BPX Chain

BPX Chain documentation

- Getting Started
  - Introduction
  - Beacon and Execution Chain
  - BPX vs Ethereum
  - BPX vs Chia
- Architecture
  - Architecture Overview
  - Beacon Client
  - Execution Client
  - Farmer
  - Harvester
  - Timelord
  - Wallet
- Consensus
  - Consensus Introduction
  - Proof of Space
  - Proof of Time
  - Challenges
  - Signage and Infusion Points
  - Harvesting
  - Multiple Blocks
  - VDF chains
  - Overflow Blocks and Weight
  - Foliage
  - Epoch and Difficulty

- Block Validation
- Block Creation
- Timelord algorithm
- Analysis

Getting Started

Introduction

BPX Chain is a new generation Ethereum-compatible blockchain based upon an innovative consensus algorithm, **Proof of Space and Time (PoST)**, originally pioneered by the Chia Network. Proof of Space is a cryptographic technique where farmers prove that they allocate unused hard disk space to the network. Proof of Time increases the overall security of the blockchain.

Compared to **Proof of Work** blockchains, BPX Chain provides the same level of security and decentralization, with a fractional power consumption. Compared to **Proof of Stake** blockchains, BPX is more censorship-resistant and avoids the centralization tendency where wealthier participants gain an outsized influence.

BPX Chain generates new blocks using hard drives and is compatible with Chia K-32 plots, enabling simultaneous farming with other PoST-based cryptocurrencies.

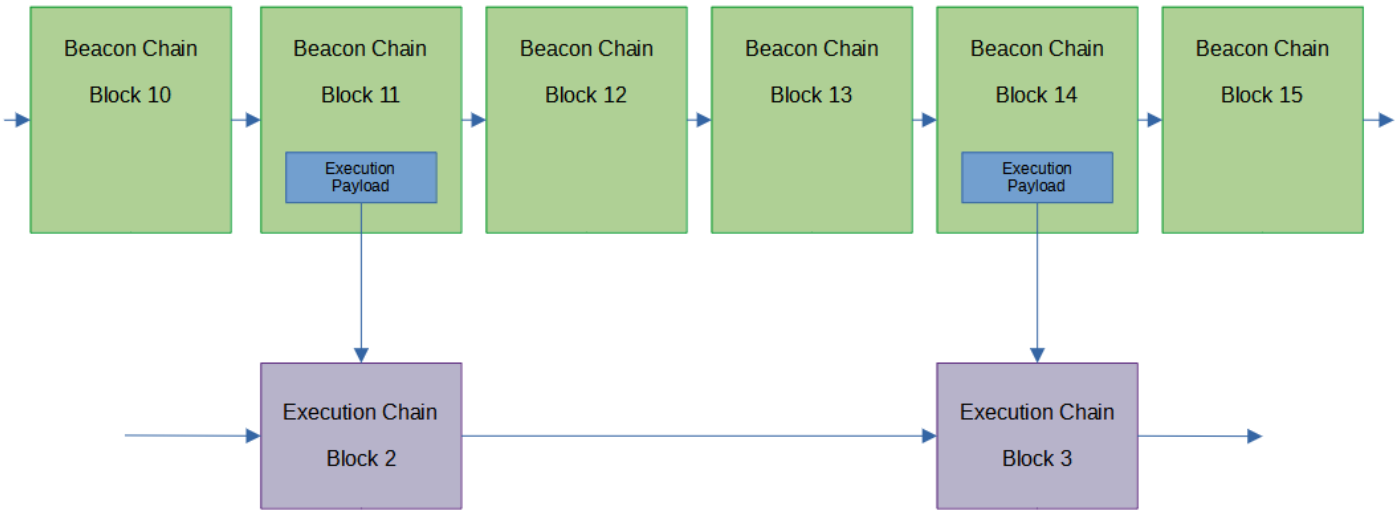
BPX Chain is fully compatible with Ethereum Virtual Machine. Existing Ethereum-based solutions can be seamlessly migrated to BPX without modifying source code. BPX also supports the standard Web3 RPC API, ensuring compatibility with popular wallets (e.g., MetaMask, Trust Wallet) and developer tools (e.g., Remix IDE, Web3.js). BPX Chain provides standard interface for fungible and non-fungible tokens, compatible with the appropriate interfaces in other blockchains (e.g. ERC-20, TRC-20, BEP-20).

The BPX cryptocurrency powers the network, facilitating transaction fees and rewarding farmers who help secure the chain.

Beacon and Execution Chain

BPX comprises two distinct blockchains: the **Beacon Chain** and the **Execution Chain**, each responsible for specific tasks, with unique block structures and differing block production intervals. As a result, the peak heights of the Beacon Chain and Execution Chain diverge.

Certain blocks on the Beacon Chain, known as **transactional blocks**, include an **execution payload** - the data structure that enables the local reconstruction of a full Execution Chain block.



Both chains require separate client applications, called the **Beacon Client** and **Execution Client**, respectively.

Synchronization with Execution Chain without synchronization with Beacon Chain is not possible because Execution Chain does not have any built-in consensus algorithm but relies on data received from Beacon Chain.

Synchronization with Beacon Chain without synchronization with Execution Chain is not possible because execution payloads in some beacon blocks must be validated by the execution client.

Such a mechanism naturally forces following both chains by each BPX full node.

Beacon Chain

Beacon Chain manages the Proof of Space and Time consensus. The beacon chain blocks contain proofs of space, VDF outputs, addresses to which the block reward should be paid, and execution

payloads that allow you to recreate the full execution block locally.

Execution Chain

Execution Chain is an EVM compatible chain. Execution blocks contains user transactions, smart contracts and logs.

BPX vs Ethereum

If you are familiar with how Ethereum works, the following tips will help you better understand the BPX Chain:

1. The BPX Execution Chain functions identically to Ethereum’s Execution Chain.
2. The BPX Execution Client is a Go-Ethereum (geth) fork, modified only for the genesis block, bootnodes, and DNS keys.
3. BPX Chain retains the Ethereum protocol, APIs, block and transaction structure unchanged.
4. All Ethereum-compatible wallets, developer tools, and applications that support a custom chainId are compatible with BPX out of the box.
5. Smart contracts written in Solidity can be deployed on the BPX network without modifying their source code.
6. The BPX Beacon Chain and Beacon Client differ significantly from Ethereum's setup but use the same Engine API.
7. BPX uses the withdrawal mechanism introduced in Ethereum’s Shanghai hard fork to distribute block rewards.

