of committees per slot, `n` is the total number of active validators, and `j` is the epoch-based committee index,

```
def compute_committee_size(n, j, N):
    start = n * j // (32 * N)
    end = n * (j + 1) // (32 * N)
    return end - start
```

The length of the vector returned will be either `n // (32 * N)` or `1 + n // (32 * N)`. The function `compute_shuffled_index()` is described in the previous section.




*Conceptually, to calculate the committee assignments for an epoch, the entire active validator set is shuffled into a list of length $n$, then sliced into $32N$ committees of as close to the same size as possible. $N$ is the number of committees per slot. The epoch-based committee number, $j$, is shown.*

In the caption to the diagram above I said that this is "conceptually" how committee membership is determined. In practice, due to our use of an oblivious shuffle, the membership of an individual committee can be calculated without shuffling the entire validator set; the result will be the same.

### Target committee size

To achieve a desirable level of security, committees need to be larger than a certain size. This makes it infeasible for an attacker to randomly end up with a super-majority in a committee even if they control a significant number of validators. The target here is a kind of lower-bound on committee size. If there are not enough validators for all committees to have at least `TARGET_COMMITTEE_SIZE` (128) members, then, as a first measure, the number of committees per slot is reduced to maintain this minimum. Only if there are fewer than `SLOTS_PER_EPOCH * TARGET_COMMITTEE_SIZE` (4096) validators in total will the committee size be reduced below `TARGET_COMMITTEE_SIZE`. With so few validators the system would be insecure in any case.

Given a proportion of the validator set controlled by an attacker, what is the probability that the attacker ends up controlling a two-thirds majority in a uniformly randomly selected committee drawn from the full set of validators? Vitalik calculated 111 to be the minimum committee size required to maintain a $2^{-40}$ chance (one-in-a-trillion) of an attacker with one third of the validators gaining by chance a two-thirds majority in any one committee. The value 128 was chosen as being the next higher power of two.

If an attacker has a proportion $p$ of the validator set, then the probability of selecting a committee of $n$ validators that has $k$ or more validators belonging to the attacker is,

$$\sum_{i=k}^{n} p^i (1-p)^{n-i} \binom{n}{i}$$

Using this we can calculate that, in fact, 109 members is sufficient to give only a $2^{-40}$ chance of an attacker with one third of the validators gaining a two-thirds majority by chance.

Code for calculating the target committee size

The following is Vitalik's Python code for calculating the probabilities.

```
def fac(n):
    return n * fac(n-1) if n else 1

def choose(n, k):
    return fac(n) / fac(k) / fac(n-k)

def prob(n, k, p):
    return p**k * (1-p)**(n-k) * choose(n,k)

def probge(n, k, p):
    return sum([prob(n,i,p) for i in range(k,n+1)])
```

Armed with this we find that the minimum committee size to avoid a two-thirds majority with a $2^{-40}$ probability is 109 rather than 111.

```
>>> probge(108, 72, 1.0 / 3) < 2**-40
False
>>> probge(109, 73, 1.0 / 3) < 2**-40
True
```

In any case, a committee size of 128 is very safe against an attacker with 1/3 of the stake:

```
>>> probge(128, 86, 1.0 / 3)
5.551560731791749e-15
```

Odds of one-in-trillion may sound like over-engineering, but we must also consider that an attacker might gain some power over the RANDAO, so some safety margin is desirable.

Notwithstanding all of this, in the current beacon chain design the minimum target committee size is irrelevant as committees never operate alone. As long as we have at least 8192 active validators, each slot has multiple committees all operating together and it is their aggregate size that confers security, not the size of any individual committee. As previously mentioned, the current committee design is influenced by an old data sharding model that is now superseded. Nonetheless, individual committees might find a role in future versions of the protocol, so the minimum target size is worth preserving.

### See also

In his survey article, Paths toward single-slot finality, Vitalik considers what it would take to introduce a single "super-committee" at each slot to replace the existing beacon committees. The super-committee would be a large enough subset of the whole validator set to achieve a satisfactorily secure level of finality within a single (extended, 16 second or longer) slot.

## Aggregator Selection

- In each committee, a subset of validators is selected to perform aggregation of the committee's messages. This improves scaling.
- Selection of aggregators is probabilistic based on BLS signatures.
- This selection method preserves both secrecy and easy verifiability of the identity of the aggregators.

### Introduction

In both beacon committees and sync committees validators create and sign their own votes (`Attestation`s and `SyncCommitteeMessage`s respectively). These votes must be aggregated into a much smaller number of aggregate signed votes, ideally into a single aggregate signature over a single vote, before being included in beacon blocks.

The goals of aggregation are three-fold: to reduce the signature verification load on the next block proposer, to reduce the network load on the global gossip channel, and to reduce the amount of block space required to store the signatures.

In the current beacon chain design, voting is done in committees with the goal of getting a majority of committee members to sign off on the same vote, although in practice there might be a number of different votes depending on the network views of the individual committee members. In any case, members of different committees are signing different data that cannot be aggregated across committees.

The process of aggregation is as follows:

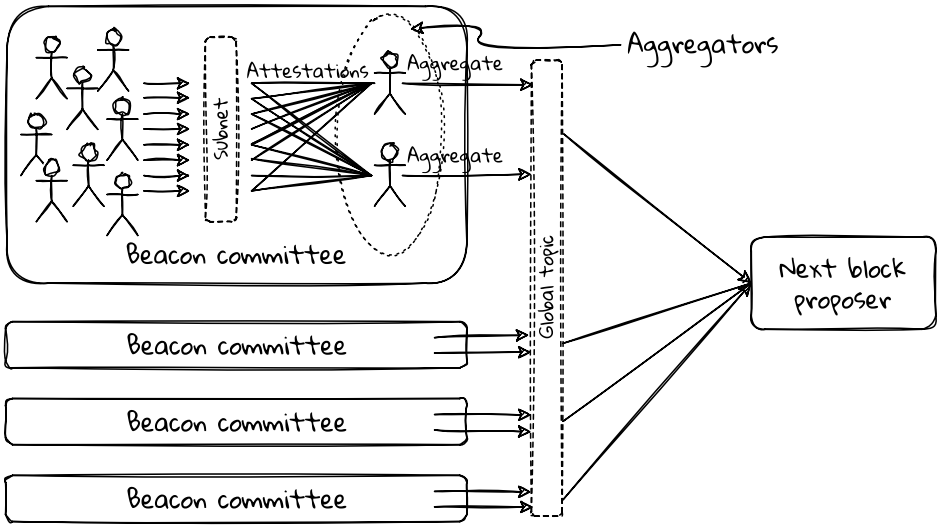
1. Committee members sign their votes (`Attestation`s or `SyncCommitteeMessage`s depending on which type of committee we are considering) and broadcast them to a peer-to-peer subnet that the whole committee is subscribed to.
2. A subset of the committee is selected to be aggregators for that committee.
3. The aggregators listen on the subnet for votes, then aggregate all the votes they receive that agree with their own view of the network into a single aggregate vote (aggregate `Attestation` or `SyncCommitteeContribution`).
4. Each aggregator wraps its aggregate vote with a proof that it was indeed an aggregator for that committee, and it signs the resulting data (`SignedAggregateAndProof` or `SignedContributionAndProof`)
5. Finally the aggregator broadcasts its aggregated vote and proof to a global channel to be received by the next block proposer.

This section is concerned with steps 2 and 4: how the aggregators are selected for duty, and how they prove that they were indeed selected.

### Aggregator selection desiderata

Aggregator selection has been designed with three properties in mind.

First, the size of the resulting aggregator set. With very high probability we want a small, non-empty subset of the committee to be selected. It doesn't matter too much if our set of aggregators is slightly on




*Within a beacon committee, all members send their individual attestations to a gossip subnet. Aggregators are a chosen subset of the committee who listen to the subnet and aggregate the attestations they receive. The aggregators broadcast their aggregates to the global channel for the next block proposer to pick up.*

the large side, but we really want to avoid having no aggregators. Bearing in mind that there's a chance of validators being down or malicious, selecting only one or two aggregators is also risky.

Second, secrecy. We'd prefer that nobody be able to calculate who the aggregators are until after they have broadcast their aggregations. This helps to avoid denial of service (DoS) attacks. Disrupting consensus would be much simpler via a network DoS attack against a small number of aggregators than against a whole committee. The secrecy property prevents this.

Third, verifiability. We want it to be easy to verify a claim that a particular validator was selected to be an aggregator. The rationale for this is explained in the p2p spec. Basically, without verifiability it would be a good strategy for *all* the validators in the committee to make and broadcast aggregate attestations to ensure that at least one aggregate includes their own attestation. This would destroy the benefits of the whole aggregator scheme.

### Aggregator selection details

The current aggregation strategy was introduced in PR 1440 and is described in the Honest Validator specs for beacon committees and sync committees.

It turns out that we can straightforwardly satisfy our three desirable properties of size, secrecy, and verifiability using BLS signatures. Each validator in the committee generates a signature over the current slot number using its secret signing key. If that signature modulo a given number is zero then it is an aggregator, otherwise it is not an aggregator.

The following are the spec functions for determining which validators are the aggregators in beacon committees.

```
def get_slot_signature(state: BeaconState, slot: Slot, privkey: int) -> BLSSignature:
    domain = get_domain(state, DOMAIN_SELECTION_PROOF, compute_epoch_at_slot(slot))
    signing_root = compute_signing_root(slot, domain)
    return bls.Sign(privkey, signing_root)


def is_aggregator(state: BeaconState, slot: Slot, index: CommitteeIndex, slot_signature: BLSSignature) ->
                                        ↪ bool:
    committee = get_beacon_committee(state, slot, index)
    modulo = max(1, len(committee) // TARGET_AGGREGATORS_PER_COMMITTEE)
    return bytes_to_uint64(hash(slot_signature)[0:8]) % modulo == 0
```

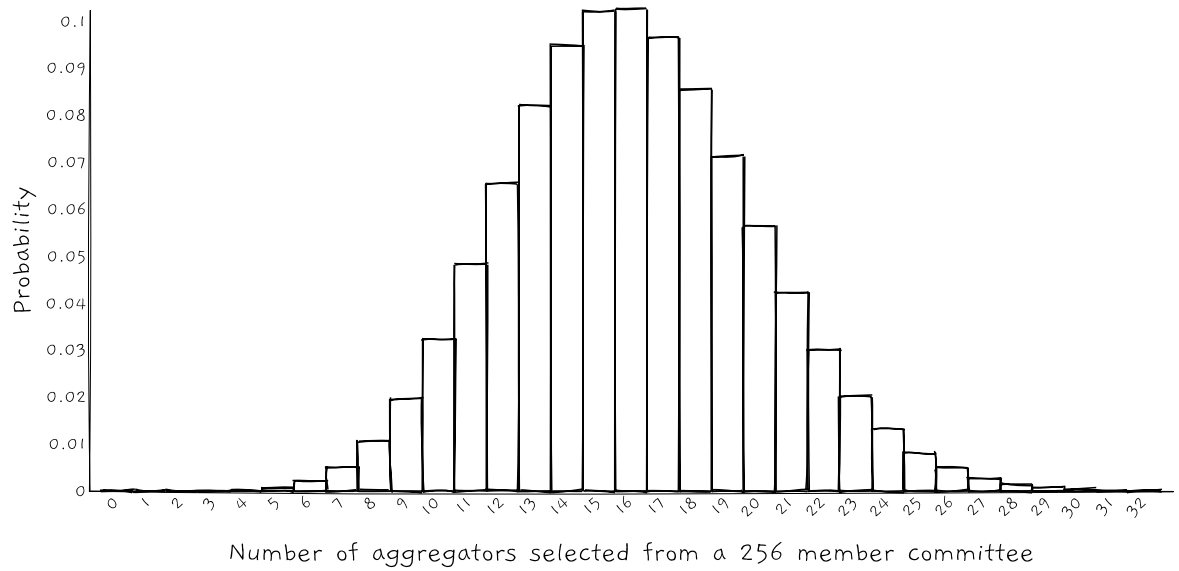
This approach provides secrecy since it relies on the validator's secret key: no-one else can determine whether or not I am an aggregator until after I have published the proof. And it provides verifiability since, once the proof is published, it is easy to check the validity of the signature using the validator's public key.

What about the size criterion?

### Beacon committee aggregators

Assuming that BLS signatures are uniformly random, then in a committee of size $N$ each validator will have a probability of being selected of `TARGET_AGGREGATORS_PER_COMMITTEE / N` (ignoring the integer arithmetic). So in expectation we will have `TARGET_AGGREGATORS_PER_COMMITTEE` (16) aggregators per committee.

The probability of having zero aggregators is $(1 - \frac{16}{N})^N$. For the minimum target committee size of $N = 128$ this is 1 in 26 million, and for the maximum committee size of $N = 2048$, 1 in 9.5 million. So we would expect to see a beacon committee with no aggregators about once every 13,000 epochs (8 weeks) in the former case and once every 5000 epochs (3 weeks) in the latter. Each committee comprises only a fraction 1/2048 of the total validator set, so occasionally having no aggregator is insignificant for the protocol, but it is unfortunate for those in that committee who will most likely not have their attestations included in a block as a result.




*The probability of having k aggregators in a beacon committee of size 256. The expected number is 16.*

### Sync committee aggregators

Sync committees operate similarly. Each committee has 512 members that are divided across four independent subnets. The target is to have 16 aggregators per subnet as above, with the aggregators changing in each slot.

The `TARGET_AGGREGATORS_PER_SYNC_SUBCOMMITTEE` value was increased from 4 to 16 ahead of the implementation of sync committees. This was based on an analysis showing that, by targeting only four aggregators, there would be an unacceptably high chance of having no aggregators on a sync committee subnet.

### Incentivisation

Aggregators are not directly incentivised by the protocol: there are no explicit rewards or penalties for performing or not performing aggregation duties.

However, there are implicit incentives. For one, if I produce a high quality aggregate signature it helps to ensure that my own signature is included in a block (there's a chance that someone else's aggregate may not include my signature). For another, since overall attestation rewards scale in proportion to participation (inclusion of attestations in blocks), aggregators benefit alongside all the other validators from slightly higher rewards when they make high quality aggregates that include many votes.

#### See also

A possible approach to implementing distributed validator technology (DVT) is for the multiple validators representing a single validator to operate independently, alongside a middleware that combines their signed attestations. This works because BLS signatures are additive: each validator has part of the key, and the signed attestations can be combined with a threshold signature scheme into a signature from the full key. While this works in principle, in practice the current interface between the beacon node and the validator client makes it difficult for these validators to determine whether they have been selected to be aggregators or not. Oisín Kyne's ethresear.ch article explores this problem and proposes a solution.

## SSZ: Simple Serialize

| First cut | ✓ | Revision | TODO |
|---|---|---|---|

- The beacon chain uses a novel serialisation method called Simple Serialize (SSZ).
- After much debate we chose to use SSZ for both consensus and communication.
- SSZ is not self-describing; you need to know in advance what you are deserialising.
- An offset scheme allows fast access to subsets of the data.
- SSZ plays nicely with Merkleization and generalised indices in Merkle proofs.

#### Introduction

Serialisation is the process of taking structured information (in our case, a data structure) and transforming it into a representation that can be stored or transmitted.

A cooking recipe is a kind of serialisation. I can write down a method for cooking something in such a way that you and others can recreate the method to cook the same thing. The recipe can be written in a book, appear online, even be spoken and memorised – this is serialisation. Using the recipe to cook something is deserialisation.

Serialisation is used for three main purposes on the beacon chain.

1. Consensus: if you and I each have information in a data structure, such as the beacon state, how can we know if our data structures are the same or not? Serialisation allows us to answer this question, as long as all clients use the same method. Note that this is also bound up with Merkleization.
2. Peer-to-peer communication: we need to exchange data structures over the Internet, such as attestations and blocks. We can't transmit structured data as-is, it must be serialised for transmission and deserialised at the other end. All clients must use the same p2p serialisation, but it doesn't need to be the same as the consensus serialisation.
3. Similarly, data structures need to be serialised for users accessing a beacon node's API. Clients are

free to choose their own API serialisation. For example, the Prysm client has an API that uses Protocol Buffers (which is being deprecated now that we have agreed a common API format that uses both SSZ and JSON).

In addition, data must be serialised before being written to disk. Each client is free to do this internally however they wish.

Ethereum 2.0 uses a bespoke serialisation scheme called Simple Serialize, or more commonly just "SSZ"[32], for all of these purposes.

### History

It seems like we spent months over the end of 2018 and the start of 2019 talking about serialisation, and the story below is highly simplified. But I think it's worth recording some of the considerations and design decisions.

Ethereum 1 has always used a serialisation format called RLP (recursive length prefix). This was deemed unsuitable for Ethereum 2, largely because it is regarded as overly complex.[33]

So, we had the freedom to choose a new serialisation protocol. What kind of decision points did we consider?

### Serialisation for consensus

Starting with serialisation in the consensus protocol, the first big question was whether to adopt an existing off-the-shelf protocol or to roll our own.

One major issue with many existing schemes is that they do not guarantee that the serialisation is deterministic: they sometimes re-order fields in unpredictable ways. This makes them totally unsuitable for consensus; the same data must result in the same output every time.

A more general concern was around using third-party libraries in a consensus-critical situation. Back in 2014, Vitalik wrote a justification, titled Why not use X?, of Ethereum implementing its own technology (such as RLP) for so many things. Here's an excerpt:

> One of our core principles in Ethereum is simplicity; the protocol should be as simple as possible, and the protocol should not contain any black boxes. Every single feature of every single sub-protocol should be precisely 100% documented on the whitepaper or wiki, and implemented using that as a specification.

Certainly, with respect to serialisation, some third-party libraries are far more generic than we need, which can lead to issues. Others don't map nicely to the data types that we want to use.

In view of these concerns momentum was in favour of adopting a bespoke, tightly specified serialisation method. It was the development of Merkleization on top of SSZ that cemented this, making SSZ (in some form) the clear leader for consensus serialisation.

### Serialisation for communications

That decision made, the next big question was whether to use the same scheme for both consensus serialisation and peer-to-peer communications serialisation (the "wire-protocol"). This was finely balanced, and good arguments were made in favour of using Protocol Buffers for p2p communication and SSZ for consensus.

Discussion around this was extensive (see the references below), but we eventually decided to use SSZ for p2p communications.

The factors that tipped the balance in favour of SSZ for communications were (1) a desire to maintain only one serialisation library, and (2) some possible performance benefit.

---

[32] Thus enshrining that ugly "z" in the full name, and the ghastly "ess-ess-zee" pronunciation.

[33] Vitalik, "As the inventor of RLP, I'm inclined to prefer SSZ", and again, "RLP honestly sucks" (with some explanation as to why!).

On the first of these, there is a bias in Ethereum 2 to favour "simplicity over efficiency". Maintaining two serialisation libraries is arguably more overhead than any potential gain from using different ones. Having said that, RLP is still used in Eth2's discovery layer (since it is shared Eth1), so this argument loses some of its force.

On the second, when we receive an object over the wire, often the first thing we will want to do is to serialise it to calculate its data root for consensus. If we receive it already serialised in the right format then it saves a deserialise/reserialise round trip.

SSZ does not make any effort to compact or compress the serialised data, and there were concerns that this might make it inefficient for the wire transfer protocol. These concerns were alleviated by adding Snappy compression on the wire, as is already done in Ethereum 1.

### SSZ development

SSZ is based on Ethereum's smart contract ABI, but with 4-byte position and size records rather than 32-byte, and different basic data types. It will immediately feel familiar to anyone who has fiddled with that. The rudiments of SSZ were laid down by Vitalik in August 2017.

The initial, more developed, spec for SSZ was merged into the beacon chain repository in October 2018, with the `Container` type being added a month later.

A big step forward in the utility of SSZ, and what established it as the serialisation protocol of choice for consensus, was the development of Merkleization (also known as tree hashing), first discussed in October 2018 and adopted into the spec in November.

Also in November 2018 we agreed to switch the byte ordering for integer types from big-endian to little-endian at the request of the Nimbus team. This means that the 32-bit number representing 66 decimal would is now serialised as `0x42000000` rather than `0x00000042`. The main motivation for the change was to map better to byte-ordering in typical microprocessors.

April 2019 saw a major change to SSZ with the adoption of offsets. This came from a scheme, Simple Offset Serialisation, previously proposed by Péter Szilágyi. The idea is to split the objects we are serialising according to whether they are fixed length or variable length. The serialisation then has two sections. The first section contains both actual serialisations of any fixed length objects, and pointers (offsets) to the serialisations of any variable length objects. The second section contains the serialisations of the variable length objects. The motivation for this is to allow fast access to arbitrary parts of the serialised data without having to deserialise the whole structure.

There was one final substantial re-work of the SSZ spec in June 2019 in which SSZ lists were required to have a maximum length specified, and bitlist and bitvector types were added.

### Overview

The specification of SSZ is maintained in the main consensus specs repo, and that's the place to go for all the details. I will only be presenting an introductory overview here, with a few examples.

The ultimate goal of SSZ is to be able to represent complex internal data structures such as the BeaconState as strings of bytes.

The formal properties that we require for SSZ to be useful for both consensus and communications areas defined in the SSZ formal verification exercise. Given objects $O_1$ and $O_2$, both of type $T$, we require that SSZ be

1. involutive: $\texttt{deserialise}\langle T\rangle(\texttt{serialise}\langle T\rangle(O_1)) = O_1$ (required for communications), and
2. injective: $\texttt{serialise}\langle T\rangle(O_1) = \texttt{serialise}\langle T\rangle(O_2)$ implies that $O_1 = O_2$ (required for consensus).

The first property says that when we serialise an object of a certain type then deserialise the result, we end up with an object identical to the one we started with. This is essential for the communications protocol.

The second says that if we serialise two objects of the same type and get the same result then the two objects are identical. Equivalently, if we have two different objects of the same type then their serialisations will differ. This is essential for the consensus protocol.

Beyond those basic functional requirements, other goals for SSZ are to be (relatively) simple, to create (fairly) compact serialisations, and to be compatible with Merkleization. It is also useful to be able to quickly access specific bits of data within the serialisation without deserialising the entire object. The adoption of offsets into SSZ improved its performance in that respect.

Unlike RLP, SSZ is not self-describing. You can decode RLP data into a structured object without knowing in advance what that object looks like. This is not the case for SSZ: you must know in advance exactly what you are deserialising. In practice this has not been a problem for Eth2: we always know in advance what class of object a particular deserialised blob of data corresponds to. A consequence of this is that, while in RLP two objects of different types cannot serialise to the same output, in SSZ they can. We'll see an example of this shortly.

### Specification

I don't plan to go into every last detail of SSZ – that's what the specification is for – rather, we'll take a general overview and then dive into a worked example.

The building blocks of SSZ are its basic types and its composite types.

### Basic types

SSZ's basic types are very simple and limited, comprising only the following two classes.

- Unsigned integers: a `uintN` is an `N`-bit unsigned integer, where `N` can be 8, 16, 32, 64, 128 or 256.
- Booleans: a `boolean` is either `True` or `False`.

The serialisation of basic types lives up to the "simple" name:

- `uintN` types are encoded as the little-endian representation in `N/8` bytes. For example, the decimal number 12345 (`0x3039` in hexadecimal) as a `uint16` type is serialised as `0x3930` (two bytes). The same number as a `uint32` type is serialised as `0x39300000` (four bytes).
- `boolean` types are always one byte and serialised as `0x01` for true and `0x00` for false.

I have embedded some examples in the following descriptions. You can run them yourself if you set up the Eth2 spec as per the instructions in the Appendices. The examples can be run via the Python REPL or by putting the commands in a file (I show both approaches).

```
>>> from eth2spec.utils.ssz.ssz_typing import uint64, boolean
>>> uint64(0x0123456789abcdef).encode_bytes().hex()
'efcdab8967452301'
>>> boolean(True).encode_bytes().hex()
'01'
>>> boolean(False).encode_bytes().hex()
'00'
```

### Composite types

Composite types hold combinations of or multiples of smaller types. The spec defines the following composite types: vectors, lists, bitvectors, bitlists, unions, and containers. I will skip unions in the following as they are not currently used in Ethereum 2.

### Vectors

A vector is an ordered fixed-length homogeneous collection with exactly `N` values. "Homogeneous" means that all the elements of a vector must be of the same type, but they do not need to be of the same size. For example, we could have a vector containing lists that each have different numbers of elements.

In the SSZ spec a vector is denoted by `Vector[type, N]`. For example `Vector[uint8, 32]` is a 32 element list of `uint8` types (bytes). The `type` can be anything, including other vectors or even containers.

Vectors provide a simple example of needing to know what kind of object you are deserialising before you attempt it. In the following example, the same string of bytes encodes both a four element set of

two-byte integers, and an eight element set of one-byte integers. When we deserialise this we need to know which of these (or many other possibilities) we are expecting to get.

```
>>> from eth2spec.utils.ssz.ssz_typing import uint8, uint16, Vector
>>> Vector[uint16, 4](1, 2, 3, 4).encode_bytes().hex()
'0100020003000400'
>>> Vector[uint8, 8](1, 0, 2, 0, 3, 0, 4, 0).encode_bytes().hex()
'0100020003000400'
```

Fun fact: in early versions of the SSZ spec, vectors were called tuples.

### Lists

A list is an ordered variable-length homogeneous collection with a maximum of `N` values.

In the SSZ spec a list is denoted by `List[type, N]`. For example, `List[uint64, 100]` is a list containing anywhere between zero and one hundred `uint64` types.

The maximum length parameter, `N`, on lists is not used in serialisation or deserialisation. It is used, however, in Merkleization, and in particular enables generalised indices in Merkle proof generation.

Both vectors and lists have the same serialisation when they are treated as stand-alone objects:

```
>>> from eth2spec.utils.ssz.ssz_typing import uint8, List, Vector
>>> List[uint8, 100](1, 2, 3).encode_bytes().hex()
'010203'
>>> Vector[uint8, 3](1, 2, 3).encode_bytes().hex()
'010203'
```

So why not use lists everywhere? Since lists are variable sized objects in SSZ they are encoded differently from fixed sized vectors when contained within another object, so there is a small overhead. The container `Foo` holding the variable sized list is encoded with an extra four byte offset at the start. We'll see why a bit later.

```
>>> from eth2spec.utils.ssz.ssz_typing import uint8, Vector, List, Container
>>> class Foo(Container):
...     x: List[uint8, 3]
>>> class Bar(Container):
...     x: Vector[uint8, 3]
>>> Foo(x = [1, 2, 3]).encode_bytes().hex()
'04000000010203'
>>> Bar(x = [1, 2, 3]).encode_bytes().hex()
'010203'
```

### Bitvectors

A bitvector is an ordered fixed-length collection of `boolean` values with `N` bits. In the SSZ spec, a bitvector is denoted by `Bitvector[N]`.

It is not obvious from the spec, but bitvectors use little-endian bit format:

```
>>> from eth2spec.utils.ssz.ssz_typing import Bitvector
>>> Bitvector[8](0,0,0,0,0,0,0,1).encode_bytes().hex()
'80'
```

Bitvectors are encoded into the minimum necessary number of whole bytes (`N // 8`) and padded with zeroes in the high bits if `N` is not a multiple of 8.

As noted in the spec, functionally we could use either `Vector[boolean, N]` or `Bitvector[N]` to represent a list of bits. However, the latter will have a serialisation up to eight times shorter in practice since the former will use a whole byte per bit.

```
>>> from eth2spec.utils.ssz.ssz_typing import Vector, Bitvector, boolean
>>> Bitvector[5](1,0,1,0,1).encode_bytes().hex()
'15'
>>> Vector[boolean,5](1,0,1,0,1).encode_bytes().hex()
'0100010001'
```

The same consideration applies for lists and bitlists.

### Bitlists

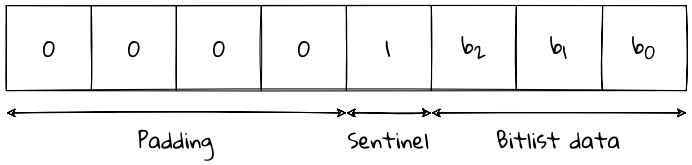
A bitlist is an ordered variable-length collection of `boolean` values with a maximum of `N` bits. In the SSZ spec, a bitlist is denoted by `Bitlist[N]`.

An interesting feature of bitlists[34] is that they use a sentinel bit to indicate the length of the list. The number of whole bytes in the bitlist is easily derived from the offsets in the serialisation, but that doesn't give us the precise number of bits. For example, in a naive scheme 13 bits would be serialised into two bytes, so we would only know that the actual list length is somewhere between 9 and 16 bits.

To resolve this problem, bitlist serialisation adds an extra `1` bit at the end of the list (which becomes the highest-order bit in the little-endian encoding). The exact length of the bitlist can then be found by ignoring any consecutive high-order zero bits and then stripping off the single sentinel bit.

As an example, this bitlist with three elements is encoded into a single byte. To deserialise this, we take the total length in bits (eight), skip the four high-order zero bits, skip the sentinel bit, and then our list comprises the remaining three bits. Equivalently, the bitlist length is the index of the highest `1` bit in the serialisation.

```
>>> from eth2spec.utils.ssz.ssz_typing import Bitlist
>>> Bitlist[100](0,0,0).encode_bytes().hex()
'08'
```




*The sentinel bit indicates the end of the bitlist. All bits beyond the sentinel are zero.*

As a consequence of the sentinel, we require an extra byte to serialise a bitlist if its actual length is a multiple of eight (irrespective of the maximum length). This is not the case for a bitvector.

```
>>> Bitlist[8](0,0,0,0,0,0,0,0).encode_bytes().hex()
'0001'
>>> Bitvector[8](0,0,0,0,0,0,0,0).encode_bytes().hex()
'00'
```

### Containers

A container is an ordered heterogeneous collection of values. Basically, a container can contain any arbitrary mix of types, including containers.

We define containers using Python's `dataclass` notation with key–type pairs. For example, this is a `Deposit` container. In the following examples I have indicated the underlying types in the appended comments.

```
class Deposit(Container):
    proof: Vector[Bytes32, DEPOSIT_CONTRACT_TREE_DEPTH + 1] # Vector[Vector[uint8, 32], N]
    data: DepositData
```

The `Deposit` container contains a `DepositData` container which is defined as follows.

[34] Though not entirely uncontroversial. Basically, if the application layer already knows what length of bitlist it expects – which it generally does in Eth2, since although committee sizes change, the sizes are known – then we could in principle dispense with the sentinel bit.

```
class DepositData(Container):
    pubkey: BLSPubkey               # Bytes48 / Vector[uint8, 48]
    withdrawal_credentials: Bytes32  # Vector[uint8, 32]
    amount: Gwei                    # uint64
    signature: BLSSignature         # Bytes96 / Vector[uint8, 96]
```

We'll see how containers are serialised in the worked example, below.

### Fixed and variable size types

SSZ distinguishes between fixed size and variable size types, and treats them differently when they are contained within other types.

- Variable size types are lists, bitlists, and any type that contains a variable size type.
- Everything else is fixed size.

This distinction is important when we serialise a compound type. The serialised output is created in two parts, as follows.

1. The serialisation of fixed length types, along with 32-bit offsets to any variable length types.
2. The serialisation of any variable length types.

This split between a fixed length part and a variable length part came about as a result of the offset encoding described earlier: it allows fast access to specific fields within a serialised data structure without needing to deserialise the whole thing.

As an example, consider the following container. It has a single fixed length `uint8` type, followed by a variable length `List[uint8,10]` type, followed again by a fixed length `uint8`.

```
>>> from eth2spec.utils.ssz.ssz_typing import uint8, List, Container
>>> class Baz(Container):
...     x: uint8
...     y: List[uint8, 10]
...     z: uint8
>>> Baz(x = 1, y = [2, 3], z = 4).encode_bytes().hex()
'0106000000040203'
```

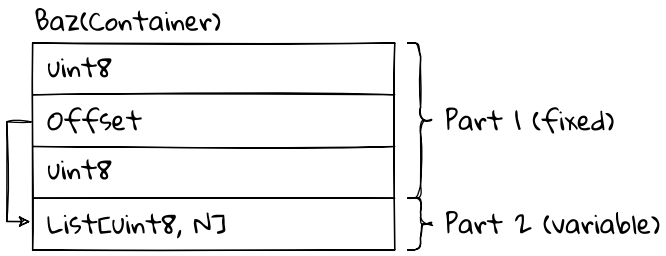
We see that the serialisation contains an unexpected `06` byte and some zero bytes. To see where they come from I'll break down the output as follows, where the first column is the byte number in the serialised string.

```
Start of Part 1 (fixed size elements)
00 01       - The serialisation of x = uint8(1)
01 06000000 - A 32-bit offset to byte 6 (in little-endian format),
              the start of the serialisation of y
05 04       - The serialisation of z = uint8(4)

Start of Part 2 (variable size elements)
06 0203     - The serialisation of y = List[uint8, N]([2, 3])
```

In Part 1, instead of directly encoding the variable size list in place, it is replaced with a pointer (an offset) to its serialisation in Part 2. So, for any container, the size of Part 1 is known and fixed no matter what kinds of variable size types are present. The actual lengths of the variable size objects can be deduced from the offsets in Part 1 and the overall length of the serialisation string.

It's not only containers that use this format, it applies to any type that contains variable size types. Here's a vector whose elements are lists. As an exercise for the reader I'll leave you to decode what's going on here.

```
>>> from eth2spec.utils.ssz.ssz_typing import uint8, List, Vector
>>> Vector[List[uint8,3],4]([1,2],[3,4,5],[],[6]).encode_bytes().hex()
'10000000120000001500000015000000010203040506'
```




*Serialisation of the `Baz` container. Fixed size parts are done first, with an offset specified for the variable size `List` data.*

### Aliases

Just quoting directly from the SSZ spec here for completeness:

> For convenience we alias:
>
> - `bit` to `boolean`
> - `byte` to `uint8` (this is a basic type)
> - `BytesN` and `ByteVector[N]` to `Vector[byte, N]` (this is *not* a basic type)
> - `ByteList[N]` to `List[byte, N]`

In the main beacon chain spec, a bunch of custom types are also defined in terms of the standard SSZ types and aliases. For example, `Slot` is an SSZ `uint64` type, `BLSPubkey` is an SSZ `Bytes48` type, and so on.

### Default values

Finally, each type has a default value. Once again directly from the SSZ spec:

| Type | Default Value |
|---|---|
| `uintN` | `0` |
| `boolean` | `False` |
| `Container` | `[default(type) for type in container]` |
| `Vector[type, N]` | `[default(type)] * N` |
| `Bitvector[N]` | `[False] * N` |
| `List[type, N]` | `[]` |
| `Bitlist[N]` | `[]` |

### Worked example

Let's explore a worked example to gather all of this together. I'd rather use a real example than make up a synthetic object, so we are going to look at the aggregate `IndexedAttestation` that was included in the beacon chain block at slot 3080831, at position 87 within the block. (It would actually have been an `Attestation` object in the block, but those bitlists are fiddly, so we'll look at the equivalent `IndexedAttestation`.)

### The data structures

The `IndexedAttestation` container looks like this.

```
class IndexedAttestation(Container):
    attesting_indices: List[ValidatorIndex, MAX_VALIDATORS_PER_COMMITTEE]
    data: AttestationData
    signature: BLSSignature
```

It contains an `AttestationData` container,

```
class AttestationData(Container):
    slot: Slot
    index: CommitteeIndex
    beacon_block_root: Root
```

### The serialisation

Now we have enough information to build the `IndexedAttestation` object and calculate its SSZ serialisation.

```
from eth2spec.utils.ssz.ssz_typing import *
from eth2spec.altair import mainnet
from eth2spec.altair.mainnet import *

attestation = IndexedAttestation(
    attesting_indices = [33652, 59750, 92360],
    data = AttestationData(
        slot = 3080829,
        index = 9,
        beacon_block_root = '0x4f4250c05956f5c2b87129cf7372f14dd576fc152543bf7042e963196b843fe6',
        source = Checkpoint (
            epoch = 96274,
            root = '0xd24639f2e661bc1adcbe7157280776cf76670fff0fee0691f146ab827f4f1ade'
        ),
        target = Checkpoint(
            epoch = 96275,
            root = '0x9bcd31881817ddeab686f878c8619d664e8bfa4f8948707cba5bc25c8d74915d'
        )
    ),
    signature = '0xaaf504503ff15ae86723c906b4b6bac91ad728e4431aea3be2e8e3acc888d8af'
                + '5dffbbcf53b234ea8e3fde67fbb09120027335ec63cf23f0213cc439e8d1b856'
                + 'c2ddfc1a78ed3326fb9b4fe333af4ad3702159dbf9caeb1a4633b752991ac437'
)

print(attestation.encode_bytes().hex())
```

The resulting serialised blob of data that represents this `IndexedAttestation` object is (in hexadecimal):

```
e40000007d022f0000000000090000000000000004f4250c05956f5c2b87129cf7372f14dd576fc15
2543bf7042e963196b843fe612780100000000009d24639f2e661bc1adcbe7157280776cf76670fff
0fee0691f146ab827f4f1ade13780100000000009bcd31881817ddeab686f878c8619d664e8bfa4f
8948707cba5bc25c8d74915daaf504503ff15ae86723c906b4b6bac91ad728e4431aea3be2e8e3ac
c888d8af5dffbbcf53b234ea8e3fde67fbb09120027335ec63cf23f0213cc439e8d1b856c2ddfc1a
78ed3326fb9b4fe333af4ad3702159dbf9caeb1a4633b752991ac437748300000000000066e90000
00000000c868010000000000
```

This can be transmitted as a string of bytes over the wire and, knowing at the other end that it represents an `IndexedAttestation`, reconstituted into an identical copy.

### The serialisation unpacked

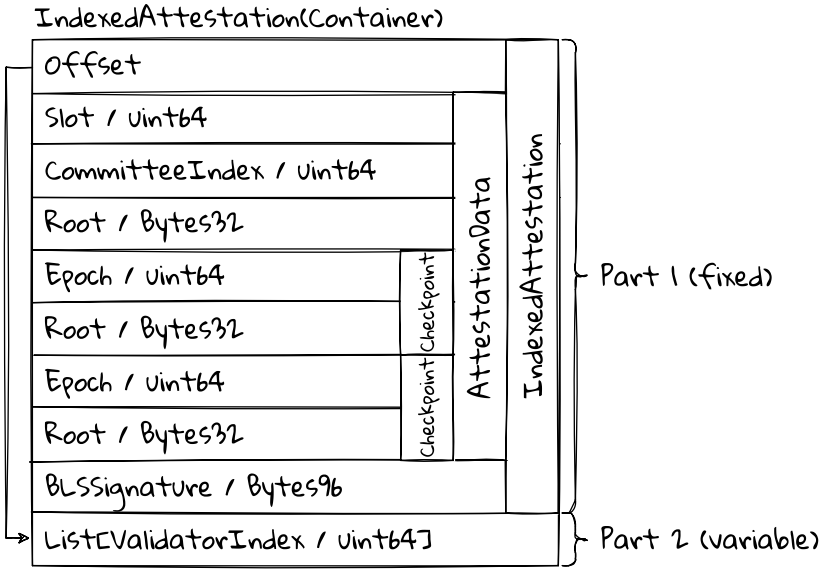
To make sense of this, we'll break down the serialisation into its parts. The first column is the byte-offset from the start of the byte string (in hexadecimal). Before each line I've indicated which part of the data structure it corresponds to, and I've translated the type aliases into their basic underlying SSZ types. Remember that all integer types are little-endian, so `7d022f0000000000` is the hexadecimal number `0x2f027d`, which is 3080829 in decimal (the slot number).

```
Start of Part 1 (fixed size elements)
   4-byte offset to the variable length attestation.attesting_indices starting at 0xe4
00 e4000000

   attestation.data.slot: Slot / uint64
04 7d022f0000000000

   attestation.data.index: CommitteeIndex / uint64
0c 0900000000000000

   attestation.data.beacon_block_root: Root / Bytes32 / Vector[uint8, 32]
14 4f4250c05956f5c2b87129cf7372f14dd576fc152543bf7042e963196b843fe6
```

```
   attestation.data.source.epoch: Epoch / uint64
34 1278010000000000

   attestation.data.source.root: Root / Bytes32 / Vector[uint8, 32]
3c d24639f2e661bc1adcbe7157280776cf76670fff0fee0691f146ab827f4f1ade

   attestation.data.target.epoch: Epoch / uint64
5c 1378010000000000

   attestation.data.target.root: Root / Bytes32 / Vector[uint8, 32]
64 9bcd31881817ddeab686f878c8619d664e8bfa4f8948707cba5bc25c8d74915d

   attestation.signature: BLSSignature / Bytes96 / Vector[uint8, 96]
84 aaf504503ff15ae86723c906b4b6bac91ad728e4431aea3be2e8e3acc888d8af
a4 5dffbbcf53b234ea8e3fde67fbb09120027335ec63cf23f0213cc439e8d1b856
c4 c2ddfc1a78ed3326fb9b4fe333af4ad3702159dbf9caeb1a4633b752991ac437

Start of Part 2 (variable size elements)
   attestation.attesting_indices: List[uint64, MAX_VALIDATORS_PER_COMMITTEE]
e4 7483000000000000066e9000000000000c868010000000000
```

The first thing to notice is that the `attesting_indices` list is variable size, so it is represented in Part 1 by an offset pointing to where the actual data is. In this case, at `0xe4` bytes (228 bytes) from the start of the serialised data. The actual length of the list can be calculated as the length of the whole string (252 bytes) minus 228 bytes (the start of the list) divided by 8 bytes, one per element. Thus we recover our list of three validator indices.

All the remaining items are fixed size, and are encoded in-place, including recursively encoding the fixed size `AttestationData` object, and its fixed size `Checkpoint` children.




*Serialisation of the `IndexedAttestation` container.*

### Multiple variable size objects

It is instructive to see how container with multiple variable size child objects is serialised. For this example we will make an `AttesterSlashing` object that contains two of the above `IndexedAttestation` objects. This is a contrived example; the slashing report is not valid since the contents are duplicates.

An `AttesterSlashing` container is defined as follows,

```
class AttesterSlashing(Container):
```

```
    attestation_1: IndexedAttestation
    attestation_2: IndexedAttestation
```

which we can populate and serialise like this, using our previously defined `IndexedAttestation` object, `attestation`.