BPX vs Chia

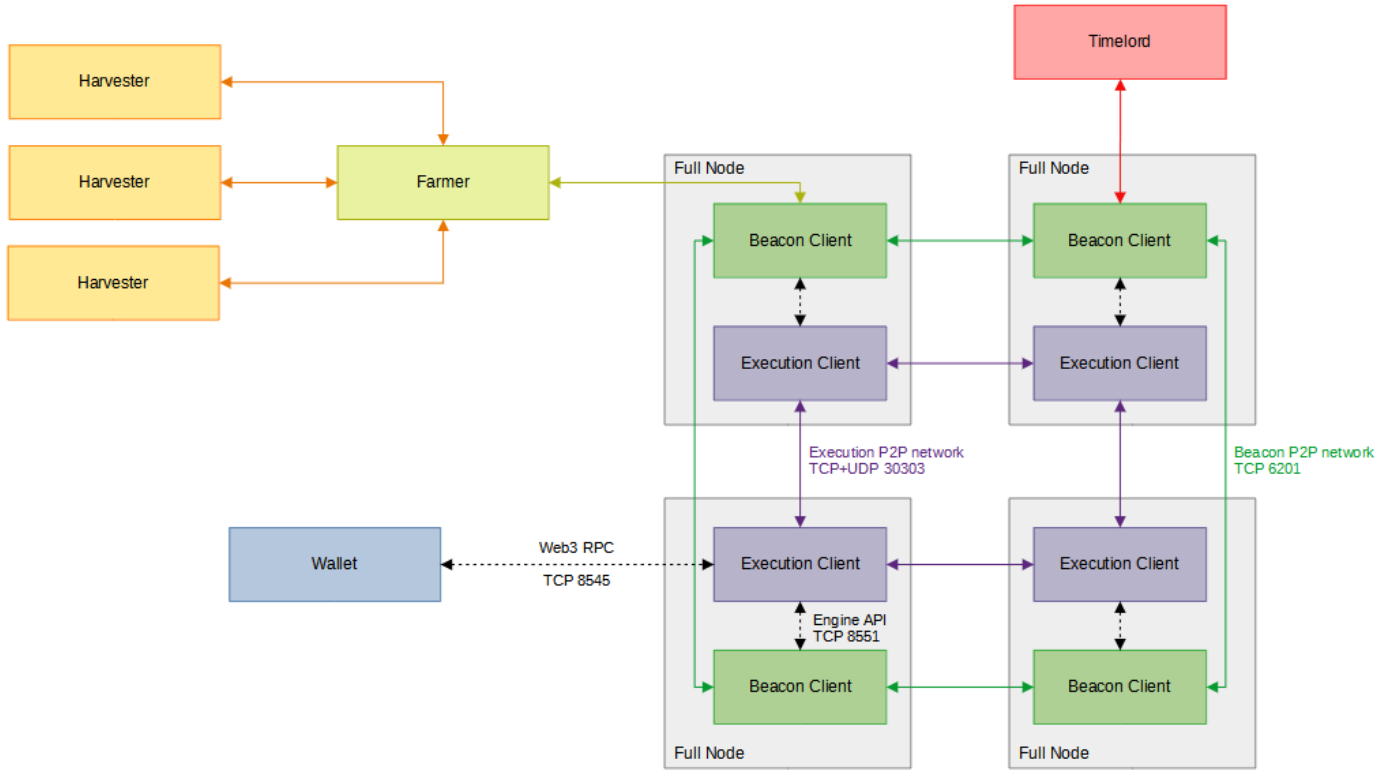
If you are familiar with how Chia works, the following tips will help you better understand the BPX Chain:

1. The BPX Beacon Chain is based on the Chia blockchain architecture, where all transactional and monetary functions have been replaced by a mechanism that controls an independent EVM-compatible execution layer.
2. The BPX Beacon Client is a heavily modified fork of the Chia reference client, though the BPX Chain itself is not a Chia fork.
3. The Beacon Client solely provides consensus updates for the EVM-based Execution Chain that runs in parallel and cannot function properly without the Execution Client.
4. BPX Beacon Chain does not handle transactions, lacks an integrated wallet, and does not display information on farmed coins; all such data is managed by the Execution Chain.
5. Many commands, APIs, and graphical interfaces of the Beacon Client are identical or similar to those in Chia.
6. BPX Chain does not use a coin set model neither CLVM.

Architecture

Architecture Overview

The above diagram shows BPX network architecture. A single machine can run more than one of these processes. Typical BPX network full node consist of two processes: Beacon Client and Execution Client. Additionally, Farmer and Harvester are activated to farm new blocks.



Beacon Client

The **Beacon Client** serves as a full node for the BPX Beacon Chain with several key responsibilities:

1. Maintaining a complete copy of the Beacon Chain.
2. Validating Beacon Chain blocks.
3. Propagating new beacon blocks and proofs across the peer-to-peer network.
4. Providing consensus updates to the Execution Client.

Beacon Clients are continually connected to a random subset of other Beacon Clients within the network. They distribute new blocks to peers, enabling all nodes to maintain a full copy of the blockchain.

Beacon Clients communicate with one another over **TCP port 6201**.

Execution Client

The **Execution Client** functions as a full node for the **BPX Execution Chain**. It listens for new transactions broadcast on the network, maintains the mempool, executes transactions within the EVM, creates and validates execution blocks, and holds the latest state and database of the Execution Chain.

The Execution Chain lacks an internal consensus algorithm. Instead, it exposes the **Engine API** on **TCP port 8551**, allowing the Beacon Client to connect and provide consensus updates, including new block candidates, chain head updates, and requests to generate new blocks.

Execution Clients communicate with each other over a listener **TCP port** and a discovery **UDP port**, both defaulting to **30303**. Each Execution Client also provides access to the **Web3 RPC API** on **TCP port 8545**, which supports interaction with wallets, developer tools, and dApps.

Farmer

BPX farmers are analogous to Bitcoin's miners. They earn block rewards and fees by finding valid proofs of space inside their stored plots. The farmer processes don't maintain a copy of the Beacon Chain, but they trust a beacon client to provide updates. The beacon client and farmer processes communicate with each other using the farmer protocol.

Farmers communicate with harvesters (individual machines, including the farmer machine, that actually store the plots) through the harvester protocol.

Farmers operate by waiting for updates from a beacon client, which gives them new signage points (equivalent to a lottery's winning numbers) approximately every nine seconds. Farmers then send the signage point to each harvester, to check whether any winning proofs of space exist. If the harvester finds any valid proofs, it sends them to the farmer. Farmer has a private key, which is used for signing blocks. After receiving signed block from the farmer, beacon client creates a new beacon chain block and propagates it accross the network.

Harvester

Harvesters are individual machines controlled by a farmer. In a large farming operation, a farmer may be connected to many harvesters.

Harvesters control the actual plot files by retrieving qualities or proofs from disk. The minimum plot size (and by far the most common) is k32, which corresponds to around 100 GiB. With each increment of a k-value, the plot size roughly doubles, so a k33 plot is around 200 GiB, k34 is around 400 GiB, etc.

The difficulty level automatically adjusts every 4608 beacon chain blocks to target one proof of space - across the entire network - for every two signage points. This is the targeted average value - there can also be zero or multiple proofs per signage point. This leads to a difficulty adjustment approximately every 12 hours.

Given a plot, the harvester must perform two tasks to find a valid proof:

1. Fetch the initial quality - this requires around seven random disk seeks, or 70 milliseconds on a slow HDD.
2. (Only performed if the initial quality is sufficiently high) Fetch the full proof - this requires around 64 disk seeks, or 640 milliseconds on a slow HDD.

For most challenges, the quality (step 1) will be very low, so fetching the entire proof (step 2) will not be necessary. A node has 28 seconds to return a proof, so disk I/O will not be a limiting factor, even when proofs are stored on slow HDDs.

> Tape drives are too slow for farming. The protocol was designed to support hard disks, but nothing slower. It is possible to use tape for long-term plot storage, only transferring the plots to disks for occasional farming, but this is likely a very rare use case.

Finally, harvesters also maintain a private key for each plot. The blocks are signed with these keys.

Timelord

Timelords support the network by creating sequential proofs of time (using a Verifiable Delay Function) and broadcasting them approximately every nine seconds. This provides "deterministic randomness", which is used to decide the winning proofs of space.

Since this computation is sequential, very little energy is consumed, as opposed to proof-of-work systems, where computation is parallelizable. For example, if 100 timelords are doing the same computation on a proof of time, they will all create the exact same output.

A timelord is required to connect to exactly one beacon client, typically on the same machine. This connection is verified with a certificate. This 1:1 architecture has a large security benefit: it keeps the timelord sandboxed in its own private network. That way, the beacon client protocol is the only protocol that requires total security. If more than one beacon client could connect to the same timelord, it would add a potential attack vector to the network.

Timelords do not directly earn rewards. Furthermore, only the fastest timelord on the network will broadcast proofs at any given time. Therefore, only one timelord is required to keep the network running, and most farmers will not feel compelled to run one. However, farmers with multi-PiB farms may want to run a timelord, both for redundancy and for protection against temporary local latency issues.

If someone controls the fastest timelord in the world, it doesn't give them much of an advantage at winning rewards. However, they could potentially orphan or censor other farmers, depending on how much faster their timelord is.

Furthermore, an attacker with a significantly faster timelord than anyone else could potentially run a long-range attack against the network with less than 42.7% of the total netspace. For security purposes, it is very important to maintain open designs of VDF hardware.

Types of Timelords

There are two primary types of Timelords: Regular and Blueboxes.

The first is the core Timelord that takes in Proofs of Space and uses a single fastest core to perform repeated squaring in a class group of unknown order as fast as possible. Beside each running VDF (referred to as a vdf_client in the application and source) is a set of proof generation threads that accumulate the proof that the time calculation's number of iterations was done correctly.

The second are Bluebox Timelords. Blueboxes are most any machine - especially things like old servers or gaming machines - that scour the historical chain looking for uncompressed proofs of time. So that the chain moves quickly, the regular Timelords use a faster method of generating proofs of time but the proofs are larger, which takes your Raspberry Pi a lot more time and effort to validate and sync the blockchain. A Bluebox picks up an uncompressed Proof of Time and recreates it, but this time with the slower and more compact proofs generated at the end. Those are then gossiped around to everyone so they can replace the large and slow to verify Proofs of Time with the compact and much quicker to validate version of exactly the same thing.

Running a Timelord

The network only requires one running Timelord to keep moving (liveness.) The way Timelords race is like they are on a series of 100 meter dashes. Each one takes off with the last good Proof of Space and tries to get to the total number of iterations required to complete a given Proof of Space. Better Proofs of Space require less iterations to prove. When the fastest Timelord announces the Proof of Time for this Proof of Space all of the other Timelords stop racing and are magically teleported to the starting line of the next 100 meter dash to start it all over again.

It's good to have a few Timelords out there. There can be things like routing flaps or the overzealous backhoe that takes large swaths of the internet offline. If the fastest Timelord was just about to win the current dash when its internet blinked off in a fury of construction misadventure, then the second fastest will win that dash and the next dashes - until the fastest returns. One of the key qualities about Proofs of Time is that given the same Proof of Space, their output and proof are always the same (though the proofs can be larger or smaller and harder or easier to validate - they all end up with the same outcome.)

BPX developers plans to run a few Timelords around the world - and some backups too - to ensure that all Farmers and nodes can hear the beat that the Timelords are calling.

Installing a Timelord

If you want to run a Timelord on Linux/MacOS, first follow the Install from Source instructions here. Then run:

```
. ./activate
sh install-timelord.sh
bpx start timelord
```

Timelords execute sequential verifiable delay functions (proofs of time or VDFs), that get added to blocks to make them valid. This requires fast CPUs and a few cores per VDF.

> Due to restrictions on how MSVC handles 128 bit numbers and how Python relies upon MSVC, it is not possible to build and run Timelords of all types on Windows.

Regular Timelords

On MacOS x86_64 and all Linux distributions, building a Timelord is as easy as running `bpx start timelord` in the virtual environment. You can also run `./vdf_bench square_asm 400000` once you've built Timelord to give you a sense of your optimal and unloaded ips. Each run of `vdf_bench` can be surprisingly variable and, in production, the actual ips you will obtain will usually be about 20% lower due to load of creating proofs. The default configuration for Timelords is good enough to just let you start it up. Set your log level to INFO and then grep for "Estimated IPS:" to get a sense of what actual ips your Timelord is achieving.

Bluebox Timelords

Once you build the Timelord with `sh install-timelord.sh` in the virtual environment, you will need to make two changes to `~/.bpxchain/beacon/config/config.yaml`. In the `timelord`: section, set `bluebox_mode` to `True`. Then you need to proceed to the `beacon` section and set `send_uncompact_interval` to something greater than 0. We recommend 300 seconds there so that your Bluebox has some time to prove through a lot of the un-compacted Proofs of Time before the node drops more into its lap. The default settings may otherwise work but if the total effort is a little too much for whatever machine you are on you can also lower the process_count: from 3 to 2, or even 1, in the `timelord_launcher` section. You know it is working if you see `VDF Client: Sent proof` in your logs at INFO level.

Timelords and Attacks

One of the things that is great about BPX consensus is that it makes it almost impossible for a Farmer with a maliciously faster Timelord to selfishly Farm. Due to the way the consensus works, a Farmer with a faster Timelord is basically compelled to prove time for all the farmers winning blocks around him also. Maliciously running a faster Timelord can give a benefit when attempting to 51% attack the network, so it is still important that over time we push the Timelord speeds as close to the maximum speeds of the silicon processes available. We expect to have the time and the resources to do that right and make open-source hardware versions widely available.

Wallet

The wallet is responsible for managing private keys, as well as generating, storing and sending transactions. Wallets communicate with the execution clients through the Web3 RPC API. Each execution client exposes the API on TCP port 8445.

You can also connect your wallet to a public RPC node provided by us, using the configuration parameters below:

Network name: BPX Chain
New RPC URL: https://rpc.bpxchain.cc
Chain ID: 279
Currency symbol: BPX
Block explorer URL: https://explorer.bpxchain.cc

BPX Chain is compatible with all popular wallets dedicated for EVM chains, e.g. **Metamask** or **Trust Wallet**.

Consensus

Consensus Introduction

Decentralized consensus algorithms require Sybil resistance, using a resource that is both cryptographically verifiable and scarce (not infinite). In previous blockchain systems, two different scarce resources have been used: computing power (Proof of Work) and staked money (Proof of Stake).

Proof of Space and Time consensus uses storage capacity as the scarce resource. This comes much closer than previous systems to Satoshi's original ideal of "one CPU, one vote," where a *vote* refers to a chance to win and validate a block, not an actual vote on-chain. For example, someone storing 500 GiB has 5 "votes," and someone storing 100 GiB has 1 "vote."

One other cryptographic puzzle piece is used to secure BPX: a verifiable delay function (VDF), which is a cryptographic proof that real time has passed.

A fair system can be created by combining proofs of space and time. In such a system, users store random-looking data on their hard drives. Their chance to win BPX is proportional to their allocated space. Furthermore, such a system scales to billions of participants in a similar way to the Proof of Work lottery. No funds, special hardware, registration, or permission is required to join, only a hard drive and an internet connection. The system is completely transparent and deterministic - anyone can efficiently and objectively verify which chain is the canonical one, without relying on any trusted parties.