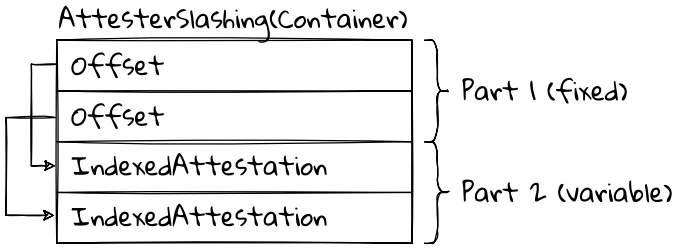
```
slashing = AttesterSlashing(
    attestation_1 = attestation,
    attestation_2 = attestation
)

print(slashing.encode_bytes().hex())
```

From this we get the following serialisation, again shown with the byte-offset within the byte string in the first column.

```
Start of Part 1 (fixed size elements)
0000 08000000
0004 04010000

Start of Part 2 (variable size elements)
0008 e40000007d022...
0104 e40000007d022...
```

This time we have two variable length types, so they are both replaced by offsets pointing to the start of the actual variable length data which appears in Part 2. The length of `attestation_1` is calculated as the difference between the two offsets, and the length of `attestation_2` is calculated as the length from its offset to the end of the string.

Another thing to note is that, since `attestation_1` and `attestation_2` are identical, their serialisations within this compound object are identical, *including* their internal offsets to their own variable length parts. That is, both attestations have variable length data at offset `0xe4` within their own serialisations; the offset is relative to the start of each sub-object's serialisation, not the entire string. This property simplifies recursive serialisation and deserialisation: a given object will have the same serialisation no matter what context it is found in.




*Serialisation of the `AttesterSlashing` container.*

### See also

The SSZ specification is the authoritative source. There is also a curated list of SSZ implementations.

The historical discussion threads around whether to use SSZ for both consensus and p2p serialisation or not are a goldmine of insight and wisdom.

- Possibly the first substantial discussion around which serialisation scheme to adopt. It covers various alternatives, touches on the p2p vs. consensus issues, and rehearses some of the desirable properties.
- An early discussion of SSZ went over some of the issues and led into the discussion below.
- Proposal to use SSZ for consensus only.

- Piper Merriam's Everything You Never Wanted To Know About Serialization remains a good summary of many of the considerations.

Other SSZ resources:

- SSZ encoding diagrams by Protolambda.
- Formal verification of the SSZ specification: Notes and Code.
- An excellent SSZ explainer by Raul Jordan with a deep dive into implementing it in Golang. (Note that the specific library referenced in the article has now been deprecated in favour of fastssz.)
- An interactive SSZ serialiser/deserialiser by ChainSafe with all the containers for Phase 0 and Altair available to play with. On the "Deserialize" tab you can paste the data from the `IndexedAttestation` above and verify that it deserialises correctly (you'll need to remove line breaks).

## Hash Tree Roots and Merkleization

| First cut | ✓ | Revision | TODO |
|---|---|---|---|

- A hash tree root provides a succinct cryptographic digest of an SSZ data structure.
- Calculating the hash tree root involves recursively Merkleizing the data structure.
- Merkleization is tightly coupled to SSZ and is defined in the same spec.
- The use of hash tree roots enables large parts of the beacon state to be cached, making it practical to operate with a monolithic beacon state.
- Eth2's Merkleization approach facilitates generalised indices and Merkle proofs which are important for light clients.

### Introduction

While discussing SSZ, I asserted that serialisation is important for consensus without going into the details. In this section we will unpack that and take a deep dive into how Ethereum 2 nodes know that they share a view of the world.

Let's say that you and I want to compare our beacon states to see if we have an identical view of the state of the chain. One way we could do this is by serialising our respective beacon states and sending them to each other. We could then compare them byte-by-byte to check that they match. The problem with this is that the serialised beacon state at the time of writing is over 41MB in size and takes several seconds to transmit over the Internet. This is completely impractical for a global consensus protocol.

What we need is a *digest* of the state: a brief summary that is enough to determine with a very high degree of confidence whether you and I have the same state, or whether they differ. The digest must also have the property that no-one can fake it. That is, you can't convince me that you have the same state as I do while actually having a different state.

Thankfully, such digests exist in the form of cryptographic hash functions. These take a (potentially) large amount of input data and mangle it up into a small number of bytes, typically 32, that for all practical purposes uniquely fingerprint the data.

Armed with such a hash function[35] we can improve on the previous idea. You and I separately serialise

[35] See the Annotated Spec for the saga of Eth2's hash function, and how we ended up with SHA256.

our beacon states and then hash (apply the hash function to) the resulting strings. This is much faster than sending all the data over the network. Now we only need to exchange and compare our very short 32 byte hashes. If they match then we have the same state; if they don't match then our states differ.

This process is very common, and was an early candidate for consensus purposes in Ethereum 2, though it was apparent fairly early that there might be better ways.

A problem with this approach is that, if you modify any part of the state – even a single bit – then you need to recalculate the hash of the entire serialised state. This is potentially a huge overhead. It was dealt with in early versions of the spec by splitting the beacon state into two parts: a slowly changing "crystallised" state that would rarely need re-hashing, and a smaller fast changing "active" state. However, this division of the state was a bit arbitrary and began to compromise some other parts of the design.

Ultimately, the split state approach was abandoned in favour of a method called "tree hashing", which is built on a technique called Merkleization[36]. The remainder of this section explores this approach.

Tree hashing brings two significant advantages over other methods of creating a beacon state digest.

The first advantage is performance. On the face of it, tree hashing is quite inefficient since it requires hashing around double the amount of data to calculate the digest (the root) of a structure compared with the other method of simply hashing the entire serialisation. However, the way that hash trees are constructed in Ethereum 2 allows us to cache the roots of entire subtrees that have not changed. So, for example, by design the list of validator records in the state does not change frequently. As a result we can cache the hash tree root of the list and do not need to recalculate it every time we recalculate the entire beacon state root. Overall this feature results in a huge reduction in the total amount of hashing required to calculate state roots, and is an important part of making the beacon chain protocol usable in practice.

The second advantage is light-client support. Indeed, the original motivation for implementing tree hashing was only about supporting light clients. Tree hashing enables efficient Merkle proofs that allow subsets of the beacon state to be provided to light clients. As long as a light client has the hash tree root by some means it can use the proofs to verify that the provided data is correct.

We will first recap Merkle trees, then extend them to Merkleization, and finally look at the construction of the hash tree root, which is our ultimate goal.

### Terminology

The SSZ specification uses the term "Merkleization" to refer to both

- the operation of finding the root of a Merkle tree given its leaves, and
- the operation of finding the hash tree root of an SSZ object.

For didactic purposes I've chosen to distinguish between these more precisely. In the following sections I'll be calling the first "Merkleization", and the second "calculating a hash tree root".

With these definitions, calculating the hash tree root of an SSZ object *uses* Merkleization, potentially multiple steps of it, but also involves other steps such packing, chunking, and length mix-ins. Moreover, Merkleization always works with full binary trees (the number of leaves is a power of two), whereas hash tree roots can be derived from much more complex binary tree structures.

### Merkle Trees

To understand Merkleization we first need to understand Merkle trees. These are not at all new, and date back to the 1970s.

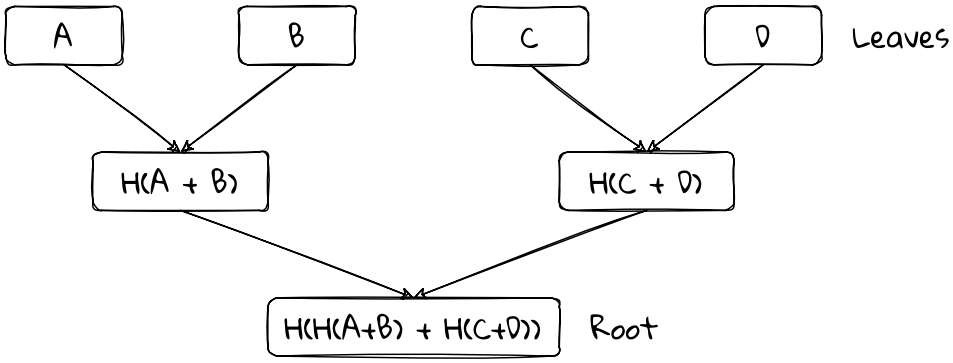
The idea is that we have a set of "leaves", which is our data, and we iteratively reduce those leaves down to a single, short root via hashing. This reduction is done by hashing the leaves in pairs to make a "parent" node. We repeat the process on the parent nodes to make grand-parent nodes, and so on

[36]The name Merkleization derives from Merkle trees, which in turn are named for the computer scientist Ralph Merkle.

I believe the noun "Merkleization", though, is ours. I've adopted the majority preferred spelling, which is also the version that made it into the SSZ spec. The ugly version won despite my best efforts.

to build a binary tree structure that culminates in a single ancestral root. In Merkleization we will be dealing only with structures that have a power of two number of leaves, so we have a full binary tree.

In the following diagram, the leaves are our four blobs of data, $A$, $B$, $C$, and $D$. These can be any string of data, though in Merkleization they will be 32 byte "chunks". The function $H$ is our hash function, and the operator $+$ concatenates strings. So $H(A+B)$ is the hash of the concatenation of strings $A$ and $B$[37].




*Example of a Merkle tree.*

In the Eth2 implementation, each box in the diagram is a 32-byte string of data: either a 32-byte leaf, or the 32-byte output of the hash function. Thus we obtain the 32-byte root of the tree, which is a "digest" of the data represented by the leaves. The root uniquely represents the data in the leaves; any change in the leaves leads to a different root.

Here's the same thing again on the Python REPL, assigning leaf values as $A = 1$, $B = 2$, $C = 3$ and $D = 4$. We construct the root of the tree starting from the leaves and descending through its levels until reaching the root, $H(H(A + B) + H(C + D))$. Note that all the leaf values are padded to 32-bytes and are little-endian (as per their SSZ serialisation).

```
>>> from eth2spec.utils.ssz.ssz_typing import uint256
>>> from eth2spec.utils.hash_function import hash
>>> a = uint256(1).to_bytes(length = 32, byteorder='little')
>>> b = uint256(2).to_bytes(length = 32, byteorder='little')
>>> c = uint256(3).to_bytes(length = 32, byteorder='little')
>>> d = uint256(4).to_bytes(length = 32, byteorder='little')
>>> ab = hash(a + b)
>>> cd = hash(c + d)
>>> abcd = hash(ab + cd)
>>> abcd.hex()
'bfe3c665d2e561f13b30606c580cb703b2041287e212ade110f0bfd8563e21bb'
```

Merkle tree constructions are a fairly common way to calculate a digest of a bunch of data, such as a blockchain state. Ethereum 1 uses a more sophisticated version of this called a hexary Merkle–Patricia trie (in Eth1 it's a "trie" not a "tree" for complicated reasons), though there are proposals to simplify that.

An extremely useful feature of Merkle trees is that it is quite efficient to construct inclusion proofs using them. This is critical functionality for light clients, and we will discuss it in depth when we look at Merkle proofs.

### Merkleization

The normal way to implement a Merkle tree is to store the entire tree structure in memory or on disk, including all the intermediate levels between the leaves and the root. As leaves are updated the affected

[37] For some reason, in computer science, trees are traditionally depicted the other way up. Call me eccentric, but I like my trees to have their leaves at the top and their roots at the bottom.

nodes in the tree are updated: changing $A$ means updating $H(A+B)$ and then the root, everything else is unchanged.

The difference with Merkleization is that the Merkle tree is computed on-the-fly from the given leaves. We can pick up where we left off from the last REPL session as follows.

```
>>> from eth2spec.utils.merkle_minimal import merkleize_chunks
>>> merkleize_chunks([a, b, c, d]).hex()
'bfe3c665d2e561f13b30606c580cb703b2041287e212ade110f0bfd8563e21bb'
```

The Merkleization function (called `merkleize()` in the SSZ spec, and `merkleize_chunks()` in the executable spec) takes a list of 32-byte chunks and returns the root of the tree for which those chunks are the leaves.

The list of chunks passed to `merkleize_chunks()` can be any length, but will be padded with zero chunks so that the total number of chunks is rounded up to the next whole power of two, such that we conceptually have a full binary tree. Thus a list of three chunks gets implicitly padded with an extra zero chunk:

```
>>> z = bytearray(32)
>>> merkleize_chunks([a, b, c]).hex()
'66c419026fee8793be7fd0011b9db46b98a79f9c9b640e25317865c358f442db'
>>> merkleize_chunks([a, b, c, z]).hex()
'66c419026fee8793be7fd0011b9db46b98a79f9c9b640e25317865c358f442db'
```

A larger tree width can be provided as a parameter to `merkleize_chunks()`, and the list will be padded with zero chunks accordingly. This capability is used when dealing with lists and bitlists.

```
>>> merkleize_chunks([a]).hex()
'0100000000000000000000000000000000000000000000000000000000000000'
>>> merkleize_chunks([a], 4).hex()
'553c8ccfd20bb4db224b1ae47359e9968a5c8098c15d8bf728b19e55749c773b'
>>> merkleize_chunks([a, z, z, z]).hex()
'553c8ccfd20bb4db224b1ae47359e9968a5c8098c15d8bf728b19e55749c773b'
```

An implementation can do this zero padding "virtually", and can optimise further by pre-computing the various levels of hashes of zero chunks: $H(0+0)$, $H(H(0+0)+H(0+0))$, and so on. In this way we don't always need to build the whole tree to find the Merkle root.

Note that the Merkleization of a single chunk is always just the chunk itself. This reduces the overall amount of hashing needed.

### The Hash Tree Root

The hash tree root is a generalisation of Merkleization that we can apply to the kind of complex, compound data structures we have in the beacon state. Calculating hash tree roots is tightly connected to the type-scheme of Simple Serialize.

Calculating the hash tree root of an SSZ object is recursive. Given a composite SSZ object, we iteratively move through the layers of its structure until we reach a basic type or collection of basic types that we can pack into chunks and Merkleize directly. Then we move back through the structure using the calculated hash tree roots as chunks themselves.

The process of calculating a hash tree root is defined in the Simple Serialize specification, and that's the place to go for the full details. However, in simplified form (once again ignoring the SSZ union type) there are basically two paths to choose from when finding an object's hash tree root.

- For basic types or collections of basic types (lists and vectors), we just pack and Merkleize directly.
- For containers and collections of composite types, we recursively find the hash tree roots of the contents.

The following two rules are a simplified summary of the first six rules listed in the specification.

1. If `X` is an SSZ basic object, a list or vector of basic objects, or a bitlist or bitvector, then `hash_tree_root(X) = merkleize_chunks(pack(X))`. The `pack()` function returns a list of chunks that can be Merkleized directly.

2. If `X` is an SSZ container, or a vector or list of composite objects, then the hash tree root is calculated recursively, `hash_tree_root(X) = merkleize_chunks([hash_tree_root(x) for x in X])`. The list comprehension is a list of hash tree roots, which is equivalent to a list of chunks.

We'll see plenty of concrete applications of these two rules in the worked example below.

### Packing and Chunking

Merkleization operates on lists of "chunks" which are 32-byte blobs of data. Lists generated by means of step 2 above are already in this form. However, step 1 involves basic objects that require a "packing and chunking" operation prior to Merkleization.

The spec gives the precise rules, but it basically looks like this:

- The object (a basic type, a list/vector of basic types, or a bitlist/bitvector) is serialised via SSZ. The sentinel bit is omitted from the serialisation of bitlist types.
- The serialisation is right-padded with zero bytes up to the next full chunk (32 byte boundary).
- The result is split into a list of 32 byte chunks.
- If necessary, further (virtual) zero chunks will be appended to reach the following total lengths (only lists and bitlists might actually need extra padding):
  - All basic types give a single chunk; no basic type has a serialisation longer than 32 bytes.
  - `Bitlist[N]` and `Bitvector[N]`: `(N + 255) // 256` (dividing by chunk size in bits and rounding up)
  - `List[B, N]` and `Vector[B, N]`, where `B` is a basic type: `(N * size_of(B) + 31) // 32` (dividing by chunk size in bytes and rounding up)

Containers and composite objects that result from rule 2 will have the following numbers of chunks, including zero-chunk padding where required for lists.

- `List[C, N]` and `Vector[C, N]`, where `C` is a composite type: `N`, since the Merkleization comprises `N` hash tree roots.
- Containers: `len(fields)`, since there is one hash tree root per field in the container.

It is not immediately obvious why lists and bitlists are padded with zero chunks up to their full maximum lengths, even if these are "virtual" chunks. However, this enables the use of generalised indices which provide a consistent way of creating Merkle proofs against hash tree roots, the topic of our next section.

Recall that, in addition to any padding added here, the Merkleization process will further pad the list with zero chunks to make it up to a power of two in length.

### Mixing in the length

We want objects that have the same type but different contents to have different hash tree roots. This presents a problem for lists. Consider the list `a` of three elements, and the list `b` which is the same three elements plus a fourth zero element on the end. These are different lists of the same type, but both Merkleize to the same value.

```
>>> from eth2spec.utils.ssz.ssz_typing import uint256, List
>>> from eth2spec.utils.merkle_minimal import merkleize_chunks
>>> a = List[uint256, 4](1, 2, 3).encode_bytes()
>>> b = List[uint256, 4](1, 2, 3, 0).encode_bytes()
>>> merkleize_chunks([a[0:32], a[32:64], a[64:96]])
0x66c419026fee8793be7fd0011b9db46b98a79f9c9b640e25317865c358f442db
>>> merkleize_chunks([b[0:32], b[32:64], b[64:96], b[96:128]])
0x66c419026fee8793be7fd0011b9db46b98a79f9c9b640e25317865c358f442db
```

We need to ensure that a list ending with a zero value has a different hash tree root from the same list without the zero value. To do this, we put lists (and bitlists) through an extra `mix_in_length()` process that involves hashing a concatenation of the Merkle root of the list and the length of the list. This is

equivalent to the Merkleization of two chunks, the first being the Merkle root of the list, the second being its length.

See the diagram for `attesting_indices` below for an illustration of this in practice.

Bitlists require a similar treatment since we remove the sentinel bit before Merkleizing.

### Summaries and Expansions

The SSZ spec describes features of Merkleization called summaries and expansions. These are not explicit functions of Merkleization, but implicitly arise as consequences of the design.

Simply put, anywhere in the process, an entire SSZ object can be replaced with its hash tree root without affecting the final result.

We make use of this in a number of ways. First and foremost is the ability to cache the hash tree roots of any unchanged parts of the state, which makes it practical to recalculate the hash tree root of the whole state when required. For example, if a validator record is unchanged we do not need to recalculate its hash tree root when finding the root of the validator registry. If the validator registry is unchanged, we do not need to recalculate its hash tree root when calculating the full state root.

As another example, consider the `BeaconBlock` and the `BeaconBlockHeader` types.

```
class BeaconBlock(Container):
    slot: Slot
    proposer_index: ValidatorIndex
    parent_root: Root
    state_root: Root
    body: BeaconBlockBody

class BeaconBlockHeader(Container):
    slot: Slot
    proposer_index: ValidatorIndex
    parent_root: Root
    state_root: Root
    body_root: Root
```

These differ only in their last fields, `body` and `body_root` respectively. If `body_root` is the hash tree root of the `BeaconBlockBody`, `body`, then these two objects will have exactly the same hash tree root. `BeaconBlock` is the expansion type of `BeaconBlockHeader`; `BeaconBlockHeader` is a summary type of `BeaconBlock`. Proposer slashing reports make use of this fact to save space by stripping out the block bodies and replacing them with their hash tree roots.

The Flashbots MEV-Boost design also makes use of this capability. In the MEV-Boost system validators are required to sign "blinded blocks". That is, blocks for which they do not have the bodies. Since the header is a summary of the block (in the sense we are using the word summary here) the same signature will be valid both for the `BeaconBlockHeader` and the corresponding full `BeaconBlock`. This simplifies the protocol design.

### Worked example

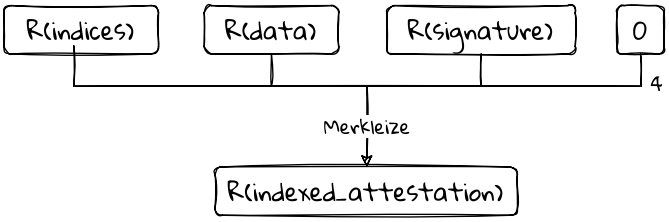
For this section's worked example we shall revisit our friend, the `IndexedAttestation`. This gives us nice instances of Merkleizing composite type, list types, and vector types, as well as demonstrating summaries and expansions.

Recall that the `IndexedAttestation` type is defined as follows,

```
class IndexedAttestation(Container):
    attesting_indices: List[ValidatorIndex, MAX_VALIDATORS_PER_COMMITTEE]
    data: AttestationData
    signature: BLSSignature
```

We will create an instance of this just as we did previously, only for brevity I shall call it `a`, rather than `attestation`. We want to calculate the hash tree root of this `IndexedAttestation`, `a`.

A container's hash tree root is the Merkleization of the list of hash tree roots of the objects it contains (by rule 2). Diagrammatically we are building the following tree and finding its root.




*Calculating the hash tree root of an* `IndexedAttestation`*. In this and the following diagrams,* $R(X)$ *is the Merkleization of* $X$*,* $S(X)$ *is the SSZ serialisation of* $X$*. Each box is a 32 byte chunk, and the small digits are the number of leaves in the Merkleization operation.*

Alternatively, in code, we have the following.

```
assert(a.hash_tree_root() == merkleize_chunks(
    [
        a.attesting_indices.hash_tree_root(),
        a.data.hash_tree_root(),
        a.signature.hash_tree_root()
    ]))
```

The `merkleize_chunks()` function is provided by the `merkle_minimal.py` library. We can apply this function directly as the hash tree roots in the list already constitute chunks. (We could also use the `get_merkle_root()` function, but then we'd have to specify a `pad_to` value of 4 to get a tree of the correct depth.)

### The `attesting_indices` root

Working down the members of the list, we need the hash tree root of the `attesting_indices` object, which has type `List[ValidatorIndex, MAX_VALIDATORS_PER_COMMITTEE]`. This is a list of basic types, namely `uint64` since that's the type of `ValidatorIndex`, and rule 1 applies.

Our `attesting_indices` list has three elements, `[33652, 59750, 92360]`, which we need to chunk and pad. First we serialise the list as usual with SSZ, then we pad it up to 32 bytes:

```
>>> serialize(a.attesting_indices).hex()
'748300000000000066e9000000000000c868010000000000'
>>> (serialize(a.attesting_indices) + bytearray(8)).hex()
'748300000000000066e9000000000000c86801000000000000000000000000000000'
```

This gives us our first chunk. However, the full number of chunks we need is `2048 // 4 = 512` (`MAX_VALIDATORS_PER_COMMITTEE` divided by `uint64`s per chunk), so we must add 511 zero chunks. In practice this padding is done "virtually". The `merkleize_chunks()` function allows us to specify the full number of chunks and takes care of adding the extras. Behind the scenes, it is creating a ten-layer deep Merkle tree with our 512 chunks as leaves and returning the tree's root.

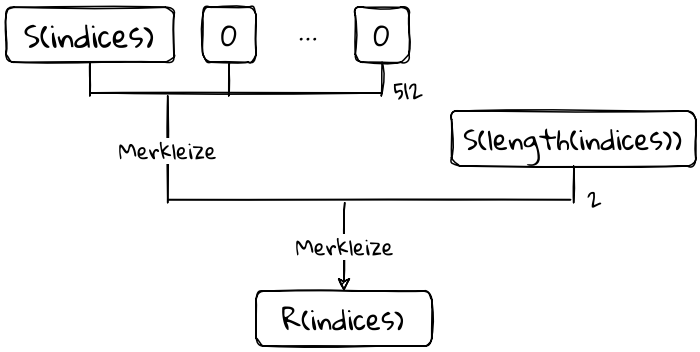
```
>>> merkleize_chunks([serialize(a.attesting_indices) + bytearray(8)], 512).hex()
'04e3bf0951474a6b06dd506648fdf8e84866542614e1c14fa832cd4bebfda0e3'
```

If this were a vector then our work would be done. However, when working with lists, there is a little further wrinkle: as a final step we need to concatenate the root that we have with the actual length of the list and hash them together. This is the `mix_in_length()` function described above which we implement here by Merkleizing the list's Merkle root with the list's length.

```
assert(a.attesting_indices.hash_tree_root() ==
       merkleize_chunks(
           [
```

```
            merkleize_chunks([a.attesting_indices.encode_bytes() + bytearray(8)], 512),
            a.attesting_indices.length().to_bytes(32, 'little')
        ]))
```

In diagram form the hash tree root calculation for the list looks like this.




*Calculating the hash tree root of the `attesting_indices`. This is a `List[uint256, 2048]` type, and our example list has three elements, comprising a single chunk. Note the extra `mix_in_length()` step that's applied to lists.*

### The `data` root

The `data` field of the `IndexedAttestation` is another container, an `AttestationData` object, defined as,

```
class AttestationData(Container):
    slot: Slot
    index: CommitteeIndex
    beacon_block_root: Root
    source: Checkpoint
    target: Checkpoint
```

As before, to find the hash tree root of a container, by rule 2 we need the root of the roots it contains. That is,

```
assert(a.data.hash_tree_root() == merkleize_chunks(
    [
        a.data.slot.hash_tree_root(),
        a.data.index.hash_tree_root(),
        a.data.beacon_block_root.hash_tree_root(),
        a.data.source.hash_tree_root(),
        a.data.target.hash_tree_root()
    ]))
```

The `Slot` and the `CommitteeIndex` are just basic `uint64` types. Their hash tree roots are their little-endian 256-bit representations.

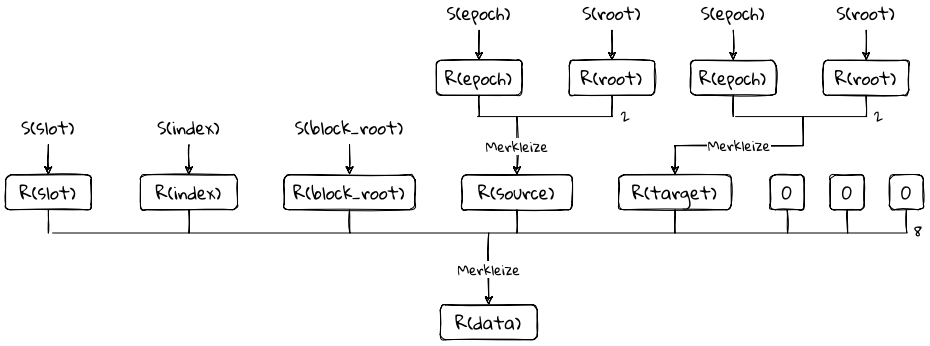
```
>>> a.data.slot.hash_tree_root().hex()
'7d022f0000000000000000000000000000000000000000000000000000000000'
>>> a.data.index.hash_tree_root().hex()
'0900000000000000000000000000000000000000000000000000000000000000'
```

The `Root` is `Bytes32` type, which is equivalent to a `Vector[unit8, 32]`. Handily, the hash tree root is just the `Root` value itself since its only a single chunk.

```
>>> a.data.beacon_block_root.hex()
'4f4250c05956f5c2b87129cf7372f14dd576fc152543bf7042e963196b843fe6'
>>> a.data.beacon_block_root.hash_tree_root().hex()
'4f4250c05956f5c2b87129cf7372f14dd576fc152543bf7042e963196b843fe6'
```

The `source` and `target` are once again containers, both having type `Checkpoint`. The `Checkpoint` type is simple to Merkleize with the knowledge we have. So, putting everything together, we can find the hash tree root of the `data` field by hand as follows.

```
assert(a.data.hash_tree_root() == merkleize_chunks(
    [
        a.data.slot.to_bytes(32, 'little'),
        a.data.index.to_bytes(32, 'little'),
        a.data.beacon_block_root,
        merkleize_chunks([a.data.source.epoch.to_bytes(32, 'little'), a.data.source.root]),
        merkleize_chunks([a.data.target.epoch.to_bytes(32, 'little'), a.data.target.root])
    ]))
```

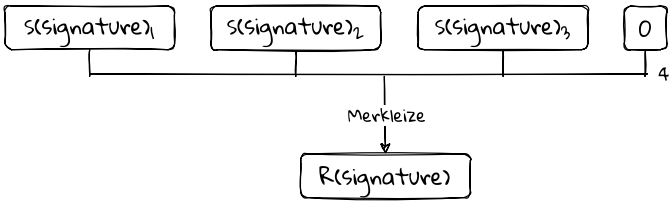



*Calculating the hash tree root of an `AttestationData` container. It contains in turn two `Checkpoint` containers, `source` and `target`.*

### The `signature` root

The final part of the `IndexedAttestation` we need to deal with is the `signature` field. This is of type `Signature`, which is a `Vector[uint8, 96]` and rule 1 applies. This is simple to Merkleize as it is exactly three chunks when packed. The `merkleize_chunks()` function takes care of adding a single virtual zero chunk to make a power-of-two number of leaves.

```
assert(a.signature.hash_tree_root() ==
       merkleize_chunks([a.signature[0:32], a.signature[32:64], a.signature[64:96]]))
```

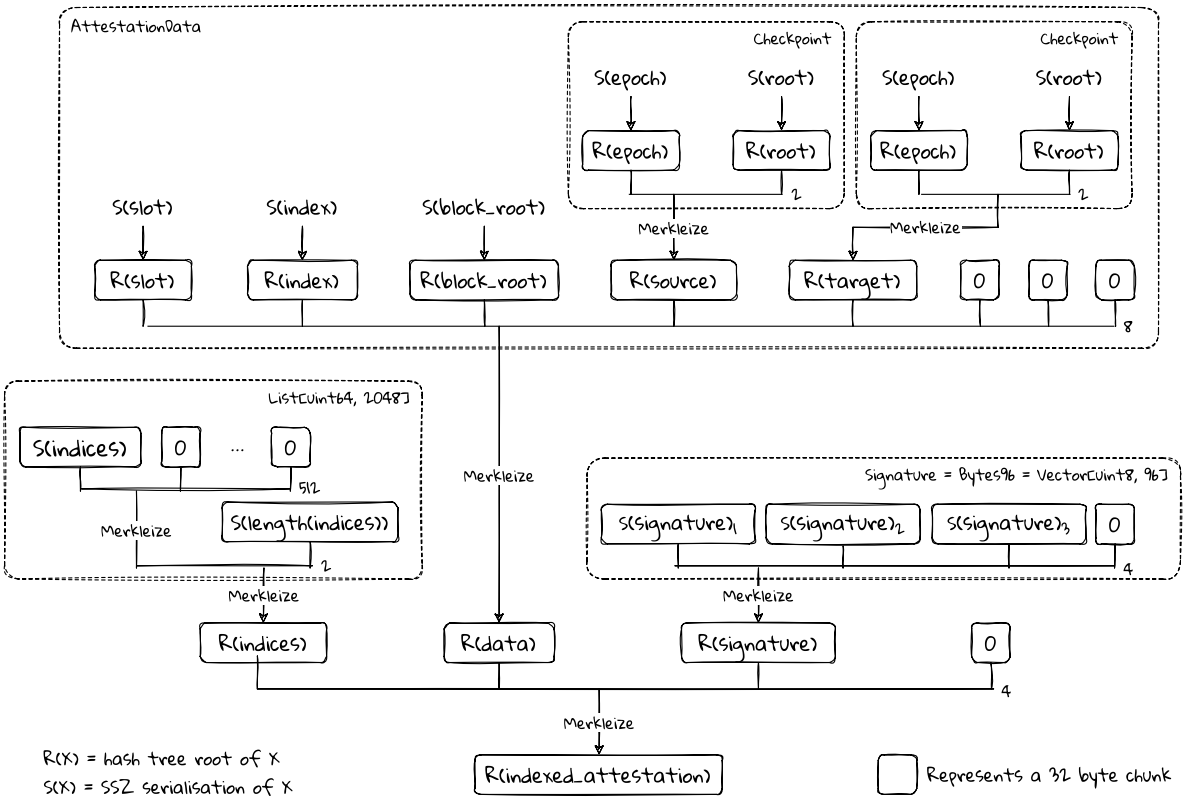



*Calculating the hash tree root of a `Signature`, which is really a `Bytes96`, or `Vector[uint8, 96]` type.*

### Putting it all together

Assembling all these parts we can illustrate in both diagram form and code form how the hash tree root of the `IndexedAttestation` is calculated from the serialisation of the underlying basic types via repeated applications of Merkleization.

### The full picture




*Illustrating the steps required to calculate the hash tree root of an* `IndexedAttestation`*. The small digits are the number of leaves in each Merkleization operation.*

### The full code

The following code illustrates all the points from the worked example. You can run it by setting up the executable spec as described in the Appendix. If everything goes well the only thing it should print is `Success!`.

```
from eth2spec.altair import mainnet
from eth2spec.altair.mainnet import *
from eth2spec.utils.ssz.ssz_typing import *
from eth2spec.utils.merkle_minimal import merkleize_chunks

# Initialise an IndexedAttestation type
a = IndexedAttestation(
    attesting_indices = [33652, 59750, 92360],
    data = AttestationData(
        slot = 3080829,
        index = 9,
        beacon_block_root = '0x4f4250c05956f5c2b87129cf7372f14dd576fc152543bf7042e963196b843fe6',
        source = Checkpoint (
            epoch = 96274,
            root = '0xd24639f2e661bc1adcbe7157280776cf76670fff0fee0691f146ab827f4f1ade'
        ),
        target = Checkpoint(
            epoch = 96275,
            root = '0x9bcd31881817ddeab686f878c8619d664e8bfa4f8948707cba5bc25c8d74915d'
        )
    ),
    signature = '0xaaf504503ff15ae86723c906b4b6bac91ad728e4431aea3be2e8e3acc888d8af'
                + '5dffbbcf53b234ea8e3fde67fbb09120027335ec63cf23f0213cc439e8d1b856'
                + 'c2ddfc1a78ed3326fb9b4fe333af4ad3702159dbf9caeb1a4633b752991ac437'
)

# A container's root is the merkleization of the roots of its fields.
# This is IndexedAttestation.
assert(a.hash_tree_root() == merkleize_chunks(
    [
        a.attesting_indices.hash_tree_root(),
        a.data.hash_tree_root(),
        a.signature.hash_tree_root()
    ]))

# A list is serialised then (virtually) padded to its full number of chunks before Merkleization.
# Finally its actual length is mixed in via a further hash/merkleization.
assert(a.attesting_indices.hash_tree_root() ==
       merkleize_chunks(
           [
               merkleize_chunks([a.attesting_indices.encode_bytes() + bytearray(8)], 512),
               a.attesting_indices.length().to_bytes(32, 'little')
           ]))

# A container's root is the merkleization of the roots of its fields.
# This is AttestationData.
assert(a.data.hash_tree_root() == merkleize_chunks(
    [
        a.data.slot.hash_tree_root(),
        a.data.index.hash_tree_root(),
        a.data.beacon_block_root.hash_tree_root(),
        a.data.source.hash_tree_root(),
        a.data.target.hash_tree_root()
    ]))

# Expanding the above AttestationData roots by "manually" calculating the roots of its fields.
assert(a.data.hash_tree_root() == merkleize_chunks(
    [
```

```
        a.data.slot.to_bytes(32, 'little'),
        a.data.index.to_bytes(32, 'little'),
        a.data.beacon_block_root,
        merkleize_chunks([a.data.source.epoch.to_bytes(32, 'little'), a.data.source.root]),
        merkleize_chunks([a.data.target.epoch.to_bytes(32, 'little'), a.data.target.root]),
    ]))

# The Signature type has a simple Merkleization.
assert(a.signature.hash_tree_root() ==
       merkleize_chunks([a.signature[0:32], a.signature[32:64], a.signature[64:96]]))

# Putting everything together, we have a "by-hand" Merkleization of the IndexedAttestation.
assert(a.hash_tree_root() == merkleize_chunks(
    [
        # a.attesting_indices.hash_tree_root()
        merkleize_chunks(
            [
                merkleize_chunks([a.attesting_indices.encode_bytes() + bytearray(8)], 512),
                a.attesting_indices.length().to_bytes(32, 'little')
            ]),
        # a.data.hash_tree_root()
        merkleize_chunks(
            [
                a.data.slot.to_bytes(32, 'little'),
                a.data.index.to_bytes(32, 'little'),
                a.data.beacon_block_root,
                merkleize_chunks([a.data.source.epoch.to_bytes(32, 'little'), a.data.source.root]),
                merkleize_chunks([a.data.target.epoch.to_bytes(32, 'little'), a.data.target.root]),
            ]),
        # a.signature.hash_tree_root()
        merkleize_chunks([a.signature[0:32], a.signature[32:64], a.signature[64:96]])
    ]))

print("Success!")
```

#### See also

What is a Merkle Tree? by Alin Tomescu is the best primer I have found on Merkle trees, and a great starting point if you are unsure about their construction and properties.

The SSZ specification is the authoritative source for Merkleization as well as serialisation. Many SSZ implementations also include Merkleization.

A formal verification of Merkleization has been performed: Notes and Code.

The Remerkleable library is a Python implementation that introduces some more advanced tools such as backing trees for the data structures. Ztyp is a further exploration of backing trees. Backing trees are a useful approach to representing and maintaining the beacon state within client implementations.

Given the limited type of hashing that's done during Merkleization (always hashing the concatenation of two 32 byte strings), it's worth looking into whether specific performance optimisations are available. Potuz has produced an optimised library, Hashtree, for Merkle tree computation that takes advantage of this.

### Generalised indices and Merkle proofs

TODO

### Sync Committees

TODO

# Upgrades

## Introduction

TODO

## History of upgrades

# Altair

TODO

## Hard Forks

TODO

## Fork Digest

TODO

# Networking

## Introduction

TODO

## Discovery

TODO

## Gossip

TODO

## RPC

TODO

## Syncing

TODO

## Message Types

TODO

# Implementation

## Introduction

TODO

## Protoarray

TODO

## SSZ backing tree

TODO

## Batch signature verification

TODO

## Slashing protection

TODO

## Checkpoint sync

TODO

# Part 3: Annotated Specification

# Introduction

The beacon chain specification is the guts of the machine. Like the guts of a computer, all the components are showing and the wires are hanging out: everything is on display. In the course of the next sections I will be dissecting the entire core beacon chain specification line by line. My aim is not only to explain how things work, but also to give some historical context, some of the reasoning behind how we ended up where we are today.

Early versions of the specs were written with much more narrative and explanation than today's. Over time they were coded up in Python for better precision and the benefits of being executable. However, in that process, most of the explanation and intuition was removed.[38] Vitalik has created his own annotated specifications that covers many of the key insights. It's hard to compete with Vitalik, but my intention here is to go one level deeper in thoroughness and detail. And perhaps to give an independent perspective.

As and when other parts of the book get written I will add links to the specific chapters on each topic (for example on Simple Serialize, consensus, networking).

Note that the online annotated specification is available in two forms:

- divided into chapters in Part 3 of the main book, and
- as a standalone single page that's useful for searching.

The contents of each are identical.

## Version information

This edition of Upgrading Ethereum is based on the Altair version of the beacon chain specification, and corresponds to Release v1.1.1, made on the 4th of October 2021.

At the time of writing, there is no single specification document for Altair. Instead, there is the Phase 0 specification and an additional Altair document describing the differences (a kind of text-based diff).

For this work, I have consolidated the two specifications into one, omitting parts that were superseded by Altair. For the most part, I have tried to reflect the existing structure of the documents to make it easier to read side-by-side with the original spec. However, I have included the separate BLS and Altair fork documents into the flow of this one.

### References

The main references:

- The Phase 0 beacon chain specification.
- Altair updates to the beacon chain specification.
- Vitalik's annotated specifications, covering Phase 0, Altair, The Merge, and beyond.

Other excellent, but in places outdated references:

- Serenity Design Rationale
- Phase 0 for Humans [v0.10.0]
- Phase 0 design notes (Justin Drake)
- My own Phase 0 annotated specification remains available for historical interest.

[38] A process called "Justinification". Iykyk ;-)

# Types, Constants, Presets, and Configuration

## Preamble

For some, a chapter on constants, presets and parameters will seem drier than the Namib desert, but I've long found these to be a rich and fertile way in to the ideas and mechanisms we'll be unpacking in detail in later chapters. Far from being a desert, this part of the spec bustles with life.

The foundation is laid with a set of custom data types. The beacon chain specification is executable in Python; the data types defined at the top of the spec represent the fundamental quantities that will reappear frequently.

Then – with constants, presets, and parameters – we will examine the numbers that define and constrain the behaviour of the chain. Each of these quantities tells a story. Each parameter encapsulates an insight, or a mechanism, or a compromise. Why is it here? How has it changed over time? Where does its value come from?

## Custom Types

The specification defines the following Python custom types, "for type hinting and readability": the data types defined here appear frequently throughout the spec; they are the building blocks for everything else.

Each type has a name, an "SSZ equivalent", and a description. SSZ is the encoding method used to pass data between clients, among other things. Here it can be thought of as just a primitive data type.

Throughout the spec, (almost) all integers are unsigned 64 bit numbers, `uint64`, but this hasn't always been the case.

Regarding "unsigned", there was much discussion around whether Eth2 should use signed or unsigned integers, and eventually unsigned was chosen. As a result, it is critical to preserve the order of operations in some places to avoid inadvertently causing underflows since negative numbers are forbidden.

And regarding "64 bit", early versions of the spec used other bit lengths than 64 (a "premature optimisation"), but arithmetic integers are now standardised at 64 bits throughout the spec, the only exception being `ParticipationFlags`, introduced in the Altair fork, which has type `uint8`, and is really a `byte` type.

| Name | SSZ equivalent | Description |
|---|---|---|
| `Slot` | `uint64` | a slot number |
| `Epoch` | `uint64` | an epoch number |
| `CommitteeIndex` | `uint64` | a committee index at a slot |
| `ValidatorIndex` | `uint64` | a validator registry index |
| `Gwei` | `uint64` | an amount in Gwei |
| `Root` | `Bytes32` | a Merkle root |
| `Hash32` | `Bytes32` | a 256-bit hash |
| `Version` | `Bytes4` | a fork version number |
| `DomainType` | `Bytes4` | a domain type |
| `ForkDigest` | `Bytes4` | a digest of the current fork data |
| `Domain` | `Bytes32` | a signature domain |
| `BLSPubkey` | `Bytes48` | a BLS12-381 public key |
| `BLSSignature` | `Bytes96` | a BLS12-381 signature |
| `ParticipationFlags` | `uint8` | a succinct representation of 8 boolean participation flags |

### Slot

Time is divided into fixed length slots. Within each slot, exactly one validator is randomly selected to propose a beacon chain block. The progress of slots is the fundamental heartbeat of the beacon chain.

### Epoch

Sequences of slots are combined into fixed-length epochs.

Epoch boundaries are the points at which the chain can be justified and finalised (by the Casper FFG mechanism). They are also the points at which validator balances are updated, validator committees get shuffled, and validator exits, entries, and slashings are processed. That is, the main state-transition work is performed per epoch, not per slot.