Some notes to keep in mind as you continue reading:

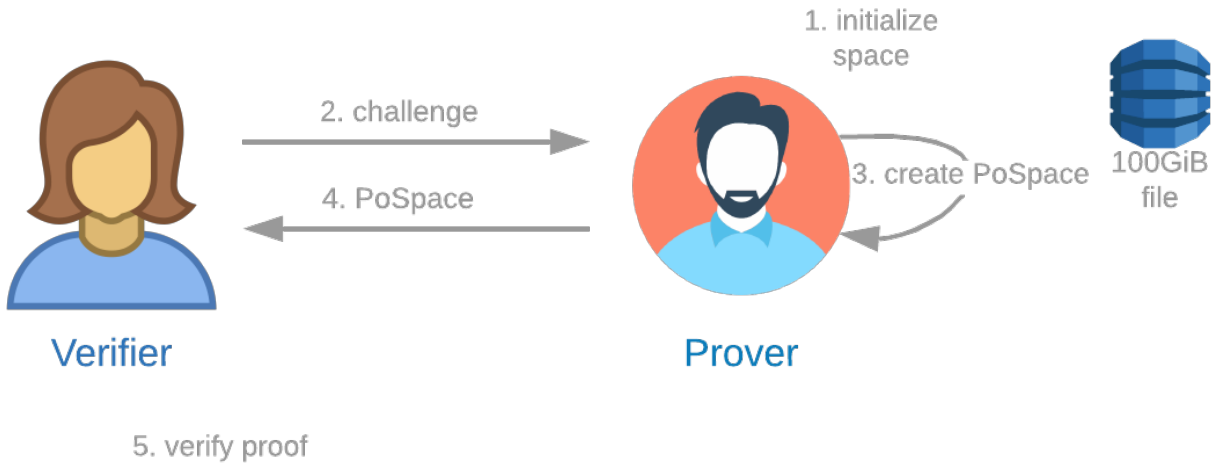
- Whenever the word *signature* is used, it refers specifically to a deterministic BLS signature, following the IETF specification with the Augmented scheme.
- The private keys performing these digital signatures are controlled and stored by the farmers.
- A unique private key is used for each plot.
- The hash function used is SHA256, except for the proofs of space which also use CHACHA8 and BLAKE3.

Proof of Space

A Proof of Space protocol is one in which:

- A Verifier can send a challenge to a Prover.
- The Prover can demonstrate to the Verifier that the Prover is reserving a specific amount of storage space at that precise time.

The Proof of Space protocol has three components: plotting, proving/farming, and verifying.



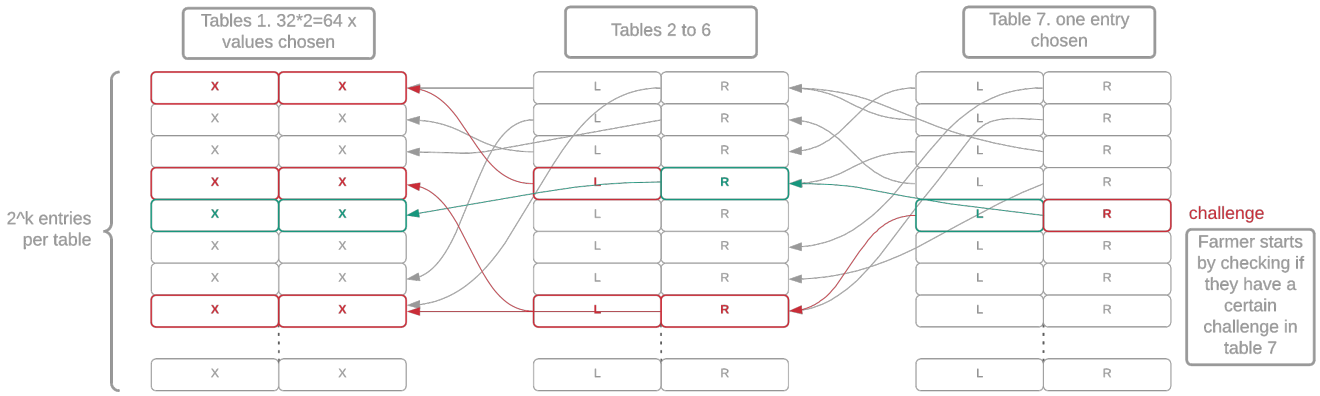
Plotting

Plotting is the process by which a Prover, who we refer to as a *farmer*, initializes a certain amount of space. To become a farmer, one must have at least 101.4 GiB available to reserve on their computer (the minimum spec is a Raspberry Pi 4). There is no upper limit to the size of a BPX farm. Several farmers have multi-PiB farms.

The k32 plot can be created in around five minutes with a high-end machine with 400 GiB of RAM, or six hours with a normal commodity machine, or 12 hours with a slow machine using one CPU core and a few GB of RAM. Opportunities still remain for huge speedups. Furthermore, each plot only needs to be created once; a farmer can farm with the same plots for many years.

Plot sizes are determined by a k parameter, where `space = 780 * k * pow(2, k - 10)`, with a minimum k of 32 (101.4 GiB). The Proof of Space construction is based on Beyond Hellman, but it is nested six times (thereby creating seven tables), and it contains other heuristics to make it practical.

Each of the seven tables in a plot is filled with random-looking data that cannot be compressed. Each table has 2^k entries. Each entry in table i contains two pointers to table i-1 (the previous table). Finally, each table-1 entry contains a pair of integers between 0 and 2^k, called "x-values." A proof of space is a collection of 64 x-values that have a certain mathematical relationship. The actual on-disk structure and the algorithm required to generate it are quite complicated, but this is the general idea.



Once the Prover has initialized 101.4 GiB, they are ready to receive a challenge and create a proof. No registration or online connection is required to create a plot using the original plot format. Nothing hits the blockchain until a reward is won, similar to PoW.

Farming

Farming is the process by which a farmer receives a sequence of 256-bit challenges to prove that they have legitimately put aside a defined amount of storage. In response to each challenge, the farmer checks their plots, generates a proof, and submits any winning proofs to the network for verification.

For each eligible plot (explained later), a farmer uses the following procedure to generate a full proof of space. Keep in mind that a plot consists of 7 tables (T1-T7) of approximately the same size, as well as 3 checkpoint tables (C1-C3), which are much smaller:

1. The farmer receives a challenge from the VDF
2. For each eligible plot, extract a k-sized value from the challenge, where *k* denotes the size of the plot (k32, k33, etc)
3. Look in the C2 table for a location at which to start scanning the C1 table
4. Scan the C1 table for the location at which to start scanning the C3 table

5. Read either one or two C3 parks. The number of parks to read depends on the index and value calculated from the C1 table. This requires an average of 5000 reads (the maximum is 10 000). These are sequential reads of 4 bytes (for an average total of 20 KiB)
6. Grab all the f7 entries matching the challenge value (which can be 0 or more), along with the index in the table at which they were found
7. For each matching f7 value, read T7 at the same index where the f7 value was found in its own table, and grab that entry, which is an index into T6
8. The T6 index contains one *line point* with two *back pointers* to T5, four to T4, eight to T3, sixteen to T2 and thirty-two to T1. Each back pointer requires 1 read, so a total of 64 disk reads (1 index from T7, 63 back pointers) are performed to fetch the whole tree of 64 x-values.

Since most proofs generated by this process are not good enough to be submitted to the network for verification, we can optimize this process by only checking one branch of the tree. This branch will return two of the 64 x-values. The position of the x-values will always be consecutive and will depend on the signage point (eg x0 and x1... or x34 and x35). We hash these x-values to produce a random 256-bit "quality string." This is combined with the difficulty and the plot size to generate the required_iterations. If the required_iterations is less than a certain number, the proof can be included in the blockchain. At this point, we look up the whole proof of space.

By only looking up one branch to determine the quality string, we can rule out most proofs. This optimization requires only around 7-9 disk seeks and reads, or about 70-90 ms on a slow hard drive

> Throughout this website, we'll make a simple assumption that a single disk seek requires 10ms. In reality, this is typically 5-10ms, so we're using a conservative estimate. The 10ms estimate also takes into account the time required to transfer data after the seek. While storage industry specs typically assume that large files are being transferred, this does not hold true for BPX farming, where proof lookups only require a tiny amount of data to be transferred. Therefore, for this website, it's safe to assume the transfer is almost instant.

BPX also uses a further optimization to disqualify a certain proportion of plots from eligibility for each challenge. This is referred to as the *plot filter*. The current requirement is that the hash of the plot ID, challenge, and signage point starts with 9 zeros. This excludes 511 out of every 512 plots. The filter hurts everyone equally (except for replotting attackers), and is therefore fair.

The plot filter effectively reduces the amount of resources required for farming by 512x - each plot only requires a few disk reads every few minutes. A farmer with 1 PiB of storage (10,000 plots) will only have an average of 20 plots that pass the filter for each challenge. Even if these plots all are stored on slow HDDs, and connected to a single Raspberry Pi, the average time required to respond to each challenge will be less than two seconds. This is well within the limits to avoid missing out on any challenges.

Each plot file has its own unique private key called a *plot key*. The plot ID is generated by hashing the plot public key, the farmer public key, and either the pool public key or the pool contract puzzle hash. The requirements for signing a proof of space depend on the type of plots being used.

In practice, the plot key is a 2/2 BLS aggregate public key between a local key stored in the plot and a key stored by the farmer software. For security and efficiency, a farmer may run on one server using this key and signature scheme. This server can then be connected to one or more harvester machines that store the actual plots. Farming requires the farmer key and the local key, but it does not require the pool key, since the pool's signature can be cached and reused for many blocks.

Verifying

After the farmer has successfully created a proof of space, the proof can be verified by performing a few hashes and making comparisons between the x-values in the proof. Recall that the proof is a list of 64 x-values, where each x-value is k bits long. For a k32 this is 256 bytes (2048 bits), and is therefore very compact.

Proof of Time

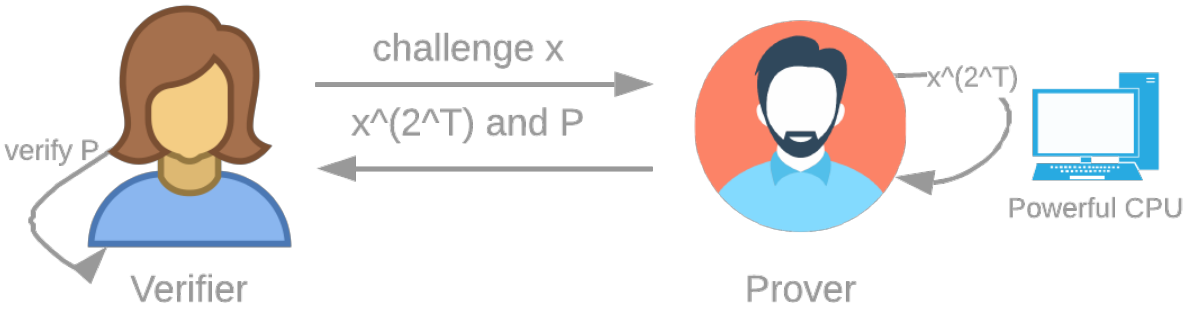
A Verifiable Delay Function, also referred to as a Proof of Time or VDF, is a proof that a sequential function was executed a certain number of times.

Verifiable: This means that after performing the computation (which takes time), the Prover can create a very small proof in a very short time, and the Verifier can verify this proof without having to redo the whole computation.

Delay: This means that the Prover actually spent a real amount of time (although we don't know exactly how much) to compute the function.

Function: This means it's deterministic: computing a VDF on an input x *always* yields the same result y.

The key word here is "sequential", like hashing a number many times: hash(hash(hash(a))), etc. This means the prover cannot just add more machines to make the function execute faster. Therefore we can assume that computing a VDF requires real (wall-clock) time. The construction that we use is repeated squaring. The Prover must square a challenge x T times. This requires time Θ(T). The Prover also must create a proof that this was performed properly.



Although the following details are not very important for understanding the consensus algorithm, the choice of what VDF to use is relevant, because if an attacker succeeds in obtaining a much faster machine, some attacks become possible.

The VDF used by BPX is repeated squaring in a class group of unknown order. There are two main ways to generate a large group that has an unknown order:

1. Use an RSA modulus, and use the integers mod N as a group. The order of the group is unknown if you can generate your modulus with many participating parties using an MPC ceremony.
2. An easier approach is to use classgroups with a large prime discriminant, which are groups of unknown order. This does not require any complex or trusted setup, so we chose this option for BPX.

To create one of these groups, one just needs a large, random, prime number. The drawbacks are that classgroup code is less tested in real life, and optimizations are less well-known than in RSA groups. We use the same initial element for the squaring (a=2, b=1 classgroup element), and instead use the challenge to generate a new random prime number for each VDF, which is used as the discriminant. The discriminant has a size of 1024 bits, which means the proof sizes are around 1024 bits. We use the Wesolowski scheme split into n (1<=n<=64) phases so that creating the proofs is very fast. Since the n-wesolowski proofs can be large, we replace them with 1-wesolowski proofs as soon as they are available. These are smaller, but require more time to make. The proofs themselves are not committed to on-chain, so they are replaceable.

Infusion

As a recap, VDFs take in an input, called a *challenge*, and produce an output, together with a proof that certifies that the function was evaluated correctly.

A *value*, in this context, can be thought of as a block with a proof of space. The value is combined with an output of a VDF, to generate a new value, which is used as the input/challenge for the next VDF. This is known as an *infusion* of a value into a VDF.

Therefore, we are chaining VDFs, but committing to a new value in between. This is used so that we have a linear progression of blocks, alternating proofs of space with proofs of time.

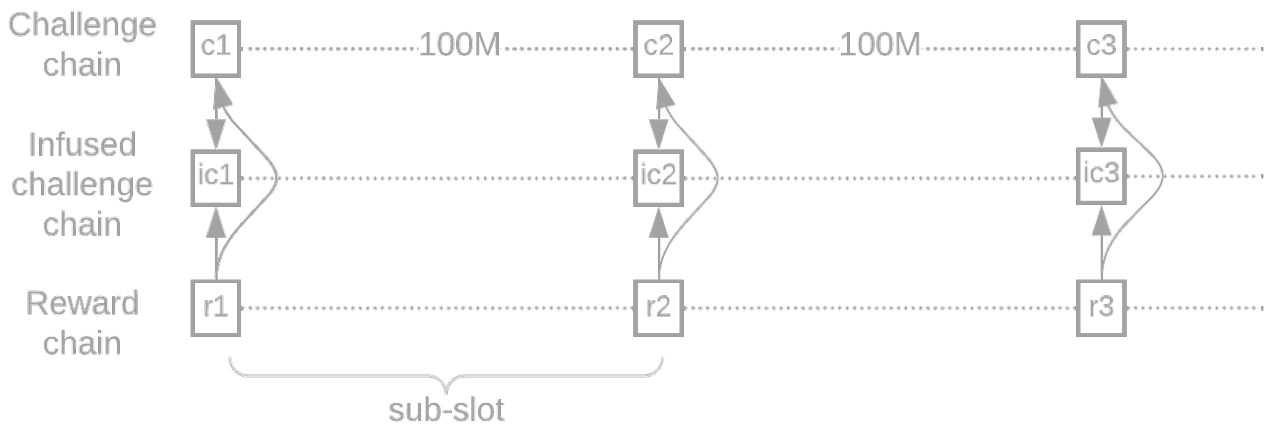
Challenges

The BPX consensus algorithm relies on timelords running VDFs for periods of time called *sub-slots*, which are adjusted periodically (and automatically) to take around 5 minutes (300 seconds). During every sub-slot, challenges are released by timelords, and a sort of mini lottery starts, where farmers check their plots for proofs of space. When farmers find a proof of space that qualifies, they broadcast it to the network.