Epochs have always felt like a slightly uncomfortable overlay on top of the slot-by-slot progress of the beacon chain, but necessitated by Casper FFG finality. There have been proposals to move away from epochs, and there are possible future developments that could allow us to do away with epochs entirely. But, for the time being, they remain.

Fun fact: Epochs were originally called Cycles.

### CommitteeIndex

Validators are organised into committees that collectively vote (make attestations) on blocks. Each committee is active at exactly one slot per epoch, but several committees are active at each slot. The `CommitteeIndex` type is an index into the list of committees active at a slot.

The beacon chain's committee-based design is a large part of what makes it practical to implement while maintaining security. If all validators were active all the time, there would be an overwhelming number of messages to deal with. The random shuffling of committees make them very hard to subvert by an attacker without a supermajority of stake.

### ValidatorIndex

Each validator making a successful deposit is consecutively assigned a unique validator index number that is permanent, remaining even after the validator exits. It is permanent because the validator's balance is associated with its index, so the data needs to be preserved when the validator exits, at least until the balance is withdrawn at an unknown future time.

### Gwei

All Ether amounts are specified in units of Gwei ($10^9$ Wei, $10^{-9}$ Ether). This is basically a hack to avoid having to use integers wider than 64 bits to store validator balances and while doing calculations, since ($2^{64}$ Wei is only 18 Ether. Even so, in some places care needs to be taken to avoid arithmetic overflow when dealing with Ether calculations.

### Root

Merkle roots are ubiquitous in the Eth2 protocol. They are a very succinct and tamper-proof way of representing a lot of data, an example of a cryptographic accumulator. Blocks are summarised by their Merkle roots; state is summarised by its Merkle root; the list of Eth1 deposits is summarised by its Merkle root; the digital signature of a message is calculated from the Merkle root of the data structure contained within the message.

### Hash32

Merkle roots are constructed with cryptographic hash functions. In the spec, a `Hash32` type is used to represent Eth1 block roots (which are also Merkle roots).

I don't know why only the Eth1 block hash has been awarded the `Hash32` type: other hashes in the spec remain `Bytes32`. In early versions of the spec `Hash32` was used for all cryptographic has quantities, but this was changed to `Bytes32`.

Anyway, it's worth taking a moment in appreciation of the humble cryptographic hash function. The hash function is arguably the single most important algorithmic innovation underpinning blockchain technology, and in fact most of our online lives. Easily taken for granted, but utterly critical in enabling our modern world.

### Version

Unlike Ethereum 1[39], the beacon chain has an in-protocol concept of a version number. It is expected that the protocol will be updated/upgraded from time to time, a process commonly known as a "hard-fork". For example, the upgrade from Phase 0 to Altair took place on the 27th of October 2021, and was assigned its own fork version.

`Version` is used when computing the `ForkDigest`.

### DomainType

`DomainType` is just a cryptographic nicety: messages intended for different purposes are tagged with different domains before being hashed and possibly signed. It's a kind of name-spacing to avoid clashes; probably unnecessary, but considered a best-practice. Ten domain types are defined in Altair.

### ForkDigest

`ForkDigest` is the unique chain identifier, generated by combining information gathered at genesis with the current chain `Version` identifier.

The `ForkDigest` serves two purposes.

1. Within the consensus protocol to prevent, for example, attestations from validators on one fork (that maybe haven't upgraded yet) being counted on a different fork.
2. Within the networking protocol to help to distinguish between useful peers that on the same chain, and useless peers that are on a different chain. This usage is described in the Ethereum 2.0 networking specification, where `ForkDigest` appears frequently.

Specifically, `ForkDigest` is the first four bytes of the hash tree root of the `ForkData` object containing the current chain `Version` and the `genesis_validators_root` which was created at beacon chain initialisation. It is computed in `compute_fork_digest()`.

### Domain

`Domain` is used when verifying protocol messages validators. To be valid, a message must have been combined with both the correct domain and the correct fork version. It calculated as the concatenation of the four byte `DomainType` and the first 28 bytes of the fork data root.

### BLSPubkey

BLS (Boneh-Lynn-Shacham) is the digital signature scheme used by Eth2. It has some very nice properties, in particular the ability to aggregate signatures. This means that many validators can sign the same message (for example, that they support block X), and these signatures can all be efficiently aggregated into a single signature for verification. The ability to do this efficiently makes Eth2 practical as a protocol. Several other protocols have adopted or will adopt BLS, such as Zcash, Chia, Dfinity and Algorand. We are using the BLS signature scheme based on the BLS12-381 (Barreto-Lynn-Scott) elliptic curve.

The `BLSPubkey` type holds a validator's public key, or the aggregation of several validators' public keys. This is used to verify messages that are claimed to have come from that validator or group of validators.

In Ethereum 2.0, BLS public keys are elliptic curve points from the BLS12-381 $G_1$ group, thus are 48 bytes long when compressed.

See the section on BLS signatures in part 2 for a more in-depth look at these things.

---

[39] Ethereum 1.0 introduced a fork identifier as defined in EIP-2124 which is similar to `Version`, but the Eth1 fork id is not part of the consensus protocol and is used only in the networking protocol.

### BLSSignature

As above, we are using BLS signatures over the BLS12-381 elliptic curve in order to sign messages between participants. As with all digital signature schemes, this guarantees both the identity of the sender and the integrity of the contents of any message.

In Ethereum 2.0, BLS signatures are elliptic curve points from the BLS12-381 $G_2$ group, thus are 96 bytes long when compressed.

### ParticipationFlags

The `ParticipationFlags` type was introduced in the Altair upgrade as part of the accounting reforms.

Prior to Altair, all attestations seen in blocks were stored in state for two epochs. At the end of an epoch, finality calculations, and reward and penalty calculations for each active validator, would be done by processing all of the attestations for the previous epoch as a batch. This created a spike in processing at epoch boundaries, and led to a noticeable increase in late blocks and attestations during the first slots of epochs. With Altair, participation flags are now used to continuously track validators' attestations, reducing the processing load at the end of epochs.

Three of the eight bits are currently used; five are reserved for future use.

As an aside, it might have been more intuitive if `ParticipationFlags` were a `Bytes1` type, rather than introducing a weird `uint8` into the spec. After all, it is not used as an arithmetic integer. However, `Bytes1` is a composite type in SSZ, really an alias for `Vector[uint8, 1]`, whereas `uint8` is a basic type. When computing the hash tree root of a `List` type, multiple basic types can be packed into a single leaf, while composite types take a leaf each. This would result in 32 times as many hashing operations for a list of `Bytes1`. For similar reasons the type of `ParticipationFlags` was changed from `bitlist` to `uint8`.

### References

- A primer on Merkle roots.
  - See also Wikipedia on Merkle Trees.
- I have written an intro to the BLS12-381 elliptic curve elsewhere.

## Constants

The distinction between "constants", "presets", and "configuration values" is not always clear, and things have moved back and forth between the sections at times[40]. In essence, "constants" are things that are expected never to change for the beacon chain, no matter what fork or test network it is running.

### Miscellaneous

| Name | Value |
|---|---|
| `GENESIS_SLOT` | `Slot(0)` |
| `GENESIS_EPOCH` | `Epoch(0)` |
| `FAR_FUTURE_EPOCH` | `Epoch(2**64 - 1)` |
| `DEPOSIT_CONTRACT_TREE_DEPTH` | `uint64(2**5)` $(= 32)$ |
| `JUSTIFICATION_BITS_LENGTH` | `uint64(4)` |
| `PARTICIPATION_FLAG_WEIGHTS` | `[TIMELY_SOURCE_WEIGHT, TIMELY_TARGET_WEIGHT, TIMELY_HEAD_WEIGHT]` |
| `ENDIANNESS` | `'little'` |

[40] See Issue 2390 for discussion and a rationale for the current categorisation into constants, presets, and configuration variables.

#### `GENESIS_SLOT`

The very first slot number for the beacon chain is zero.

Perhaps this seems uncontroversial, but it actually featured heavily in the Great Signedness Wars mentioned previously. The issue was that calculations on unsigned integers might have negative intermediate values, which would cause problems. A proposed work-around for this was to start the chain at a non-zero slot number. It was initially set to 2^19, then 2^63, then 2^32, and finally back to zero. In my humble opinion, this madness only confirms that we should have been using signed integers all along.

#### `GENESIS_EPOCH`

As above. When the chain starts, it starts at epoch zero.

#### `FAR_FUTURE_EPOCH`

A candidate for the dullest constant. It's used as a default initialiser for validators' activation and exit times before they are properly set. No epoch number will ever be bigger than this one.

#### `DEPOSIT_CONTRACT_TREE_DEPTH`

`DEPOSIT_CONTRACT_TREE_DEPTH` specifies the size of the (sparse) Merkle tree used by the Eth1 deposit contract to store deposits made. With a value of 32, this allows for $2^{32} = 4.3$ billion deposits. Given that the minimum deposit it 1 Ether, that number is clearly enough.

Since deposit receipts contain Merkle proofs, their size depends on the value of this constant.

#### `JUSTIFICATION_BITS_LENGTH`

As an optimisation to Casper FFG – the process by which finality is conferred on epochs – the beacon chain uses a "$k$-finality" rule. We will describe this more fully when we look at processing justification and finalisation. For now, this constant is just the number of bits we need to store in state to implement $k$-finality. With $k = 2$, we track the justification status of the last four epochs.

#### `PARTICIPATION_FLAG_WEIGHTS`

This array is just a convenient way to access the various weights given to different validator achievements when calculating rewards. The three weights are defined under incentivization weights, and each weight corresponds to a flag stored in state and defined under participation flag indices.

#### `ENDIANNESS`

Endianness refers to the order of bytes in the binary representation of a number: most significant byte first is big-endian; least significant byte first is little-endian. For the most part these details are hidden by compilers, and we don't need to worry about endianness. But endianness matters when converting between integers and bytes, which is relevant to shuffling and proposer selection, the RANDAO, and when serialising with SSZ.

The spec began life as big-endian, but the Nimbus team from Status successfully lobbied for it to be changed to little-endian to better match processor hardware implementations, and the endianness of WASM. SSZ was changed first, and then the rest of the spec followed.

### Participation flag indices

| Name | Value |
|---|---|
| `TIMELY_SOURCE_FLAG_INDEX` | `0` |
| `TIMELY_TARGET_FLAG_INDEX` | `1` |
| `TIMELY_HEAD_FLAG_INDEX` | `2` |

Validators making attestations that get included on-chain are rewarded for three things:

- getting attestations included with the correct source checkpoint within 5 slots (`integer_squareroot(SLOTS_PER_EPOCH)`);
- getting attestations included with the correct target checkpoint within 32 slots (`SLOTS_PER_EPOCH`); and,
- getting attestations included with the correct head within 1 slot (`MIN_ATTESTATION_INCLUSION_DELAY`), basically immediately.

These flags are temporarily recorded in the `BeaconState` when attestations are processed, then used at the ends of epochs to update finality and to calculate validator rewards for making attestations.

The mechanism for rewarding timely inclusion of attestations (thus penalising late attestations) differs between Altair and Phase 0. In Phase 0, attestations included within 32 slots would receive the full reward for the votes they got correct (source, target, head), plus a declining reward based on the delay in inclusion: $\frac{1}{2}$ for a two slot delay, $\frac{1}{3}$ for a three slot delay, and so on. With Altair, for each vote, we now have a cliff before which the validator receives the full reward and after which a penalty. The cliffs differ in duration, which is intended to more accurately target incentives at behaviours that genuinely help the chain (there is little value in rewarding a correct head vote made 30 slots late, for example). See `get_attestation_participation_flag_indices()` for how this is implemented in code.

### Incentivization weights

| Name | Value |
|---|---|
| `TIMELY_SOURCE_WEIGHT` | `uint64(14)` |
| `TIMELY_TARGET_WEIGHT` | `uint64(26)` |
| `TIMELY_HEAD_WEIGHT` | `uint64(14)` |
| `SYNC_REWARD_WEIGHT` | `uint64(2)` |
| `PROPOSER_WEIGHT` | `uint64(8)` |
| `WEIGHT_DENOMINATOR` | `uint64(64)` |

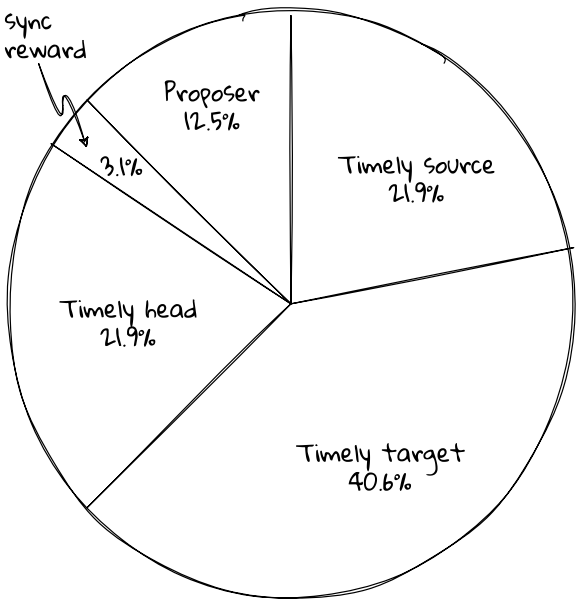
These weights are used to calculate the reward earned by a validator for performing its duties. There are five duties in total. Three relate to making attestations: attesting to the source epoch, attesting to the target epoch, and attesting to the head block. There are also rewards for proposing blocks, and for participating in sync committees. Note that the sum of five the weights is equal to `WEIGHT_DENOMINATOR`.

On a long-term average, a validator can expect to earn a total amount of `get_base_reward()` per epoch, with these weights being the relative portions for each of the duties comprising that total. Proposing blocks and participating in sync committees do not happen in every epoch, but are randomly assigned, so over small periods of time validator earnings may differ from `get_base_reward()`.

The apportioning of rewards was overhauled in the Altair upgrade to better reflect the importance of each activity within the protocol. The total reward amount remains the same, but sync committee rewards were added, and the relative weights were adjusted. Previously, the weights corresponded to 16 for correct source, 16 for correct target, 16 for correct head, 14 for inclusion (equivalent to correct source), and 2 for block proposals. The factor of four increase in the proposer reward addressed a long-standing spec bug.

### Withdrawal Prefixes

| Name | Value |
|---|---|
| `BLS_WITHDRAWAL_PREFIX` | `Bytes1('0x00')` |
| `ETH1_ADDRESS_WITHDRAWAL_PREFIX` | `Bytes1('0x01')` |




*The proportion of the total reward derived from each of the micro-rewards.*

Withdrawal prefixes relate to the withdrawal credentials provided when deposits are made for validators. The withdrawal credential is a commitment to a private key that may be used later to withdraw funds from the validator's balance on the beacon chain.

Two ways to specify the withdrawal credentials are currently available, versioned with these prefixes, with others such as `0x02` and `0x03` under discussion.

These withdrawal credential prefixes are not yet significant in the core beacon chain spec, but will become significant when withdrawals are enabled in a future upgrade. The withdrawal credentials data is not consensus-critical, and future withdrawal credential types can be added without a hard fork. There are suggestions as to how existing credentials might be changed between methods which would be consensus critical.

The presence of these prefixes in the spec indicates a "social consensus" among the dev teams and protocol designers that we will in future support these methods for making withdrawals.

See the Withdrawals section for discussion on what the mechanism might look like.

#### `BLS_WITHDRAWAL_PREFIX`

The beacon chain launched with only BLS-style withdrawal credentials available, so all early stakers used this. The `0x00` prefix on the credential distinguishes this type from the others: it replaces the first byte of the hash of the BLS public key that corresponds to the BLS private key of the staker.

With this type of credential, in addition to a BLS signing key, stakers need a second BLS key that they will later use for withdrawals. The credential registered in the deposit data is the 32 byte SHA256 hash of the validators withdrawal public key, with the first byte set to `BLS_WITHDRAWAL_PREFIX`.

#### `ETH1_ADDRESS_WITHDRAWAL_PREFIX`

Eth1 withdrawal credentials are much simpler, and were adopted once it became clear that Ethereum 2.0 would not be using a BLS-based address scheme for accounts at any time soon. These provide a commitment that stakers will be able to withdraw their beacon chain funds to a normal Ethereum account (possibly a contract account) at a future date.

### Domain types

| Name | Value |
|---|---|
| DOMAIN_BEACON_PROPOSER | DomainType('0x00000000') |
| DOMAIN_BEACON_ATTESTER | DomainType('0x01000000') |
| DOMAIN_RANDAO | DomainType('0x02000000') |
| DOMAIN_DEPOSIT | DomainType('0x03000000') |
| DOMAIN_VOLUNTARY_EXIT | DomainType('0x04000000') |
| DOMAIN_SELECTION_PROOF | DomainType('0x05000000') |
| DOMAIN_AGGREGATE_AND_PROOF | DomainType('0x06000000') |
| DOMAIN_SYNC_COMMITTEE | DomainType('0x07000000') |
| DOMAIN_SYNC_COMMITTEE_SELECTION_PROOF | DomainType('0x08000000') |
| DOMAIN_CONTRIBUTION_AND_PROOF | DomainType('0x09000000') |

These domain types are used in three ways: for seeds, for signatures, and for selecting aggregators.

### As seeds

When random numbers are required in-protocol, one way they are generated is by hashing the RANDAO mix with other quantities, one of them being a domain type (see `get_seed()`). The original motivation was to avoid occasional collisions between Phase 0 committees and Phase 1 persistent committees, back when they were a thing. So, when computing the beacon block proposer, `DOMAIN_BEACON_PROPOSER` is hashed into the seed, when computing attestation committees, `DOMAIN_BEACON_ATTESTER` is hashed in, and when computing sync committees, `DOMAIN_SYNC_COMMITTEE` is hashed in.

### As signatures

In addition, as a cryptographic nicety, each of the protocol's signature types is augmented with the appropriate domain before being signed:

- Signed block proposals incorporate `DOMAIN_BEACON_PROPOSER`
- Signed attestations incorporate `DOMAIN_BEACON_ATTESTER`
- RANDAO reveals are BLS signatures, and use `DOMAIN_RANDAO`
- Deposit data messages incorporate `DOMAIN_DEPOSIT`
- Validator voluntary exit messages incorporate `DOMAIN_VOLUNTARY_EXIT`
- Sync committee signatures incorporate `DOMAIN_SYNC_COMMITTEE`

In each case, except for deposits, the fork version is also incorporated before signing. Deposits are valid across forks, but other messages are not. Note that this would allow validators to participate, if they wish, in two independent forks of the beacon chain without fear of being slashed.

### Aggregator selection

The remaining four types, suffixed `_PROOF` are not used directly in the beacon chain specification. They were introduced to implement attestation subnet validations for denial of service resistance. The technique was extended to sync committees with the Altair upgrade.

Briefly, at each slot, validators are selected to aggregate attestations from their committees. The selection is done based on the validator's signature over the slot number, mixing in `DOMAIN_SELECTION_PROOF`. The validator then signs the whole aggregated attestation, including the previous signature as proof that it was selected to be a validator, using `DOMAIN_AGGREGATE_AND_PROOF`. And similarly for sync committees. In this way, everything is verifiable and attributable, making it hard to flood the network with fake messages.

These four are not part of the consensus-critical state-transition, but are nonetheless important to the healthy functioning of the chain.

This mechanism is described in the Phase 0 honest validator spec for attestation aggregation, and in the Altair honest validator spec for sync committee aggregation.

### Crypto

| Name | Value |
|---|---|
| `G2_POINT_AT_INFINITY` | `BLSSignature(b'\xc0' + b'\x00' * 95)` |

This is the compressed serialisation of the "point at infinity", the identity point, of the G2 group of the BLS12-381 curve that we are using for signatures. Note that it is in big-endian format (unlike all other constants in the spec).

It was introduced as a convenience when verifying aggregate signatures that contain no public keys in `eth_fast_aggregate_verify()`. The underlying FastAggregateVerify function from the BLS signature standard would reject these.

`G2_POINT_AT_INFINITY` is described in the separate BLS Extensions document, but included here for convenience.

## Preset

The "presets" are consistent collections of configuration variables that are bundled together. The specs repo currently defines two sets of presets, mainnet and minimal. The mainnet configuration is running in production on the beacon chain; minimal is often used for testing. Other configurations are possible. For example, Teku uses a swift configuration for acceptance testing.

All the values discussed below are from the mainnet configuration.

You'll notice that most of these values are powers of two. There's no huge significance to this. Computer scientists think it's neat, and it ensures that things cleanly divide other things in general. There is a view that this practice helps to minimise bike-shedding (endless arguments over trivial matters).

Some of the configuration parameters below are quite technical and perhaps obscure. I'll take the opportunity here to introduce some concepts, and give more detailed explanations when they appear in later chapters.

### Misc

| Name | Value |
|---|---|
| `MAX_COMMITTEES_PER_SLOT` | `uint64(2**6)` (= 64) |
| `TARGET_COMMITTEE_SIZE` | `uint64(2**7)` (= 128) |
| `MAX_VALIDATORS_PER_COMMITTEE` | `uint64(2**11)` (= 2,048) |
| `SHUFFLE_ROUND_COUNT` | `uint64(90)` |

#### `MAX_COMMITTEES_PER_SLOT`

Validators are organised into committees to do their work. At any one time, each validator is a member of exactly one beacon chain committee, and is called on to make an attestation exactly once per epoch. An attestation is a vote for, or a statement of, the validator's view of the chain at that point in time.

On the beacon chain, up to 64 committees are active in a slot and effectively act as a single committee as far as the fork-choice rule is concerned. They all vote on the proposed block for the slot, and their votes/attestations are pooled. In a similar way, all committees active during an epoch (that is, the whole active validator set) act effectively as a single committee as far as justification and finalisation are concerned.

The number 64 is intended to map to one committee per shard once data shards are deployed, since these committees will also vote on shard crosslinks.

Note that sync committees are a different thing: there is only one sync committee active at any time.

#### TARGET_COMMITTEE_SIZE

To achieve a desirable level of security, committees need to be larger than a certain size. This makes it infeasible for an attacker to randomly end up with a super-majority in a committee even if they control a significant number of validators. The target here is a kind of lower-bound on committee size. If there are not enough validators for all committees to have at least 128 members, then, as a first measure, the number of committees per slot is reduced to maintain this minimum. Only if there are fewer than `SLOTS_PER_EPOCH * TARGET_COMMITTEE_SIZE` = 4096 validators in total will the committee size be reduced below `TARGET_COMMITTEE_SIZE`. With so few validators, the system would be insecure in any case.

For further discussion and an explanation of how the value of `TARGET_COMMITTEE_SIZE` was set, see the section on committees.

#### MAX_VALIDATORS_PER_COMMITTEE

This is just used for sizing some data structures, and is not particularly interesting. Reaching this limit would imply over 4 million active validators, staked with a total of 128 million Ether, which exceeds the total supply today.

#### SHUFFLE_ROUND_COUNT

The beacon chain implements a rather interesting way of shuffling validators in order to select committees, called the "swap-or-not shuffle". This shuffle proceeds in rounds, and the degree of shuffling is determined by the number of rounds, `SHUFFLE_ROUND_COUNT`. The time taken to shuffle is linear in the number of rounds, so for light-weight, non-mainnet configurations, the number of rounds can be reduced.

The value 90 was introduced in Vitalik's initial commit without explanation. The original paper describing the shuffling technique seems to suggest that a cryptographically safe number of rounds is $6 \log N$. With 90 rounds, then, we should be good for shuffling 3.3 million validators, which is close to the maximum number possible (given the Ether supply).

### Hysteresis parameters

| Name | Value |
|---|---|
| `HYSTERESIS_QUOTIENT` | `uint64(4)` |
| `HYSTERESIS_DOWNWARD_MULTIPLIER` | `uint64(1)` |
| `HYSTERESIS_UPWARD_MULTIPLIER` | `uint64(5)` |

The parameters prefixed `HYSTERESIS_` control the way that effective balance is changed (see `EFFECTIVE_BALANCE_INCREMENT`). As described there, the effective balance of a validator follows changes to the actual balance in a step-wise way, with hysteresis applied. This ensures that the effective balance does not change often.

The original hysteresis design had an unintended effect that might have encouraged stakers to over-deposit or make multiple deposits in order to maintain a balance above 32 Ether at all times. If a validator's balance were to drop below 32 Ether soon after depositing, however briefly, the effective balance would have immediately dropped to 31 Ether and taken a long time to recover. This would have resulted in a 3% reduction in rewards for a period.

This problem was addressed by making the hysteresis configurable via these parameters. Specifically, these settings mean:

1. if a validators' balance falls 0.25 Ether below its effective balance, then its effective balance is reduced by 1 Ether
2. if a validator's balance rises 1.25 Ether above its effective balance, then its effective balance is increased by 1 Ether

These calculations are done in `process_effective_balance_updates()` during end of epoch processing.

### Gwei values

| Name | Value |
|---|---|
| `MIN_DEPOSIT_AMOUNT` | `Gwei(2**0 * 10**9)` (= 1,000,000,000) |
| `MAX_EFFECTIVE_BALANCE` | `Gwei(2**5 * 10**9)` (= 32,000,000,000) |
| `EFFECTIVE_BALANCE_INCREMENT` | `Gwei(2**0 * 10**9)` (= 1,000,000,000) |

#### `MIN_DEPOSIT_AMOUNT`

`MIN_DEPOSIT_AMOUNT` is not actually used anywhere within the beacon chain specification document. Rather, it is enforced in the deposit contract that was deployed to the Ethereum 1 chain. Any amount less than this value sent to the deposit contract is reverted.

Allowing stakers to make deposits smaller than a full stake is useful for topping-up a validator's balance if its effective balance has dropped below 32 Ether, so as to maintain full productivity. However, this actually led to a vulnerability for some staking pools, involving the front-running of deposits. In some circumstances, a front-running attacker could change a genuine depositor's withdrawal credentials to their own.

#### `MAX_EFFECTIVE_BALANCE`

There is a concept of "effective balance" for validators: whatever a validator's total balance, its voting power is weighted by its effective balance, even if its actual balance is higher. Effective balance is also the amount on which all rewards, penalties, and slashings are calculated - it's used a lot in the protocol

The `MAX_EFFECTIVE_BALANCE` is the highest effective balance that a validator can have: 32 Ether. Any balance above this is ignored. Note that this means that staking rewards don't compound in the usual case (unless a validator's effective balance somehow falls below 32 Ether, in which case rewards kind of compound).

There is a discussion in the Design Rationale of why 32 Ether was chosen as the staking amount. In short, we want enough validators to keep the chain both alive and secure under attack, but not so many that the message overhead on the network becomes too high.

#### `EFFECTIVE_BALANCE_INCREMENT`

Throughout the protocol, a quantity called "effective balance" is used instead of the validators' actual balances. Effective balance tracks the actual balance, with two differences: (1) effective balance is capped at `MAX_EFFECTIVE_BALANCE` no matter how high the actual balance of a validator is, and (2) effective balance is much more granular - it changes only in steps of `EFFECTIVE_BALANCE_INCREMENT` rather than `Gwei`.

This discretisation of effective balance is intended to reduce the amount of hashing required when making state updates. The goal is to avoid having to re-calculate the hash tree root of validator records too often. Validators' actual balances, which change frequently, are stored as a contiguous list in BeaconState, outside of validators' records. Effective balances are stored inside validators' individual records, which are more costly to update (more hashing required). So we try to update effective balances relatively infrequently.

Effective balance is changed according to a process with hysteresis to avoid situations where it might change frequently. See `HYSTERESIS_QUOTIENT`.

You can read more about effective balance in the Design Rationale and in this article.

### Time parameters

| Name | Value | Unit | Duration |
|---|---|---|---|
| `MIN_ATTESTATION_INCLUSION_DELAY` | `uint64(2**0)` (= 1) | slots | 12 seconds |
| `SLOTS_PER_EPOCH` | `uint64(2**5)` (= 32) | slots | 6.4 minutes |
| `MIN_SEED_LOOKAHEAD` | `uint64(2**0)` (= 1) | epochs | 6.4 minutes |
| `MAX_SEED_LOOKAHEAD` | `uint64(2**2)` (= 4) | epochs | 25.6 minutes |
| `MIN_EPOCHS_TO_INACTIVITY_PENALTY` | `uint64(2**2)` (= 4) | epochs | 25.6 minutes |
| `EPOCHS_PER_ETH1_VOTING_PERIOD` | `uint64(2**6)` (= 64) | epochs | ~6.8 hours |
| `SLOTS_PER_HISTORICAL_ROOT` | `uint64(2**13)` (= 8,192) | slots | ~27 hours |

#### `MIN_ATTESTATION_INCLUSION_DELAY`

A design goal of Ethereum 2.0 is not to heavily disadvantage validators that are running on lower-spec systems, or, conversely, to reduce any advantage gained by running on high-spec systems.

One aspect of performance is network bandwidth. When a validator becomes the block proposer, it needs to gather attestations from the rest of its committee. On a low-bandwidth link, this takes longer, and could result in the proposer not being able to include as many past attestations as other better-connected validators might, thus receiving lower rewards.

`MIN_ATTESTATION_INCLUSION_DELAY` was an attempt to "level the playing field" by setting a minimum number of slots before an attestation can be included in a beacon block. It was originally set at 4, with a 6 second slot time, allowing 24 seconds for attestations to propagate around the network.

It was later set to one – attestations are included as early as possible – and, now that we plan to crosslink shards every slot, this is the only value that makes sense. So `MIN_ATTESTATION_INCLUSION_DELAY` exists today as a kind of relic of the earlier design.

The current slot time of 12 seconds is assumed to allow sufficient time for attestations to propagate and be aggregated sufficiently within one slot.

#### `SLOTS_PER_EPOCH`

We currently have 12 second slots and 32 slot epochs. In earlier designs slots were six seconds and there were 64 slots per epoch. So the time between epoch boundaries was unchanged when slots were lengthened.

The choice of 32 slots per epoch is a trade-off between time to finality (we need two epochs to finalise, so we prefer to keep them as short as we can) and being as certain as possible that at least one honest proposer per epoch will make a block to update the RANDAO (for which we prefer longer epochs).

In addition, epoch boundaries are where the heaviest part of the beacon chain state-transition calculation occurs, so that's another reason for not having them too close together.

Since every validator attests one every epoch, there is an interplay between the number of slots per epoch, the number of committees per slot, committee sizes, and the total number of validators.

#### `MIN_SEED_LOOKAHEAD`

A random seed is used to select all the committees and proposers for an epoch. During each epoch, the beacon chain accumulates randomness from proposers via the RANDAO and stores it. The seed for the current epoch is based on the RANDAO output from the epoch `MIN_SEED_LOOKAHEAD + 1` ago. With `MIN_SEED_LOOKAHEAD` set to one, the effect is that we can know the seed for the current epoch and the next epoch, but not beyond, since the next-but-one epoch depends on randomness from the current epoch that hasn't been accumulated yet.

This mechanism is designed to allow sufficient time for committee members to find each other on the peer-to-peer network, and in future to sync up any shard data they need. But preventing committee makeup being known too far ahead limits the opportunity for coordinated collusion between validators.

#### `MAX_SEED_LOOKAHEAD`

The above notwithstanding, if an attacker has a large proportion of the stake, or is, for example, able to DoS block proposers for a while, then it might be possible for the attacker to predict the output of the RANDAO further ahead than `MIN_SEED_LOOKAHEAD` would normally allow. This might enable the attacker to manipulate committee memberships to their advantage by performing well-timed exits and activations of their validators.

To prevent this, we assume a maximum feasible lookahead that an attacker might achieve (`MAX_SEED_LOOKAHEAD`) and delay all activations and exits by this amount, which allows new randomness to come in via block proposals from honest validators. With `MAX_SEED_LOOKAHEAD` set to 4, if only 10% of validators are online and honest, then the chance that an attacker can succeed in forecasting the seed beyond (`MAX_SEED_LOOKAHEAD - MIN_SEED_LOOKAHEAD`) = 3 epochs is $0.9^{3\times32}$, which is about 1 in 25,000.

#### `MIN_EPOCHS_TO_INACTIVITY_PENALTY`

The inactivity penalty is discussed below. This parameter sets the length of time until it kicks in. If the last finalised epoch is longer ago than `MIN_EPOCHS_TO_INACTIVITY_PENALTY`, then the beacon chain starts operating in "leak" mode. In this mode, participating validators no longer get rewarded, and validators that are not participating get penalised.

#### `EPOCHS_PER_ETH1_VOTING_PERIOD`

In order to safely onboard new validators, the beacon chain needs to take a view on what the Eth1 chain looks like. This is done by collecting votes from beacon block proposers - they are expected to consult an available Eth1 client in order to construct their vote.

`EPOCHS_PER_ETH1_VOTING_PERIOD * SLOTS_PER_EPOCH` is the total number of votes for Eth1 blocks that are collected. As soon as half of this number of votes are for the same Eth1 block, that block is adopted by the beacon chain and deposit processing can continue.

Rules for how validators select the right block to vote for are set out in the validator guide. `ETH1_FOLLOW_DISTANCE` is the (approximate) minimum depth of block that can be considered.

This parameter was increased from 32 to 64 epochs for the beacon chain mainnet. This increase is intended to allow devs more time to respond if there is any trouble on the Eth1 chain, in addition to the eight hours grace provided by `ETH1_FOLLOW_DISTANCE`.

For a detailed analysis of these parameters, see this article.

#### `SLOTS_PER_HISTORICAL_ROOT`

There have been several redesigns of the way the beacon chain stores its past history. The current design is a double batched accumulator. The block root and state root for every slot are stored in the state for `SLOTS_PER_HISTORICAL_ROOT` slots. When that list is full, both lists are Merkleized into a single Merkle root, which is added to the ever-growing `state.historical_roots` list.

### State list lengths

The following parameters set the sizes of some lists in the beacon chain state. Some lists have natural sizes, others such as the validator registry need an explicit maximum size to guide SSZ serialisation.

| Name | Value | Unit | Duration |
|---|---|---|---|
| EPOCHS_PER_HISTORICAL_VECTOR | uint64(2**16) (= 65,536) | epochs | ~0.8 years |
| EPOCHS_PER_SLASHINGS_VECTOR | uint64(2**13) (= 8,192) | epochs | ~36 days |
| HISTORICAL_ROOTS_LIMIT | uint64(2**24) (= 16,777,216) | historical roots | ~52,262 years |
| VALIDATOR_REGISTRY_LIMIT | uint64(2**40) (= 1,099,511,627,776) | validators | |

#### EPOCHS_PER_HISTORICAL_VECTOR

This is the number of epochs of previous RANDAO mixes that are stored (one per epoch). Having access to past randao mixes allows historical shufflings to be recalculated. Since Validator records keep track of the activation and exit epochs of all past validators, we can thus reconstitute past committees as far back as we have the RANDAO values. This information can be used for slashing long-past attestations, for example. It is not clear how the value of this parameter was decided.

#### EPOCHS_PER_SLASHINGS_VECTOR

In the epoch in which a misbehaving validator is slashed, its effective balance is added to an accumulator in the state. In this way, the `state.slashings` list tracks the total effective balance of all validators slashed during the last `EPOCHS_PER_SLASHINGS_VECTOR` epochs.

At a time `EPOCHS_PER_SLASHINGS_VECTOR // 2` after being slashed, a further penalty is applied to the slashed validator, based on the total amount of value slashed during the 4096 epochs before and the 4096 epochs after it was originally slashed.

The idea of this is to disproportionately punish coordinated attacks, in which many validators break the slashing conditions around the same time, while only lightly penalising validators that get slashed by making a mistake. Early designs for Eth2 would always slash a validator's entire deposit.

See also `PROPORTIONAL_SLASHING_MULTIPLIER_ALTAIR`.

#### HISTORICAL_ROOTS_LIMIT

Every `SLOTS_PER_HISTORICAL_ROOT` slots, the list of block roots and the list of state roots in the beacon state are Merkleized and added to `state.historical_roots` list. Although `state.historical_roots` is in principle unbounded, all SSZ lists must have maximum sizes specified. The size `HISTORICAL_ROOTS_LIMIT` will be fine for the next few millennia, after which it will be somebody else's problem. The list grows at less than 10 KB per year.

Storing past roots like this allows Merkle proofs to be constructed about anything in the beacon chain's history if required.

#### VALIDATOR_REGISTRY_LIMIT

Every time the Eth1 deposit contract processes a deposit from a new validator (as identified by its public key), a new entry is appended to the `state.validators` list.

In the current design, validators are never removed from this list, even after exiting from being a validator. This is largely because there is nowhere yet to send a validator's remaining deposit and staking rewards, so they continue to need to be tracked in the beacon chain.

The maximum length of this list is `VALIDATOR_REGISTRY_LIMIT`, which is one trillion, so we ought to be OK for a while, especially given that the minimum deposit amount is 1 Ether.

### Rewards and penalties

| Name | Value |
|---|---|
| `BASE_REWARD_FACTOR` | `uint64(2**6)` (= 64) |
| `WHISTLEBLOWER_REWARD_QUOTIENT` | `uint64(2**9)` (= 512) |
| `PROPOSER_REWARD_QUOTIENT` | `uint64(2**3)` (= 8) |
| `INACTIVITY_PENALTY_QUOTIENT` | `uint64(2**26)` (= 67,108,864) |
| `MIN_SLASHING_PENALTY_QUOTIENT` | `uint64(2**7)` (= 128) |
| `PROPORTIONAL_SLASHING_MULTIPLIER` | `uint64(1)` |
| `INACTIVITY_PENALTY_QUOTIENT_ALTAIR` | `uint64(3 * 2**24)` (= 50,331,648) |
| `MIN_SLASHING_PENALTY_QUOTIENT_ALTAIR` | `uint64(2**6)` (= 64) |
| `PROPORTIONAL_SLASHING_MULTIPLIER_ALTAIR` | `uint64(2)` |

Note that there are similar constants with different values here, one version with an `_ALTAIR` suffix. This is explained in the specs repo as follows:

> Variables are not replaced but extended with forks. This is to support syncing from one state to another over a fork boundary, without hot-swapping a config. Instead, for forks that introduce changes in a variable, the variable name is suffixed with the fork name.

So, the unsuffixed versions are the Phase 0 values, and the `_ALTAIR` suffixed versions are the values that apply to the current Altair fork.

#### `BASE_REWARD_FACTOR`

This is the big knob to turn to change the issuance rate of Eth2. Almost all validator rewards are calculated in terms of a "base reward per increment" which is formulated as,

```
EFFECTIVE_BALANCE_INCREMENT * BASE_REWARD_FACTOR // integer_squareroot(get_total_active_balance(state))
```

Thus, the total validator rewards per epoch (the Eth2 issuance rate) could be tuned by increasing or decreasing `BASE_REWARD_FACTOR`.

The exception is proposer rewards for including slashing reports in blocks. However, these are more than offset by the amount of stake burnt, so do not increase the overall issuance rate.

#### `WHISTLEBLOWER_REWARD_QUOTIENT`

One reward that is not tied to the base reward is the whistleblower reward. This is an amount awarded to the proposer of a block containing one or more proofs that a proposer or attester has violated a slashing condition. The whistleblower reward is set at $\frac{1}{512}$ of the effective balance of the slashed validator.

The whistleblower reward comes from new issuance of Ether on the beacon chain, but is more than offset by the Ether burned due to slashing penalties.

#### `PROPOSER_REWARD_QUOTIENT`

`PROPOSER_REWARD_QUOTIENT` was removed in the Altair upgrade in favour of `PROPOSER_WEIGHT`. It was used to apportion rewards between attesters and proposers when including attestations in blocks.

#### `INACTIVITY_PENALTY_QUOTIENT_ALTAIR`

This value supersedes `INACTIVITY_PENALTY_QUOTIENT`.

If the beacon chain hasn't finalised a checkpoint for longer than `MIN_EPOCHS_TO_INACTIVITY_PENALTY` epochs, then it enters "leak" mode. In this mode, any validator that does not vote (or votes for an

incorrect target) is penalised an amount each epoch of `(effective_balance * inactivity_score) // ( INACTIVITY_SCORE_BIAS * INACTIVITY_PENALTY_QUOTIENT_ALTAIR )`.

In Altair, `inactivity_score` is a per-validator quantity, whereas previously validators were penalised by a globally calculated amount when they missed a duty during a leak. See inactivity penalties for more on the rationale for this and how this score is calculated per validator.

During a leak, no validators receive rewards, and they continue to accrue the normal penalties when they fail to fulfil duties. In addition, for epochs in which validators do not make a correct, timely target vote, they receive a leak penalty.

To examine the effect of the leak on a single validator's balance, assume that during a period of inactivity leak (non-finalisation) the validator is completely offline. At each epoch, the offline validator will be penalised an amount $nB/\alpha$, where $n$ is the number of epochs since the leak started, $B$ is the validator's effective balance, and $\alpha$ is the prevailing `INACTIVITY_PENALTY_QUOTIENT`.

The effective balance $B$ will remain constant for a while, by design, during which time the total amount of the penalty after $n$ epochs would be $n(n+1)B/2\alpha$: the famous "quadratic leak". If $B$ were continuously variable, the penalty would satisfy $\frac{dB}{dt} = -\frac{Bt}{\alpha}$, which can be solved to give $B(t) = B_0 e^{-t^2/2\alpha}$. The actual behaviour is somewhere between these two since the effective balance decreases in a step-wise fashion.

In the continuous case, the `INACTIVITY_PENALTY_QUOTIENT`, $\alpha$, is the square of the time it takes to reduce the balance of a non-participating validator to $1/\sqrt{e}$, or around 60.7% of its initial value. With the value of `INACTIVITY_PENALTY_QUOTIENT_ALTAIR` at `3 * 2**24`, this equates to around seven thousand epochs, or 31.5 days.

The idea for the inactivity leak (aka the quadratic leak) was proposed in the original Casper FFG paper. The problem it addresses is that, if a large fraction of the validator set were to go offline at the same time, it would not be possible to continue finalising checkpoints, since a majority vote from validators representing 2/3 of the total stake is required for finalisation.

In order to recover, the inactivity leak gradually reduces the stakes of validators who are not making attestations until, eventually, the participating validators control 2/3 of the remaining stake. They can then begin to finalise checkpoints once again.

This inactivity penalty mechanism is designed to protect the chain long-term in the face of catastrophic events (sometimes referred to as the ability to survive World War III). The result might be that the beacon chain could permanently split into two independent chains either side of a network partition, and this is assumed to be a reasonable outcome for any problem that can't be fixed in a few weeks. In this sense, the beacon chain formally prioritises availability over consistency. (You can't have both.)

The value of `INACTIVITY_PENALTY_QUOTIENT` was increased by a factor of four from `2**24` to `2**26` for the beacon chain launch, with the intention of penalising validators less severely in case of non-finalisation due to implementation problems in the early days. As it happens, there were no instances of non-finalisation during the eleven months of Phase 0 of the beacon chain.