The difficulty adjusts automatically to target 32 winning proofs for the entire network in each sub-slot, or about one winner every 9.38 seconds on average (32 winners per 300 seconds). The winning proofs are infused into the VDF at different times within the sub-slot.

> A sub-slot is always targeted to last 5 minutes. There is also a period of time called a *slot*. Typically, a slot and a sub-slot are exactly the same thing. However, in order to prevent long-range attacks, slots are required to have at least 16 blocks (and sub-slots are not). If a sub-slot ends with fewer than 16 blocks having been created, the same slot must continue for another sub-slot.

The consensus requires farmers to follow the heaviest chain, which is the chain that has the highest accumulated difficulty (usually the chain with the most blocks).



We can see three challenge points, c1, c2, and c3. At the these points timelords create challenges (256-bit hashes) which are provided as input to VDFs. Timelords take these hashes, and start computing a VDF on this challenge, for the specified number of iterations. In this example, each slot is 100,000,000 iterations. When the VDF is finished, the timelord publishes the new challenge

and the proof of the VDF. An infusion of end-of-slot information happens at the end of each sub-slot.

A challenge is always a 256-bit hash. The base info that is always included in this hash is the challenge chain VDF. However, the infused challenge chain, SubEpochSummary, difficulty, and sub slot iters might also be included, depending on where we are in the epoch cycle:

```
class ChallengeChainSubSlot(Streamable):
    challenge_chain_end_of_slot_vdf: VDFInfo
    infused_challenge_chain_sub_slot_hash: Optional[bytes32]  # Only at the end of a slot
    subepoch_summary_hash: Optional[bytes32]  # Only once per sub-epoch, and one sub-epoch
delayed
    new_sub_slot_iters: Optional[uint64]  # Only at the end of epoch, sub-epoch, and slot
    new_difficulty: Optional[uint64]  # Only at the end of epoch, sub-epoch, and slot
```

Sub-slot: a segment of a fixed number of VDF iterations, subject to periodic work difficulty adjustments, which automatically target a time of 5 minutes.

Sub-slot iterations: determines how many VDF iterations each sub-slot must have. This number is periodically adjusted.

Challenge: a sha256 output string. It is used as a proof-of-space challenge for farmers' plots. It is also used for the challenge chain VDF, and is sometimes referred to as a *challenge hash*.

As you can see on the image above, there are three VDFs being executed concurrently, each of which serves a different purpose. In the networking protocol, the three VDF proofs are usually passed around together, in what is called an *end of sub-slot bundle*.

Signage and Infusion Points

Each sub-slot in both the challenge chain and the reward chain is divided into 64 smaller VDFs. Between each of these smaller VDFs is a point called a **signage point**. Timelords publish the VDF output and proof when they reach each signage point.

> The challenge and reward chains both have signage points. The infused challenge chain, however, does not.

The signage points occur every 4.69 seconds (64 signage points per 300-second sub-slot). The number of iterations between each signage point is **sp_interval_iterations**, which is equal to sub-slot_iterations / 64.

The challenge at the start of the sub-slot is also a valid signage point. As each of the 64 signage points in the sub-slot is reached, those points are broadcast, starting from the fastest timelord's beacon client, and propagating to every other beacon client on the network.

Farmers receive these signage points and compute a hash for each plot, at each signage point. If the hash starts with nine zeros, the plot passes the filter for that signage point, and can proceed. This disqualifies around 511/512 of all plot files in the network, for that signage point. The formula to compute the filter hash is:

```
plot filter bits = sha256(plot id + sub slot challenge + cc signage point)
```

The proof of space challenge is computed as the hash of the plot filter bits:

```
PoSpace challenge = sha256(plot filter bits)
```

Using this challenge, the farmers fetch quality strings for each plot that made it past the filter. Recall that this process requires around seven random disk seeks, which takes around 70 ms on a slow HDD. The quality string is a hash derived from part of the proof of space (but the whole proof of space has yet to be retrieved).

> For both of our previous example, as well as the next example, we'll use the following values:
> sub_slot_iterations = 100,000,000
> sp_interval_iterations = sub_slot_iterations / 64 = 1,562,500

The farmer computes the **required_iterations** for each proof of space. If the required_iterations < sp_interval_iterations, the proof of space is eligible for inclusion into the blockchain. At this point,

the farmer fetches the entire proof of space from disk (which requires 64 disk seeks, or 640 ms on a slow HDD), creates an unfinished beacon block, and broadcasts it to the network.

> For the vast majority of eligible plots, required_iterations will be far too high, since on average 32 will qualify for the whole network for each 5-minute sub-slot. This is a random process so it's possible (though unlikely) for a large number of proofs to qualify. The signage_point_iterations is the number of iterations from the start of the sub-slot to the signage point. Any plot that does meet the required_iterations for a signage point will qualify as there is no rivalry between winning plots.

The exact method for required_iterations is the following:

```
sp_quality_string = sha256(quality_string + cc_signage_point)
required_iterations = (difficulty
    * difficulty_constant_factor
    * int.from_bytes(sp_quality_string, "big", signed=False)
    // pow(2, 256) * expected_plot_size(size))
```

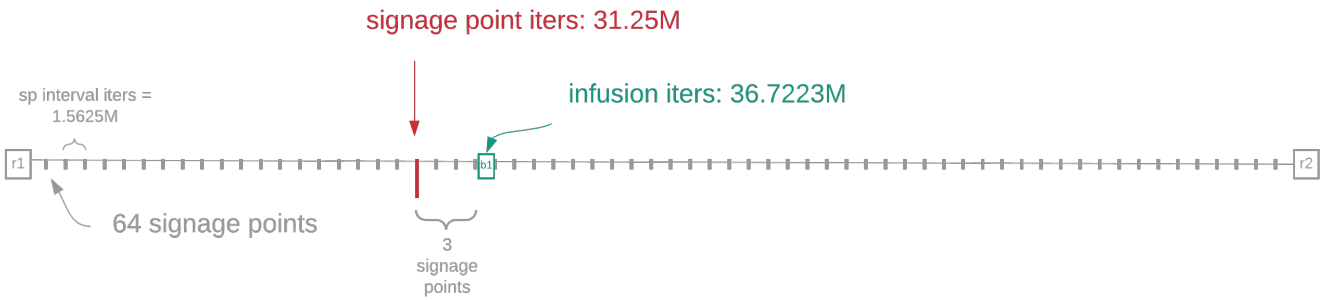
The difficulty constant factor is based on the initial constants of the beacon chain. The difficulty varies per epoch. As you can see, the **sp_quality_string** is converted into a random number between 0 and 1, by dividing it by `2^256`, and then multiplied by the plot size.

For consensus purposes, the `expected_plot_size` is `((2 * k) + 1) * (2 ** (k - 1)).`, where k>=32<50. The actual plot size is that value times a constant factor, in bytes. This is because each entry in the plot is around `k+0.5` bits, and there are around `2^(k)` entries.

The **infusion_iterations** is the number of iterations from the start of the sub-slot at which the block with at least the required quality can be included into the blockchain. This is calculated as:

`infusion_iterations = ( signage_point_iterations + 3 * sp_interval_iterations + required_iterations) % sub-slot_iterations`

Therefore, infusion_iterations will be between 3 and 4 signage points after the current signage point. Farmers must submit their proofs and blocks before the infusion point is reached. The modulus is there to allow overflows into the next sub-slot, if the signage point is near the end of the sub-slot.



A drawing shows the infusion point as a green square marked `b1`. The first and last blocks of the sub-slot are marked `r1` and `r2`, respectively. For this example, the farmer will create the block at the time of the signage point marked with a red arrow, which we'll call `b1'`.