The value was decreased by one quarter in the Altair upgrade from `2**26` to `3 * 2**24` as a step towards eventually setting it to its final value. Decreasing the inactivity penalty quotient speeds up recovery of finalisation in the event of an inactivity leak.

#### `MIN_SLASHING_PENALTY_QUOTIENT_ALTAIR`

When a validator is first convicted of a slashable offence, an initial penalty is applied. This is calculated as, `validator.effective_balance // MIN_SLASHING_PENALTY_QUOTIENT_ALTAIR`.

Thus, the initial slashing penalty is between 0.25 Ether and 0.5 Ether depending on the validator's effective balance (which is between 16 and 32 Ether; note that effective balance is denominated in Gwei).

A further slashing penalty is applied later based on the total amount of balance slashed during a period of `EPOCHS_PER_SLASHINGS_VECTOR`.

The value of `MIN_SLASHING_PENALTY_QUOTIENT` was increased by a factor of four from `2**5` to `2**7` for the beacon chain launch, anticipating that unfamiliarity with the rules of Ethereum 2.0 staking was likely to result in some unwary users getting slashed. In the event, a total of 157 validators were slashed during Phase 0, all as a result of user error or misconfiguration as far as can be determined.

The value was halved in the Altair upgrade from `2**7` to `2**6` as a step towards eventually setting it to its final value of `2**5`.

#### `PROPORTIONAL_SLASHING_MULTIPLIER_ALTAIR`

When a validator has been slashed, a further penalty is later applied to the validator based on how many other validators were slashed during a window of size `EPOCHS_PER_SLASHINGS_VECTOR` epochs centred on that slashing event (approximately 18 days before and after).

The proportion of the validator's remaining effective balance that will be subtracted is calculated as, `PROPORTIONAL_SLASHING_MULTIPLIER` multiplied by the sum of the effective balances of the slashed validators in the window, divided by the total effective balance of all validators. The idea of this mechanism is to punish accidents lightly (in which only a small number of validators were slashed) and attacks heavily (where many validators coordinated to double vote).

To finalise conflicting checkpoints, at least a third of the balance must have voted for both. That's why the "natural" setting of `PROPORTIONAL_SLASHING_MULTIPLIER` is three: in the event of an attack that finalises conflicting checkpoints, the attackers lose their entire stake. This provides "the maximal minimum accountable safety margin".

However, for the initial stage of the beacon chain, Phase 0, `PROPORTIONAL_SLASHING_MULTIPLIER` was set to one, and increased to two at the Altair upgrade. These lower values provide some insurance against client bugs that might cause mass slashings in the early days. It will eventually be increased to its final value of three in a later upgrade.

### Max operations per block

| Name | Value |
|---|---|
| `MAX_PROPOSER_SLASHINGS` | `2**4` (= 16) |
| `MAX_ATTESTER_SLASHINGS` | `2**1` (= 2) |
| `MAX_ATTESTATIONS` | `2**7` (= 128) |
| `MAX_DEPOSITS` | `2**4` (= 16) |
| `MAX_VOLUNTARY_EXITS` | `2**4` (= 16) |

These parameters are used to size lists in the beacon block bodies for the purposes of SSZ serialisation, as well as constraining the maximum size of beacon blocks so that they can propagate efficiently, and avoid DoS attacks.

Some comments on the chosen values:

- I have suggested elsewhere reducing `MAX_DEPOSITS` from sixteen to one to ensure that more validators must process deposits, which encourages them to run Eth1 clients.
- At first sight, there looks to be a disparity between the number of proposer slashings and the number of attester slashings that may be included in a block. But note that an attester slashing (a) can be much larger than a proposer slashing, and (b) can result in many more validators getting slashed than a proposer slashing.
- `MAX_ATTESTATIONS` is double the value of `MAX_COMMITTEES_PER_SLOT`. This allows there to be an empty slot (with no block proposal), yet still include all the attestations for the empty slot in the next slot. Since, ideally, each committee produces a single aggregate attestation, a block can hold two slots' worth of aggregate attestations.

### Sync committee

| Name | Value | Unit | Duration |
|---|---|---|---|
| `SYNC_COMMITTEE_SIZE` | `uint64(2**9)` (= 512) | Validators | |

| Name | Value | Unit | Duration |
|---|---|---|---|
| `EPOCHS_PER_SYNC_COMMITTEE_PERIOD` | `uint64(2**8)` (= 256) | epochs | ~27 hours |

Sync committees were introduced by the Altair upgrade to allow light clients to quickly and trustlessly determine the head of the beacon chain.

Why did we need a new committee type? Couldn't this be built on top of existing committees, say committees 0 to 3 at a slot? After all, voting for the head of the chain is already one of their duties. The reason is that it is important for reducing the load on light clients that sync committees do not change very often. Standard committees change every slot; we need something much longer lived here.

Only a single sync committee is active at any one time, and contains a randomly selected subset of size `SYNC_COMMITTEE_SIZE` of the total validator set.

A sync committee does its duties (and receives rewards for doing so) for only `EPOCHS_PER_SYNC_COMMITTEE_PERIOD` epochs until the next committee takes over.

With 262,144 validators ($2^{18}$), the expected time between being selected for sync committee duty is over 19 months. The probability of being in the current sync committee would be 1/512 per validator.

`SYNC_COMMITTEE_SIZE` is a trade-off between security (ensuring that enough honest validators are always present) and efficiency for light clients (ensuring that they do not have to handle too much computation). The value 512 is conservative in terms of safety. It would be catastrophic for trustless bridges to other protocols, for example, if a sync committee voted in an invalid block.

`EPOCHS_PER_SYNC_COMMITTEE_PERIOD` is around a day, and again is a trade-off between security (short enough that it's hard for an attacker to find and corrupt committee members) and efficiency (reducing the data load on light clients).

## Configuration

### Genesis Settings

With Altair, beacon chain genesis is long behind us. Nevertheless, the ability to spin-up testnets is useful in all sorts of scenarios, so the Altair spec retains genesis functionality, now called initialisation.

The following parameters refer to the actual mainnet beacon chain genesis and I'll explain them in that context. When starting up new testnets, these will of course be changed. For example, see the configuration file for the Prater testnet.

| Name | Value |
|---|---|
| `MIN_GENESIS_ACTIVE_VALIDATOR_COUNT` | `uint64(2**14)` (= 16,384) |
| `MIN_GENESIS_TIME` | `uint64(1606824000)` (Dec 1, 2020, 12pm UTC) |
| `GENESIS_FORK_VERSION` | `Version('0x00000000')` |
| `GENESIS_DELAY` | `uint64(604800)` (7 days) |

#### `MIN_GENESIS_ACTIVE_VALIDATOR_COUNT`

`MIN_GENESIS_ACTIVE_VALIDATOR_COUNT` is the minimum number of full validator stakes that must have been deposited before the beacon chain can start producing blocks. The number is chosen to ensure a degree of security. It allows for four 128 member committees per slot, rather than the 64 committees per slot eventually desired to support fully operational data shards. Fewer validators means higher rewards per validator, so it is designed to attract early participants to get things bootstrapped.

`MIN_GENESIS_ACTIVE_VALIDATOR_COUNT` used to be much higher (65,536 = 2 million Ether staked), but was reduced when `MIN_GENESIS_TIME`, below, was added.

In the actual event of beacon chain genesis, there were 21,063 participating validators, comfortably

exceeding the minimum necessary count.

#### MIN_GENESIS_TIME

MIN_GENESIS_TIME is the earliest date that the beacon chain can start.

Having a MIN_GENESIS_TIME allows us to start the chain with fewer validators than was previously thought necessary. The previous plan was to start the chain as soon as there were MIN_GENESIS_ACTIVE_VALIDATOR_COUNT validators staked. But there were concerns that with a lowish initial validator count, a single entity could form the majority of them and then act to prevent other validators from entering (a "gatekeeper attack"). A minimum genesis time allows time for all those who wish to make deposits to do so before they could be excluded by a gatekeeper attack.

The beacon chain actually started at 12:00:23 UTC on the 1st of December 2020. The extra 23 seconds comes from the timestamp of the first Eth1 block to meet the genesis criteria, block 11320899. I like to think of this as a little remnant of proof of work forever embedded in the beacon chain's history.

#### GENESIS_FORK_VERSION

Unlike Ethereum 1.0, the beacon chain gives in-protocol versions to its forks. See the Version custom type for more explanation.

GENESIS_FORK_VERSION is the fork version the beacon chain starts with at its "genesis" event: the point at which the chain first starts producing blocks. In Altair, this value is used only when computing the cryptographic domain for deposit messages, which are valid across all forks.

ALTAIR_FORK_VERSION is defined elsewhere.

#### GENESIS_DELAY

The GENESIS_DELAY is a grace period to allow nodes and node operators time to prepare for the genesis event. The genesis event cannot occur before MIN_GENESIS_TIME. If there are not MIN_GENESIS_ACTIVE_VALIDATOR_COUNT registered validators sufficiently in advance of MIN_GENESIS_TIME, then Genesis will occur GENESIS_DELAY seconds after enough validators have been registered.

Seven days' notice was regarded as sufficient to allow client dev teams time to make a release once the genesis parameters were known, and for node operators to upgrade to that release. And, of course, to organise some parties. It was increased from 2 days over time due to lessons learned on some of the pre-genesis testnets.

### Time parameters

| Name | Value | Unit | Duration |
|---|---|---|---|
| SECONDS_PER_SLOT | uint64(12) | seconds | 12 seconds |
| SECONDS_PER_ETH1_BLOCK | uint64(14) | seconds | 14 seconds |
| MIN_VALIDATOR_WITHDRAWABILITY_DELAY | uint64(2**8) (= 256) | epochs | ~27 hours |
| SHARD_COMMITTEE_PERIOD | uint64(2**8) (= 256) | epochs | ~27 hours |
| ETH1_FOLLOW_DISTANCE | uint64(2**11) (= 2,048) | Eth1 blocks | ~8 hours |

#### `SECONDS_PER_SLOT`

This was originally six seconds, but is now twelve, and has been other values in between.

Network delays are the main limiting factor in shortening the slot length. Three communication activities need to be accomplished within a slot, and it is supposed that four seconds is enough for the vast majority of nodes to have participated in each:

1. Blocks are proposed at the start of a slot and should have propagated to most of the network within the first four seconds.
2. At four seconds into a slot, committee members create and broadcast attestations, including attesting to this slot's block. During the next four seconds, these attestations are collected by aggregators in each committee.
3. At eight seconds into the slot, the aggregators broadcast their aggregate attestations which then have four seconds to reach the validator who is proposing the next block.

This slot length has to account for shard blocks as well in later phases. There was some discussion around having the beacon chain and shards on differing cadences, but the latest sharding design tightly couples the beacon chain with the shards. Shard blocks under this design will be much larger, which led to the extension of the slot to 12 seconds.

There is a general intention to shorten the slot time in future, perhaps to 8 seconds, if it proves possible to do this in practice. Or perhaps to lengthen it to 16 seconds.

#### `SECONDS_PER_ETH1_BLOCK`

The assumed block interval on the Eth1 chain, used in conjunction with `ETH1_FOLLOW_DISTANCE` when considering blocks on the Eth1 chain, either at genesis, or when voting on the deposit contract state.

The average Eth1 block time since January 2020 has actually been nearer 13 seconds, but never mind. The net effect is that we will be going a little deeper back in the Eth1 chain than `ETH1_FOLLOW_DISTANCE` would suggest, which ought to be safer.

#### `MIN_VALIDATOR_WITHDRAWABILITY_DELAY`

A validator can stop participating once it has made it through the exit queue. However, its funds remain locked for the duration of `MIN_VALIDATOR_WITHDRAWABILITY_DELAY`. Initially, this is to allow some time for any slashable behaviour to be detected and reported so that the validator can still be penalised (in which case the validator's withdrawable time is pushed `EPOCHS_PER_SLASHINGS_VECTOR` into the future). When data shards are introduced this delay will also allow for shard rewards to be credited and for proof of custody challenges to be mounted.

Note that, for the time being, there is no mechanism to withdraw a validator's balance in any case. Nonetheless, being in a "withdrawable" state means that a validator has now fully exited from the protocol.

#### `SHARD_COMMITTEE_PERIOD`

This really anticipates the implementation of data shards. The idea is that it's bad for the stability of longer-lived committees if validators can appear and disappear very rapidly. Therefore, a validator cannot initiate a voluntary exit until `SHARD_COMMITTEE_PERIOD` epochs after it is activated. Note that it could still be ejected by slashing before this time.

#### `ETH1_FOLLOW_DISTANCE`

This is used to calculate the minimum depth of block on the Ethereum 1 chain that can be considered by the Eth2 chain: it applies to the Genesis process and the processing of deposits by validators. The Eth1 chain depth is estimated by multiplying this value by the target average Eth1 block time, `SECONDS_PER_ETH1_BLOCK`.

The value of `ETH1_FOLLOW_DISTANCE` is not based on the expected depth of any reorgs of the Eth1 chain, which are rarely if ever more than 2-3 blocks deep. It is about providing time to respond to an incident on the Eth1 chain such as a consensus failure between clients.

This parameter was increased from 1024 to 2048 blocks for the beacon chain mainnet, to allow devs more time to respond if there were any trouble on the Eth1 chain.

### Validator Cycle

| Name | Value |
|---|---|
| `EJECTION_BALANCE` | `Gwei(2**4 * 10**9)` (= 16,000,000,000) |
| `MIN_PER_EPOCH_CHURN_LIMIT` | `uint64(2**2)` (= 4) |
| `CHURN_LIMIT_QUOTIENT` | `uint64(2**16)` (= 65,536) |

#### `EJECTION_BALANCE`

If a validator's effective balance falls to 16 Ether or below then it is exited from the system. This is most likely to happen as a result of the "inactivity leak", which gradually reduces the balances of inactive validators in order to maintain the liveness of the beacon chain.

Note that the dependence on effective balance means that the validator is queued for ejection as soon as its actual balance falls to 16.75 Ether.

#### `MIN_PER_EPOCH_CHURN_LIMIT`

Validators are allowed to exit the system and cease validating, and new validators may apply to join at any time. For interesting reasons, a design decision was made to apply a rate-limit to entries (activations) and exits. Basically, it is important in proof of stake protocols that the validator set not change too quickly.

In the normal case, a validator is able to exit fairly swiftly: it just needs to wait `MAX_SEED_LOOKAHEAD` (currently four) epochs. However, if there are large numbers of validators wishing to exit at the same time, a queue forms with a limited number of exits allowed per epoch. The minimum number of exits per epoch (the minimum "churn") is `MIN_PER_EPOCH_CHURN_LIMIT`, so that validators can always eventually exit. The actual allowed churn per epoch is calculated in conjunction with `CHURN_LIMIT_QUOTIENT`.

The same applies to new validator activations, once a validator has been marked as eligible for activation.

In concrete terms, this means that up to four validators can enter or exit the active validator set each epoch (900 per day) until we have 327,680 active validators, at which point the limit rises to five.

The rate at which validators can exit is strongly related to the concept of weak subjectivity, and the weak subjectivity period.

#### `CHURN_LIMIT_QUOTIENT`

This is used in conjunction with `MIN_PER_EPOCH_CHURN_LIMIT` to calculate the actual number of validator exits and activations allowed per epoch. The number of exits allowed is `max(MIN_PER_EPOCH_CHURN_LIMIT, n // CHURN_LIMIT_QUOTIENT)`, where `n` is the number of active validators. The same applies to activations.

### Inactivity penalties

| Name | Value | Description |
|---|---|---|
| `INACTIVITY_SCORE_BIAS` | `uint64(2**2)` (= 4) | score points per inactive epoch |
| `INACTIVITY_SCORE_RECOVERY_RATE` | `uint64(2**4)` (= 16) | score points per leak-free epoch |

#### `INACTIVITY_SCORE_BIAS`

If the beacon chain hasn't finalised an epoch for longer than `MIN_EPOCHS_TO_INACTIVITY_PENALTY` epochs, then it enters "leak" mode. In this mode, any validator that does not vote (or votes for an incorrect target) is penalised an amount each epoch of `(effective_balance * inactivity_score) // (INACTIVITY_SCORE_`

`BIAS * INACTIVITY_PENALTY_QUOTIENT_ALTAIR)`. See `INACTIVITY_PENALTY_QUOTIENT_ALTAIR` for discussion of the inactivity leak itself.

The per-validator `inactivity-score` is new in Altair. During Phase 0, inactivity penalties were an increasing global amount applied to all validators that did not participate in an epoch, regardless of their individual track records of participation. So a validator that was able to participate for a significant fraction of the time nevertheless could be quite severely penalised due to the growth of the per-epoch inactivity penalty. Vitalik gives a simplified example: "if fully [off]line validators get leaked and lose 40% of their balance, someone who has been trying hard to stay online and succeeds at 90% of their duties would still lose 4% of their balance. Arguably this is unfair."

In addition, if many validators are able to participate intermittently, it indicates that whatever event has befallen the chain is potentially recoverable (unlike a permanent network partition, or a super-majority network fork, for example). The inactivity leak is intended to bring finality to irrecoverable situations, so prolonging the time to finality if it's not irrecoverable is likely a good thing.

With Altair, each validator has an individual inactivity score in the beacon state which is updated by `process_inactivity_updates()` as follows.

- Every epoch, irrespective of the inactivity leak,
  - decrease the score by one when the validator makes a correct timely target vote, and
  - increase the score by `INACTIVITY_SCORE_BIAS` otherwise.
- When *not* in an inactivity leak
  - decrease every validator's score by `INACTIVITY_SCORE_RECOVERY_RATE`.

There is a floor of zero on the score. So, outside a leak, validators' scores will rapidly return to zero and stay there, since `INACTIVITY_SCORE_RECOVERY_RATE` is greater than `INACTIVITY_SCORE_BIAS`.

When in a leak, if $p$ is the participation rate between 0 and 1, and $\lambda$ is `INACTIVITY_SCORE_BIAS`, then the expected score after $N$ epochs is $\max(0, N((1-p)\lambda - p))$. For $\lambda = 4$ this is $\max(0, N(4-5p))$. So a validator that is participating 80% of the time or more can maintain a score that is bounded near zero. With less than 80% average participation, its score will increase unboundedly.

#### `INACTIVITY_SCORE_RECOVERY_RATE`

When not in an inactivity leak, validators' inactivity scores are reduced by `INACTIVITY_SCORE_RECOVERY_RATE + 1` per epoch when they make a timely target vote, and by `INACTIVITY_SCORE_RECOVERY_RATE - INACTIVITY_SCORE_BIAS` when they don't. So, even for non-performing validators, scores decrease three times faster than they increase.

The new scoring system means that some validators will continue to be penalised due to the leak, even after finalisation starts again. This is intentional. When the leak causes the beacon chain to finalise, at that point we have just 2/3 of the stake online. If we immediately stop the leak (as we used to), then the amount of stake online would remain close to 2/3 and the chain would be vulnerable to flipping in and out of finality as small numbers of validators come and go. We saw this behaviour on some of the testnets prior to launch. Continuing the leak after finalisation serves to increase the balances of participating validators to greater than 2/3, providing a margin that should help to prevent such behaviour.

See the section on the Inactivity Leak for some more analysis of the inactivity score and some graphs of its effect.

# Containers

## Preamble

We are about to see our first Python code in the executable spec. For specification purposes, these Container data structures are just Python data classes that are derived from the base SSZ `Container` class.

SSZ is the serialisation and Merkleization format used everywhere in Ethereum 2.0. It is not self-describing, so you need to know ahead of time what you are unpacking when deserialising. SSZ deals with basic types and composite types. Classes like the below are handled as SSZ containers, a composite type defined as an "ordered heterogeneous collection of values".

Client implementations in different languages will obviously use their own paradigms to represent these data structures.

Two notes directly from the spec:

- The definitions are ordered topologically to facilitate execution of the spec.
- Fields missing in container instantiations default to their zero value.

## Misc dependencies

**`Fork`**

```
class Fork(Container):
    previous_version: Version
    current_version: Version
    epoch: Epoch  # Epoch of latest fork
```

Fork data is stored in the BeaconState to indicate the current and previous fork versions. The fork version gets incorporated into the cryptographic domain in order to invalidate messages from validators on other forks. The previous fork version and the epoch of the change are stored so that pre-fork messages can still be validated (at least until the next fork). This ensures continuity of attestations across fork boundaries.

Note that this is all about planned protocol forks (upgrades), and nothing to do with the fork-choice rule, or inadvertent forks due to errors in the state transition.

**`ForkData`**

```
class ForkData(Container):
    current_version: Version
    genesis_validators_root: Root
```

`ForkData` is used only in `compute_fork_data_root()`. This is used when distinguishing between chains for the purpose of peer-to-peer gossip, and for domain separation. By including both the current fork version and the genesis validators root, we can cleanly distinguish between, say, mainnet and a testnet. Even if they have the same fork history, the genesis validators roots will differ.

`Version` is the datatype for a fork version number.

**`Checkpoint`**

```
class Checkpoint(Container):
    epoch: Epoch
    root: Root
```

`Checkpoints` are the points of justification and finalisation used by the Casper FFG protocol. Validators use them to create `AttestationData` votes, and the status of recent checkpoints is recorded in `BeaconState`.

As per the Casper paper, checkpoints contain a height, and a block root. In this implementation of Casper FFG, checkpoints occur whenever the slot number is a multiple of `SLOTS_PER_EPOCH`, thus they correspond to `epoch` numbers. In particular, checkpoint $N$ is the first slot of epoch $N$. The genesis block is checkpoint 0, and starts off both justified and finalised.

Thus, the `root` element here is the block root of the first block in the `epoch`. (This might be the block root of an earlier block if some slots have been skipped, that is, if there are no blocks for those slots.).

It is very common to talk about justifying and finalising epochs. This is not strictly correct: checkpoints are justified and finalised.

Once a checkpoint has been finalised, the slot it points to and all prior slots will never be reverted.

**`Validator`**

```
class Validator(Container):
    pubkey: BLSPubkey
    withdrawal_credentials: Bytes32  # Commitment to pubkey for withdrawals
    effective_balance: Gwei  # Balance at stake
    slashed: boolean
    # Status epochs
    activation_eligibility_epoch: Epoch  # When criteria for activation were met
    activation_epoch: Epoch
    exit_epoch: Epoch
    withdrawable_epoch: Epoch  # When validator can withdraw funds
```

This is the data structure that stores most of the information about an individual validator, with only validators' balances and inactivity scores stored elsewhere.

Validators' actual balances are stored separately in the `BeaconState` structure, and only the slowly changing "effective balance" is stored here. This is because actual balances are liable to change quite frequently (every epoch): the Merkleization process used to calculate state roots means that only the parts that change need to be recalculated; the roots of unchanged parts can be cached. Separating out the validator balances potentially means that only 1/15th (8/121) as much data needs to be rehashed every epoch compared to storing them here, which is an important optimisation.

For similar reasons, validators' inactivity scores are stored outside validator records as well, as they are also updated every epoch.

A validator's record is created when its deposit is first processed. Sending multiple deposits does not create multiple validator records: deposits with the same public key are aggregated in one record. Validator records never expire; they are stored permanently, even after the validator has exited the system. Thus there is a 1:1 mapping between a validator's index in the list and the identity of the validator (validator records are only ever appended to the list).

Also stored in `Validator`:

- `pubkey` serves as both the unique identity of the validator and the means of cryptographically verifying messages purporting to have been signed by it. The public key is stored raw, unlike in Eth1, where it is hashed to form the account address. This allows public keys to be aggregated for verifying aggregated attestations.
- Validators actually have two private/public key pairs, the one above used for signing protocol messages, and a separate "withdrawal key". `withdrawal_credentials` is a commitment generated from the validator's withdrawal key so that, at some time in the future, a validator can prove it owns the funds and will be able to withdraw them. There are two types of withdrawal credential currently defined, one corresponding to BLS keys, and one corresponding to standard Ethereum ECDSA keys.
- `effective_balance` is a topic of its own that we've touched upon already, and will discuss more fully when we look at effective balances updates.
- `slashed` indicates that a validator has been slashed, that is, punished for violating the slashing conditions. A validator can be slashed only once.
- The remaining values are the epochs in which the validator changed, or is due to change state.

A detailed explanation of the stages in a validator's lifecycle is here, and we'll be covering it in detail as we work through the beacon chain logic. But, in simplified form, progress is as follows:

1. A 32 ETH deposit has been made on the Ethereum 1 chain. No validator record exists yet.

2. The deposit is processed by the beacon chain at some slot. A validator record is created with all epoch fields set to `FAR_FUTURE_EPOCH`.
3. At the end of the current epoch, the `activation_eligibility_epoch` is set to the next epoch.
4. After the epoch `activation_eligibility_epoch` has been finalised, the validator is added to the activation queue by setting its `activation_epoch` appropriately, taking into account the per-epoch churn limit and `MAX_SEED_LOOKAHEAD`.
5. On reaching `activation_epoch` the validator becomes active, and should carry out its duties.
6. At any time after `SHARD_COMMITTEE_PERIOD` epochs have passed, a validator may request a voluntary exit. `exit_epoch` is set according to the validator's position in the exit queue and `MAX_SEED_LOOKAHEAD`, and `withdrawable_epoch` is set `MIN_VALIDATOR_WITHDRAWABILITY_DELAY` epochs after that.
7. From `exit_epoch` onward the validator is no longer active. There is no mechanism for exited validators to rejoin: exiting is permanent.
8. After `withdrawable_epoch`, the validator's balance can in principle be withdrawn, although there is no mechanism for doing this for the time being.

The above does not account for slashing or forced exits due to low balance.

#### `AttestationData`

```
class AttestationData(Container):
    slot: Slot
    index: CommitteeIndex
    # LMD GHOST vote
    beacon_block_root: Root
    # FFG vote
    source: Checkpoint
    target: Checkpoint
```

The beacon chain relies on a combination of two different consensus mechanisms: LMD GHOST keeps the chain moving, and Casper FFG brings finalisation. These are documented in the Gasper paper. Attestations from (committees of) validators are used to provide votes simultaneously for each of these consensus mechanisms.

This container is the fundamental unit of attestation data. It provides the following elements.

- `slot`: each active validator should be making exactly one attestation per epoch. Validators have an assigned slot for their attestation, and it is recorded here for validation purposes.
- `index`: there can be several committees active in a single slot. This is the number of the committee that the validator belongs to in that slot. It can be used to reconstruct the committee to check that the attesting validator is a member. Ideally, all (or the majority at least) of the attestations in a slot from a single committee will be identical, and can therefore be aggregated into a single aggregate attestation.
- `beacon_block_root` is the validator's vote on the head block for that slot after locally running the LMD GHOST fork-choice rule. It may be the root of a block from a previous slot if the validator believes that the current slot is empty.
- `source` is the validator's opinion of the best currently justified checkpoint for the Casper FFG finalisation process.
- `target` is the validator's opinion of the block at the start of the current epoch, also for Casper FFG finalisation.

This `AttestationData` structure gets wrapped up into several other similar but distinct structures:

- `Attestation` is the form in which attestations normally make their way around the network. It is signed and aggregatable, and the list of validators making this attestation is compressed into a bitlist.

- `IndexedAttestation` is used primarily for attester slashing. It is signed and aggregated, with the list of attesting validators being an uncompressed list of indices.
- `PendingAttestation`. In Phase 0, after having their validity checked during block processing, `PendingAttestation`s were stored in the beacon state pending processing at the end of the epoch. This was reworked in the Altair upgrade and `PendingAttestation`s are no longer used.

#### `IndexedAttestation`

```
class IndexedAttestation(Container):
    attesting_indices: List[ValidatorIndex, MAX_VALIDATORS_PER_COMMITTEE]
    data: AttestationData
    signature: BLSSignature
```

This is one of the forms in which aggregated attestations – combined identical attestations from multiple validators in the same committee – are handled.

`Attestation`s and `IndexedAttestation`s contain essentially the same information. The difference being that the list of attesting validators is stored uncompressed in `IndexedAttestation`s. That is, each attesting validator is referenced by its global validator index, and non-attesting validators are not included. To be valid, the validator indices must be unique and sorted, and the signature must be an aggregate signature from exactly the listed set of validators.

`IndexedAttestation`s are primarily used when reporting attester slashing. An `Attestation` can be converted to an `IndexedAttestation` using `get_indexed_attestation()`.

#### `PendingAttestation`

```
class PendingAttestation(Container):
    aggregation_bits: Bitlist[MAX_VALIDATORS_PER_COMMITTEE]
    data: AttestationData
    inclusion_delay: Slot
    proposer_index: ValidatorIndex
```

`PendingAttestation`s were removed in the Altair upgrade and now appear only in the process for upgrading the state during the fork. The following is provided for historical reference.

Prior to Altair, `Attestation`s received in blocks were verified then temporarily stored in beacon state in the form of `PendingAttestation`s, pending further processing at the end of the epoch.

A `PendingAttestation` is an `Attestation` minus the signature, plus a couple of fields related to reward calculation:

- `inclusion_delay` is the number of slots between the attestation having been made and it being included in a beacon block by the block proposer. Validators are rewarded for getting their attestations included in blocks, but the reward used to decline in inverse proportion to the inclusion delay. This incentivised swift attesting and communicating by validators.
- `proposer_index` is the block proposer that included the attestation. The block proposer gets a micro reward for every validator's attestation it includes, not just for the aggregate attestation as a whole. This incentivises efficient finding and packing of aggregations, since the number of aggregate attestations per block is capped.

Taken together, these rewards are designed to incentivise the whole network to collaborate to do efficient attestation aggregation (proposers want to include only well-aggregated attestations; validators want to get their attestations included, so will ensure that they get well aggregated).

With Altair, this whole mechanism has been replaced by `ParticipationFlags`.

#### `Eth1Data`

```
class Eth1Data(Container):
    deposit_root: Root
    deposit_count: uint64
    block_hash: Hash32
```

Proposers include their view of the Ethereum 1 chain in blocks, and this is how they do it. The beacon chain stores these votes up in the beacon state until there is a simple majority consensus, then the winner is committed to beacon state. This is to allow the processing of Eth1 deposits, and creates a simple "honest-majority" one-way bridge from Eth1 to Eth2. The 1/2 majority assumption for this (rather than 2/3 for committees) is considered safe as the number of validators voting each time is large: `EPOCHS_PER_ETH1_VOTING_PERIOD` * `SLOTS_PER_EPOCH` = 64 * 32 = 2048.

- `deposit_root` is the result of the `get_deposit_root()` method of the Eth1 deposit contract after executing the Eth1 block being voted on - it's the root of the (sparse) Merkle tree of deposits.
- `deposit_count` is the number of deposits in the deposit contract at that point, the result of the `get_deposit_count` method on the contract. This will be equal to or greater than (if there are pending unprocessed deposits) the value of `state.eth1_deposit_index`.
- `block_hash` is the hash of the Eth1 block being voted for. This doesn't have any current use within the Eth2 protocol, but is "too potentially useful to not throw in there", to quote Danny Ryan.

#### `HistoricalBatch`

```
class HistoricalBatch(Container):
    block_roots: Vector[Root, SLOTS_PER_HISTORICAL_ROOT]
    state_roots: Vector[Root, SLOTS_PER_HISTORICAL_ROOT]
```

This is used to implement part of the double batched accumulator for the past history of the chain. Once `SLOTS_PER_HISTORICAL_ROOT` block roots and the same number of state roots have been accumulated in the beacon state, they are put in a `HistoricalBatch` object and the hash tree root of that is appended to the `historical_roots` list in beacon state. The corresponding block and state root lists in the beacon state are circular and just get overwritten in the next period. See `process_historical_roots_update()`.

#### `DepositMessage`

```
class DepositMessage(Container):
    pubkey: BLSPubkey
    withdrawal_credentials: Bytes32
    amount: Gwei
```

The basic information necessary to either add a validator to the registry, or to top up an existing validator's stake.

`pubkey` is the unique public key of the validator. If it is already present in the registry (the list of validators in beacon state) then `amount` is added to its balance. Otherwise a new `Validator` entry is appended to the list and credited with `amount`.

See the `Validator` container for more on `withdrawal_credentials`.

There are two protections that `DepositMessages` get at different points.

1. `DepositData` is included in beacon blocks as a `Deposit`, which adds a Merkle proof that the data has been registered with the Eth1 deposit contract.
2. When the containing beacon block is processed, deposit messages are stored, pending processing at the end of the epoch, in the beacon state as `DepositData`. This includes the pending validator's BLS signature so that the authenticity of the `DepositMessage` can be verified before a validator is added.

#### `DepositData`

```
class DepositData(Container):
    pubkey: BLSPubkey
    withdrawal_credentials: Bytes32
    amount: Gwei
    signature: BLSSignature  # Signing over DepositMessage
```

A signed DepositMessage. The comment says that the signing is done over DepositMessage. What actually happens is that a DepositMessage is constructed from the first three fields; the root of that is combined with DOMAIN_DEPOSIT in a SigningData object; finally the root of this is signed and included in DepositData.

**BeaconBlockHeader**

```
class BeaconBlockHeader(Container):
    slot: Slot
    proposer_index: ValidatorIndex
    parent_root: Root
    state_root: Root
    body_root: Root
```

A standalone version of a beacon block header: BeaconBlocks contain their own header. It is identical to BeaconBlock, except that body is replaced by body_root. It is BeaconBlock-lite.

BeaconBlockHeader is stored in beacon state to record the last processed block header. This is used to ensure that we proceed along a continuous chain of blocks that always point to their predecessor[41]. See process_block_header().

The signed version is used in proposer slashings.

**SyncCommittee**

```
class SyncCommittee(Container):
    pubkeys: Vector[BLSPubkey, SYNC_COMMITTEE_SIZE]
    aggregate_pubkey: BLSPubkey
```

Sync committees were introduced in the Altair upgrade to support light clients to the beacon chain protocol. The list of committee members for each of the current and next sync committees is stored in the beacon state. Members are updated every EPOCHS_PER_SYNC_COMMITTEE_PERIOD epochs by get_next_sync_committee().

Including the aggregate_pubkey of the sync committee is an optimisation intended to save light clients some work when verifying the sync committee's signature. All the public keys of the committee members (including any duplicates) are aggregated into this single public key. If any signatures are missing from the SyncAggregate, the light client can "de-aggregate" them by performing elliptic curve subtraction. As long as more than half of the committee contributed to the signature, then this will be faster than constructing the aggregate of participating members from scratch. If less than half contributed to the signature, the light client can start instead with the identity public key and use elliptic curve addition to aggregate those public keys that are present.

See also SYNC_COMMITTEE_SIZE.

**SigningData**

```
class SigningData(Container):
    object_root: Root
    domain: Domain
```

This is just a convenience container, used only in compute_signing_root() to calculate the hash tree root of an object along with a domain. The resulting root is the message data that gets signed with a BLS signature. The SigningData object itself is never stored or transmitted.

## Beacon operations

The following are the various protocol messages that can be transmitted in a block' on the beacon chain.

For most of these, the proposer is rewarded either explicitly or implicitly for including the object in a block.

The proposer receives explicit in-protocol rewards for including the following in blocks:

- ProposerSlashings,

[41] It's a blockchain, yo!

- AttesterSlashings,
- Attestations, and
- SyncAggregates.

Including Deposit objects in blocks is only implicitly rewarded, in that, if there are pending deposits that the block proposer does not include then the block is invalid, so the proposer receives no reward.

There is no direct reward for including VoluntaryExit objects. However, for each validator exited, rewards for the remaining validators increase very slightly, so it's still beneficial for proposers not to ignore VoluntaryExits.

#### ProposerSlashing

```
class ProposerSlashing(Container):
    signed_header_1: SignedBeaconBlockHeader
    signed_header_2: SignedBeaconBlockHeader
```

ProposerSlashings may be included in blocks to prove that a validator has broken the rules and ought to be slashed. Proposers receive a reward for correctly submitting these.

In this case, the rule is that a validator may not propose two different blocks at the same height, and the payload is the signed headers of the two blocks that evidence the crime. The signatures on the SignedBeaconBlockHeaders are checked to verify that they were both signed by the accused validator.

#### AttesterSlashing

```
class AttesterSlashing(Container):
    attestation_1: IndexedAttestation
    attestation_2: IndexedAttestation
```

AttesterSlashings may be included in blocks to prove that one or more validators in a committee has broken the rules and ought to be slashed. Proposers receive a reward for correctly submitting these.

The contents of the IndexedAttestations are checked against the attester slashing conditions in is_slashable_attestation_data(). If there is a violation, then any validator that attested to both attestation_1 and attestation_2 is slashed, see process_attester_slashing().

AttesterSlashings can be very large since they could in principle list the indices of all the validators in a committee. However, in contrast to proposer slashings, many validators can be slashed as a result of a single report.

#### Attestation

```
class Attestation(Container):
    aggregation_bits: Bitlist[MAX_VALIDATORS_PER_COMMITTEE]
    data: AttestationData
    signature: BLSSignature
```

This is the form in which attestations make their way around the network. It is designed to be easily aggregatable: Attestations containing identical AttestationData can be combined into a single attestation by aggregating the signatures.

Attestations contain the same information as IndexedAttestations, but use knowledge of the validator committees at slots to compress the list of attesting validators down to a bitlist. Thus, Attestations are at least 5 times smaller than IndexedAttestations, and up to 35 times smaller (with 128 or 2048 validators per committee, respectively).

When a validator first broadcasts its attestation to the network, the aggregation_bits list will contain only a single bit set, and calling get_attesting_indices() on it will return a list containing only a single entry, the validator's own index.

#### Deposit

```
class Deposit(Container):
    proof: Vector[Bytes32, DEPOSIT_CONTRACT_TREE_DEPTH + 1]  # Merkle path to deposit root
    data: DepositData
```

This container is used to include deposit data from prospective validators in beacon blocks so that they can be processed into beacon state.

The `proof` is a Merkle proof constructed by the block proposer that the `DepositData` corresponds to the previously agreed deposit root of the Eth1 contract's deposit tree. It is verified in `process_deposit()` by `is_valid_merkle_branch()`.

#### `VoluntaryExit`

```
class VoluntaryExit(Container):
    epoch: Epoch  # Earliest epoch when voluntary exit can be processed
    validator_index: ValidatorIndex
```

Voluntary exit messages are how a validator signals that it wants to cease being a validator. Blocks containing `VoluntaryExit` data for an epoch later than the current epoch are invalid, so nodes should buffer or ignore any future-dated exits they see.

`VoluntaryExit` objects are never used naked; they are always wrapped up into a `SignedVoluntaryExit` object.

#### `SyncAggregate`

```
class SyncAggregate(Container):
    sync_committee_bits: Bitvector[SYNC_COMMITTEE_SIZE]
    sync_committee_signature: BLSSignature
```

The prevailing sync committee is stored in the beacon state, so the `SyncAggregate`s included in blocks need only use a bit vector to indicate which committee members signed off on the message.

The `sync_committee_signature` is the aggregate signature of all the validators referenced in the bit vector over the block root of the previous slot.

`SyncAggregate`s are handled by `process_sync_aggregate()`.

## Beacon blocks

#### `BeaconBlockBody`

```
class BeaconBlockBody(Container):
    randao_reveal: BLSSignature
    eth1_data: Eth1Data  # Eth1 data vote
    graffiti: Bytes32  # Arbitrary data
    # Operations
    proposer_slashings: List[ProposerSlashing, MAX_PROPOSER_SLASHINGS]
    attester_slashings: List[AttesterSlashing, MAX_ATTESTER_SLASHINGS]
    attestations: List[Attestation, MAX_ATTESTATIONS]
    deposits: List[Deposit, MAX_DEPOSITS]
    voluntary_exits: List[SignedVoluntaryExit, MAX_VOLUNTARY_EXITS]
    sync_aggregate: SyncAggregate  # [New in Altair]
```

The two fundamental data structures for nodes are the `BeaconBlock` and the `BeaconState`. The `BeaconBlock` is how the leader (the chosen proposer in a slot) communicates network updates to all the other validators, and those validators update their own `BeaconState` by applying `BeaconBlock`s. The idea is that (eventually) all validators on the network come to agree on the same `BeaconState`.

Validators are randomly selected to propose beacon blocks, and there ought to be exactly one beacon block per slot if things are running correctly. If a validator is offline, or misses its slot, or proposes an invalid block, or has its block orphaned, then a slot can be empty.

The following objects are always present in a valid beacon block.

- `randao_reveal`: the block is invalid if the RANDAO reveal does not verify correctly against the proposer's public key. This is the block proposer's contribution to the beacon chain's randomness. The proposer generates it by signing the current epoch number (combined with `DOMAIN_RANDAO`) with its private key. To the best of anyone's knowledge, the result is indistinguishable from random. This gets mixed into the beacon state RANDAO.

- See Eth1Data for `eth1_data`. In principle, this is mandatory, but it is not checked, and there is no penalty for making it up.
- `graffiti` is left free for the proposer to insert whatever data it wishes. It has no protocol level significance. It can be left as zero; most clients set the client name and version string as their own default graffiti value.
- `sync_aggregate` is a record of which validators in the current sync committee voted for the chain head in the previous slot.

Deposits are a special case. They are mandatory only if there are pending deposits to be processed. There is no explicit reward for including deposits, except that a block is invalid without any that ought to be there.

- `deposits`: if the block does not contain either all the outstanding `Deposit`s, or `MAX_DEPOSITS` of them in deposit order, then it is invalid.

Including any of the remaining objects is optional. They are handled, if present, in the `process_operations()` function.

The proposer earns rewards for including any of the following. Rewards for attestations and sync aggregates are available every slot. Slashings, however, are very rare.

- `proposer_slashings`: up to `MAX_PROPOSER_SLASHINGS` `ProposerSlashing` objects may be included.
- `attester_slashings`: up to `MAX_ATTESTER_SLASHINGS` `AttesterSlashing` objects may be included.
- `attestations`: up to `MAX_ATTESTATIONS` (aggregated) `Attestation` objects may be included. The block proposer is incentivised to include well-packed aggregate attestations, as it receives a micro reward for each unique attestation. In a perfect world, with perfectly aggregated attestations, `MAX_ATTESTATIONS` would be equal to `MAX_COMMITTEES_PER_SLOT`; in our configuration it is double. This provides capacity in blocks to catch up with attestations after skip slots, and also room to include some imperfectly aggregated attestations.

Including voluntary exits is optional, and there are no explicit rewards for including them.

- `voluntary_exits`: up to `MAX_VOLUNTARY_EXITS` `SignedVoluntaryExit` objects may be included.

#### `BeaconBlock`

```
class BeaconBlock(Container):
    slot: Slot
    proposer_index: ValidatorIndex
    parent_root: Root
    state_root: Root
    body: BeaconBlockBody
```

`BeaconBlock` just adds some blockchain paraphernalia to `BeaconBlockBody`. It is identical to `BeaconBlockHeader`, except that the `body_root` is replaced by the actual block `body`.