At `b1`, the farmer's block gets combined with the VDF output for that point. This creates a new input for the VDF from that point on, i.e. we infuse the farmer's block into the VDF. `b1` is only fully valid after two events have occurred:

1. infusion_iterations has been reached, and
2. Two VDF proofs have been included: one from `r1` to the signage point and one from `r1` to `b1`. (Actually it's more since there are three VDF chains).

The farmer creates the block at the time of the signage point, `b1'`. However, `b1'` is not finished yet, since it needs the infusion point VDF. Once the infusion_iterations VDF has been released, it is added to `b1'` to form the finished block at `b1`.

Recall that in this example,

- sub-slot_iterations = 100M
- sp_interval_iterations is 1.5625M. Furthermore, let's say a farmer has a total of 1000 plots.

For each of the 64 signage points, as they are released to the network every 4.69 seconds, or every 1.5625M iterations, the farmer computes the plot filter and sees how many plots pass. For each passing plot, the farmer calculates required_iterations.

Let's say the farmer calculates required_iterations < 1.5625M once in the sub-slot. (We'll assume the exact required_iterations = 0.7828M in this instance.) Figure 5 shows this happening at the 20th signage point.

infusion_iterations is then computed as:

infusion_iterations = signage_point_iterations + (3 * sp_interval_iterations) + required_iterations

= (signage * point * sp * interval_iterations) + (3 * sp_interval_iterations) + required_iterations

= (20 * 1.5625M) + (3 * 1.5626M) + 0.7827M

= 36.7223M

After realizing they have won (at the 20th infusion point), the farmer fetches the whole proof of space, makes a beacon block (optionally including execution payload), and broadcasts this to the network. The block has until infusion_iterations (typically a few seconds) to reach timelords, who will infuse the block, creating the infusion point VDFs. With these VDFs, the block can be finished and added to the beacon chain by other beacon clients.

Harvesting

Approximately every 4.69 seconds, the beacon client sends a new signage point to the farmer, who sends it to each harvester.

The exact protocol message sent for each signage point is the following:

```
class NewSignagePointHarvester:
    challenge_hash: bytes32
    difficulty: uint64
    sub_slot_iters: uint64
    signage_point_index: uint8
    sp_hash: bytes32
```

1. The harvester receives a signage point, and computes the plot filter:

`plot filter bits = sha256(plot_id + challenge_hash + sp_hash)`

If the resulting bits start with 9 zeroes, then the plot passes the filter. This does not require disk access, since the plot_ids are stored in memory.

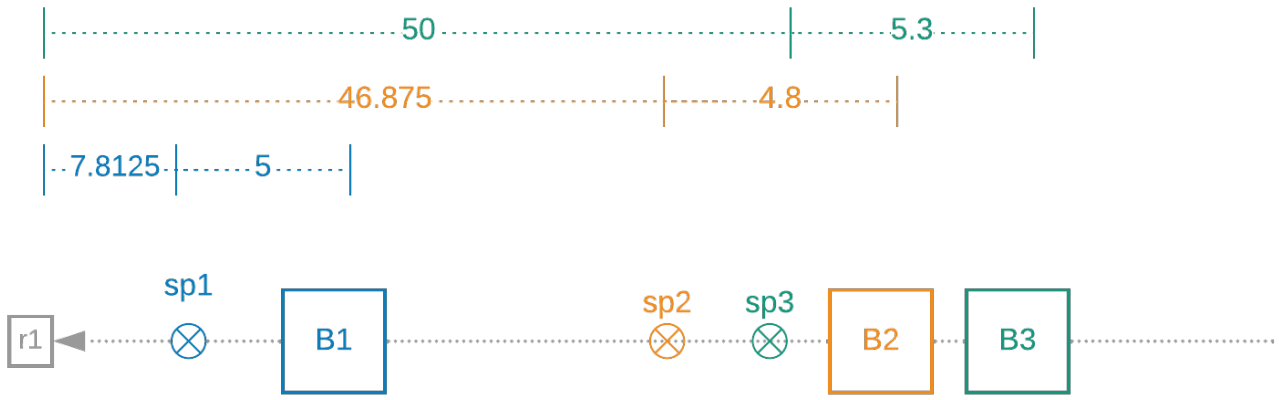
2. For each of the plots passing the filter, a new thread is started, which performs the quality lookups. Recall that this requires around 7-9 random reads into the plot, one for each table. This is where the majority of the disk activity will be. On average, 1 of every 512 plots will perform this step.

3. required_iterations is computed. If required_iterations is less than the interval_iterations, this proof of space is good (it has won a block). Most proofs will not pass this step.

5. For winning proofs, the whole proof is fetched on disk (approximately 64 random reads in the plot).

6. The proof is sent back to the farmer.

Multiple Blocks



As you can see, multiple blocks can get infused into the same sub-slot. BPX targets one beacon block every 9.38 seconds on average (32 blocks per sub-slot), and this is adjusted every 4608 blocks (around 12 hours) through the work difficulty algorithm.

VDF proofs span:

- from the previous infusion point before the current signage point to the current signage point, and
- from the previous infusion point to the current infusion point. This means that the VDF proofs required for each block can overlap.

In the example, B2 contains a VDF proof from B1 to sp2, and from B1 to B2. B3 contains a proof from B1 to sp3, and from B2 to B3. B2 does not depend at all on B3, but B3 depends on B2, since its VDF is from B2's infusion point.

The blocks get created at the signage points, but they are missing the infusion point VDF. Once this VDF is added, the block is finished, and forms part of the beacon chain.

The signatures get created and added by the farmers at the signage points, and broadcast to the whole network. There are no signatures at the infusion point; the only things added at the infusion point are the VDFs.

Finally, note that there can be multiple winners at the same signage point, all of which can be included into the blockchain. That would be the case in the diagram, if `sp2 == sp3`. The one which gets included first is the one with the lower `required_iters`, and thus earlier infusion point.

You may be wondering what happens if a farmer makes a copy of a plot and the plot becomes eligible for infusion. Do the plots each win a block reward? No - two blocks get created, but only one will be infused. The beacon client will only propagate the first copy of the block they see. The timelord node is ultimately connected to exactly one beacon client, so even if multiple identical blocks make it to that beacon client, they will not both be sent to the timelord for infusion.

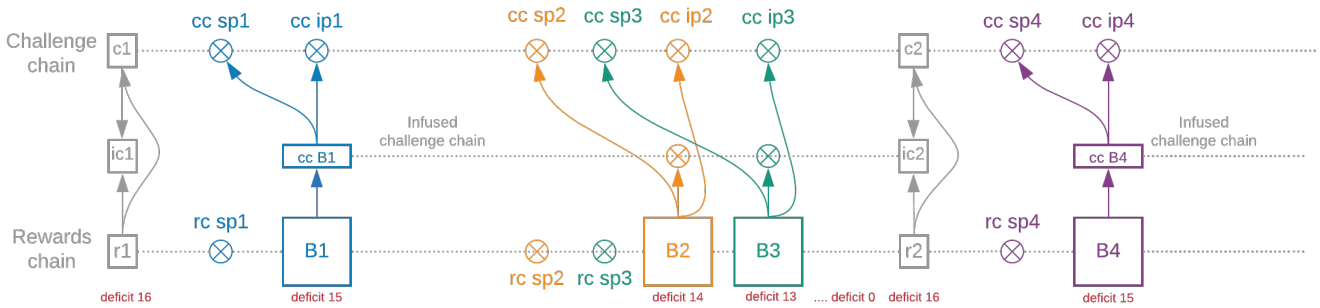
It is possible (albeit very unlikely) for two non-identical blocks to have the same infusion point, even though their hashes don't match. In this case, the beacon client will reject the second block they receive because each block must have total_iters > prev block total_iters.