`slot` is the slot the block is proposed for.

`proposer_index` was added to avoid a potential DoS vector, and to allow clients without full access to the state to still know useful things.

`parent_root` is used to make sure that this block is a direct child of the last block we processed.

In order to calculate `state_root`, the proposer is expected to run the state transition with the block before propagating it. After the beacon node has processed the block, the state roots are compared to ensure they match. This is the mechanism for tying the whole system together and making sure that all validators and beacon nodes are always working off the same version of state (absent any short-term forks).

If any of these is incorrect, then the block is invalid with respect to the current beacon state and will be ignored.

## Beacon state

**`BeaconState`**

```
class BeaconState(Container):
    # Versioning
    genesis_time: uint64
    genesis_validators_root: Root
    slot: Slot
    fork: Fork
    # History
    latest_block_header: BeaconBlockHeader
    block_roots: Vector[Root, SLOTS_PER_HISTORICAL_ROOT]
    state_roots: Vector[Root, SLOTS_PER_HISTORICAL_ROOT]
    historical_roots: List[Root, HISTORICAL_ROOTS_LIMIT]
    # Eth1
    eth1_data: Eth1Data
    eth1_data_votes: List[Eth1Data, EPOCHS_PER_ETH1_VOTING_PERIOD * SLOTS_PER_EPOCH]
    eth1_deposit_index: uint64
    # Registry
    validators: List[Validator, VALIDATOR_REGISTRY_LIMIT]
    balances: List[Gwei, VALIDATOR_REGISTRY_LIMIT]
    # Randomness
    randao_mixes: Vector[Bytes32, EPOCHS_PER_HISTORICAL_VECTOR]
    # Slashings
    slashings: Vector[Gwei, EPOCHS_PER_SLASHINGS_VECTOR]  # Per-epoch sums of slashed effective balances
    # Participation
    previous_epoch_participation: List[ParticipationFlags, VALIDATOR_REGISTRY_LIMIT]  # [Modified in
                                       ↪ Altair]
    current_epoch_participation: List[ParticipationFlags, VALIDATOR_REGISTRY_LIMIT]  # [Modified in
                                       ↪ Altair]
    # Finality
    justification_bits: Bitvector[JUSTIFICATION_BITS_LENGTH]  # Bit set for every recent justified epoch
    previous_justified_checkpoint: Checkpoint
    current_justified_checkpoint: Checkpoint
    finalized_checkpoint: Checkpoint
    # Inactivity
    inactivity_scores: List[uint64, VALIDATOR_REGISTRY_LIMIT]  # [New in Altair]
    # Sync
    current_sync_committee: SyncCommittee  # [New in Altair]
    next_sync_committee: SyncCommittee  # [New in Altair]
```

All roads lead to the `BeaconState`. Maintaining this data structure is the sole purpose of all the apparatus in all of the spec documents. This state is the focus of consensus among the beacon nodes; it is what everybody, eventually, must agree on.

The beacon chain's state is monolithic: everything is bundled into a single state object (sometimes referred to as the "God object"). Some have argued for more granular approaches that might be more efficient, but at least the current approach is simple.

Let's break this thing down.

```
    # Versioning
    genesis_time: uint64
    genesis_validators_root: Root
    slot: Slot
    fork: Fork
```

How do we know which chain we're on, and where we are on it? The information here ought to be sufficient. A continuous path back to the genesis block would also suffice.

`genesis_validators_root` is calculated at Genesis time (when the chain starts) and is fixed for the life of

the chain. This, combined with the `fork` identifier, should serve to uniquely identify the chain that we are on.

The fork choice rule uses `genesis_time` to work out what slot we're in.

The `fork` object is manually updated as part of beacon chain upgrades, also called hard forks. This invalidates blocks and attestations from validators not following the new fork.

```
    # History
    latest_block_header: BeaconBlockHeader
    block_roots: Vector[Root, SLOTS_PER_HISTORICAL_ROOT]
    state_roots: Vector[Root, SLOTS_PER_HISTORICAL_ROOT]
    historical_roots: List[Root, HISTORICAL_ROOTS_LIMIT]
```

`latest_block_header` is only used to make sure that the next block we process is a direct descendent of the previous block. It's a blockchain thing.

Past `block_roots` and `state_roots` are stored in lists here until the lists are full. Once they are full, the Merkle root is taken of both the lists together and appended to `historical_roots`. `historical_roots` effectively grows without bound (`HISTORICAL_ROOTS_LIMIT` is *large*), but at a rate of only 10KB per year. Keeping this data is useful for light clients, and also allows Merkle proofs to be created against past states, for example historical deposit data.

```
    # Eth1
    eth1_data: Eth1Data
    eth1_data_votes: List[Eth1Data, EPOCHS_PER_ETH1_VOTING_PERIOD * SLOTS_PER_EPOCH]
    eth1_deposit_index: uint64
```

`eth1_data` is the latest agreed upon state of the Eth1 chain and deposit contract. `eth1_data_votes` accumulates `Eth1Data` from blocks until there is an overall majority in favour of one Eth1 state. If a majority is not achieved by the time the list is full then it is cleared down and voting starts again from scratch. `eth1_deposit_index` is the total number of deposits that have been processed by the beacon chain (which is greater than or equal to the number of validators, as a deposit can top-up the balance of an existing validator).

```
    # Registry
    validators: List[Validator, VALIDATOR_REGISTRY_LIMIT]
    balances: List[Gwei, VALIDATOR_REGISTRY_LIMIT]
```

The registry of `Validator`s and their balances. The `balances` list is separated out as it changes much more frequently than the `validators` list. Roughly speaking, balances of active validators are updated every epoch, while the `validators` list has only minor updates per epoch. When combined with SSZ tree hashing, this results in a big saving in the amount of data to be rehashed on registry updates. See also validator inactivity scores under Inactivity which we treat similarly.

```
    # Randomness
    randao_mixes: Vector[Bytes32, EPOCHS_PER_HISTORICAL_VECTOR]
```

Past randao mixes are stored in a fixed-size circular list for `EPOCHS_PER_HISTORICAL_VECTOR` epochs (~290 days). These can be used to recalculate past committees, which allows slashing of historical attestations. See `EPOCHS_PER_HISTORICAL_VECTOR` for more information.

```
    # Slashings
    slashings: Vector[Gwei, EPOCHS_PER_SLASHINGS_VECTOR]
```

A fixed-size circular list of past slashed amounts. Each epoch, the total effective balance of all validators slashed in that epoch is stored as an entry in this list. When the final slashing penalty for a slashed validator is calculated, it is weighted with the sum of this list. This mechanism is designed to less heavily penalise one-off slashings that are most likely accidental, and more heavily penalise mass slashings during a window of time, which are more likely to be a coordinated attack.

```
    # Participation
    previous_epoch_participation: List[ParticipationFlags, VALIDATOR_REGISTRY_LIMIT]  # [Modified in
                                      ↪ Altair]
    current_epoch_participation: List[ParticipationFlags, VALIDATOR_REGISTRY_LIMIT]  # [Modified in
                                      ↪ Altair]
```

These lists record which validators participated in attesting during the current and previous epochs by recording flags for timely votes for the correct source, the correct target and the correct head. We store two epochs' worth since Validators have up to 32 slots to include a correct target vote. The flags are used to calculate finality and to assign rewards at the end of epochs.

Previously, during Phase 0, we stored two epochs' worth of actual attestations in the state and processed them en masse at the end of epochs. This was slow, and was thought to be contributing to observed late block production in the first slots of epochs. The change to the new scheme was implemented in the Altair upgrade under the title of Accounting Reforms.

```
    # Finality
    justification_bits: Bitvector[JUSTIFICATION_BITS_LENGTH]
    previous_justified_checkpoint: Checkpoint
    current_justified_checkpoint: Checkpoint
    finalized_checkpoint: Checkpoint
```

Ethereum 2.0 uses the Casper FFG finality mechanism, with a k-finality optimisation, where k = 2. The above objects in the state are the data that need to be tracked in order to apply the finality rules.

- `justification_bits` is only four bits long. It tracks the justification status of the last four epochs: 1 if justified, 0 if not. This is used when calculating whether we can finalise an epoch.
- Outside of the finality calculations, `previous_justified_checkpoint` and `current_justified_checkpoint` are used to filter attestations: valid blocks include only attestations with a source checkpoint that matches the justified checkpoint in the state for the attestation's epoch.
- `finalized_checkpoint`: the network has agreed that the beacon chain state at or before that epoch will never be reverted. So, for one thing, the fork choice rule doesn't need to go back any further than this. The Casper FFG mechanism is specifically constructed so that two conflicting finalized checkpoints cannot be created without at least one third of validators being slashed.

```
    # Inactivity
    inactivity_scores: List[uint64, VALIDATOR_REGISTRY_LIMIT]  # [New in Altair]
```

This is logically part of "Registry", above, and would be better placed there. It is a per-validator record of inactivity scores that is updated every epoch. This list is stored outside the main list of Validator objects since it is updated very frequently. See the Registry for more explanation.

```
    # Sync
    current_sync_committee: SyncCommittee  # [New in Altair]
    next_sync_committee: SyncCommittee  # [New in Altair]
```

Sync committees were introduced in the Altair upgrade. The next sync committee is calculated and stored so that participating validators can prepare in advance by subscribing to the required p2p subnets.

### Historical Note

There was a period during which beacon state was split into "crystallized state" and "active state". The active state was constantly changing; the crystallized state changed only once per epoch (or what passed for epochs back then). Separating out the fast-changing state from the slower-changing state was an attempt to avoid having to constantly rehash the whole state every slot. With the introduction of SSZ tree hashing, this was no longer necessary, as the roots of the slower changing parts could simply be cached, which was a nice simplification. There remains an echo of this approach, however, in the splitting out of validator balances and inactivity scores into different structures withing the beacon state.

## Signed envelopes

The following are just wrappers for more basic types, with an added signature.

#### `SignedVoluntaryExit`

```
class SignedVoluntaryExit(Container):
    message: VoluntaryExit
    signature: BLSSignature
```

A voluntary exit is currently signed with the validator's online signing key.

There has been some discussion about changing this to also allow signing of a voluntary exit with the validator's offline withdrawal key. The introduction of multiple types of withdrawal credential makes this more complex, however, and it is no longer likely to be practical.

**`SignedBeaconBlock`**

```
class SignedBeaconBlock(Container):
    message: BeaconBlock
    signature: BLSSignature
```

`BeaconBlock`s are signed by the block proposer and unwrapped for block processing.

This signature is what makes proposing a block "accountable". If two correctly signed conflicting blocks turn up, the signatures guarantee that the same proposer produced them both, and is thus subject to being slashed. This is also why stakers need to closely guard their signing keys.

**`SignedBeaconBlockHeader`**

```
class SignedBeaconBlockHeader(Container):
    message: BeaconBlockHeader
    signature: BLSSignature
```

This is used only when reporting proposer slashing, within a `ProposerSlashing` object.

Through the magic of SSZ hash tree roots, a valid signature for a `SignedBeaconBlock` is also a valid signature for a `SignedBeaconBlockHeader`. Proposer slashing makes use of this to save space in slashing reports.

# Helper Functions

## Preamble

> *Note*: The definitions below are for specification purposes and are not necessarily optimal implementations.

This note in the spec is super important for implementers! There are many, many optimisations of the below routines that are being used in practice; a naive implementation would be impractically slow for mainnet configurations. As long as the optimised code produces identical results to the code here, then all is fine.

## Math

### `integer_squareroot`

```
def integer_squareroot(n: uint64) -> uint64:
    """
    Return the largest integer ``x`` such that ``x**2 <= n``.
    """
    x = n
    y = (x + 1) // 2
    while y < x:
        x = y
        y = (x + n // x) // 2
    return x
```

Validator rewards scale with the reciprocal of the square root of the total active balance of all validators. This is calculated in `get_base_reward_per_increment()`.

In principle `integer_squareroot` is also used in `get_attestation_participation_flag_indices()`, to specify the maximum delay for source votes to receive a reward. But this is just the constant, `integer_squareroot(SLOTS_PER_EPOCH)`, which is `5`.

Newton's method is used which has pretty good convergence properties, but implementations may use any method that gives identical results.

| Used by | `get_base_reward_per_increment()`, `get_attestation_participation_flag_indices()` |
|---|---|

### `xor`

```
def xor(bytes_1: Bytes32, bytes_2: Bytes32) -> Bytes32:
    """
    Return the exclusive-or of two 32-byte strings.
    """
    return Bytes32(a ^ b for a, b in zip(bytes_1, bytes_2))
```

The bitwise `xor` of two 32-byte quantities is defined here in Python terms.

This is used only in `process_randao()` when mixing in the new randao reveal.

Fun fact: if you `xor` two `byte` types in Java, the result is a 32 bit (signed) integer. This is one reason we need to define the "obvious" here. But mainly, because the spec is executable, we need to tell Python what it doesn't already know.

| Used by | `process_randao()` |
|---|---|

#### uint_to_bytes

> `def uint_to_bytes(n: uint) -> bytes` is a function for serializing the `uint` type object to bytes in `ENDIANNESS`-endian. The expected length of the output is the byte-length of the `uint` type.

For the most part, integers are integers and bytes are bytes, and they don't mix much. But there are a few places where we need to convert from integers to bytes:

- several times in the `compute_shuffled_index()` algorithm;
- in `compute_proposer_index()` for selecting a proposer weighted by stake;
- in `get_seed()` to mix the epoch number into the randao mix;
- in `get_beacon_proposer_index()` to mix the slot number into the per-epoch randao seed; and
- in `get_next_sync_committee_indices()`.

You'll note that in every case, the purpose of the conversion is for the integer to form part of a byte string that is hashed to create (pseudo-)randomness.

The result of this conversion is dependent on our arbitrary choice of endianness, that is, how we choose to represent integers as strings of bytes. For Eth2, we have chosen little-endian: see the discussion of `ENDIANNESS` for more background.

The `uint_to_bytes()` function is not given an explicit implementation in the specification, which is unusual. This to avoid exposing the innards of the Python SSZ implementation (of `uint`) to the rest of the spec. When running the spec as an executable, it uses the definition in the SSZ utilities.

| | |
|---|---|
| Used by | `compute_shuffled_index()`, `compute_proposer_index()`, `get_seed()`, `get_beacon_proposer_index()`, `get_next_sync_committee_indices()` |
| See also | `ENDIANNESS`, SSZ utilities |

#### bytes_to_uint64

```
def bytes_to_uint64(data: bytes) -> uint64:
    """
    Return the integer deserialization of ``data`` interpreted as ``ENDIANNESS``-endian.
    """
    return uint64(int.from_bytes(data, ENDIANNESS))
```

`bytes_to_uint64()` is the inverse of `uint_to_bytes()`, and is used by the shuffling algorithm to create a random index from the output of a hash.

It is also used in the validator specification when selecting validators to aggregate attestations, and sync committee messages.

`int.from_bytes` is a built-in Python 3 method. The `uint64` cast is provided by the spec's SSZ implementation.

| | |
|---|---|
| Used by | `compute_shuffled_index` |
| See also | attestation aggregator selection, sync committee aggregator selection |

## Crypto

#### hash

> `def hash(data: bytes) -> Bytes32` is SHA256.

SHA256 was chosen as the protocol's base hash algorithm for easier cross-chain interoperability: many other chains use SHA256, and Eth1 has a SHA256 precompile.

There was a lot of discussion about this choice early in the design process. The original plan had been to use the BLAKE2b-512 hash function – that being a modern hash function that's faster than SHA3 – and to move to a STARK/SNARK friendly hash function at some point (such as MiMC). However, to keep interoperability with Eth1, in particular for the implementation of the deposit contract, the hash function was changed to Keccak256. Finally, we settled on SHA256 as having even broader compatibility.

The hash function serves two purposes within the protocol. The main use, computationally, is in Merkleization, the computation of hash tree roots, which is ubiquitous in the protocol. Its other use is to harden the randomness used in various places.

| | |
|---|---|
| Used by | `hash_tree_root`, `is_valid_merkle_branch()`, `compute_shuffled_index()`, `compute_proposer_index()`, `get_seed()`, `get_beacon_proposer_index()`, `get_next_sync_committee_indices()`, `process_randao()` |

#### `hash_tree_root`

> `def hash_tree_root(object: SSZSerializable) -> Root` is a function for hashing objects into a single root by utilizing a hash tree structure, as defined in the SSZ spec.

The development of the tree hashing process was transformational for the Ethereum 2.0 specification, and it is now used everywhere.

The naive way to create a digest of a data structure is to serialise it and then just run a hash function over the result. In tree hashing, the basic idea is to treat each element of an ordered, compound data structure as the leaf of a Merkle tree, recursively if necessary until a primitive type is reached, and to return the Merkle root of the resulting tree.

At first sight, this all looks quite inefficient. Twice as much data needs to be hashed when tree hashing, and actual speeds are 4-6 times slower compared with the linear hash. However, it is good for supporting light clients, because it allows Merkle proofs to be constructed easily for subsets of the full state.

The breakthrough insight was realising that much of the re-hashing work can be cached: if part of the state data structure has not changed, that part does not need to be re-hashed: the whole subtree can be replaced with its cached hash. This turns out to be a huge efficiency boost, allowing the previous design, with cumbersome separate crystallised and active state, to be simplified into a single state object.

Merkleization, the process of calculating the `hash_tree_root()` of an object, is defined in the SSZ specification, and explained further in the section on SSZ.

### BLS signatures

See the main write-up on BLS Signatures for a more in-depth exploration of this topic.

> The IETF BLS signature draft standard v4 with ciphersuite `BLS_SIG_BLS12381G2_XMD:SHA-256_SSWU_RO_POP_` defines the following functions:
>
> - `def Sign(privkey: int, message: Bytes) -> BLSSignature`
> - `def Verify(pubkey: BLSPubkey, message: Bytes, signature: BLSSignature) -> bool`
> - `def Aggregate(signatures: Sequence[BLSSignature]) -> BLSSignature`
> - `def FastAggregateVerify(pubkeys: Sequence[BLSPubkey], message: Bytes, signature: BLSSignature) -> bool`
> - `def AggregateVerify(pubkeys: Sequence[BLSPubkey], messages: Sequence[Bytes], signature: BLSSignature) -> bool`

> - `def KeyValidate(pubkey: BLSPubkey) -> bool`
>
> The above functions are accessed through the `bls` module, e.g. `bls.Verify`.

The detailed specification of the cryptographic functions underlying Ethereum 2.0's BLS signing scheme is delegated to the draft IETF standard as described in the spec. This includes specifying the elliptic curve BLS12-381 as our domain of choice.

Our intention in conforming to the in-progress standard is to provide for maximal interoperability with other chains, applications, and cryptographic libraries. Ethereum Foundation researchers and Eth2 developers had input to the development of the standard. Nevertheless, there were some challenges involved in trying to keep up as the standard evolved. For example, the Hashing to Elliptic Curves standard was still changing rather late in the beacon chain testing phase. In the end, everything worked out fine.

The following two functions are described in the separate BLS Extensions document, but included here for convenience.

#### `eth_aggregate_pubkeys`

```
def eth_aggregate_pubkeys(pubkeys: Sequence[BLSPubkey]) -> BLSPubkey:
    """
    Return the aggregate public key for the public keys in ``pubkeys``.

    NOTE: the ``+`` operation should be interpreted as elliptic curve point addition, which takes as input
    elliptic curve points that must be decoded from the input ``BLSPubkey``s.
    This implementation is for demonstrative purposes only and ignores encoding/decoding concerns.
    Refer to the BLS signature draft standard for more information.
    """
    assert len(pubkeys) > 0
    # Ensure that the given inputs are valid pubkeys
    assert all(bls.KeyValidate(pubkey) for pubkey in pubkeys)

    result = copy(pubkeys[0])
    for pubkey in pubkeys[1:]:
        result += pubkey
    return result
```

Stand-alone aggregation of public keys is not defined by the BLS signature standard. In the standard, public keys are aggregated only in the context of performing an aggregate signature verification via `AggregateVerify()` or `FastAggregateVerify()`.

The `eth_aggregate_pubkeys()` function was added in the Altair upgrade to implement an optimisation for light clients when verifying the signatures on `SyncAggregate`s.

| | |
|---|---|
| Used by | `get_next_sync_committee()` |
| Uses | `bls.KeyValidate()` |

#### `eth_fast_aggregate_verify`

```
def eth_fast_aggregate_verify(pubkeys: Sequence[BLSPubkey], message: Bytes32, signature: BLSSignature) ->
                                    ↪ bool:
    """
    Wrapper to ``bls.FastAggregateVerify`` accepting the ``G2_POINT_AT_INFINITY`` signature when
                                    ↪ ``pubkeys`` is empty.
    """
    if len(pubkeys) == 0 and signature == G2_POINT_AT_INFINITY:
        return True
    return bls.FastAggregateVerify(pubkeys, message, signature)
```

The specification of `FastAggregateVerify()` in the BLS signature standard returns `INVALID` if there are zero public keys given.

This function was introduced in Altair to handle SyncAggregates that no sync committee member had signed off on, in which case the G2_POINT_AT_INFINITY can be considered a "correct" signature (in our case, but not according to the standard).

The networking and validator specs were later clarified to require that SyncAggregates have at least one signature. But this requirement is not enforced in the consensus layer (in process_sync_aggregate()), so we need to retain this eth_fast_aggregate_verify() wrapper to allow the empty signature to be valid.

| | |
|---|---|
| Used by | process_sync_aggregate() |
| Uses | FastAggregateVerify() |
| See also | G2_POINT_AT_INFINITY |

## Predicates

### is_active_validator

```
def is_active_validator(validator: Validator, epoch: Epoch) -> bool:
    """
    Check if ``validator`` is active.
    """
    return validator.activation_epoch <= epoch < validator.exit_epoch
```

Validators don't explicitly track their own state (eligible for activation, active, exited, withdrawable - the sole exception being whether they have been slashed or not). Instead, a validator's state is calculated by looking at the fields in the Validator record that store the epoch numbers of state transitions.

In this case, if the validator was activated in the past and has not yet exited, then it is active.

This is used a few times in the spec, most notably in get_active_validator_indices() which returns a list of all active validators at an epoch.

| | |
|---|---|
| Used by | get_active_validator_indices(), get_eligible_validator_indices(), process_registry_updates(), process_voluntary_exit() |
| See also | Validator |

### is_eligible_for_activation_queue

```
def is_eligible_for_activation_queue(validator: Validator) -> bool:
    """
    Check if ``validator`` is eligible to be placed into the activation queue.
    """
    return (
        validator.activation_eligibility_epoch == FAR_FUTURE_EPOCH
        and validator.effective_balance == MAX_EFFECTIVE_BALANCE
    )
```

When a deposit is been processed with a previously unseen public key, a new Validator record is created with all the state-transition fields set to the default value of FAR_FUTURE_EPOCH.

It is possible to deposit any amount over MIN_DEPOSIT_AMOUNT (currently 1 Ether) into the deposit contract. However, validators do not become eligible for activation until their effective balance is equal to MAX_EFFECTIVE_BALANCE, which corresponds to an actual balance of 32 Ether or more.

This predicate is used during epoch processing to find validators that have acquired the minimum necessary balance, but have not yet been added to the queue for activation. These validators are then marked as eligible for activation by setting the validator.activation_eligibility_epoch to the next epoch.

| | |
|---|---|
| Used by | `process_registry_updates()` |
| See also | `Validator`, `FAR_FUTURE_EPOCH`, `MAX_EFFECTIVE_BALANCE` |

#### `is_eligible_for_activation`

```
def is_eligible_for_activation(state: BeaconState, validator: Validator) -> bool:
    """
    Check if ``validator`` is eligible for activation.
    """
    return (
        # Placement in queue is finalized
        validator.activation_eligibility_epoch <= state.finalized_checkpoint.epoch
        # Has not yet been activated
        and validator.activation_epoch == FAR_FUTURE_EPOCH
    )
```

A validator that `is_eligible_for_activation()` has had its `activation_eligibility_epoch` set, but its `activation_epoch` is not yet set.

To avoid any ambiguity or confusion on the validator side about its state, we wait until its eligibility activation epoch has been finalised before adding it to the activation queue by setting its `activation_epoch`. Otherwise, it might at one point become active, and then the beacon chain could flip to a fork in which it is not active. This could happen if the latter fork had fewer blocks and had thus processed fewer deposits.

| | |
|---|---|
| Used by | `process_registry_updates()` |
| See also | `Validator`, `FAR_FUTURE_EPOCH` |

#### `is_slashable_validator`

```
def is_slashable_validator(validator: Validator, epoch: Epoch) -> bool:
    """
    Check if ``validator`` is slashable.
    """
    return (not validator.slashed) and (validator.activation_epoch <= epoch <
                                        ↪ validator.withdrawable_epoch)
```

Validators can be slashed only once: the flag `validator.slashed` is set when the first correct slashing report for the validator is processed.

An unslashed validator remains eligible to be slashed from when it becomes active right up until it becomes withdrawable. This is `MIN_VALIDATOR_WITHDRAWABILITY_DELAY` epochs (around 27 hours) after it has exited from being a validator and ceased validation duties.

| | |
|---|---|
| Used by | `process_proposer_slashing()`, `process_attester_slashing()` |
| See also | `Validator` |

#### `is_slashable_attestation_data`

```
def is_slashable_attestation_data(data_1: AttestationData, data_2: AttestationData) -> bool:
    """
    Check if ``data_1`` and ``data_2`` are slashable according to Casper FFG rules.
    """
    return (
        # Double vote
        (data_1 != data_2 and data_1.target.epoch == data_2.target.epoch) or
```

```
        # Surround vote
        (data_1.source.epoch < data_2.source.epoch and data_2.target.epoch < data_1.target.epoch)
    )
```

This predicate is used by `process_attester_slashing()` to check that the two sets of alleged conflicting attestation data in an `AttesterSlashing` do in fact qualify as slashable.

There are two ways for validators to get slashed under Casper FFG:

1. A double vote: voting more than once for the same target epoch, or
2. A surround vote: the source–target interval of one attestation entirely contains the source–target interval of a second attestation from the same validator or validators. The reporting block proposer needs to take care to order the `IndexedAttestation`s within the `AttesterSlashing` object so that the first set of votes surrounds the second. (The opposite ordering also describes a slashable offence, but is not checked for here.)

| | |
|---|---|
| Used by | `process_attester_slashing()` |
| See also | `AttestationData`, `AttesterSlashing` |

#### `is_valid_indexed_attestation`

```
def is_valid_indexed_attestation(state: BeaconState, indexed_attestation: IndexedAttestation) -> bool:
    """
    Check if ``indexed_attestation`` is not empty, has sorted and unique indices and has a valid
                                        ↪ aggregate signature.
    """
    # Verify indices are sorted and unique
    indices = indexed_attestation.attesting_indices
    if len(indices) == 0 or not indices == sorted(set(indices)):
        return False
    # Verify aggregate signature
    pubkeys = [state.validators[i].pubkey for i in indices]
    domain = get_domain(state, DOMAIN_BEACON_ATTESTER, indexed_attestation.data.target.epoch)
    signing_root = compute_signing_root(indexed_attestation.data, domain)
    return bls.FastAggregateVerify(pubkeys, signing_root, indexed_attestation.signature)
```

`is_valid_indexed_attestation()` is used in attestation processing and attester slashing.

IndexedAttestations differ from Attestations in that the latter record the contributing validators in a bitlist and the former explicitly list the global indices of the contributing validators.

An IndexedAttestation passes this validity test only if all of the following apply.

1. There is at least one validator index present.
2. The list of validators contains no duplicates (the Python `set` function performs deduplication).
3. The indices of the validators are sorted. (It is not clear to me why this is required. It's used in the duplicate check here, but that could just be replaced by checking the set size.)
4. Its aggregated signature verifies against the aggregated public keys of the listed validators.

Verifying the signature uses the magic of aggregated BLS signatures. The indexed attestation contains a BLS signature that is supposed to be the combined individual signatures of each of the validators listed in the attestation. This is verified by passing it to `bls.FastAggregateVerify()` along with the list of public keys from the same validators. The verification succeeds only if exactly the same set of validators signed the message (`signing_root`) as appear in the list of public keys. Note that `get_domain()` mixes in the fork version, so that attestations are not valid across forks.

No check is done here that the `attesting_indices` (which are the global validator indices) are all members of the correct committee for this attestation. In `process_attestation()` they must be, by construction. In `process_attester_slashing()` it doesn't matter: *any* validator signing conflicting attestations is liable to be slashed.

| | |
|---|---|
| Used by | process_attester_slashing(), process_attestation() |
| Uses | get_domain(), compute_signing_root(), bls.FastAggregateVerify() |
| See also | IndexedAttestation, Attestation |

#### is_valid_merkle_branch

```
def is_valid_merkle_branch(leaf: Bytes32, branch: Sequence[Bytes32], depth: uint64, index: uint64, root:
                                        ↪ Root) -> bool:
    """
    Check if ``leaf`` at ``index`` verifies against the Merkle ``root`` and ``branch``.
    """
    value = leaf
    for i in range(depth):
        if index // (2**i) % 2:
            value = hash(branch[i] + value)
        else:
            value = hash(value + branch[i])
    return value == root
```

This is the classic algorithm for verifying a Merkle branch (also called a Merkle proof). Nodes are iteratively hashed as the tree is traversed from leaves to root. The bits of `index` select whether we are the right or left child of our parent at each level. The result should match the given `root` of the tree.

In this way we prove that we know that `leaf` is the value at position `index` in the list of leaves, and that we know the whole structure of the rest of the tree, as summarised in `branch`.

We use this function in `process_deposit()` to check whether the deposit data we've received is correct or not. Based on the deposit data they have seen, Eth2 clients build a replica of the Merkle tree of deposits in the deposit contract. The proposer of the block that includes the deposit constructs the Merkle proof using its view of the deposit contract, and all other nodes use `is_valid_merkle_branch()` to check that their view matches the proposer's. It is a consensus failure if there is a mismatch, perhaps due to one client considering a deposit valid while another considers it invalid for some reason.

| | |
|---|---|
| Used by | process_deposit() |

## Misc

#### compute_shuffled_index

```
def compute_shuffled_index(index: uint64, index_count: uint64, seed: Bytes32) -> uint64:
    """
    Return the shuffled index corresponding to ``seed`` (and ``index_count``).
    """
    assert index < index_count

    # Swap or not (https://link.springer.com/content/pdf/10.1007%2F978-3-642-32009-5_1.pdf)
    # See the 'generalized domain' algorithm on page 3
    for current_round in range(SHUFFLE_ROUND_COUNT):
        pivot = bytes_to_uint64(hash(seed + uint_to_bytes(uint8(current_round)))[0:8]) % index_count
        flip = (pivot + index_count - index) % index_count
        position = max(index, flip)
        source = hash(
            seed
            + uint_to_bytes(uint8(current_round))
            + uint_to_bytes(uint32(position // 256))
        )
        byte = uint8(source[(position % 256) // 8])
        bit = (byte >> (position % 8)) % 2
```

```
        index = flip if bit else index

    return index
```

Selecting random, distinct committees of validators is a big part of Ethereum 2.0; it is foundational for both its scalability and security. This selection is done by shuffling.

Shuffling a list of objects is a well understood problem in computer science. Notice, however, that this routine manages to shuffle a *single index* to a new location, knowing only the total length of the list. To use the technical term for this, it is *oblivious*. To shuffle the whole list, this routine needs to be called once per validator index in the list. By construction, each input index maps to a distinct output index. Thus, when applied to all indices in the list, it results in a permutation, also called a shuffling.

Why do this rather than a simpler, more efficient, conventional shuffle? It's all about light clients. Beacon nodes will generally need to know the whole shuffling, but light clients will often be interested only in a small number of committees. Using this technique allows the composition of a single committee to be calculated without having to shuffle the entire set: potentially a big saving on time and memory.

As stated in the code comments, this is an implementation of the "swap-or-not" shuffle, described in the cited paper. Vitalik kicked off a search for a shuffle with these properties in late 2018. With the help of Professor Dan Boneh of Stanford University, the swap-or-not was identified as a candidate a couple of months later, and adopted into the spec.

The algorithm breaks down as follows. For each iteration (each round), we start with a current `index`.

1. Pseudo-randomly select a pivot. This is a 64-bit integer based on the seed and current round number. This domain is large enough that any non-uniformity caused by taking the modulus in the next step is entirely negligible.
2. Use `pivot` to find another index in the list of validators, `flip`, which is `pivot - index` accounting for wrap-around in the list.
3. Calculate a single pseudo-random bit based on the seed, the current round number, and some bytes from either `index` or `flip` depending on which is greater.
4. If our bit is zero, we keep `index` unchanged; if it is one, we set `index` to `flip`.

We are effectively swapping cards in a deck based on a deterministic algorithm.

The way that `position` is broken down is worth noting:

- Bits 0-2 (3 bits) are used to select a single bit from the eight bits of `byte`.
- Bits 3-7 (5 bits) are used to select a single byte from the thirty-two bytes of `source`.
- Bits 8-39 (32 bits) are used in generating `source`. Note that the upper two bytes of this will always be zero in practice, due to limits on the number of active validators.

`SHUFFLE_ROUND_COUNT` is, and always has been, 90 in the mainnet configuration, as explained there.

See the section on Shuffling for a more structured exposition and analysis of this algorithm (with diagrams!).

In practice, full beacon node implementations will run this once per epoch using an optimised version that shuffles the whole list, and cache the result of that for the epoch.

| | |
|---|---|
| Used by | `compute_committee()`, `compute_proposer_index()`, `get_next_sync_committee_indices()` |
| Uses | `bytes_to_uint64()` |
| See also | `SHUFFLE_ROUND_COUNT` |

#### `compute_proposer_index`

```
def compute_proposer_index(state: BeaconState, indices: Sequence[ValidatorIndex], seed: Bytes32) ->
                                        ↪ ValidatorIndex:
    """
```

```
    Return from ``indices`` a random index sampled by effective balance.
    """
    assert len(indices) > 0
    MAX_RANDOM_BYTE = 2**8 - 1
    i = uint64(0)
    total = uint64(len(indices))
    while True:
        candidate_index = indices[compute_shuffled_index(i % total, total, seed)]
        random_byte = hash(seed + uint_to_bytes(uint64(i // 32)))[i % 32]
        effective_balance = state.validators[candidate_index].effective_balance
        if effective_balance * MAX_RANDOM_BYTE >= MAX_EFFECTIVE_BALANCE * random_byte:
            return candidate_index
        i += 1
```

There is exactly one beacon block proposer per slot, selected randomly from among all the active validators. The seed parameter is set in `get_beacon_proposer_index` based on the epoch and slot. Note that there is a small but finite probability of the same validator being called on to propose a block more than once in an epoch.

A validator's chance of being the proposer is weighted by its effective balance: a validator with a 32 Ether effective balance is twice as likely to be chosen as a validator with a 16 Ether effective balance.

To account for the need to weight by effective balance, this function implements as a try-and-increment algorithm. A counter `i` starts at zero. This counter does double duty:

- First `i` is used to uniformly select a candidate proposer with probability $1/N$ where, $N$ is the number of active validators. This is done by using the `compute_shuffled_index` routine to shuffle index `i` to a new location, which is then the `candidate_index`.
- Then `i` is used to generate a pseudo-random byte using the hash function as a seeded PRNG with at least 256 bits of output. The lower 5 bits of `i` select a byte in the hash function, and the upper bits salt the seed. (An obvious optimisation is that the output of the hash changes only once every 32 iterations.)

The `if` test is where the weighting by effective balance is done. If the candidate has `MAX_EFFECTIVE_BALANCE`, it will always pass this test and be returned as the proposer. If the candidate has a fraction of `MAX_EFFECTIVE_BALANCE` then that fraction is the probability of being returned as proposer.

If the candidate is not chosen, then `i` is incremented and we try again. Since the minimum effective balance is half of the maximum, then this ought to terminate fairly swiftly. In the worst case, all validators have 16 Ether effective balance and the chance of having to do another iteration is 50%, in which case there is a one in a million chance of having to do 20 iterations.

Note that this dependence on the validators' effective balances, which are updated at the end of each epoch, means that proposer assignments are valid only in the current epoch. This is different from attestation committee assignments, which are valid with a one epoch look-ahead.

| | |
|---|---|
| Used by | `get_beacon_proposer_index()` |
| Uses | `compute_shuffled_index()` |
| See also | `MAX_EFFECTIVE_BALANCE` |

#### compute_committee

```
def compute_committee(indices: Sequence[ValidatorIndex],
                      seed: Bytes32,
                      index: uint64,
                      count: uint64) -> Sequence[ValidatorIndex]:
    """
    Return the committee corresponding to ``indices``, ``seed``, ``index``, and committee ``count``.
    """
    start = (len(indices) * index) // count
    end = (len(indices) * uint64(index + 1)) // count
```

```
    return [indices[compute_shuffled_index(uint64(i), uint64(len(indices)), seed)] for i in range(start,
                                        ↪ end)]
```

`compute_committee` is used by `get_beacon_committee()` to find the specific members of one of the committees at a slot.

Every epoch, a fresh set of committees is generated; during an epoch, the committees are stable.

Looking at the parameters in reverse order:

- `count` is the total number of committees in an epoch. This is `SLOTS_PER_EPOCH` times the output of `get_committee_count_per_slot()`.
- `index` is the committee number within the epoch, running from `0` to `count - 1`. It is calculated in (`get_beacon_committee()` from the committee number in the slot `index` and the slot number as `(slot % SLOTS_PER_EPOCH) * committees_per_slot + index`.
- `seed` is the seed value for computing the pseudo-random shuffling, based on the epoch number and a domain parameter (`get_beacon_committee()` uses `DOMAIN_BEACON_ATTESTER`).
- `indices` is the list of validators eligible for inclusion in committees, namely the whole list of indices of active validators.

Random sampling among the validators is done by taking a contiguous slice of array indices from `start` to `end` and seeing where each one gets shuffled to by `compute_shuffled_index()`. Note that `ValidatorIndex(i)` is a type-cast in the above: it just turns `i` into a ValidatorIndex type for input into the shuffling. The output value of the shuffling is then used as an index into the `indices` list. There is much here that client implementations will optimise with caching and batch operations.

It may not be immediately obvious, but not all committees returned will be the same size (they can vary by one), and every validator in `indices` will be a member of exactly one committee. As we increment `index` from zero, clearly `start` for `index == j + 1` is `end` for `index == j`, so there are no gaps. In addition, the highest `index` is `count - 1`, so every validator in `indices` finds its way into a committee.[42]

This method of selecting committees is light client friendly. Light clients can compute only the committees that they are interested in without needing to deal with the entire validator set. See the section on Shuffling for explanation of how this works.

Sync committees are assigned by a different process that is more akin to repeatedly performing `compute_proposer_index()`.

| | |
|---|---|
| Used by | `get_beacon_committee` |
| Uses | `compute_shuffled_index()` |

**`compute_epoch_at_slot`**

```
def compute_epoch_at_slot(slot: Slot) -> Epoch:
    """
    Return the epoch number at ``slot``.
    """
    return Epoch(slot // SLOTS_PER_EPOCH)
```

This is trivial enough that I won't explain it. But note that it does rely on `GENESIS_SLOT` and `GENESIS_EPOCH` being zero. The more pernickety among us might prefer it to read,

```
    return GENESIS_EPOCH + Epoch((slot - GENESIS_SLOT) // SLOTS_PER_EPOCH)
```

**`compute_start_slot_at_epoch`**

```
def compute_start_slot_at_epoch(epoch: Epoch) -> Slot:
    """
```

[42] Also not immediately obvious is that there is a subtle issue with committee sizes that was discovered by formal verification, although, given the max supply of ETH it will never be triggered.

```
    Return the start slot of ``epoch``.
    """
    return Slot(epoch * SLOTS_PER_EPOCH)
```

Maybe should read,

```
    return GENESIS_SLOT + Slot((epoch - GENESIS_EPOCH) * SLOTS_PER_EPOCH))
```

| | |
|---|---|
| Used by | `get_block_root()` |
| See also | `SLOTS_PER_EPOCH`, `GENESIS_SLOT`, `GENESIS_EPOCH` |

#### compute_activation_exit_epoch

```
def compute_activation_exit_epoch(epoch: Epoch) -> Epoch:
    """
    Return the epoch during which validator activations and exits initiated in ``epoch`` take effect.
    """
    return Epoch(epoch + 1 + MAX_SEED_LOOKAHEAD)
```

When queuing validators for activation or exit in `process_registry_updates()` and `initiate_validator_exit()` respectively, the activation or exit is delayed until the next epoch, plus `MAX_SEED_LOOKAHEAD` epochs, currently 4.

See `MAX_SEED_LOOKAHEAD` for the details, but in short it is designed to make it extremely hard for an attacker to manipulate the make up of committees via activations and exits.

| | |
|---|---|
| Used by | `initiate_validator_exit()`, `process_registry_updates()` |
| See also | `MAX_SEED_LOOKAHEAD` |

#### compute_fork_data_root

```
def compute_fork_data_root(current_version: Version, genesis_validators_root: Root) -> Root:
    """
    Return the 32-byte fork data root for the ``current_version`` and ``genesis_validators_root``.
    This is used primarily in signature domains to avoid collisions across forks/chains.
    """
    return hash_tree_root(ForkData(
        current_version=current_version,
        genesis_validators_root=genesis_validators_root,
    ))
```

The fork data root serves as a unique identifier for the chain that we are on. `genesis_validators_root` identifies our unique genesis event, and `current_version` our own hard fork subsequent to that genesis event. This is useful, for example, to differentiate between a testnet and mainnet: both might have the same fork versions, but will definitely have different genesis validator roots.

It is used by `compute_fork_digest()` and `compute_domain()`.

| | |
|---|---|
| Used by | `compute_fork_digest()`, `compute_domain()` |
| Uses | `hash_tree_root()` |
| See also | `ForkData` |

#### compute_fork_digest

```
def compute_fork_digest(current_version: Version, genesis_validators_root: Root) -> ForkDigest:
    """
```

```
    Return the 4-byte fork digest for the ``current_version`` and ``genesis_validators_root``.
    This is a digest primarily used for domain separation on the p2p layer.
    4-bytes suffices for practical separation of forks/chains.
    """
    return ForkDigest(compute_fork_data_root(current_version, genesis_validators_root)[:4])
```

Extracts the first four bytes of the fork data root as a `ForkDigest` type. It is primarily used for domain separation on the peer-to-peer networking layer.

`compute_fork_digest()` is used extensively in the Ethereum 2.0 networking specification to distinguish between independent beacon chain networks or forks: it is important that activity on one chain does not interfere with other chains.

| | |
|---|---|
| Uses | `compute_fork_data_root()` |
| See also | `ForkDigest` |

#### compute_domain

```
def compute_domain(domain_type: DomainType, fork_version: Version=None, genesis_validators_root:
                                        ↪ Root=None) -> Domain:
    """
    Return the domain for the ``domain_type`` and ``fork_version``.
    """
    if fork_version is None:
        fork_version = GENESIS_FORK_VERSION
    if genesis_validators_root is None:
        genesis_validators_root = Root()  # all bytes zero by default
    fork_data_root = compute_fork_data_root(fork_version, genesis_validators_root)
    return Domain(domain_type + fork_data_root[:28])
```

When dealing with signed messages, the signature "domains" are separated according to three independent factors:

1. All signatures include a `DomainType` relevant to the message's purpose, which is just some cryptographic hygiene in case the same message is to be signed for different purposes at any point.
2. All but signatures on deposit messages include the fork version. This ensures that messages across different forks of the chain become invalid, and that validators won't be slashed for signing attestations on two different chains (this is allowed).
3. And, now, the root hash of the validator Merkle tree at Genesis is included. Along with the fork version this gives a unique identifier for our chain.

This function is mainly used by `get_domain()`. It is also used in deposit processing, in which case `fork_version` and `genesis_validators_root` take their default values since deposits are valid across forks.

Fun fact: this function looks pretty simple, but I found a subtle bug in the way tests were generated in a previous implementation.

| | |
|---|---|
| Used by | `get_domain()`, `process_deposit()` |
| Uses | `compute_fork_data_root()` |
| See also | `Domain`, `DomainType` `GENESIS_FORK_VERSION` |

#### compute_signing_root

```
def compute_signing_root(ssz_object: SSZObject, domain: Domain) -> Root:
    """
    Return the signing root for the corresponding signing data.
    """
    return hash_tree_root(SigningData(
        object_root=hash_tree_root(ssz_object),
```

```
        domain=domain,
    ))
```

This is a pre-processor for signing objects with BLS signatures:

1. calculate the hash tree root of the object;
2. combine the hash tree root with the `Domain` inside a temporary `SigningData` object;
3. return the hash tree root of that, which is the data to be signed.

The `domain` is usually the output of `get_domain()`, which mixes in the cryptographic domain, the fork version, and the genesis validators root to the message hash. For deposits, it is the output of `compute_domain()`, ignoring the fork version and genesis validators root.

This is exactly equivalent to adding the domain to an object and taking the hash tree root of the whole thing. Indeed, this function used to be called `compute_domain_wrapper_root()`.

| | |
|---|---|
| Used by | Many places |
| Uses | `hash_tree_root()` |
| See also | `SigningData`, `Domain` |

## Participation flags

These two simple utilities were added in the Altair upgrade.

#### `add_flag`

```
def add_flag(flags: ParticipationFlags, flag_index: int) -> ParticipationFlags:
    """
    Return a new ``ParticipationFlags`` adding ``flag_index`` to ``flags``.
    """
    flag = ParticipationFlags(2**flag_index)
    return flags | flag
```

This is simple and self-explanatory. The `2**flag_index` is a bit Pythonic. In a C-like language it would use a bit-shift:

```
    1 << flag_index
```

| | |
|---|---|
| Used by | `process_attestation()`, `translate_participation()` |
| See also | `ParticipationFlags` |

#### `has_flag`

```
def has_flag(flags: ParticipationFlags, flag_index: int) -> bool:
    """
    Return whether ``flags`` has ``flag_index`` set.
    """
    flag = ParticipationFlags(2**flag_index)
    return flags & flag == flag
```

Move along now, nothing to see here.

| | |
|---|---|
| Used by | `get_unslashed_participating_indices()`, `process_attestation()` |
| See also | `ParticipationFlags` |

## Beacon State Accessors

As the name suggests, these functions access the beacon state to calculate various useful things, without modifying it.

#### get_current_epoch

```
def get_current_epoch(state: BeaconState) -> Epoch:
    """
    Return the current epoch.
    """
    return compute_epoch_at_slot(state.slot)
```

A getter for the current epoch, as calculated by `compute_epoch_at_slot()`.

| | |
|---|---|
| Used by | Everywhere |
| Uses | `compute_epoch_at_slot()` |

#### get_previous_epoch

```
def get_previous_epoch(state: BeaconState) -> Epoch:
    """`
    Return the previous epoch (unless the current epoch is ``GENESIS_EPOCH``).
    """
    current_epoch = get_current_epoch(state)
    return GENESIS_EPOCH if current_epoch == GENESIS_EPOCH else Epoch(current_epoch - 1)
```

Return the previous epoch number as an `Epoch` type. Returns `GENESIS_EPOCH` if we are in the `GENESIS_EPOCH`, since it has no prior, and we don't do negative numbers.

| | |
|---|---|
| Used by | Everywhere |
| Uses | `get_current_epoch()` |
| See also | `GENESIS_EPOCH` |

#### get_block_root

```
def get_block_root(state: BeaconState, epoch: Epoch) -> Root:
    """
    Return the block root at the start of a recent ``epoch``.
    """
    return get_block_root_at_slot(state, compute_start_slot_at_epoch(epoch))
```

The Casper FFG part of consensus deals in `Checkpoint`s that are the first slot of an epoch. `get_block_root` is a specialised version of `get_block_root_at_slot()` that returns the block root of the checkpoint, given only an epoch.

| | |
|---|---|
| Used by | `get_attestation_participation_flag_indices()`, `weigh_justification_and_finalization()` |
| Uses | `get_block_root_at_slot()`, `compute_start_slot_at_epoch()` |
| See also | `Root` |

#### get_block_root_at_slot

```
def get_block_root_at_slot(state: BeaconState, slot: Slot) -> Root:
    """
    Return the block root at a recent ``slot``.
```

```
    """
    assert slot < state.slot <= slot + SLOTS_PER_HISTORICAL_ROOT
    return state.block_roots[slot % SLOTS_PER_HISTORICAL_ROOT]
```

Recent block roots are stored in a circular list in state, with a length of `SLOTS_PER_HISTORICAL_ROOT` (currently ~27 hours).

`get_block_root_at_slot()` is used by `get_attestation_participation_flag_indices()` to check whether an attestation has voted for the correct chain head. It is also used in `process_sync_aggregate()` to find the block that the sync committee is signing-off on.

| | |
|---|---|
| Used by | `get_block_root()`, `get_attestation_participation_flag_indices()`, `process_sync_aggregate()` |
| See also | `SLOTS_PER_HISTORICAL_ROOT`, `Root` |

#### get_randao_mix

```
def get_randao_mix(state: BeaconState, epoch: Epoch) -> Bytes32:
    """
    Return the randao mix at a recent ``epoch``.
    """
    return state.randao_mixes[epoch % EPOCHS_PER_HISTORICAL_VECTOR]
```

RANDAO mixes are stored in a circular list of length `EPOCHS_PER_HISTORICAL_VECTOR`. They are used when calculating the seed for assigning beacon proposers and committees.

| | |
|---|---|
| Used by | `get_seed`, `process_randao_mixes_reset()` , `process_randao()` |
| See also | `EPOCHS_PER_HISTORICAL_VECTOR` |

#### get_active_validator_indices

```
def get_active_validator_indices(state: BeaconState, epoch: Epoch) -> Sequence[ValidatorIndex]:
    """
    Return the sequence of active validator indices at ``epoch``.
    """
    return [ValidatorIndex(i) for i, v in enumerate(state.validators) if is_active_validator(v, epoch)]
```

Steps through the entire list of validators and returns the list of only the active ones. That is, the list of validators that have been activated but not exited as determined by `is_active_validator()`.

This function is heavily used and I'd expect it to be memoised in practice.

| | |
|---|---|
| Used by | Many places |
| Uses | `is_active_validator()` |

#### get_validator_churn_limit

```
def get_validator_churn_limit(state: BeaconState) -> uint64:
    """
    Return the validator churn limit for the current epoch.
    """
    active_validator_indices = get_active_validator_indices(state, get_current_epoch(state))
    return max(MIN_PER_EPOCH_CHURN_LIMIT, uint64(len(active_validator_indices)) // CHURN_LIMIT_QUOTIENT)
```

The "churn limit" applies when activating and exiting validators and acts as a rate-limit on changes to the validator set. The value returned by this function provides the number of validators that may

become active in an epoch, and the number of validators that may exit in an epoch.

Some small amount of churn is always allowed, set by MIN_PER_EPOCH_CHURN_LIMIT, and the amount of per-epoch churn allowed increases by one for every extra CHURN_LIMIT_QUOTIENT validators that are currently active (once the minimum has been exceeded).

In concrete terms, this means that up to four validators can enter or exit the active validator set each epoch (900 per day) until we have 327,680 active validators, at which point the limit rises to five.

| | |
|---|---|
| Used by | initiate_validator_exit(), process_registry_updates() |
| Uses | get_active_validator_indices() |
| See also | MIN_PER_EPOCH_CHURN_LIMIT, CHURN_LIMIT_QUOTIENT |

#### get_seed

```
def get_seed(state: BeaconState, epoch: Epoch, domain_type: DomainType) -> Bytes32:
    """
    Return the seed at ``epoch``.
    """
    mix = get_randao_mix(state, Epoch(epoch + EPOCHS_PER_HISTORICAL_VECTOR - MIN_SEED_LOOKAHEAD - 1))  #
                                    ↪ Avoid underflow
    return hash(domain_type + uint_to_bytes(epoch) + mix)
```

Used in get_beacon_committee(), get_beacon_proposer_index(), and get_next_sync_committee_indices() to provide the randomness for computing proposers and committees. domain_type is DOMAIN_BEACON_ATTESTER, DOMAIN_BEACON_PROPOSER, and DOMAIN_SYNC_COMMITTEE respectively.

RANDAO mixes are stored in a circular list of length EPOCHS_PER_HISTORICAL_VECTOR. The seed for an epoch is based on the randao mix from MIN_SEED_LOOKAHEAD epochs ago. This is to limit the forward visibility of randomness: see the explanation there.

The seed returned is not based only on the domain and the randao mix, but the epoch number is also mixed in. This is to handle the pathological case of no blocks being seen for more than two epochs, in which case we run out of randao updates. That could lock in forever a non-participating set of block proposers. Mixing in the epoch number means that fresh committees and proposers can continue to be selected.

| | |
|---|---|
| Used by | get_beacon_committee(), get_beacon_proposer_index(), get_next_sync_committee_indices() |
| Uses | get_randao_mix() |
| See also | EPOCHS_PER_HISTORICAL_VECTOR, MIN_SEED_LOOKAHEAD |

#### get_committee_count_per_slot

```
def get_committee_count_per_slot(state: BeaconState, epoch: Epoch) -> uint64:
    """
    Return the number of committees in each slot for the given ``epoch``.
    """
    return max(uint64(1), min(
        MAX_COMMITTEES_PER_SLOT,
        uint64(len(get_active_validator_indices(state, epoch))) // SLOTS_PER_EPOCH //
                                    ↪ TARGET_COMMITTEE_SIZE,
    ))
```

Every slot in a given epoch has the same number of beacon committees, as calculated by this function.

As far as the LMD GHOST consensus protocol is concerned, all the validators attesting in a slot effectively

act as a single large committee. However, organising them into multiple committees gives two benefits.

1. Having multiple smaller committees reduces the load on the aggregators that collect and aggregate the attestations from committee members. This is important, as validating the signatures and aggregating them takes time. The downside is that blocks need to be larger, as, in the best case, there are up to 64 aggregate attestations to store per block rather than a single large aggregate signature over all attestations.
2. It maps well onto the future plans for data shards, when each committee will be responsible for committing to a block on one shard in addition to its current duties.

There is always at least one committee per slot, and never more than `MAX_COMMITTEES_PER_SLOT`, currently 64.

Subject to these constraints, the actual number of committees per slot is $N/4096$, where $N$ is the total number of active validators.

The intended behaviour looks like this:

- The ideal case is that there are `MAX_COMMITTEES_PER_SLOT` = 64 committees per slot. This maps to one committee per slot per shard once data sharding has been implemented. These committees will be responsible for voting on shard crosslinks. There must be at least 262,144 active validators to achieve this.
- If there are fewer active validators, then the number of committees per shard is reduced below 64 in order to maintain a minimum committee size of `TARGET_COMMITTEE_SIZE` = 128. In this case, not every shard will get crosslinked at every slot (once sharding is in place).
- Finally, only if the number of active validators falls below 4096 will the committee size be reduced to less than 128. With so few validators, the chain has no meaningful security in any case.

| | |
|---|---|
| Used by | `get_beacon_committee()`, `process_attestation()` |
| Uses | `get_active_validator_indices()` |
| See also | `MAX_COMMITTEES_PER_SLOT`, `TARGET_COMMITTEE_SIZE` |

**`get_beacon_committee`**

```
def get_beacon_committee(state: BeaconState, slot: Slot, index: CommitteeIndex) ->
                                        ↪ Sequence[ValidatorIndex]:
    """
    Return the beacon committee at ``slot`` for ``index``.
    """
    epoch = compute_epoch_at_slot(slot)
    committees_per_slot = get_committee_count_per_slot(state, epoch)
    return compute_committee(
        indices=get_active_validator_indices(state, epoch),
        seed=get_seed(state, epoch, DOMAIN_BEACON_ATTESTER),
        index=(slot % SLOTS_PER_EPOCH) * committees_per_slot + index,
        count=committees_per_slot * SLOTS_PER_EPOCH,
    )
```

Beacon committees vote on the beacon block at each slot via attestations. There are up to `MAX_COMMITTEES_PER_SLOT` beacon committees per slot, and each committee is active exactly once per epoch.

This function returns the list of committee members given a slot number and an index within that slot to select the desired committee, relying on `compute_committee()` to do the heavy lifting.

Note that, since this uses `get_seed()`, we can obtain committees only up to `EPOCHS_PER_HISTORICAL_VECTOR` epochs into the past (minus `MIN_SEED_LOOKAHEAD`).

`get_beacon_committee` is used by `get_attesting_indices()` and `process_attestation()` when processing attestations coming from a committee, and by validators when checking their committee assignments and aggregation duties.

| | |
|---|---|
| Used by | `get_attesting_indices()`, `process_attestation()` |
| Uses | `get_committee_count_per_slot()`, `compute_committee()`, `get_active_validator_indices()`, `get_seed()` |
| See also | `MAX_COMMITTEES_PER_SLOT`, `DOMAIN_BEACON_ATTESTER` |

#### `get_beacon_proposer_index`

```
def get_beacon_proposer_index(state: BeaconState) -> ValidatorIndex:
    """
    Return the beacon proposer index at the current slot.
    """
    epoch = get_current_epoch(state)
    seed = hash(get_seed(state, epoch, DOMAIN_BEACON_PROPOSER) + uint_to_bytes(state.slot))
    indices = get_active_validator_indices(state, epoch)
    return compute_proposer_index(state, indices, seed)
```

Each slot, exactly one of the active validators is randomly chosen to be the proposer of the beacon block for that slot. The probability of being selected is weighted by the validator's effective balance in `compute_proposer_index()`.

The chosen block proposer does not need to be a member of one of the beacon committees for that slot: it is chosen from the entire set of active validators for that epoch.

The RANDAO seed returned by `get_seed()` is updated once per epoch. The slot number is mixed into the seed using a hash to allow us to choose a different proposer at each slot. This also protects us in the case that there is an entire epoch of empty blocks. If that were to happen the RANDAO would not be updated, but we would still be able to select a different set of proposers for the next epoch via this slot number mix-in process.

There is a chance of the same proposer being selected in two consecutive slots, or more than once per epoch. If every validator has the same effective balance, then the probability of being selected in a particular slot is simply $\frac{1}{N}$ independent of any other slot, where $N$ is the number of active validators in the epoch corresponding to the slot.

| | |
|---|---|
| Used by | `slash_validator()`, `process_block_header()`, `process_randao()`, `process_attestation()`, `process_sync_aggregate()` |
| Uses | `get_seed()`, `uint_to_bytes()`, `get_active_validator_indices()`, `compute_proposer_index()` |

#### `get_total_balance`

```
def get_total_balance(state: BeaconState, indices: Set[ValidatorIndex]) -> Gwei:
    """
    Return the combined effective balance of the ``indices``.
    ``EFFECTIVE_BALANCE_INCREMENT`` Gwei minimum to avoid divisions by zero.
    Math safe up to ~10B ETH, afterwhich this overflows uint64.
    """
    return Gwei(max(EFFECTIVE_BALANCE_INCREMENT, sum([state.validators[index].effective_balance for index
                                        ↪ in indices])))
```

A simple utility that returns the total balance of all validators in the list, `indices`, passed in.

As an aside, there is an interesting example of some fragility in the spec lurking here. This function used to return a minimum of 1 Gwei to avoid a potential division by zero in the calculation of rewards and penalties. However, the rewards calculation was modified to avoid a possible integer overflow condition, without modifying this function, which re-introduced the possibility of a division by zero. This was

later fixed by returning a minimum of `EFFECTIVE_BALANCE_INCREMENT`. The formal verification of the specification is helpful in avoiding issues like this.

| | |
|---|---|
| Used by | `get_total_active_balance()`, `get_flag_index_deltas()`, `process_justification_and_finalization()` |
| See also | `EFFECTIVE_BALANCE_INCREMENT` |

#### `get_total_active_balance`

```
def get_total_active_balance(state: BeaconState) -> Gwei:
    """
    Return the combined effective balance of the active validators.
    Note: ``get_total_balance`` returns ``EFFECTIVE_BALANCE_INCREMENT`` Gwei minimum to avoid divisions
                                    ↪ by zero.
    """
    return get_total_balance(state, set(get_active_validator_indices(state, get_current_epoch(state))))
```

Uses `get_total_balance()` to calculate the sum of the effective balances of all active validators in the current epoch.

This quantity is frequently used in the spec. For example, Casper FFG uses the total active balance to judge whether the 2/3 majority threshold of attestations has been reached in justification and finalisation. And it is a fundamental part of the calculation of rewards and penalties. The base reward is proportional to the reciprocal of the square root of the total active balance. Thus, validator rewards are higher when little balance is at stake (few active validators) and lower when much balance is at stake (many active validators).

Since it is calculated from effective balances, total active balance does not change during an epoch, so is a great candidate for being cached.

| | |
|---|---|
| Used by | `get_flag_index_deltas()`, `process_justification_and_finalization()`, `get_base_reward_per_increment()`, `process_slashings()`, `process_sync_aggregate()` |
| Uses | `get_total_balance()`, `get_active_validator_indices()` |

#### `get_domain`

```
def get_domain(state: BeaconState, domain_type: DomainType, epoch: Epoch=None) -> Domain:
    """
    Return the signature domain (fork version concatenated with domain type) of a message.
    """
    epoch = get_current_epoch(state) if epoch is None else epoch
    fork_version = state.fork.previous_version if epoch < state.fork.epoch else state.fork.current_version
    return compute_domain(domain_type, fork_version, state.genesis_validators_root)
```

`get_domain()` pops up whenever signatures need to be verified, since a `DomainType` is always mixed in to the signed data. For the science behind domains, see Domain types and `compute_domain()`.

With the exception of `DOMAIN_DEPOSIT`, domains are always combined with the fork version before being used in signature generation. This is to distinguish messages from different chains, and ensure that validators don't get slashed if they choose to participate on two independent forks. (That is, deliberate forks, aka hard-forks. Participating on both branches of temporary consensus forks is punishable: that's basically the whole point of slashing.)

Note that a message signed under one fork version will be valid during the next fork version, but not thereafter. So, for example, voluntary exit messages signed during Altair will be valid after the Bellatrix

beacon chain upgrade, but not after the Capella upgrade (the one after Bellatrix). Voluntary exit messages signed during Phase 0 are valid under Altair but will be made invalid by the Bellatrix upgrade.

| | |
|---|---|
| Used by | `is_valid_indexed_attestation()`, `verify_block_signature()`, `process_randao()`, `process_proposer_slashing()`, `process_voluntary_exit()`, `process_sync_aggregate()` |
| Uses | `compute_domain()` |
| See also | `DomainType`, Domain types |

#### `get_indexed_attestation`

```
def get_indexed_attestation(state: BeaconState, attestation: Attestation) -> IndexedAttestation:
    """
    Return the indexed attestation corresponding to ``attestation``.
    """
    attesting_indices = get_attesting_indices(state, attestation.data, attestation.aggregation_bits)

    return IndexedAttestation(
        attesting_indices=sorted(attesting_indices),
        data=attestation.data,
        signature=attestation.signature,
    )
```

Lists of validators within committees occur in two forms in the specification.

- They can be compressed into a bitlist, in which each bit represents the presence or absence of a validator from a particular committee. The committee is referenced by slot, and committee index within that slot. This is how sets of validators are represented in `Attestation`s.
- Or they can be listed explicitly by their validator indices, as in `IndexedAttestation`s. Note that the list of indices is sorted: an attestation is invalid if not.

`get_indexed_attestation()` converts from the former representation to the latter. The slot number and the committee index are provided by the `AttestationData` and are used to reconstruct the committee members via `get_beacon_committee()`. The supplied bitlist will have come from an `Attestation`.

Attestations are aggregatable, which means that attestations from multiple validators making the same vote can be rolled up into a single attestation through the magic of BLS signature aggregation. However, in order to be able to verify the signature later, a record needs to be kept of which validators actually contributed to the attestation. This is so that those validators' public keys can be aggregated to match the construction of the signature.

The conversion from the bit-list format to the list format is performed by `get_attesting_indices()`, below.

| | |
|---|---|
| Used by | `process_attestation()` |
| Uses | `get_attesting_indices()` |
| See also | `Attestation`, `IndexedAttestation` |

#### `get_attesting_indices`

```
def get_attesting_indices(state: BeaconState,
                          data: AttestationData,
                          bits: Bitlist[MAX_VALIDATORS_PER_COMMITTEE]) -> Set[ValidatorIndex]:
    """
    Return the set of attesting indices corresponding to ``data`` and ``bits``.
    """
    committee = get_beacon_committee(state, data.slot, data.index)
```

```
    return set(index for i, index in enumerate(committee) if bits[i])
```

As described under `get_indexed_attestation()`, lists of validators come in two forms. This routine converts from the compressed form, in which validators are represented as a subset of a committee with their presence or absence indicated by a 1 or 0 bit respectively, to an explicit list of `ValidatorIndex` types.

| | |
|---|---|
| Used by | `get_indexed_attestation()`, `process_attestation()`, `translate_participation()` |
| Uses | `get_beacon_committee()` |
| See also | `AttestationData`, `IndexedAttestation` |

**`get_next_sync_committee_indices`**

```
def get_next_sync_committee_indices(state: BeaconState) -> Sequence[ValidatorIndex]:
    """
    Return the sync committee indices, with possible duplicates, for the next sync committee.
    """
    epoch = Epoch(get_current_epoch(state) + 1)

    MAX_RANDOM_BYTE = 2**8 - 1
    active_validator_indices = get_active_validator_indices(state, epoch)
    active_validator_count = uint64(len(active_validator_indices))
    seed = get_seed(state, epoch, DOMAIN_SYNC_COMMITTEE)
    i = 0
    sync_committee_indices: List[ValidatorIndex] = []
    while len(sync_committee_indices) < SYNC_COMMITTEE_SIZE:
        shuffled_index = compute_shuffled_index(uint64(i % active_validator_count),
                                                ↪ active_validator_count, seed)
        candidate_index = active_validator_indices[shuffled_index]
        random_byte = hash(seed + uint_to_bytes(uint64(i // 32)))[i % 32]
        effective_balance = state.validators[candidate_index].effective_balance
        if effective_balance * MAX_RANDOM_BYTE >= MAX_EFFECTIVE_BALANCE * random_byte:
            sync_committee_indices.append(candidate_index)
        i += 1
    return sync_committee_indices
```

`get_next_sync_committee_indices()` is used to select the subset of validators that will make up a sync committee. The committee size is `SYNC_COMMITTEE_SIZE`, and the committee is allowed to contain duplicates, that is, the same validator more than once. This is to handle gracefully the situation of there being fewer active validators than `SYNC_COMMITTEE_SIZE`.

Similarly to being chosen to propose a block, the probability of any validator being selected for a sync committee is proportional to its effective balance. Thus, the algorithm is almost the same as that of `compute_proposer_index()`, except that this one exits only after finding `SYNC_COMMITTEE_SIZE` members, rather than exiting as soon as a candidate is found. Both routines use the try-and-increment method to weight the probability of selection with the validators' effective balances.

It's fairly clear why block proposers are selected with a probability proportional to their effective balances: block production is subject to slashing, and proposers with less at stake have less to slash, so we reduce their influence accordingly. It is not so clear why the probability of being in a sync committee is also proportional to a validator's effective balance; sync committees are not subject to slashing. It has to do with keeping calculations for light clients simple. We don't want to burden light clients with summing up validators' balances to judge whether a 2/3 supermajority of stake in the committee has voted for a block. Ideally, they can just count the participation flags. To make this somewhat reliable, we weight the probability that a validator participates in proportion to its effective balance.

| | |
|---|---|
| Used by | `get_next_sync_committee()` |
| Uses | `get_active_validator_indices()`, `get_seed()`, `compute_shuffled_index()`, `uint_to_bytes()` |

| See also | SYNC_COMMITTEE_SIZE, compute_proposer_index() |
|---|---|

#### get_next_sync_committee

> *Note*: The function `get_next_sync_committee` should only be called at sync committee period boundaries and when upgrading state to Altair.

The random seed that generates the sync committee is based on the number of the next epoch. `get_next_sync_committee_indices()` doesn't contain any check that the epoch corresponds to a sync-committee change boundary, which allowed the timing of the Altair upgrade to be more flexible. But a consequence is that you will get an incorrect committee if you call `get_next_sync_committee()` at the wrong time.

```
def get_next_sync_committee(state: BeaconState) -> SyncCommittee:
    """
    Return the next sync committee, with possible pubkey duplicates.
    """
    indices = get_next_sync_committee_indices(state)
    pubkeys = [state.validators[index].pubkey for index in indices]
    aggregate_pubkey = eth_aggregate_pubkeys(pubkeys)
    return SyncCommittee(pubkeys=pubkeys, aggregate_pubkey=aggregate_pubkey)
```

`get_next_sync_committee()` is a simple wrapper around `get_next_sync_committee_indices()` that packages everything up into a nice `SyncCommittee` object.

See the `SyncCommittee` type for an explanation of how the `aggregate_pubkey` is intended to be used.

| Used by | process_sync_committee_updates(), initialize_beacon_state_from_eth1(), upgrade_to_altair() |
|---|---|
| Uses | get_next_sync_committee_indices(), eth_aggregate_pubkeys() |
| See also | SyncCommittee |

#### get_unslashed_participating_indices

```
def get_unslashed_participating_indices(state: BeaconState, flag_index: int, epoch: Epoch) ->
                                        ↪ Set[ValidatorIndex]:
    """
    Return the set of validator indices that are both active and unslashed for the given ``flag_index``
                                        ↪ and ``epoch``.
    """
    assert epoch in (get_previous_epoch(state), get_current_epoch(state))
    if epoch == get_current_epoch(state):
        epoch_participation = state.current_epoch_participation
    else:
        epoch_participation = state.previous_epoch_participation
    active_validator_indices = get_active_validator_indices(state, epoch)
    participating_indices = [i for i in active_validator_indices if has_flag(epoch_participation[i],
                                        ↪ flag_index)]
    return set(filter(lambda index: not state.validators[index].slashed, participating_indices))
```

`get_unslashed_participating_indices()` returns the list of validators that made a timely attestation with the type `flag_index` during the `epoch` in question.

It is used with the `TIMELY_TARGET_FLAG_INDEX` flag in `process_justification_and_finalization()` to calculate the proportion of stake that voted for the candidate checkpoint in the current and previous epochs.

It is also used with the `TIMELY_TARGET_FLAG_INDEX` for applying inactivity penalties in `process_inactivity_updates()` and `get_inactivity_penalty_deltas()`. If a validator misses a correct target vote during an

inactivity leak then it is considered not to have participated at all (it is not contributing anything useful).

And it is used in `get_flag_index_deltas()` for calculating rewards due for each type of correct vote.

Slashed validators are ignored. Once slashed, validators no longer receive rewards or participate in consensus, although they are subject to penalties until they have finally been exited.

| | |
|---|---|
| Used by | `get_flag_index_deltas()`, `process_justification_and_finalization()`, `process_inactivity_updates()`, `get_inactivity_penalty_deltas()` |
| Uses | `get_active_validator_indices()`, `has_flag()` |
| See also | Participation flag indices |

**`get_attestation_participation_flag_indices`**

```
def get_attestation_participation_flag_indices(state: BeaconState,
                                               data: AttestationData,
                                               inclusion_delay: uint64) -> Sequence[int]:
    """
    Return the flag indices that are satisfied by an attestation.
    """
    if data.target.epoch == get_current_epoch(state):
        justified_checkpoint = state.current_justified_checkpoint
    else:
        justified_checkpoint = state.previous_justified_checkpoint

    # Matching roots
    is_matching_source = data.source == justified_checkpoint
    is_matching_target = is_matching_source and data.target.root == get_block_root(state,
                                      ↪ data.target.epoch)
    is_matching_head = is_matching_target and data.beacon_block_root == get_block_root_at_slot(state,
                                      ↪ data.slot)
    assert is_matching_source

    participation_flag_indices = []
    if is_matching_source and inclusion_delay <= integer_squareroot(SLOTS_PER_EPOCH):
        participation_flag_indices.append(TIMELY_SOURCE_FLAG_INDEX)
    if is_matching_target and inclusion_delay <= SLOTS_PER_EPOCH:
        participation_flag_indices.append(TIMELY_TARGET_FLAG_INDEX)
    if is_matching_head and inclusion_delay == MIN_ATTESTATION_INCLUSION_DELAY:
        participation_flag_indices.append(TIMELY_HEAD_FLAG_INDEX)

    return participation_flag_indices
```

This is called by `process_attestation()` during block processing, and is the heart of the mechanism for recording validators' votes as contained in their attestations. It filters the given attestation against the beacon state's current view of the chain, and returns participation flag indices only for the votes that are both correct and timely.

`data` is an `AttestationData` object that contains the source, target, and head votes of the validators that contributed to the attestation. The attestation may represent the votes of one or more validators.

`inclusion_delay` is the difference between the current slot on the beacon chain and the slot for which the attestation was created. For the block containing the attestation to be valid, `inclusion_delay` must be between `MIN_ATTESTATION_INCLUSION_DELAY` and `SLOTS_PER_EPOCH` inclusive. In other words, attestations must be included in the next block, or in any block up to 32 slots later, after which they are ignored.

Since the attestation may be up to 32 slots old, it might have been generated in the current epoch or the previous epoch, so the first thing we do is to check the attestation's target vote epoch to see which epoch we should be looking at in the beacon state.

Next, we check whether each of the votes in the attestation are correct:

- Does the attestation's source vote match what we believe to be the justified checkpoint in the epoch in question?
- If so, does the attestation's target vote match the head block at the epoch's checkpoint, that is, the first slot of the epoch?
- If so, does the attestation's head vote match what we believe to be the head block at the attestation's slot? Note that the slot may not contain a block – it may be a skip slot – in which case the last known block is considered to be the head.

These three build on each other, so that it is not possible to have a correct target vote without a correct source vote, and it is not possible to have a correct head vote without a correct target vote.

The `assert` statement is interesting. If an attestation does not have the correct source vote, the block containing it is invalid and is discarded. Having an incorrect source vote means that the block proposer disagrees with me about the last justified checkpoint, which is an irreconcilable difference.

After checking the validity of the votes, the timeliness of each vote is checked. Let's take them in reverse order.

- Correct head votes must be included immediately, that is, in the very next slot.
  - Head votes, used for LMD GHOST consensus, are not useful after one slot.
- Correct target votes must be included within 32 slots, one epoch.
  - Target votes are useful at any time, but it is simpler if they don't span more than a couple of epochs, so 32 slots is a reasonable limit. This check is actually redundant since attestations in blocks cannot be older than 32 slots.
- Correct source votes must be included within 5 slots (`integer_squareroot(32)`).
  - This is the geometric mean of 1 (the timely head threshold) and 32 (the timely target threshold). This is an arbitrary choice. Vitalik's view[43] is that, with this setting, the cumulative timeliness rewards most closely match an exponentially decreasing curve, which "feels more logical".

The timely inclusion requirements are new in Altair. In Phase 0, all correct votes received a reward, and there was an additional reward for inclusion the was proportional to the reciprocal of the inclusion distance. This led to a oddity where it was always more profitable to vote for a correct head, even if that meant waiting longer and risking not being included in the next slot.

| | |
|---|---|
| Used by | `process_attestation()`, `translate_participation()` |
| Uses | `get_block_root()`, `get_block_root_at_slot()`, `integer_squareroot()` |
| See also | Participation flag indices, `AttestationData`, `MIN_ATTESTATION_INCLUSION_DELAY` |

**get_flag_index_deltas**

```
def get_flag_index_deltas(state: BeaconState, flag_index: int) -> Tuple[Sequence[Gwei], Sequence[Gwei]]:
    """
    Return the deltas for a given ``flag_index`` by scanning through the participation flags.
    """
    rewards = [Gwei(0)] * len(state.validators)
    penalties = [Gwei(0)] * len(state.validators)
    previous_epoch = get_previous_epoch(state)
    unslashed_participating_indices = get_unslashed_participating_indices(state, flag_index,
                                      ↪ previous_epoch)
    weight = PARTICIPATION_FLAG_WEIGHTS[flag_index]
```

[43] From a conversation on the Ethereum Research Discord server.

```
    unslashed_participating_balance = get_total_balance(state, unslashed_participating_indices)
    unslashed_participating_increments = unslashed_participating_balance // EFFECTIVE_BALANCE_INCREMENT
    active_increments = get_total_active_balance(state) // EFFECTIVE_BALANCE_INCREMENT
    for index in get_eligible_validator_indices(state):
        base_reward = get_base_reward(state, index)
        if index in unslashed_participating_indices:
            if not is_in_inactivity_leak(state):
                reward_numerator = base_reward * weight * unslashed_participating_increments
                rewards[index] += Gwei(reward_numerator // (active_increments * WEIGHT_DENOMINATOR))
        elif flag_index != TIMELY_HEAD_FLAG_INDEX:
            penalties[index] += Gwei(base_reward * weight // WEIGHT_DENOMINATOR)
    return rewards, penalties
```

This function is used during epoch processing to assign rewards and penalties to individual validators based on their voting record in the previous epoch. Rewards for block proposers for including attestations are calculated during block processing. The "deltas" in the function name are the separate lists of rewards and penalties returned. Rewards and penalties are always treated separately to avoid negative numbers.

The function is called once for each of the flag types corresponding to correct attestation votes: timely source, timely target, timely head.

The list of validators returned by `get_unslashed_participating_indices()` contains the ones that will be rewarded for making this vote type in a timely and correct manner. That routine uses the flags set in state for each validator by `process_attestation()` during block processing and returns the validators for which the corresponding flag is set.

Every active validator is expected to make an attestation exactly once per epoch, so we then cycle through the entire set of active validators, rewarding them if they appear in `unslashed_participating_indices`, as long as we are not in an inactivity leak. If we are in a leak, no validator is rewarded for any of its votes, but penalties still apply to non-participating validators.

Notice that the reward is weighted with `unslashed_participating_increments`, which is proportional to the total stake of the validators that made a correct vote with this flag. This means that, if participation by other validators is lower, then my rewards are lower, even if I perform my duties perfectly. The reason for this is to do with discouragement attacks (see also this nice explainer). In short, with this mechanism, validators are incentivised to help each other out (e.g. by forwarding gossip messages, or aggregating attestations well) rather than to attack or censor one-another.

Validators that did not make a correct and timely vote are penalised with a full weighted base reward for each flag that they missed, except for missing the head vote. Head votes have only a single slot to get included, so a missing block in the next slot is sufficient to cause a miss, but is completely outside the attester's control. Thus head votes are only rewarded, not penalised. This also allows perfectly performing validators to break even during an inactivity leak, when we expect at least a third of blocks to be missing: they receive no rewards, but ideally no penalties either.

Untangling the arithmetic, the maximum total issuance due to rewards for attesters in an epoch, $I_A$, comes out as follows, in the notation described later.

$$I_A = \frac{W_s + W_t + W_h}{W_\Sigma} NB$$

| | |
|---|---|
| Used by | `process_rewards_and_penalties()` |
| Uses | `get_unslashed_participating_indices()`, `get_total_balance()`, `get_total_active_balance()`, `get_eligible_validator_indices()`, `get_base_reward()`, `is_in_inactivity_leak()` |
| See also | `process_attestation()`, participation flag indices, rewards and penalties |

## Beacon State Mutators

#### `increase_balance`

```
def increase_balance(state: BeaconState, index: ValidatorIndex, delta: Gwei) -> None:
    """
    Increase the validator balance at index ``index`` by ``delta``.
    """
    state.balances[index] += delta
```

After creating a validator with its deposit balance, this and `decrease_balance()` are the only places in the spec where validator balances are ever modified.

We need two separate functions to change validator balances, one to increase them and one to decrease them, since we are using only unsigned integers.

Fun fact: A typo around this led to Teku's one and only consensus failure at the initial client interop event. Unsigned integers induce bugs!

| | |
|---|---|
| Used by | `slash_validator()`, `process_rewards_and_penalties()`, `process_attestation()`, `process_deposit()`, `process_sync_aggregate()` |
| See also | `decrease_balance()` |

#### `decrease_balance`

```
def decrease_balance(state: BeaconState, index: ValidatorIndex, delta: Gwei) -> None:
    """
    Decrease the validator balance at index ``index`` by ``delta``, with underflow protection.
    """
    state.balances[index] = 0 if delta > state.balances[index] else state.balances[index] - delta
```

The counterpart to `increase_balance()`. This has a little extra work to do to check for unsigned int underflow since balances may not go negative.

| | |
|---|---|
| Used by | `slash_validator()`, `process_rewards_and_penalties()`, `process_slashings()`, `process_sync_aggregate()` |
| See also | `increase_balance()` |

#### `initiate_validator_exit`

```
def initiate_validator_exit(state: BeaconState, index: ValidatorIndex) -> None:
    """
    Initiate the exit of the validator with index ``index``.
    """
    # Return if validator already initiated exit
    validator = state.validators[index]
    if validator.exit_epoch != FAR_FUTURE_EPOCH:
        return

    # Compute exit queue epoch
    exit_epochs = [v.exit_epoch for v in state.validators if v.exit_epoch != FAR_FUTURE_EPOCH]
    exit_queue_epoch = max(exit_epochs + [compute_activation_exit_epoch(get_current_epoch(state))])
    exit_queue_churn = len([v for v in state.validators if v.exit_epoch == exit_queue_epoch])
    if exit_queue_churn >= get_validator_churn_limit(state):
        exit_queue_epoch += Epoch(1)

    # Set validator exit epoch and withdrawable epoch
```

```
    validator.exit_epoch = exit_queue_epoch
    validator.withdrawable_epoch = Epoch(validator.exit_epoch + MIN_VALIDATOR_WITHDRAWABILITY_DELAY)
```

Exits may be initiated voluntarily, as a result of being slashed, or by dropping to the `EJECTION_BALANCE` threshold.

In all cases, a dynamic "churn limit" caps the number of validators that may exit per epoch. This is calculated by `get_validator_churn_limit()`. The mechanism for enforcing this is the exit queue: the validator's `exit_epoch` is set such that it is at the end of the queue.

The exit queue is not maintained as a separate data structure, but is continually re-calculated from the exit epochs of all validators and allowing for a fixed number to exit per epoch. I expect there are some optimisations to be had around this in actual implementations.

An exiting validator is expected to continue with its proposing and attesting duties until its `exit_epoch` has passed, and will continue to receive rewards and penalties accordingly.

In addition, an exited validator remains eligible to be slashed until its `withdrawable_epoch`, which is set to `MIN_VALIDATOR_WITHDRAWABILITY_DELAY` epochs after its `exit_epoch`. This is to allow some extra time for any slashable offences by the validator to be detected and reported.

| | |
|---|---|
| Used by | `slash_validator()`, `process_registry_updates()`, `process_voluntary_exit()` |
| Uses | `compute_activation_exit_epoch()`, `get_validator_churn_limit()` |
| See also | Voluntary Exits, `MIN_VALIDATOR_WITHDRAWABILITY_DELAY` |

**`slash_validator`**

```
def slash_validator(state: BeaconState,
                    slashed_index: ValidatorIndex,
                    whistleblower_index: ValidatorIndex=None) -> None:
    """
    Slash the validator with index ``slashed_index``.
    """
    epoch = get_current_epoch(state)
    initiate_validator_exit(state, slashed_index)
    validator = state.validators[slashed_index]
    validator.slashed = True
    validator.withdrawable_epoch = max(validator.withdrawable_epoch, Epoch(epoch +
                                       ↪ EPOCHS_PER_SLASHINGS_VECTOR))
    state.slashings[epoch % EPOCHS_PER_SLASHINGS_VECTOR] += validator.effective_balance
    decrease_balance(state, slashed_index, validator.effective_balance //
                                       ↪ MIN_SLASHING_PENALTY_QUOTIENT_ALTAIR)

    # Apply proposer and whistleblower rewards
    proposer_index = get_beacon_proposer_index(state)
    if whistleblower_index is None:
        whistleblower_index = proposer_index
    whistleblower_reward = Gwei(validator.effective_balance // WHISTLEBLOWER_REWARD_QUOTIENT)
    proposer_reward = Gwei(whistleblower_reward * PROPOSER_WEIGHT // WEIGHT_DENOMINATOR)
    increase_balance(state, proposer_index, proposer_reward)
    increase_balance(state, whistleblower_index, Gwei(whistleblower_reward - proposer_reward))
```

Both proposer slashings and attester slashings end up here when a report of a slashable offence has been verified during block processing.

When a validator is slashed, several things happen immediately:

- The validator is processed for exit via `initiate_validator_exit()`, so it joins the exit queue.

- The validator is marked as slashed. This information is used when calculating rewards and penalties: while being exited, whatever it does, a slashed validator receives penalties as if it had failed to propose or attest, including the inactivity leak if applicable.
- Normally, as part of the exit process, the `withdrawable_epoch` for a validator (the point at which a validator's stake is in principle unlocked) is set to `MIN_VALIDATOR_WITHDRAWABILITY_DELAY` epochs after it exits. When a validator is slashed, a much longer period of lock-up applies, namely `EPOCHS_PER_SLASHINGS_VECTOR`. This is to allow a further, potentially much greater, slashing penalty to be applied later once the chain knows how many validators have been slashed together around the same time. The postponement of the withdrawable epoch is twice as long as required to apply the extra penalty, which is applied half-way through the period. This simply means that slashed validators continue to accrue attestation penalties for some 18 days longer than necessary. Treating slashed validators fairly is not a big priority for the protocol.
- The effective balance of the validator is added to the accumulated effective balances of validators slashed this epoch, and stored in the circular list, `state.slashings`. This will later be used by the slashing penalty calculation mentioned in the previous point.
- An initial "slap on the wrist" slashing penalty of the validator's effective balance (in Gwei) divided by the `MIN_SLASHING_PENALTY_QUOTIENT_ALTAIR` is applied. With current values, this is a maximum of 0.5 Ether, increased from 0.25 Ether in Phase 0. The plan is to increase this to 1 Ether at The Merge.
- The block proposer that included the slashing proof receives a reward.

In short, a slashed validator receives an initial minor penalty, can expect to receive a further penalty later, and is marked for exit.

Note that the `whistleblower_index` defaults to `None` in the parameter list. This is never used in Phase 0, with the result that the proposer that included the slashing gets the entire whistleblower reward; there is no separate whistleblower reward for the finder of proposer or attester slashings. One reason is simply that reports are too easy to steal: if I report a slashable event to a block proposer, there is nothing to prevent that proposer claiming the report as its own. We could introduce some fancy ZK protocol to make this trustless, but this is what we're going with for now. Later developments, such as the proof-of-custody game, may reward whistleblowers directly.

| | |
|---|---|
| Used by | `process_proposer_slashing()`, `process_attester_slashing()` |
| Uses | `initiate_validator_exit()`, `get_beacon_proposer_index()`, `decrease_balance()`, `increase_balance()` |
| See also | `EPOCHS_PER_SLASHINGS_VECTOR`, `MIN_SLASHING_PENALTY_QUOTIENT_ALTAIR`, `process_slashings()` |

# Beacon Chain State Transition Function

## Preamble

> The post-state corresponding to a pre-state `state` and a signed block `signed_block` is defined as `state_transition(state, signed_block)`. State transitions that trigger an unhandled exception (e.g. a failed `assert` or an out-of-range list access) are considered invalid. State transitions that cause a `uint64` overflow or underflow are also considered invalid.

This is a very important statement of how the spec deals with invalid conditions and errors. Basically, if any block is processed that would trigger any kind of exception in the Python code of the specification, then that block is invalid and must be rejected. That means having to undo any state modifications already made in the course of processing the block.

People who do formal verification of the specification don't much like this, as having assert statements in running code is an anti-pattern: it is better to ensure that your code can simply never fail.

Anyway, the beacon chain state transition has three elements:

1. slot processing, which is performed for every slot regardless of what else is happening;
2. epoch processing, which happens every `SLOTS_PER_EPOCH` (32) slots, again regardless of whatever else is going on; and,
3. block processing, which happens only in slots for which a beacon block has been received.

```
def state_transition(state: BeaconState, signed_block: SignedBeaconBlock, validate_result: bool=True) ->
                                        ↪ None:
    block = signed_block.message
    # Process slots (including those with no blocks) since block
    process_slots(state, block.slot)
    # Verify signature
    if validate_result:
        assert verify_block_signature(state, signed_block)
    # Process block
    process_block(state, block)
    # Verify state root
    if validate_result:
        assert block.state_root == hash_tree_root(state)
```

As the spec is written, a state transition is triggered by receiving a block to process. That means that we first need to fast forward from our current slot number in the state (which is the slot at which we last processed a block) to the slot of the block we are processing. We treat intervening slots, if any, as empty. This "fast-forward" is done by `process_slots()`, which also triggers epoch processing as required.

In actual client implementations, state updates will usually be time-based, triggered by moving to the next slot if a block has not been received. However, the fast-forward functionality will be used when exploring different forks in the block tree.

The `validate_result` parameter defaults to `True`, meaning that the block's signature will be checked, and that the result of applying the block to the state results in the same state root that the block claims it does (the "post-states" must match). When creating blocks, however, proposers can set `validate_result` to `False` to allow the state root to be calculated, else we'd have a circular dependency. The signature over the initial candidate block is omitted to avoid bad interactions with slashing protection when signing twice in a slot.

| | |
|---|---|
| Uses | `process_slots()`, `verify_block_signature`, `process_block` |

```
def verify_block_signature(state: BeaconState, signed_block: SignedBeaconBlock) -> bool:
    proposer = state.validators[signed_block.message.proposer_index]
    signing_root = compute_signing_root(signed_block.message, get_domain(state, DOMAIN_BEACON_PROPOSER))
    return bls.Verify(proposer.pubkey, signing_root, signed_block.signature)
```

Check that the signature on the block matches the block's contents and the public key of the claimed proposer of the block. This ensures that blocks cannot be forged, or tampered with in transit. All the public keys for validators are stored in the `Validator`s list in state.

| | |
|---|---|
| Used by | `state_transition()` |
| Uses | `compute_signing_root()`, `get_domain()`, `bls.Verify()` |
| See also | `DOMAIN_BEACON_PROPOSER` |

```
def process_slots(state: BeaconState, slot: Slot) -> None:
    assert state.slot < slot
    while state.slot < slot:
        process_slot(state)
        # Process epoch on the start slot of the next epoch
        if (state.slot + 1) % SLOTS_PER_EPOCH == 0:
            process_epoch(state)
        state.slot = Slot(state.slot + 1)
```

Updates the state from its current slot up to the given slot number assuming that all the intermediate slots are empty (that they do not contain blocks). Iteratively calls `process_slot()` to apply the empty slot state-transition.

This is where epoch processing is triggered when required. Empty slot processing is extremely light weight, but any epoch transitions that need to be processed require the full rewards and penalties, and justification–finalisation apparatus.

| | |
|---|---|
| Used by | `state_transition()` |
| Uses | `process_slot()`, `process_epoch()` |
| See also | `SLOTS_PER_EPOCH` |

```
def process_slot(state: BeaconState) -> None:
    # Cache state root
    previous_state_root = hash_tree_root(state)
    state.state_roots[state.slot % SLOTS_PER_HISTORICAL_ROOT] = previous_state_root
    # Cache latest block header state root
    if state.latest_block_header.state_root == Bytes32():
        state.latest_block_header.state_root = previous_state_root
    # Cache block root
    previous_block_root = hash_tree_root(state.latest_block_header)
    state.block_roots[state.slot % SLOTS_PER_HISTORICAL_ROOT] = previous_block_root
```

Apply a single slot state-transition (but updating the slot number, and any required epoch processing is handled by `process_slots()`). This is done at each slot whether or not there is a block present; if there is no block present then it is the only thing that is done.

Slot processing is almost trivial and consists only of calculating the updated state and block hash tree roots (as necessary), and storing them in the historical lists in the state. In a circular way, the state roots only change over an the empty slot state transition due to updating the lists of state and block roots.

`SLOTS_PER_HISTORICAL_ROOT` is a multiple of `SLOTS_PER_EPOCH`, so there is no danger of overwriting the circular lists of `state_roots` and `block_roots`. These will be dealt with correctly during epoch processing.

The only curiosity here is the lines,

```
    if state.latest_block_header.state_root == Bytes32():
        state.latest_block_header.state_root = previous_state_root
```

This logic was introduced to avoid a circular dependency while also keeping the state transition clean. Each block that we receive contains a post-state root, but as part of state processing we store the block in the state (in `state.latest_block_header`), thus changing the post-state root.

Therefore, to be able to verify the state transition, we use the convention that the state root of the incoming block, and the state root that we calculate after inserting the block into the state, are both based on a *temporary* block header that has a stubbed state root, namely `Bytes32()`. This allows the block's claimed post-state root to validated without the circularity. The next time that `process_slots()` is called, the block's stubbed state root is updated to the actual post-state root, as above.

| | |
|---|---|
| Used by | `process_slots()` |
| Uses | `hash_tree_root` |
| See also | `SLOTS_PER_HISTORICAL_ROOT` |

## Epoch processing

```
def process_epoch(state: BeaconState) -> None:
    process_justification_and_finalization(state)  # [Modified in Altair]
    process_inactivity_updates(state)  # [New in Altair]
    process_rewards_and_penalties(state)  # [Modified in Altair]
    process_registry_updates(state)
    process_slashings(state)  # [Modified in Altair]
    process_eth1_data_reset(state)
    process_effective_balance_updates(state)
    process_slashings_reset(state)
    process_randao_mixes_reset(state)
    process_historical_roots_update(state)
    process_participation_flag_updates(state)  # [New in Altair]
    process_sync_committee_updates(state)  # [New in Altair]
```

The long laundry list of things that need to be done at the end of an epoch. You can see from the comments that a bunch of extra work was added in Altair.

| | |
|---|---|
| Used by | `process_slots()` |
| Uses | All the things below |

### Justification and finalization

```
def process_justification_and_finalization(state: BeaconState) -> None:
    # Initial FFG checkpoint values have a `0x00` stub for `root`.
    # Skip FFG updates in the first two epochs to avoid corner cases that might result in modifying this
                                        ↪ stub.
    if get_current_epoch(state) <= GENESIS_EPOCH + 1:
        return
    previous_indices = get_unslashed_participating_indices(state, TIMELY_TARGET_FLAG_INDEX,
                                        ↪ get_previous_epoch(state))
    current_indices = get_unslashed_participating_indices(state, TIMELY_TARGET_FLAG_INDEX,
                                        ↪ get_current_epoch(state))
    total_active_balance = get_total_active_balance(state)
    previous_target_balance = get_total_balance(state, previous_indices)
    current_target_balance = get_total_balance(state, current_indices)
    weigh_justification_and_finalization(state, total_active_balance, previous_target_balance,
                                        ↪ current_target_balance)
```

I believe the corner cases mentioned in the comments are related to Issue 849[44]. In any case, skipping justification and finalisation calculations during the first two epochs definitely simplifies things.

[44]Worth a visit if only to have a chuckle at Jacek's description of `uints` as "ugly integers".

For the purposes of the Casper FFG finality calculations, we want attestations that have both source and target votes we agree with. If the source vote is incorrect, then the attestation is never processed into the state, so we just need the validators that voted for the correct target, according to their participation flag indices.

Since correct target votes can be included up to 32 slots after they are made, we collect votes from both the previous epoch and the current epoch to ensure that we have them all.

Once we know which validators voted for the correct source and head in the current and previous epochs, we add up their effective balances (not actual balances). `total_active_balance` is the sum of the effective balances for all validators that ought to have voted during the current epoch. Slashed, but not exited validators are not included in these calculations.

These aggregate balances are passed to `weigh_justification_and_finalization()` to do the actual work of updating justification and finalisation.

| | |
|---|---|
| Used by | `process_epoch()` |
| Uses | `get_unslashed_participating_indices()`, `get_total_active_balance()`, `get_total_balance()`, `weigh_justification_and_finalization()` |
| See also | participation flag indices |

```
def weigh_justification_and_finalization(state: BeaconState,
                                         total_active_balance: Gwei,
                                         previous_epoch_target_balance: Gwei,
                                         current_epoch_target_balance: Gwei) -> None:
    previous_epoch = get_previous_epoch(state)
    current_epoch = get_current_epoch(state)
    old_previous_justified_checkpoint = state.previous_justified_checkpoint
    old_current_justified_checkpoint = state.current_justified_checkpoint

    # Process justifications
    state.previous_justified_checkpoint = state.current_justified_checkpoint
    state.justification_bits[1:] = state.justification_bits[:JUSTIFICATION_BITS_LENGTH - 1]
    state.justification_bits[0] = 0b0
    if previous_epoch_target_balance * 3 >= total_active_balance * 2:
        state.current_justified_checkpoint = Checkpoint(epoch=previous_epoch,
                                                        root=get_block_root(state, previous_epoch))
        state.justification_bits[1] = 0b1
    if current_epoch_target_balance * 3 >= total_active_balance * 2:
        state.current_justified_checkpoint = Checkpoint(epoch=current_epoch,
                                                        root=get_block_root(state, current_epoch))
        state.justification_bits[0] = 0b1

    # Process finalizations
    bits = state.justification_bits
    # The 2nd/3rd/4th most recent epochs are justified, the 2nd using the 4th as source
    if all(bits[1:4]) and old_previous_justified_checkpoint.epoch + 3 == current_epoch:
        state.finalized_checkpoint = old_previous_justified_checkpoint
    # The 2nd/3rd most recent epochs are justified, the 2nd using the 3rd as source
    if all(bits[1:3]) and old_previous_justified_checkpoint.epoch + 2 == current_epoch:
        state.finalized_checkpoint = old_previous_justified_checkpoint
    # The 1st/2nd/3rd most recent epochs are justified, the 1st using the 3rd as source
    if all(bits[0:3]) and old_current_justified_checkpoint.epoch + 2 == current_epoch:
        state.finalized_checkpoint = old_current_justified_checkpoint
    # The 1st/2nd most recent epochs are justified, the 1st using the 2nd as source
    if all(bits[0:2]) and old_current_justified_checkpoint.epoch + 1 == current_epoch:
        state.finalized_checkpoint = old_current_justified_checkpoint
```

This routine handles justification first, and then finalisation.

**Justification**

A supermajority link is a vote with a justified source checkpoint $C_m$ and a target checkpoint $C_n$ that was made by validators controlling more than two-thirds of the stake. If a checkpoint has a supermajority link pointing to it then we consider it justified. So, if more than two-thirds of the validators agree that checkpoint 3 was justified (their source vote) and have checkpoint 4 as their target vote, then we justify checkpoint 4.

We know that all the attestations have source votes that we agree with. The first `if` statement tries to justify the previous epoch's checkpoint seeing if the (source, target) pair is a supermajority. The second `if` statement tries to justify the current epoch's checkpoint. Note that the previous epoch's checkpoint might already have been justified; this is not checked but does not affect the logic.

The justification status of the last four epochs is stored in an array of bits in the state. After shifting the bits along by one at the outset of the routine, the justification status of the current epoch is stored in element 0, the previous in element 1, and so on.

Note that the `total_active_balance` is the current epoch's total balance, so it may not be strictly correct for calculating the supermajority for the previous epoch. However, the rate at which the validator set can change between epochs is tightly constrained, so this is not a significant issue.

**Finalisation**

The version of Casper FFG described in the Gasper paper uses $k$-finality, which extends the handling of finality in the original Casper FFG paper.

In $k$-finality, if we have a consecutive set of $k$ justified checkpoints $C_j, \ldots, C_{j+k-1}$, and a supermajority link from $C_j$ to $C_{j+k}$, then $C_j$ is finalised. Also note that this justifies $C_{j+k}$, by the rules above.

The Casper FFG version of this is 1-finality. So, a supermajority link from a justified checkpoint $C_n$ to the very next checkpoint $C_{n+1}$ both justifies $C_{n+1}$ and finalises $C_n$.

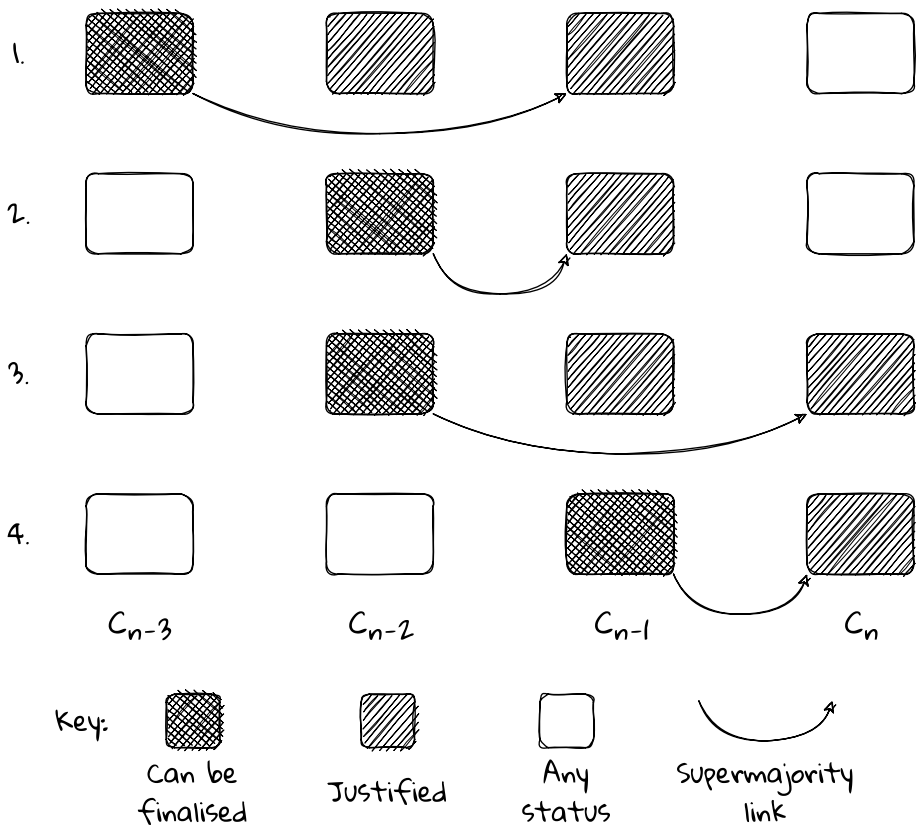
On the beacon chain we are using 2-finality, since target votes may be included up to an epoch late. In 2-finality, we keep records of checkpoint justification status for four epochs and have the following conditions for finalisation, where the checkpoint for the current epoch is $C_n$. Note that we have already updated the justification status of $C_n$ and $C_{n-1}$ in this routine, which implies the existence of supermajority links pointing to them if the corresponding bits are set, respectively.

1. Checkpoints $C_{n-3}$ and $C_{n-2}$ are justified, and there is a supermajority link from $C_{n-3}$ to $C_{n-1}$: finalise $C_{n-3}$.
2. Checkpoint $C_{n-2}$ is justified, and there is a supermajority link from $C_{n-2}$ to $C_{n-1}$: finalise $C_{n-2}$. This is equivalent to 1-finality applied to the previous epoch.
3. Checkpoints $C_{n-2}$ and $C_{n-1}$ are justified, and there is a supermajority link from $C_{n-2}$ to $C_n$: finalise $C_{n-2}$.
4. Checkpoint $C_{n-1}$ is justified, and there is a supermajority link from $C_{n-1}$ to $C_n$: finalise $C_{n-1}$. This is equivalent to 1-finality applied to the current epoch.

Almost always we would expect to see only the 1-finality cases, in particular, case 4. The 2-finality cases would occur only in situations where many attestations are delayed, or when we are very close to the 2/3rds participation threshold. Note that these evaluations stack, so it is possible for rule 2 to finalise $C_{n-2}$ and then for rule 4 to immediately finalise $C_{n-1}$, for example.

For the uninitiated, in Python's array slice syntax, `bits[1:4]` means bits 1, 2, and 3 (but not 4). This always trips me up.

| | |
|---|---|
| Used by | `process_justification_and_finalization()` |
| Uses | `get_block_root()` |
| See also | `JUSTIFICATION_BITS_LENGTH`, `Checkpoint` |



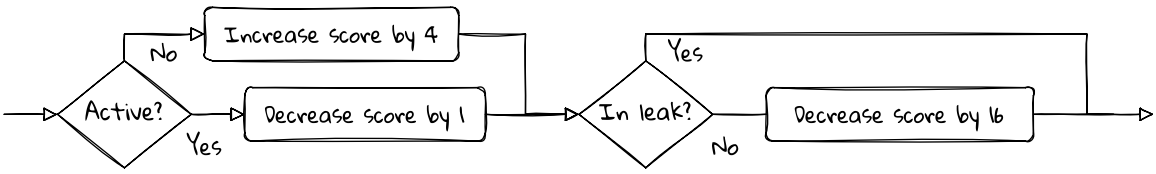

*The four k-finality scenarios. Checkpoint numbers are along the bottom.*

### Inactivity scores

```
def process_inactivity_updates(state: BeaconState) -> None:
    # Skip the genesis epoch as score updates are based on the previous epoch participation
    if get_current_epoch(state) == GENESIS_EPOCH:
        return

    for index in get_eligible_validator_indices(state):
        # Increase the inactivity score of inactive validators
        if index in get_unslashed_participating_indices(state, TIMELY_TARGET_FLAG_INDEX,
                                    ↪ get_previous_epoch(state)):
            state.inactivity_scores[index] -= min(1, state.inactivity_scores[index])
        else:
            state.inactivity_scores[index] += INACTIVITY_SCORE_BIAS
        # Decrease the inactivity score of all eligible validators during a leak-free epoch
        if not is_in_inactivity_leak(state):
            state.inactivity_scores[index] -= min(INACTIVITY_SCORE_RECOVERY_RATE,
                                    ↪ state.inactivity_scores[index])
```

With Altair, each validator has an individual inactivity score in the beacon state which is updated as follows.

- Every epoch, irrespective of the inactivity leak,
  - decrease the score by one when the validator makes a correct timely target vote, and
  - increase the score by `INACTIVITY_SCORE_BIAS` otherwise. Note that `get_eligible_validator_indices()` includes slashed but not yet withdrawable validators: slashed validators are treated as not participating, whatever they actually do.
- When *not* in an inactivity leak
  - decrease all validators' scores by `INACTIVITY_SCORE_RECOVERY_RATE`.




*How each validator's inactivity score is updated. The happy flow is right through the middle.*

There is a floor of zero on the score. So, outside a leak, validators' scores will rapidly return to zero and stay there, since `INACTIVITY_SCORE_RECOVERY_RATE` is greater than `INACTIVITY_SCORE_BIAS`.

| | |
|---|---|
| Used by | `process_epoch()` |
| Uses | `get_eligible_validator_indices()`, `get_unslashed_participating_indices()`, `is_in_inactivity_leak()` |
| See also | `INACTIVITY_SCORE_BIAS`, `INACTIVITY_SCORE_RECOVERY_RATE`, `INACTIVITY_SCORE_RECOVERY_RATE` |

### Reward and penalty calculations

Without wanting to go full Yellow Paper on you, I am going to adopt a little notation to help analyse the rewards.

We will define a base reward $B$ that we will see turns out to be the expected long-run average income of an optimally performing validator per epoch (ignoring validator set size changes). The total number of active validators is $N$.

The base reward is calculated from a base reward per increment, $b$. An "increment" is a unit of effective balance in terms of `EFFECTIVE_BALANCE_INCREMENT`. $B = 32b$ because `MAX_EFFECTIVE_BALANCE = 32 * EFFECTIVE_BALANCE_INCREMENT`

Other quantities we will use in rewards calculation are the incentivization weights: $W_s$, $W_t$, $W_h$, and $W_y$ being the weights for correct source, target, head, and sync committee votes respectively; $W_p$ being the proposer weight; and the weight denominator $W_\Sigma$ which is the sum of the weights.

Issuance for regular rewards happens in four ways:

- $I_A$ is the maximum total reward for all validators attesting in an epoch;
- $I_{A_P}$ is the maximum reward issued to proposers in an epoch for including attestations;
- $I_S$ is the maximum total reward for all sync committee participants in an epoch; and
- $I_{S_P}$ is the maximum reward issued to proposers in an epoch for including sync aggregates;

Under `get_flag_index_deltas()`, `process_attestation()`, and `process_sync_aggregate()` we find that these work out as follows in terms of $B$ and $N$:

$$I_A = \frac{W_s + W_t + W_h}{W_\Sigma} NB$$
$$I_{A_P} = \frac{W_p}{W_\Sigma - W_p} I_A$$
$$I_S = \frac{W_y}{W_\Sigma} NB$$
$$I_{S_P} = \frac{W_p}{W_\Sigma - W_p} I_S$$

To find the total optimal issuance per epoch, we can first sum $I_A$ and $I_S$,

$$I_A + I_S = \frac{W_s + W_t + W_h + W_y}{W_\Sigma} NB = \frac{W_\Sigma - W_p}{W_\Sigma} NB$$

Now adding in the proposer rewards

So, we see that every epoch, $NB$ Gwei is awarded to $N$ validators. Every validator participates in attesting, and proposing and sync committee duties are uniformly random, so the long-term expected income per optimally performing validator per epoch is $B$ Gwei.

### Helpers

```
def get_base_reward_per_increment(state: BeaconState) -> Gwei:
    return Gwei(EFFECTIVE_BALANCE_INCREMENT * BASE_REWARD_FACTOR //
                                        ↪ integer_squareroot(get_total_active_balance(state)))
```

The base reward per increment is the fundamental unit of reward in terms of which all other regular rewards and penalties are calculated. We will denote the base reward per increment, $b$.

As I noted under `BASE_REWARD_FACTOR`, this is the big knob to turn if we wish to increase or decrease the total reward for participating in Eth2, otherwise known as the issuance rate of new Ether.

An increment is a single unit of a validator's effective balance, denominated in terms of `EFFECTIVE_BALANCE_INCREMENT`, which happens to be one Ether. So, an increment is 1 Ether of effective balance, and maximally effective validator has 32 increments.

The base reward per increment is inversely proportional to the square root of the total balance of all active validators. This means that, as the number $N$ of validators increases, the reward per validator decreases as $\frac{1}{\sqrt{N}}$, and the overall issuance per epoch increases as $\sqrt{N}$.

The decrease with increasing $N$ in per-validator rewards provides a price discovery mechanism: the idea is that an equilibrium will be found where the total number of validators results in a reward similar to returns available elsewhere for similar risk. A different curve could have been chosen for the rewards profile. For example, the inverse of total balance rather than its square root would keep total issuance constant. The section on Issuance has a deeper exploration of these topics.

| | |
|---|---|
| Used by | `get_base_reward()`, `process_sync_aggregate()` |
| Uses | `integer_squareroot()`, `get_total_active_balance()` |

```
def get_base_reward(state: BeaconState, index: ValidatorIndex) -> Gwei:
    """
    Return the base reward for the validator defined by ``index`` with respect to the current ``state``.
    """
    increments = state.validators[index].effective_balance // EFFECTIVE_BALANCE_INCREMENT
    return Gwei(increments * get_base_reward_per_increment(state))
```

The base reward is the reward that an optimally performing validator can expect to earn on average per epoch, over the long term. It is proportional to the validator's effective balance; a validator with `MAX_EFFECTIVE_BALANCE` can expect to receive the full base reward $B = 32b$ per epoch on a long-term average.

| | |
|---|---|
| Used by | `get_flag_index_deltas()`, `process_attestation()` |
| Uses | `get_base_reward_per_increment()` |
| See also | `EFFECTIVE_BALANCE_INCREMENT` |

```
def get_finality_delay(state: BeaconState) -> uint64:
    return get_previous_epoch(state) - state.finalized_checkpoint.epoch
```

Returns the number of epochs since the last finalised checkpoint (minus one). In ideal running this ought to be zero: during epoch processing we aim to have justified the checkpoint in the current epoch and finalised the checkpoint in the previous epoch. A delay in finalisation suggests a chain split or a large fraction of validators going offline.

| Used by | is_in_inactivity_leak() |
|---|---|

```
def is_in_inactivity_leak(state: BeaconState) -> bool:
    return get_finality_delay(state) > MIN_EPOCHS_TO_INACTIVITY_PENALTY
```

If the beacon chain has not managed to finalise a checkpoint for MIN_EPOCHS_TO_INACTIVITY_PENALTY epochs (that is, four epochs), then the chain enters the inactivity leak. In this mode, penalties for non-participation are heavily increased, with the goal of reducing the proportion of stake controlled by non-participants, and eventually regaining finality.

| Used by | get_flag_index_deltas(), process_inactivity_updates() |
|---|---|
| Uses | get_finality_delay() |
| See also | inactivity leak, MIN_EPOCHS_TO_INACTIVITY_PENALTY |

```
def get_eligible_validator_indices(state: BeaconState) -> Sequence[ValidatorIndex]:
    previous_epoch = get_previous_epoch(state)
    return [
        ValidatorIndex(index) for index, v in enumerate(state.validators)
        if is_active_validator(v, previous_epoch) or (v.slashed and previous_epoch + 1 <
                                    ↪ v.withdrawable_epoch)
    ]
```

These are the validators that were subject to rewards and penalties in the previous epoch.

The list differs from the active validator set returned by get_active_validator_indices() by including slashed but not fully exited validators in addition to the ones marked active. Slashed validators are subject to penalties right up to when they become withdrawable and are thus fully exited.

| Used by | get_flag_index_deltas(), process_inactivity_updates(), get_inactivity_penalty_deltas() |
|---|---|
| Uses | is_active_validator() |

### Inactivity penalty deltas

```
def get_inactivity_penalty_deltas(state: BeaconState) -> Tuple[Sequence[Gwei], Sequence[Gwei]]:
    """
    Return the inactivity penalty deltas by considering timely target participation flags and inactivity
                                    ↪ scores.
    """
    rewards = [Gwei(0) for _ in range(len(state.validators))]
    penalties = [Gwei(0) for _ in range(len(state.validators))]
    previous_epoch = get_previous_epoch(state)
    matching_target_indices = get_unslashed_participating_indices(state, TIMELY_TARGET_FLAG_INDEX,
                                    ↪ previous_epoch)
    for index in get_eligible_validator_indices(state):
        if index not in matching_target_indices:
            penalty_numerator = state.validators[index].effective_balance * state.inactivity_scores[index]
            penalty_denominator = INACTIVITY_SCORE_BIAS * INACTIVITY_PENALTY_QUOTIENT_ALTAIR
            penalties[index] += Gwei(penalty_numerator // penalty_denominator)
    return rewards, penalties
```

Validators receive penalties proportional to their individual inactivity scores, even when the beacon chain is not in an inactivity leak. However, these scores reduce to zero fairly rapidly outside a leak. This is a change from Phase 0 in which inactivity penalties were applied only during leaks.

All unslashed validators that made a correct and timely target vote in the previous epoch are identified by `get_unslashed_participating_indices()`, and all other active validators receive a penalty, including slashed validators.

The penalty is proportional to the validator's effective balance and its inactivity score. See `INACTIVITY_PENALTY_QUOTIENT_ALTAIR` for more details of the calculation, and `INACTIVITY_SCORE_RECOVERY_RATE` for some charts of how the penalties accrue.

The returned `rewards` is always an array of zeros. It's here just to make the Python syntax simpler in the calling routine.

| | |
|---|---|
| Used by | `def_process_rewards_and_penalties()` |
| Uses | `get_unslashed_participating_indices()`, `get_eligible_validator_indices()` |
| See also | Inactivity Scores, `INACTIVITY_PENALTY_QUOTIENT_ALTAIR`, `INACTIVITY_SCORE_RECOVERY_RATE` |

### Process rewards and penalties

```
def process_rewards_and_penalties(state: BeaconState) -> None:
    # No rewards are applied at the end of `GENESIS_EPOCH` because rewards are for work done in the
                                        ↪ previous epoch
    if get_current_epoch(state) == GENESIS_EPOCH:
        return

    flag_deltas = [get_flag_index_deltas(state, flag_index) for flag_index in
                                        ↪ range(len(PARTICIPATION_FLAG_WEIGHTS))]
    deltas = flag_deltas + [get_inactivity_penalty_deltas(state)]
    for (rewards, penalties) in deltas:
        for index in range(len(state.validators)):
            increase_balance(state, ValidatorIndex(index), rewards[index])
            decrease_balance(state, ValidatorIndex(index), penalties[index])
```

This is where validators are rewarded and penalised according to their attestation records.

Attestations included in beacon blocks were processed by `process_attestation` as blocks were received, and flags were set in the beacon state according to their timeliness and correctness. These flags are now processed into rewards and penalties for each validator by calling `get_flag_index_deltas()` for each of the flag types.

Once the normal attestation rewards and penalties have been calculated, additional penalties based on validators' inactivity scores are accumulated.

As noted elsewhere, rewards and penalties are handled separately from each other since we don't do negative numbers.

For reference, the only other places where rewards and penalties are applied are as follows:

- during block processing: for sync committee participation, when applying the proposer reward, and when applying initial slashing rewards and penalties.
- during epoch processing: when applying extended slashing penalties.

| | |
|---|---|
| Used by | `process_epoch()` |
| Uses | `get_flag_index_deltas()`, `get_inactivity_penalty_deltas()`, `increase_balance()`, `decrease_balance()` |
| See also | `ParticipationFlags`, `PARTICIPATION_FLAG_WEIGHTS` |

### Registry updates

```
def process_registry_updates(state: BeaconState) -> None:
    # Process activation eligibility and ejections
    for index, validator in enumerate(state.validators):
        if is_eligible_for_activation_queue(validator):
            validator.activation_eligibility_epoch = get_current_epoch(state) + 1

        if (
            is_active_validator(validator, get_current_epoch(state))
            and validator.effective_balance <= EJECTION_BALANCE
        ):
            initiate_validator_exit(state, ValidatorIndex(index))

    # Queue validators eligible for activation and not yet dequeued for activation
    activation_queue = sorted([
        index for index, validator in enumerate(state.validators)
        if is_eligible_for_activation(state, validator)
        # Order by the sequence of activation_eligibility_epoch setting and then index
    ], key=lambda index: (state.validators[index].activation_eligibility_epoch, index))
    # Dequeued validators for activation up to churn limit
    for index in activation_queue[:get_validator_churn_limit(state)]:
        validator = state.validators[index]
        validator.activation_epoch = compute_activation_exit_epoch(get_current_epoch(state))
```

The `Registry` is the part of the beacon state that stores `Validator` records. These particular updates are, for the most part, concerned with moving validators through the activation queue.

`is_eligible_for_activation_queue()` finds validators that have a sufficient deposit amount yet their `activation_eligibility_epoch` is still set to `FAR_FUTURE_EPOCH`. These will be at most the validators for which deposits were processed during the last epoch, potentially up to `MAX_DEPOSITS * SLOTS_PER_EPOCH`, which is 512 (minus any partial deposits that don't yet add up to a whole deposit). These have their `activation_eligibility_epoch` set to the next epoch. They will become eligible for activation once that epoch is finalised – "eligible for activation" means only that they can be added to the activation queue; they will not become active until they reach the end of the queue.

Next, any validators whose effective balance has fallen to `EJECTION_BALANCE` have their exit initiated.

`is_eligible_for_activation()` selects validators whose `activation_eligibility_epoch` has just been finalised. The list of these is ordered by eligibility epoch, and then by index. There might be multiple eligibility epochs in the list if finalisation got delayed for some reason.

Finally, the first `get_validator_churn_limit()` validators in the list get their activation epochs set to `compute_activation_exit_epoch()`.

On first sight, you'd think that the activation epochs of the whole queue could be set here, rather than just a single epoch's worth. But at some point, `get_validator_churn_limit()` will change unpredictably (we don't know when validators will exit), which makes that infeasible. Though, curiously, that is exactly what `initiate_validator_exit()` does. Anyway, clients could optimise this by persisting the sorted activation queue rather than recalculating it.

| | |
|---|---|
| Used by | `process_epoch()` |
| Uses | `is_eligible_for_activation_queue()`, `is_active_validator()`, `initiate_validator_exit()`, `is_eligible_for_activation()`, `get_validator_churn_limit()`, `compute_activation_exit_epoch()` |
| See also | `Validator`, `EJECTION_BALANCE` |

### Slashings

```
def process_slashings(state: BeaconState) -> None:
    epoch = get_current_epoch(state)
    total_balance = get_total_active_balance(state)
    adjusted_total_slashing_balance = min(sum(state.slashings) * PROPORTIONAL_SLASHING_MULTIPLIER_ALTAIR,
                                          ↪ total_balance)
    for index, validator in enumerate(state.validators):
        if validator.slashed and epoch + EPOCHS_PER_SLASHINGS_VECTOR // 2 == validator.withdrawable_epoch:
            increment = EFFECTIVE_BALANCE_INCREMENT  # Factored out from penalty numerator to avoid
                                          ↪ uint64 overflow
            penalty_numerator = validator.effective_balance // increment * adjusted_total_slashing_balance
            penalty = penalty_numerator // total_balance * increment
            decrease_balance(state, ValidatorIndex(index), penalty)
```

Slashing penalties are applied in two stages: the first stage is in `slash_validator()`, immediately on detection; the second stage is here.

In `slash_validator()` the withdrawable epoch is set `EPOCHS_PER_SLASHINGS_VECTOR` in the future, so in this function we are considering all slashed validators that are halfway to being withdrawable, that is, completely exited from the protocol. Equivalently, they were slashed `EPOCHS_PER_SLASHINGS_VECTOR // 2` epochs ago (about 18 days).

To calculate the additional slashing penalty, we do the following:

1. Find the sum of the effective balances (at the time of the slashing) of all validators that were slashed in the previous `EPOCHS_PER_SLASHINGS_VECTOR` epochs (36 days). These are stored as a vector in the state.
2. Multiply this sum by `PROPORTIONAL_SLASHING_MULTIPLIER_ALTAIR`, but cap the result at `total_balance`, the total active balance of all validators.
3. For each slashed validator being considered, multiply its effective balance by the result of #2 and then divide by the `total_balance`. This results in an amount between zero and the full effective balance of the validator. That amount is subtracted from its actual balance as the penalty. Note that the effective balance could exceed the actual balance in odd corner cases, but `decrease_balance()` ensures the balance does not go negative.

If only a single validator were slashed within the 36 days, then this secondary penalty is tiny (actually zero, see below). If one-third of validators were slashed (the minimum required to finalise conflicting blocks), then, with `PROPORTIONAL_SLASHING_MULTIPLIER` set to two, each slashed validator would lose two thirds of its effective balance. When `PROPORTIONAL_SLASHING_MULTIPLIER` is eventually set to its final value of three, a successful chain attack will result in the attackers losing their entire effective balances.

Interestingly, due to the way the integer arithmetic is constructed in this routine, in particular the factoring out of `increment`, the result of this calculation will be zero if `validator.effective_balance * adjusted_total_slashing_balance` is less than `total_balance`. Effectively, the penalty is rounded down to the nearest whole amount of Ether. Issues 1322 and 2161 discuss this. In the end, the consequence is that when there are few slashings there is no extra correlated slashing penalty at all, which is probably a good thing.

| | |
|---|---|
| Used by | `process_epoch()` |
| Uses | `get_total_active_balance()`, `decrease_balance()` |
| See also | `slash_validator()`, `EPOCHS_PER_SLASHINGS_VECTOR`, `PROPORTIONAL_SLASHING_MULTIPLIER_ALTAIR` |

### Eth1 data votes updates

```
def process_eth1_data_reset(state: BeaconState) -> None:
    next_epoch = Epoch(get_current_epoch(state) + 1)
    # Reset eth1 data votes
    if next_epoch % EPOCHS_PER_ETH1_VOTING_PERIOD == 0:
        state.eth1_data_votes = []
```

There is a fixed period during which beacon block proposers vote on their view of the Eth1 deposit contract and try to come to a simple majority agreement. At the end of the period, the record of votes is cleared and voting begins again, whether or not agreement was reached during the period.

| | |
|---|---|
| Used by | `process_epoch()` |
| See also | `EPOCHS_PER_ETH1_VOTING_PERIOD`, `Eth1Data` |

### Effective balances updates

```
def process_effective_balance_updates(state: BeaconState) -> None:
    # Update effective balances with hysteresis
    for index, validator in enumerate(state.validators):
        balance = state.balances[index]
        HYSTERESIS_INCREMENT = uint64(EFFECTIVE_BALANCE_INCREMENT // HYSTERESIS_QUOTIENT)
        DOWNWARD_THRESHOLD = HYSTERESIS_INCREMENT * HYSTERESIS_DOWNWARD_MULTIPLIER
        UPWARD_THRESHOLD = HYSTERESIS_INCREMENT * HYSTERESIS_UPWARD_MULTIPLIER
        if (
            balance + DOWNWARD_THRESHOLD < validator.effective_balance
            or validator.effective_balance + UPWARD_THRESHOLD < balance
        ):
            validator.effective_balance = min(balance - balance % EFFECTIVE_BALANCE_INCREMENT,
                                              ↪ MAX_EFFECTIVE_BALANCE)
```

Each validator's balance is represented twice in the state: once accurately in a list separate from validator records, and once in a coarse-grained format within the validator's record. Only effective balances are used in calculations within the spec, but rewards and penalties are applied to actual balances. This routine is where effective balances are updated once per epoch to follow the actual balances.

A hysteresis mechanism is used when calculating the effective balance of a validator when its actual balance changes. See Hysteresis Parameters for more discussion of this, and the values of the related constants. With the current values, a validator's effective balance drops to `X` ETH when its actual balance drops below `X.75` ETH, and increases to `Y` ETH when its actual balance rises above `Y.25` ETH. The hysteresis mechanism ensures that effective balances change infrequently, which means that the list of validator records needs to be re-hashed only infrequently when calculating the state root, saving considerably on work.

| | |
|---|---|
| Used by | `process_epoch()` |
| See also | Hysteresis Parameters |

### Slashings balances updates

```
def process_slashings_reset(state: BeaconState) -> None:
    next_epoch = Epoch(get_current_epoch(state) + 1)
    # Reset slashings
    state.slashings[next_epoch % EPOCHS_PER_SLASHINGS_VECTOR] = Gwei(0)
```

`state.slashings` is a circular list of length `EPOCHS_PER_SLASHINGS_VECTOR` that contains the total of the effective balances of all validators that have been slashed at each epoch. These are used to apply a correlated slashing penalty to slashed validators before they are exited. Each epoch we overwrite the oldest entry with zero, and it becomes the current entry.

| | |
|---|---|
| Used by | `process_epoch()` |
| See also | `process_slashings()`, `EPOCHS_PER_SLASHINGS_VECTOR` |

### Randao mixes updates

```
def process_randao_mixes_reset(state: BeaconState) -> None:
    current_epoch = get_current_epoch(state)
    next_epoch = Epoch(current_epoch + 1)
    # Set randao mix
    state.randao_mixes[next_epoch % EPOCHS_PER_HISTORICAL_VECTOR] = get_randao_mix(state, current_epoch)
```

`state.randao_mixes` is a circular list of length `EPOCHS_PER_HISTORICAL_VECTOR`. The current value of the RANDAO, which is updated with every block that arrives, is stored at position `state.randao_mixes[current_epoch % EPOCHS_PER_HISTORICAL_VECTOR]`, as per `get_randao_mix()`.

At the end of every epoch, final value of the RANDAO for this epoch is copied over to become the starting value of the randao for the next, preserving the remaining entries as historical values.

| | |
|---|---|
| Used by | `process_epoch()` |
| Uses | `get_randao_mix()` |
| See also | `process_randao()`, `EPOCHS_PER_HISTORICAL_VECTOR` |

### Historical roots updates

```
def process_historical_roots_update(state: BeaconState) -> None:
    # Set historical root accumulator
    next_epoch = Epoch(get_current_epoch(state) + 1)
    if next_epoch % (SLOTS_PER_HISTORICAL_ROOT // SLOTS_PER_EPOCH) == 0:
        historical_batch = HistoricalBatch(block_roots=state.block_roots, state_roots=state.state_roots)
        state.historical_roots.append(hash_tree_root(historical_batch))
```

Every `SLOTS_PER_HISTORICAL_ROOT` slots, the historical roots accumulator is updated. This is implements part of the double batched accumulator for the past history of the chain. Once `SLOTS_PER_HISTORICAL_ROOT` block roots and the same number of state roots have been accumulated in the beacon state, they are put in a `HistoricalBatch` object and the hash tree root of that is appended to the `historical_roots` list in the beacon state. The corresponding block and state root lists in the beacon state are circular and just get overwritten in the next period.

Storing past roots like this allows historical Merkle proofs to be constructed if required.

| | |
|---|---|
| Used by | `process_epoch()` |
| See also | `HistoricalBatch`, `SLOTS_PER_HISTORICAL_ROOT` |

### Participation flags updates

```
def process_participation_flag_updates(state: BeaconState) -> None:
    state.previous_epoch_participation = state.current_epoch_participation
    state.current_epoch_participation = [ParticipationFlags(0b0000_0000) for _ in
                                         ↪ range(len(state.validators))]
```

Two epochs' worth of validator participation flags (that record validators' attestation activity) are stored. At the end of every epoch the current becomes the previous, and a new empty list becomes current.

| | |
|---|---|
| Used by | `process_epoch()` |
| See also | `ParticipationFlags` |

### Sync committee updates

```
def process_sync_committee_updates(state: BeaconState) -> None:
    next_epoch = get_current_epoch(state) + Epoch(1)
    if next_epoch % EPOCHS_PER_SYNC_COMMITTEE_PERIOD == 0:
        state.current_sync_committee = state.next_sync_committee
        state.next_sync_committee = get_next_sync_committee(state)
```

Sync committees are rotated every `EPOCHS_PER_SYNC_COMMITTEE_PERIOD`. The next sync committee is ready and waiting so that validators can prepare in advance by subscribing to the necessary subnets. That becomes the current sync committee, and the next is calculated.

| | |
|---|---|
| Used by | `process_epoch()` |
| Uses | `get_next_sync_committee()` |
| See also | `EPOCHS_PER_SYNC_COMMITTEE_PERIOD` |

## Block processing

```
def process_block(state: BeaconState, block: BeaconBlock) -> None:
    process_block_header(state, block)
    process_randao(state, block.body)
    process_eth1_data(state, block.body)
    process_operations(state, block.body)  # [Modified in Altair]
    process_sync_aggregate(state, block.body.sync_aggregate)  # [New in Altair]
```

These are the tasks that the beacon node performs in order to process a block and update the state. If any of the called functions triggers an `assert` failure or any other kind of exception, then the entire block is invalid, and any state changes must be rolled back.

`process_operations()` covers the processing of any slashing reports (proposer and attester) in the block, any attestations, any deposits, and any voluntary exits.

| | |
|---|---|
| Used by | `state_transition()` |
| Uses | `process_block_header()`, `process_randao()`, `process_eth1_data()`, `process_operations()`, `process_sync_aggregate()` |

### Block header

```
def process_block_header(state: BeaconState, block: BeaconBlock) -> None:
    # Verify that the slots match
    assert block.slot == state.slot
    # Verify that the block is newer than latest block header
    assert block.slot > state.latest_block_header.slot
    # Verify that proposer index is the correct index
    assert block.proposer_index == get_beacon_proposer_index(state)
    # Verify that the parent matches
    assert block.parent_root == hash_tree_root(state.latest_block_header)
    # Cache current block as the new latest block
    state.latest_block_header = BeaconBlockHeader(
        slot=block.slot,
        proposer_index=block.proposer_index,
        parent_root=block.parent_root,
        state_root=Bytes32(),  # Overwritten in the next process_slot call
        body_root=hash_tree_root(block.body),
    )

    # Verify proposer is not slashed
    proposer = state.validators[block.proposer_index]
    assert not proposer.slashed
```

A straightforward set of validity conditions for the block header data.

The version of the block header object that this routine stores in the state is a duplicate of the incoming block's header, but with its `state_root` set to its default empty `Bytes32()` value. See `process_slot()` for the explanation of this.

| | |
|---|---|
| Used by | `process_block()` |
| Uses | `get_beacon_proposer_index()`, `hash_tree_root()` |
| See also | BeaconBlockHeader, `process_slot()` |

### RANDAO

```
def process_randao(state: BeaconState, body: BeaconBlockBody) -> None:
    epoch = get_current_epoch(state)
    # Verify RANDAO reveal
    proposer = state.validators[get_beacon_proposer_index(state)]
    signing_root = compute_signing_root(epoch, get_domain(state, DOMAIN_RANDAO))
    assert bls.Verify(proposer.pubkey, signing_root, body.randao_reveal)
    # Mix in RANDAO reveal
    mix = xor(get_randao_mix(state, epoch), hash(body.randao_reveal))
    state.randao_mixes[epoch % EPOCHS_PER_HISTORICAL_VECTOR] = mix
```

A good source of randomness is foundational to the operation of the beacon chain. Security of the protocol depends significantly on being able to unpredictably and uniformly select block proposers and committee members. In fact, the very name "beacon chain" was inspired by Dfinity's concept of a randomness beacon.

The current mechanism for providing randomness is a RANDAO, in which each block proposer provides some randomness and all the contributions are mixed together over the course of an epoch. This is not unbiasable (a malicious proposer may choose to skip a block if it is to its advantage to do so), but is good enough. In future, Ethereum might use a verifiable delay function (VDF) to provide unbiasable randomness.

Early designs had the validators pre-committing to "hash onions", peeling off one layer of hashing at each block proposal. This was changed to using a BLS signature over the epoch number as the entropy source. Using signatures is both a simplification, and an enabler for multi-party (distributed) validators. The (reasonable) assumption is that sufficient numbers of validators generated their secret keys with good entropy to ensure that the RANDAO's entropy is adequate.

The `process_randao()` function simply uses the proposer's public key to verify that the RANDAO reveal in the block is indeed the epoch number signed with the proposer's private key. It then mixes the hash of the reveal into the current epoch's RANDAO accumulator. The hash is used in order to reduce the signature down from 96 to 32 bytes, and to make it uniform. `EPOCHS_PER_HISTORICAL_VECTOR` past values of the RANDAO accumulator at the ends of epochs are stored in the state.

From Justin Drake's notes: > Using `xor` in `process_randao` is (slightly) more secure than using `hash`. To illustrate why, imagine an attacker can grind randomness in the current epoch such that two of his validators are the last proposers, in a different order, in two resulting samplings of the next epochs. The commutativity of `xor` makes those two samplings equivalent, hence reducing the attacker's grinding opportunity for the next epoch versus `hash` (which is not commutative). The strict security improvement may simplify the derivation of RANDAO security formal lower bounds.

Note that the `assert` statement means that the whole block is invalid if the RANDAO reveal is incorrectly formed.

| | |
|---|---|
| Used by | `process_block()` |
| Uses | `get_beacon_proposer_index()`, `compute_signing_root()`, `get_domain()`, `bls.Verify()`, `hash()`, `xor()`, `get_randao_mix()` |
| See also | `EPOCHS_PER_HISTORICAL_VECTOR` |

### Eth1 data

```
def process_eth1_data(state: BeaconState, body: BeaconBlockBody) -> None:
    state.eth1_data_votes.append(body.eth1_data)
```

```
    if state.eth1_data_votes.count(body.eth1_data) * 2 > EPOCHS_PER_ETH1_VOTING_PERIOD * SLOTS_PER_EPOCH:
        state.eth1_data = body.eth1_data
```

Blocks may contain `Eth1Data` which is supposed to be the proposer's best view of the Eth1 chain and the deposit contract at the time. There is no incentive to get this data correct, or penalty for it being incorrect.

If there is a simple majority of the same vote being cast by proposers during each voting period of `EPOCHS_PER_ETH1_VOTING_PERIOD` epochs (6.8 hours) then the Eth1 data is committed to the beacon state. This updates the chain's view of the deposit contract, and new deposits since the last update will start being processed.

This mechanism has proved to be fragile in the past, but appears to be workable if not perfect.

| | |
|---|---|
| Used by | `process_block()` |
| See also | `Eth1Data`, `EPOCHS_PER_ETH1_VOTING_PERIOD` |

### Operations

```
def process_operations(state: BeaconState, body: BeaconBlockBody) -> None:
    # Verify that outstanding deposits are processed up to the maximum number of deposits
    assert len(body.deposits) == min(MAX_DEPOSITS, state.eth1_data.deposit_count -
                                     ↪ state.eth1_deposit_index)

    def for_ops(operations: Sequence[Any], fn: Callable[[BeaconState, Any], None]) -> None:
        for operation in operations:
            fn(state, operation)

    for_ops(body.proposer_slashings, process_proposer_slashing)
    for_ops(body.attester_slashings, process_attester_slashing)
    for_ops(body.attestations, process_attestation)
    for_ops(body.deposits, process_deposit)
    for_ops(body.voluntary_exits, process_voluntary_exit)
```

Just a dispatcher for handling the various optional contents in a block.

Deposits are optional only in the sense that some blocks have them and some don't. However, as per the `assert` statement, if, according to the beacon chain's view of the Eth1 chain, there are deposits pending, then the block *must* include them, otherwise the block is invalid. On the face of it, this suggests that it is important for a block proposer to have access to an Eth1 node, so as to be able to obtain the deposit data. In practice, this turns out to be not so important, although. with Altair, the proposer reward was increased by a factor of four, which increases the importance of the Eth1 node.

Block proposers are explicitly rewarded for including any available attestations and slashing reports. There is a validity condition, and thus an implicit reward, related to including deposit messages. The incentive for including voluntary exits is that a smaller validator set means higher rewards for the remaining validators.

| | |
|---|---|
| Used by | `process_block()` |
| Uses | `process_proposer_slashing()`, `process_attester_slashing()`, `process_attestation()`, `process_deposit()`, `process_voluntary_exit()` |
| See also | `BeaconBlockBody` |

### Proposer slashings

```
def process_proposer_slashing(state: BeaconState, proposer_slashing: ProposerSlashing) -> None:
    header_1 = proposer_slashing.signed_header_1.message
    header_2 = proposer_slashing.signed_header_2.message
```

```
    # Verify header slots match
    assert header_1.slot == header_2.slot
    # Verify header proposer indices match
    assert header_1.proposer_index == header_2.proposer_index
    # Verify the headers are different
    assert header_1 != header_2
    # Verify the proposer is slashable
    proposer = state.validators[header_1.proposer_index]
    assert is_slashable_validator(proposer, get_current_epoch(state))
    # Verify signatures
    for signed_header in (proposer_slashing.signed_header_1, proposer_slashing.signed_header_2):
        domain = get_domain(state, DOMAIN_BEACON_PROPOSER,
                                    ↪ compute_epoch_at_slot(signed_header.message.slot))
        signing_root = compute_signing_root(signed_header.message, domain)
        assert bls.Verify(proposer.pubkey, signing_root, signed_header.signature)

    slash_validator(state, header_1.proposer_index)
```

A `ProposerSlashing` is a proof that a proposer has signed two blocks at the same height. Up to `MAX_PROPOSER_SLASHINGS` of them may be included in a block. It contains the evidence in the form of a pair of `SignedBeaconBlockHeader`s.

The proof is simple: the two proposals come from the same slot, have the same proposer, but differ in one or more of `parent_root`, `state_root`, or `body_root`. In addition, they were both signed by the proposer. The conflicting blocks do not need to be valid: any pair of headers that meet the criteria, irrespective of the blocks' contents, are liable to be slashed.

As ever, the `assert` statements ensure that the containing block is invalid if it contains any invalid slashing claims.

Fun fact: the first slashing to occur on the beacon chain was a proposer slashing. Two clients running side-by-side with the same keys will often produce the same attestations since the protocol is designed to encourage that. Independently producing the same block is very unlikely as blocks contain much more data.

| | |
|---|---|
| Used by | `process_block()` |
| Uses | `is_slashable_validator()`, `get_domain()`, `compute_signing_root()`, `bls.Verify()`, `slash_validator()` |
| See also | `ProposerSlashing` |

### Attester slashings

```
def process_attester_slashing(state: BeaconState, attester_slashing: AttesterSlashing) -> None:
    attestation_1 = attester_slashing.attestation_1
    attestation_2 = attester_slashing.attestation_2
    assert is_slashable_attestation_data(attestation_1.data, attestation_2.data)
    assert is_valid_indexed_attestation(state, attestation_1)
    assert is_valid_indexed_attestation(state, attestation_2)

    slashed_any = False
    indices = set(attestation_1.attesting_indices).intersection(attestation_2.attesting_indices)
    for index in sorted(indices):
        if is_slashable_validator(state.validators[index], get_current_epoch(state)):
            slash_validator(state, index)
            slashed_any = True
    assert slashed_any
```

`AttesterSlashing`s are similar to proposer slashings in that they just provide the evidence of the two aggregate `IndexedAttestation`s that conflict with each other. Up to `MAX_ATTESTER_SLASHINGS` of them may be included in a block.

The validity checking is done by `is_slashable_attestation_data()`, which checks the double vote and surround vote conditions, and by `is_valid_indexed_attestation()` which verifies the signatures on the attestations.

Any validators that appear in both attestations are slashed. If no validator is slashed, then the attester slashing claim was not valid after all, and therefore its containing block is invalid.

Examples: a double vote attester slashing; surround vote attester slashings.

| | |
|---|---|
| Used by | `process_block()` |
| Uses | `is_slashable_attestation_data()`, `is_valid_indexed_attestation()`, `is_slashable_validator()`, `slash_validator()` |
| See also | `AttesterSlashing` |

### Attestations

```
def process_attestation(state: BeaconState, attestation: Attestation) -> None:
    data = attestation.data
    assert data.target.epoch in (get_previous_epoch(state), get_current_epoch(state))
    assert data.target.epoch == compute_epoch_at_slot(data.slot)
    assert data.slot + MIN_ATTESTATION_INCLUSION_DELAY <= state.slot <= data.slot + SLOTS_PER_EPOCH
    assert data.index < get_committee_count_per_slot(state, data.target.epoch)

    committee = get_beacon_committee(state, data.slot, data.index)
    assert len(attestation.aggregation_bits) == len(committee)

    # Participation flag indices
    participation_flag_indices = get_attestation_participation_flag_indices(state, data, state.slot -
                                    ↪ data.slot)

    # Verify signature
    assert is_valid_indexed_attestation(state, get_indexed_attestation(state, attestation))

    # Update epoch participation flags
    if data.target.epoch == get_current_epoch(state):
        epoch_participation = state.current_epoch_participation
    else:
        epoch_participation = state.previous_epoch_participation

    proposer_reward_numerator = 0
    for index in get_attesting_indices(state, data, attestation.aggregation_bits):
        for flag_index, weight in enumerate(PARTICIPATION_FLAG_WEIGHTS):
            if flag_index in participation_flag_indices and not has_flag(epoch_participation[index],
                                    ↪ flag_index):
                epoch_participation[index] = add_flag(epoch_participation[index], flag_index)
                proposer_reward_numerator += get_base_reward(state, index) * weight

    # Reward proposer
    proposer_reward_denominator = (WEIGHT_DENOMINATOR - PROPOSER_WEIGHT) * WEIGHT_DENOMINATOR //
                                    ↪ PROPOSER_WEIGHT
    proposer_reward = Gwei(proposer_reward_numerator // proposer_reward_denominator)
    increase_balance(state, get_beacon_proposer_index(state), proposer_reward)
```

Block proposers are rewarded here for including attestations during block processing, while attesting validators receive their rewards and penalties during epoch processing.

This routine processes each attestation included in the block. First a bunch of validity checks are performed. If any of these fails, then the whole block is invalid (it is most likely from a proposer on a different fork, and so useless to us):

- The target vote of the attestation must be either the previous epoch's checkpoint or the current epoch's checkpoint.
- The target checkpoint and the attestation's slot must belong to the same epoch.
- The attestation must be no newer than `MIN_ATTESTATION_INCLUSION_DELAY` slots, which is one. So this condition rules out attestations from the current or future slots.
- The attestation must be no older than `SLOTS_PER_EPOCH` slots, which is 32.
- The attestation must come from a committee that existed when the attestation was created.
- The size of the committee and the size of the aggregate must match (`aggregation_bits`).
- The (aggregate) signature on the attestation must be valid and must correspond to the aggregated public keys of the validators that it claims to be signed by. This (and other criteria) is checked by `is_valid_indexed_attestation()`.

Once the attestation has passed the checks it is processed by converting the votes from validators that it contains into flags in the state.

It's easy to skip over amidst all the checking, but the actual attestation processing is done by `get_attestation_participation_flag_indices()`. This takes the source, target, and head votes of the attestation, along with its inclusion delay (how many slots late was it included in a block) and returns a list of up to three flags corresponding to the votes that were both correct and timely, in `participation_flag_indices`.

For each validator that signed the attestation, if each flag in `participation_flag_indices` is not already set for it in its `epoch_participation` record, then the flag is set, and the proposer is rewarded. Recall that the validator making the attestation is not rewarded until the end of the epoch. If the flag is already set in the corresponding epoch for a validator, no proposer reward is accumulated: the attestation for this validator was included in an earlier block.

The proposer reward is accumulated, and weighted according to the weight assigned to each of the flags (timely source, timely target, timely head).

If a proposer includes all the attestations only for one slot, and all the relevant validators vote, then its reward will be, in the notation established earlier,

$$I_{A_P} = \frac{W_p}{32(W_\Sigma - W_p)} I_A$$

Where $I_A$ is the total maximum reward per epoch for attesters, calculated in `get_flag_index_deltas()`. The total available reward in an epoch for proposers including attestations is 32 times this.

| | |
|---|---|
| Used by | `process_operations()` |
| Uses | `get_committee_count_per_slot()`, `get_beacon_committee()`, `get_attestation_participation_flag_indices()`, `is_valid_indexed_attestation()`, `get_indexed_attestation()`, `get_attesting_indices()`, `has_flag()`, `add_flag()`, `get_base_reward()`, `increase_balance()` |
| See also | Participation flag indices, `PARTICIPATION_FLAG_WEIGHTS`, `get_flag_index_deltas()` |

### Deposits

```
def get_validator_from_deposit(state: BeaconState, deposit: Deposit) -> Validator:
    amount = deposit.data.amount
    effective_balance = min(amount - amount % EFFECTIVE_BALANCE_INCREMENT, MAX_EFFECTIVE_BALANCE)
```

```
    return Validator(
        pubkey=deposit.data.pubkey,
        withdrawal_credentials=deposit.data.withdrawal_credentials,
        activation_eligibility_epoch=FAR_FUTURE_EPOCH,
        activation_epoch=FAR_FUTURE_EPOCH,
        exit_epoch=FAR_FUTURE_EPOCH,
        withdrawable_epoch=FAR_FUTURE_EPOCH,
        effective_balance=effective_balance,
    )
```

Create a newly initialised validator object from a deposit. This was factored out of `process_deposit()` for better code reuse between the Phase 0 spec and the sharding spec.

The `state` parameter in the input argument list is an oversight: it is not used or required.

| | |
|---|---|
| Used by | `process_deposit()` |
| See also | `Validator`, `Deposit`, `FAR_FUTURE_EPOCH`, `MAX_EFFECTIVE_BALANCE` |

```
def process_deposit(state: BeaconState, deposit: Deposit) -> None:
    # Verify the Merkle branch
    assert is_valid_merkle_branch(
        leaf=hash_tree_root(deposit.data),
        branch=deposit.proof,
        depth=DEPOSIT_CONTRACT_TREE_DEPTH + 1,  # Add 1 for the List length mix-in
        index=state.eth1_deposit_index,
        root=state.eth1_data.deposit_root,
    )

    # Deposits must be processed in order
    state.eth1_deposit_index += 1

    pubkey = deposit.data.pubkey
    amount = deposit.data.amount
    validator_pubkeys = [validator.pubkey for validator in state.validators]
    if pubkey not in validator_pubkeys:
        # Verify the deposit signature (proof of possession) which is not checked by the deposit contract
        deposit_message = DepositMessage(
            pubkey=deposit.data.pubkey,
            withdrawal_credentials=deposit.data.withdrawal_credentials,
            amount=deposit.data.amount,
        )
        domain = compute_domain(DOMAIN_DEPOSIT)  # Fork-agnostic domain since deposits are valid across
                                                 ↪ forks
        signing_root = compute_signing_root(deposit_message, domain)
        # Initialize validator if the deposit signature is valid
        if bls.Verify(pubkey, signing_root, deposit.data.signature):
            state.validators.append(get_validator_from_deposit(state, deposit))
            state.balances.append(amount)
            state.previous_epoch_participation.append(ParticipationFlags(0b0000_0000))
            state.current_epoch_participation.append(ParticipationFlags(0b0000_0000))
            state.inactivity_scores.append(uint64(0))
    else:
        # Increase balance by deposit amount
        index = ValidatorIndex(validator_pubkeys.index(pubkey))
        increase_balance(state, index, amount)
```

Here, we process a deposit from a block. If the deposit is valid, either a new validator is created or the deposit amount is added to an existing validator.

The call to `is_valid_merkle_branch()` ensures that it is not possible to fake a deposit. The `eth1data.deposit_root` from the deposit contract has been agreed by the beacon chain and includes

all pending deposits visible to the beacon chain. The deposit itself contains a Merkle proof that it is included in that root. The `state.eth1_deposit_index` counter ensures that deposits are processed in order. In short, the proposer provides `leaf` and `branch`, but neither `index` nor `root`.

Deposits are signed with the private key of the depositing validator, and the corresponding public key is included in the deposit data. This constitutes a "proof of possession" of the private key, and prevents nastiness like the rogue key attack. Note that `compute_domain()` is used directly here when validating the deposit's signature, rather than the more usual `get_domain()` wrapper. This is because deposit messages are valid across beacon chain forks (such as Phase 0 and Altair), so we don't want to mix the fork version into the domain. In addition, deposits can be made before `genesis_validators_root` is known.

If the Merkle branch check fails, then the whole block is invalid. However, individual deposits can fail the signature check without invalidating the block. This allows incorrectly signed deposits to be de-queued (via `state.eth1_deposit_index += 1`) without blocking further progress (that increment would have to be reverted if the block were invalid).

Note that it is not possible to change a validator's withdrawal credentials after the initial deposit: the withdrawal credentials of subsequent deposits for the same validator are ignored; only the credentials appearing on the initial deposit are stored on the beacon chain. This is an important security measure. If an attacker steals a validator's signing key, we don't want them to be able to change the withdrawal credentials in order to steal the stake for themselves. However, it works both ways, and a vulnerability was identified for staking pools in which a malicious operator could potentially front-run a deposit transaction with a 1 ETH deposit to set the withdrawal credentials to their own.

| | |
|---|---|
| Used by | `process_operations()` |
| Uses | `is_valid_merkle_branch()`, `hash_tree_root()`, `compute_domain()`, `compute_signing_root()`, `bls.Verify()`, `get_validator_from_deposit()` |
| See also | `Deposit` |

### Voluntary exits

```
def process_voluntary_exit(state: BeaconState, signed_voluntary_exit: SignedVoluntaryExit) -> None:
    voluntary_exit = signed_voluntary_exit.message
    validator = state.validators[voluntary_exit.validator_index]
    # Verify the validator is active
    assert is_active_validator(validator, get_current_epoch(state))
    # Verify exit has not been initiated
    assert validator.exit_epoch == FAR_FUTURE_EPOCH
    # Exits must specify an epoch when they become valid; they are not valid before then
    assert get_current_epoch(state) >= voluntary_exit.epoch
    # Verify the validator has been active long enough
    assert get_current_epoch(state) >= validator.activation_epoch + SHARD_COMMITTEE_PERIOD
    # Verify signature
    domain = get_domain(state, DOMAIN_VOLUNTARY_EXIT, voluntary_exit.epoch)
    signing_root = compute_signing_root(voluntary_exit, domain)
    assert bls.Verify(validator.pubkey, signing_root, signed_voluntary_exit.signature)
    # Initiate exit
    initiate_validator_exit(state, voluntary_exit.validator_index)
```

A voluntary exit message is submitted by a validator to indicate that it wishes to cease being an active validator. A proposer receives voluntary exit messages via gossip or via its own API and then includes the message in a block so that it can be processed by the network.

Most of the checks are straightforward, as per the comments in the code. Note the following.

- Voluntary exits are ignored if they are included in blocks before the given `epoch`, so nodes might buffer any future-dated exits they see before putting them in a block.
- A validator must have been active for at least `SHARD_COMMITTEE_PERIOD` epochs (27 hours). See there for the rationale.

- Voluntary exits are signed with the validator's usual signing key. There is some discussion about changing this to also allow signing of a voluntary exit with the validator's withdrawal key.

If the voluntary exit message is valid then the validator is added to the exit queue by calling `initiate_validator_exit()`.

At present it is not possible for a validator to exit and re-enter, but this functionality may be introduced in future.

| | |
|---|---|
| Used by | `process_operations()` |
| Uses | `is_active_validator()`, `get_domain()`, `compute_signing_root()`, `bls.Verify()`, `initiate_validator_exit()` |
| See also | `VoluntaryExit`, `SHARD_COMMITTEE_PERIOD` |

### Sync aggregate processing

```
def process_sync_aggregate(state: BeaconState, sync_aggregate: SyncAggregate) -> None:
    # Verify sync committee aggregate signature signing over the previous slot block root
    committee_pubkeys = state.current_sync_committee.pubkeys
    participant_pubkeys = [pubkey for pubkey, bit in zip(committee_pubkeys,
                                        ↪ sync_aggregate.sync_committee_bits) if bit]
    previous_slot = max(state.slot, Slot(1)) - Slot(1)
    domain = get_domain(state, DOMAIN_SYNC_COMMITTEE, compute_epoch_at_slot(previous_slot))
    signing_root = compute_signing_root(get_block_root_at_slot(state, previous_slot), domain)
    assert eth_fast_aggregate_verify(participant_pubkeys, signing_root,
                                        ↪ sync_aggregate.sync_committee_signature)

    # Compute participant and proposer rewards
    total_active_increments = get_total_active_balance(state) // EFFECTIVE_BALANCE_INCREMENT
    total_base_rewards = Gwei(get_base_reward_per_increment(state) * total_active_increments)
    max_participant_rewards = Gwei(total_base_rewards * SYNC_REWARD_WEIGHT // WEIGHT_DENOMINATOR //
                                        ↪ SLOTS_PER_EPOCH)
    participant_reward = Gwei(max_participant_rewards // SYNC_COMMITTEE_SIZE)
    proposer_reward = Gwei(participant_reward * PROPOSER_WEIGHT // (WEIGHT_DENOMINATOR - PROPOSER_WEIGHT))

    # Apply participant and proposer rewards
    all_pubkeys = [v.pubkey for v in state.validators]
    committee_indices = [ValidatorIndex(all_pubkeys.index(pubkey)) for pubkey in
                                        ↪ state.current_sync_committee.pubkeys]
    for participant_index, participation_bit in zip(committee_indices,
                                        ↪ sync_aggregate.sync_committee_bits):
        if participation_bit:
            increase_balance(state, participant_index, participant_reward)
            increase_balance(state, get_beacon_proposer_index(state), proposer_reward)
        else:
            decrease_balance(state, participant_index, participant_reward)
```

Similarly to how attestations are handled, the beacon block proposer includes in its block an aggregation of sync committee votes that agree with its local view of the chain. Specifically, the sync committee votes are for the head block that the proposer saw in the previous slot. (If the previous slot is empty, then the head block will be from an earlier slot.)

We validate these votes against our local view of the chain, and if they agree then we reward the participants that voted. If they do not agree with our local view, then the entire block is invalid: it is on another branch.

To perform the validation, we form the signing root of the block at the previous slot, with `DOMAIN_SYNC_COMMITTEE` mixed in. Then we check if the aggregate signature received in the `SyncAggregate` verifies against it, using the aggregate public key of the validators who claimed to have signed it. If either the signing root (that is, the head block) is wrong, or the list of participants is wrong, then the verification will fail and the block is invalid.

Like proposer rewards, but unlike attestation rewards, sync committee rewards are not weighted with the participants' effective balances. This is already taken care of by the committee selection process that weights the probability of selection with the effective balance of the validator.

Running through the calculations:

- `total_active_increments`: the sum of the effective balances of the entire active validator set normalised with the `EFFECTIVE_BALANCE_INCREMENT` to give the total number of increments.
- `total_base_rewards`: the maximum rewards that will be awarded to all validators for all duties this epoch. It is at most $NB$ in the notation established earlier.
- `max_participant_rewards`: the amount of the total reward to be given to the entire sync committee in this slot.
- `participant_reward`: the reward per participating validator, and the penalty per non-participating validator.
- `proposer_reward`: one seventh of the participant reward.

Each committee member that voted receives a reward of `participant_reward`, and the proposer receives one seventh of this in addition.

Each committee member that failed to vote receives a penalty of `participant_reward`, and the proposer receives nothing.

In our notation the maximum issuance (reward) due to sync committees per slot is as follows.

$$I_S = \frac{W_y}{32 \cdot W_\Sigma} NB$$

The per-epoch reward is thirty-two times this. The maximum reward for the proposer in respect of sync aggregates:

$$I_{S_P} = \frac{W_p}{W_\Sigma - W_p} I_S$$

| | |
|---|---|
| Used by | `process_operations()` |
| Uses | `get_domain()`, `compute_signing_root()`, `eth_fast_aggregate_verify()`, `get_total_active_balance()`, `get_base_reward_per_increment()`, `increase_balance()`, `decrease_balance()` |
| See also | Incentivization weights, `SYNC_COMMITTEE_SIZE` |

# Initialise State

## Introduction

TODO: rework and synthesis - this text is from the original Genesis

Before the Ethereum beacon chain genesis has been triggered, and for every Ethereum proof-of-work block, let `candidate_state = initialize_beacon_state_from_eth1(eth1_block_hash, eth1_timestamp, deposits)` where:

- `eth1_block_hash` is the hash of the Ethereum proof-of-work block
- `eth1_timestamp` is the Unix timestamp corresponding to `eth1_block_hash`
- `deposits` is the sequence of all deposits, ordered chronologically, up to (and including) the block with hash `eth1_block_hash`

Proof-of-work blocks must only be considered once they are at least `SECONDS_PER_ETH1_BLOCK * ETH1_FOLLOW_DISTANCE` seconds old (i.e. `eth1_timestamp + SECONDS_PER_ETH1_BLOCK * ETH1_FOLLOW_DISTANCE <= current_unix_time`). Due to this constraint, if `GENESIS_DELAY < SECONDS_PER_ETH1_BLOCK * ETH1_FOLLOW_DISTANCE`, then the `genesis_time` can happen before the time/state is first known. Values should be configured to avoid this case.

## Initialisation

Aka genesis.

This helper function is only for initializing the state for pure Altair testnets and tests.

```
def initialize_beacon_state_from_eth1(eth1_block_hash: Bytes32,
                                      eth1_timestamp: uint64,
                                      deposits: Sequence[Deposit]) -> BeaconState:
    fork = Fork(
        previous_version=ALTAIR_FORK_VERSION,  # [Modified in Altair] for testing only
        current_version=ALTAIR_FORK_VERSION,  # [Modified in Altair]
        epoch=GENESIS_EPOCH,
    )
    state = BeaconState(
        genesis_time=eth1_timestamp + GENESIS_DELAY,
        fork=fork,
        eth1_data=Eth1Data(block_hash=eth1_block_hash, deposit_count=uint64(len(deposits))),
        latest_block_header=BeaconBlockHeader(body_root=hash_tree_root(BeaconBlockBody())),
        randao_mixes=[eth1_block_hash] * EPOCHS_PER_HISTORICAL_VECTOR,  # Seed RANDAO with Eth1 entropy
    )

    # Process deposits
    leaves = list(map(lambda deposit: deposit.data, deposits))
    for index, deposit in enumerate(deposits):
        deposit_data_list = List[DepositData, 2**DEPOSIT_CONTRACT_TREE_DEPTH](*leaves[:index + 1])
        state.eth1_data.deposit_root = hash_tree_root(deposit_data_list)
        process_deposit(state, deposit)

    # Process activations
    for index, validator in enumerate(state.validators):
        balance = state.balances[index]
        validator.effective_balance = min(balance - balance % EFFECTIVE_BALANCE_INCREMENT,
                                          ↪ MAX_EFFECTIVE_BALANCE)
        if validator.effective_balance == MAX_EFFECTIVE_BALANCE:
            validator.activation_eligibility_epoch = GENESIS_EPOCH
            validator.activation_epoch = GENESIS_EPOCH

    # Set genesis validators root for domain separation and chain versioning
    state.genesis_validators_root = hash_tree_root(state.validators)

    # [New in Altair] Fill in sync committees
```

```
    # Note: A duplicate committee is assigned for the current and next committee at genesis
    state.current_sync_committee = get_next_sync_committee(state)
    state.next_sync_committee = get_next_sync_committee(state)

    return state
```

TODO

## Genesis state

Let `genesis_state = candidate_state` whenever `is_valid_genesis_state(candidate_state) is True` for the first time.

```
def is_valid_genesis_state(state: BeaconState) -> bool:
    if state.genesis_time < MIN_GENESIS_TIME:
        return False
    if len(get_active_validator_indices(state, GENESIS_EPOCH)) < MIN_GENESIS_ACTIVE_VALIDATOR_COUNT:
        return False
    return True
```

TODO

## Genesis block

Let `genesis_block = BeaconBlock(state_root=hash_tree_root(genesis_state))`.

TODO

# Altair Fork Logic

## Introduction

TODO

From fork.md

## Configuration

TODO

| Name | Value |
|---|---|
| `ALTAIR_FORK_VERSION` | `Version('0x01000000')` |
| `ALTAIR_FORK_EPOCH` | `Epoch(74240)` (Oct 27, 2021, 10:56:23am UTC) |

## Fork to Altair

### Fork trigger

The fork is triggered at epoch `ALTAIR_FORK_EPOCH`.

Note that for the pure Altair networks, we don't apply `upgrade_to_altair` since it starts with Altair version logic.

### Upgrading the state

If `state.slot % SLOTS_PER_EPOCH == 0` and `compute_epoch_at_slot(state.slot) == ALTAIR_FORK_EPOCH`, an irregular state change is made to upgrade to Altair.

The upgrade occurs after the completion of the inner loop of `process_slots` that sets `state.slot` equal to `ALTAIR_FORK_EPOCH * SLOTS_PER_EPOCH`. Care must be taken when transitioning through the fork boundary as implementations will need a modified state transition function that deviates from the Phase 0 document. In particular, the outer `state_transition` function defined in the Phase 0 document will not expose the precise fork slot to execute the upgrade in the presence of skipped slots at the fork boundary. Instead the logic must be within `process_slots`.

```
def translate_participation(state: BeaconState, pending_attestations:
                                    ↪ Sequence[phase0.PendingAttestation]) -> None:
    for attestation in pending_attestations:
        data = attestation.data
        inclusion_delay = attestation.inclusion_delay
        # Translate attestation inclusion info to flag indices
        participation_flag_indices = get_attestation_participation_flag_indices(state, data,
                                    ↪ inclusion_delay)

        # Apply flags to all attesting validators
        epoch_participation = state.previous_epoch_participation
        for index in get_attesting_indices(state, data, attestation.aggregation_bits):
            for flag_index in participation_flag_indices:
                epoch_participation[index] = add_flag(epoch_participation[index], flag_index)
```

TODO

```
def upgrade_to_altair(pre: phase0.BeaconState) -> BeaconState:
    epoch = phase0.get_current_epoch(pre)
    post = BeaconState(
        # Versioning
        genesis_time=pre.genesis_time,
        genesis_validators_root=pre.genesis_validators_root,
        slot=pre.slot,
        fork=Fork(
```

```
            previous_version=pre.fork.current_version,
            current_version=ALTAIR_FORK_VERSION,
            epoch=epoch,
        ),
        # History
        latest_block_header=pre.latest_block_header,
        block_roots=pre.block_roots,
        state_roots=pre.state_roots,
        historical_roots=pre.historical_roots,
        # Eth1
        eth1_data=pre.eth1_data,
        eth1_data_votes=pre.eth1_data_votes,
        eth1_deposit_index=pre.eth1_deposit_index,
        # Registry
        validators=pre.validators,
        balances=pre.balances,
        # Randomness
        randao_mixes=pre.randao_mixes,
        # Slashings
        slashings=pre.slashings,
        # Participation
        previous_epoch_participation=[ParticipationFlags(0b0000_0000) for _ in
                                      ↪ range(len(pre.validators))],
        current_epoch_participation=[ParticipationFlags(0b0000_0000) for _ in range(len(pre.validators))],
        # Finality
        justification_bits=pre.justification_bits,
        previous_justified_checkpoint=pre.previous_justified_checkpoint,
        current_justified_checkpoint=pre.current_justified_checkpoint,
        finalized_checkpoint=pre.finalized_checkpoint,
        # Inactivity
        inactivity_scores=[uint64(0) for _ in range(len(pre.validators))],
    )
    # Fill in previous epoch participation from the pre state's pending attestations
    translate_participation(post, pre.previous_epoch_attestations)

    # Fill in sync committees
    # Note: A duplicate committee is assigned for the current and next committee at the fork boundary
    post.current_sync_committee = get_next_sync_committee(post)
    post.next_sync_committee = get_next_sync_committee(post)
    return post
```

TODO

# Part 4: Future

## Introduction

TODO

# The Merge

## Introduction

TODO

## Architecture

TODO

## Engine API

TODO

## Optimistic Sync

TODO

## The Transition

TODO

## Withdrawals

TODO

## Data Availability Sampling

TODO

### Proto-Danksharding

TODO

### Full Danksharding

TODO

# Distributed Validator Technology

## Introduction

TODO

## Multi-party Compute

TODO

## Consensus

TODO

# Light Clients

## Introduction

TODO

## Syncing

TODO

## Protocol

TODO

## Active Research Topics

### Introduction

TODO

### Proofs of Custody

TODO

### Builder / proposer split

TODO

### Consensus changes

TODO

### Single slot finality

TODO

### Verkle trees

TODO

### Statelessness

TODO

### Single Secret Leader Election

TODO

### Verifiable Delay Function

TODO

### Post-quantum crypto

TODO

### S[NT]ARK-friendly state transitions

TODO

# Appendices

# Staking

## Introduction

TODO

## Ways to Stake

TODO

## Client Diversity

TODO

## FAQ

TODO

# How to become a core dev

## So you wanna be a core dev?

TODO

## Resources

TODO

# Reference

TODO

## Running the spec

### Introduction

Being written in Python, the spec itself is executable. This is wonderful for generating test cases and there is a whole infrastructure in the specs repo for doing just that.

We can also run the spec ourselves to do interesting things. In this exercise we will calculate the minimum and maximum sizes of the various containers defined by the spec. The following code is from Protolambda, lightly modified to simplify and update it.

```
from inspect import getmembers, isclass
from eth2spec.utils.ssz.ssz_typing import Container
from eth2spec.altair import mainnet

def get_spec_ssz_types():
    return [
        value for (_, value) in getmembers(mainnet, isclass)
        if issubclass(value, Container) and value != Container  # only the subclasses, not the imported
                                        ↪ base class
    ]

type_bounds = {
    value.__name__: ({
        'size': value.type_byte_length()
    } if value.is_fixed_byte_length() else {
        'min_size': value.min_byte_length(),
        'max_size': value.max_byte_length(),
    }) for value in get_spec_ssz_types()
}

import json
print(json.dumps(type_bounds))
```

### Set up

We have a bunch of hoops to jump through to get things installed for the first time. The below works well for me on Linux, but I haven't tested extensive variations. Just use the commands prefixed with >, I've included some of the output so you can check whether things are on the right lines.

First, set up a Python virtual environment.

```
> git clone https://github.com/ethereum/consensus-specs.git
Cloning into 'consensus-specs'...
...
> cd consensus-specs/
> python3 -m venv .
> source bin/activate
(consensus-specs) > python --version
Python 3.8.10
```

Now we install and build all the dependencies required for the actual specs.

```
(consensus-specs) > python setup.py install
... tons of output ...
(consensus-specs) > make install_test
... some initial failures reported but it installs cytoolz and sorts itself out ...
(consensus-specs) > python setup.py pyspecdev
running pyspecdev
running build_py command
running pyspec
```

```
...
```

### Run

Finally we can simply run the Python script from above. Copy it into a file called `sizes.py` and run it as follows.

```
(consensus-specs) > python sizes.py | jq
{
  "AggregateAndProof": {
    "min_size": 337,
    "max_size": 593
  },
...
```

The pipe to `jq` is optional, you will just get less pretty output without it.

### Full output

Values are bytes. Don't be too alarmed that the maximum size of `BeaconState` turns out to be 139TiB!


```
{
  "AggregateAndProof": {
    "min_size": 337,
    "max_size": 593
  },
  "Attestation": {
    "min_size": 229,
    "max_size": 485
  },
  "AttestationData": {
    "size": 128
  },
  "AttesterSlashing": {
    "min_size": 464,
    "max_size": 33232
  },
  "BeaconBlock": {
    "min_size": 464,
    "max_size": 157816
  },
  "BeaconBlockBody": {
    "min_size": 380,
    "max_size": 157732
  },
  "BeaconBlockHeader": {
    "size": 112
  },
  "BeaconState": {
    "min_size": 2736629,
    "max_size": 152832656015861
  },
  "Checkpoint": {
    "size": 40
  },
  "ContributionAndProof": {
    "size": 264
  },
  "Deposit": {
    "size": 1240
  },
  "DepositData": {
    "size": 184
  },
```

```
"DepositMessage": {
  "size": 88
},
"Eth1Block": {
  "size": 48
},
"Eth1Data": {
  "size": 72
},
"Fork": {
  "size": 16
},
"ForkData": {
  "size": 36
},
"HistoricalBatch": {
  "size": 524288
},
"IndexedAttestation": {
  "min_size": 228,
  "max_size": 16612
},
"LightClientUpdate": {
  "size": 25364
},
"PendingAttestation": {
  "min_size": 149,
  "max_size": 405
},
"ProposerSlashing": {
  "size": 416
},
"SignedAggregateAndProof": {
  "min_size": 437,
  "max_size": 693
},
"SignedBeaconBlock": {
  "min_size": 564,
  "max_size": 157916
},
"SignedBeaconBlockHeader": {
  "size": 208
},
"SignedContributionAndProof": {
  "size": 360
},
"SignedVoluntaryExit": {
  "size": 112
},
"SigningData": {
  "size": 64
},
"SyncAggregate": {
  "size": 160
},
"SyncAggregatorSelectionData": {
  "size": 16
},
"SyncCommittee": {
  "size": 24624
},
"SyncCommitteeContribution": {
  "size": 160
```

```
  },
  "SyncCommitteeMessage": {
    "size": 144
  },
  "Validator": {
    "size": 121
  },
  "VoluntaryExit": {
    "size": 16
  }
}
```


### Sizes of containers

TODO

## Glossary

TODO