VDF chains

If we only used one VDF (for the reward chain), the inclusion or exclusion of blocks would allow control of the challenge for the next slot. This means that an attacker could try many different combinations of blocks, and choose the challenge that suits them best, to obtain more wins in the next slot.

These types of attacks are called grinding attacks, and they are one of the main difficulties of changing from Proof of Work to Proof of Space or Proof of Stake.

To mitigate this, the challenges will be based only on the first block to be infused in a slot.



There is a lot going on in this diagram! Let's break it down.

There are 4 **blocks**: B1, B2, B3, and B4. Farmers create these blocks. The blocks have pointers (the arrows), and the data the pointers are pointing to is all contained within the blocks themselves. At least 16 blocks have been created in the diagram's sub-slot, but we don't draw all of them due to space constraints.

The challenge chain and the reward chain each create 64 signage points, released every 4.69 seconds (on average) by timelords. Blocks must include the signage point VDFs (which mark the signage points) for both chains.

The timelords send their VDF output to their beacon client, which adds it into an EndOfSubSlotBundle. This bundle includes the output from each chain (for example c1, ic1, and r1 in the diagram). The bundle is propagated to all other beacon clients. Blocks must also include the infusion point VDFs for all three chains.

The challenge chain broadcasts the challenges (c1 and c2). The same chain also executes the VDF from the start of the sub-slot to the end with nothing infused into it (the circles are VDF proofs but they do not interrupt the VDF). That is, in the challenge chain, the "lottery" is completely pre-

determined, and not affected by blocks in the slot, until the end of the slot.

The reward chain infuses every block that is included.

The chain in the middle is called the **infused challenge chain**. It starts at the first infused block for each challenge, and goes on until the end of the slot.

Recall that a **slot** must have at least 16 reward-chain blocks. A sub-slot doesn't have a minimum number of blocks (though it targets 32 blocks). Instead, a sub-slot always ends when sub-slot_iterations has been reached (this is targeted to take 5 minutes).

Because a sub-slot is targeted to produce more than 16 blocks, a slot usually only needs one sub-slot to meet its minimum-block requirement, but that is not always the case. For example, we may have only 10 blocks in a sub-slot, and then 3 and then 7, which means those three sub-slots form one slot. The **deficit** is the number of blocks still necessary to end the slot.

At the end of the slot, the challenge chain is combined with the infused challenge chain to generate the new challenge c2, which is used to start the challenge chain for the next sub-slot.

The only block which affects the challenge chain (and thus the PoSpace lottery) is the first block in the slot, which here is B1. In fact, it's only a deterministic part of B1 called "cc B1", which only depends on challenge chain data. An attacker who wants to grind cannot change the challenge by withholding B2, B3, or any other block apart from the first one.

An honest farmer who holds the first block (B1) will release it. If an attacker controls the first block (B1), they have two additional options: delay it or withhold it.

- Delay it: In order to know whether the new challenge will benefit them, they will need to execute the VDF all the way up to c2. By that time, their chance to get included in the reward chain is gone, since honest farmers sign only one block per proof of space.
- Withhold it: This does not provide much benefit to the attacker, since they must release it before sp2 in order to get the farmers on their chain. Farmers will choose the heaviest chain, which is the one with the most (heaviest) reward chain blocks.

Why do we commit to any blocks at all in the challenge chain? If we did not, an attacker with a faster VDF could look ahead, since they would not need the help of honest participants in order to compute the challenge chain into the future. The challenge chain would be totally deterministic. This would enable some advantage by replotting.

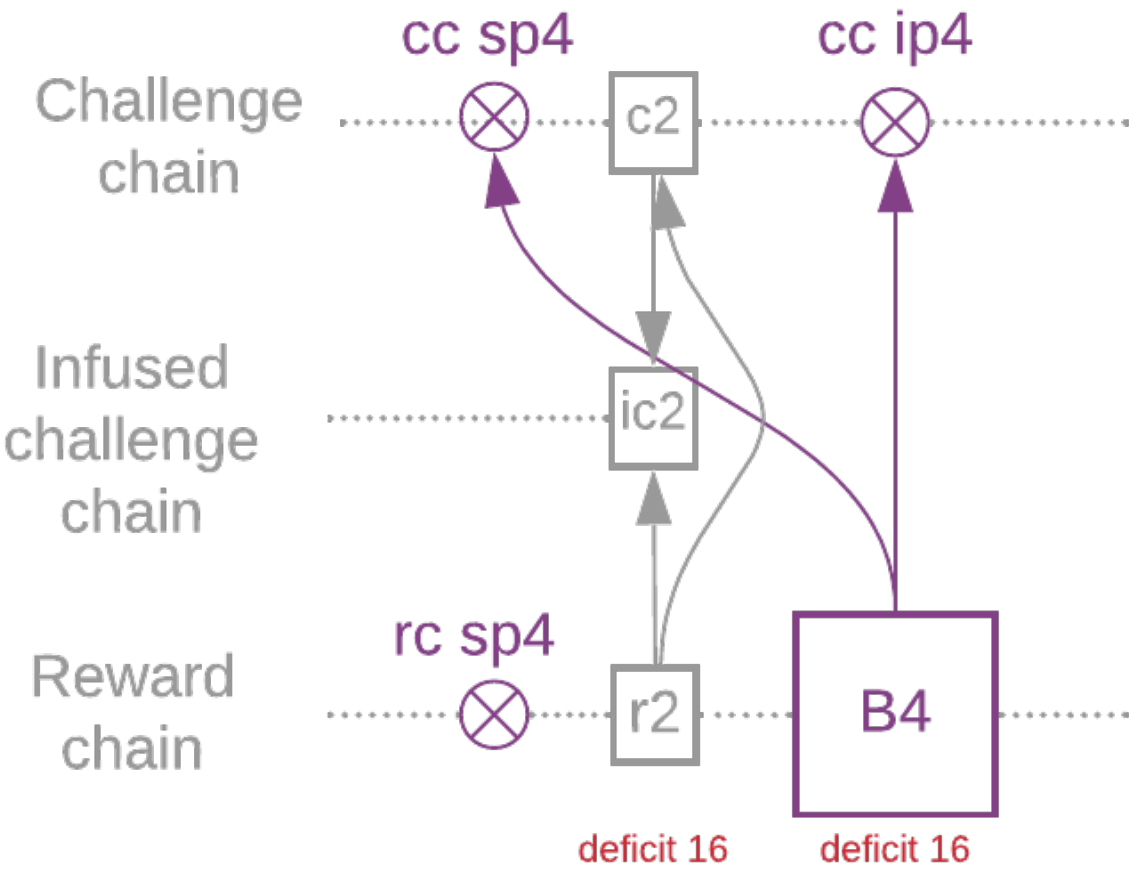
For a block to be considered valid, it has to provide VDFs for the challenge chain and reward chain, and optionally for the infused challenge chain if it is present. Forcing all VDFs to be included means that all three chains are guaranteed to move forward at the same rate.

Overflow Blocks and Weight

For a farmer to create a block, their required_iterations must be less than sub-slot_iterations / 64. This means that infusion_iterations might be greater than the sub-slot_iterations, and therefore the infusion must happen in the next sub-slot.

Overflow block: a block whose infusion point is in a different sub-slot than its signage point.

Current-slot challenge: Any given block's current-slot challenges include all challenges starting at the first challenge in the slot, and ending at the end of the slot (non-inclusive). This is relevant because sometimes a slot spans multiple sub-slots, and thus multiple challenges.



Overflow blocks cannot exist in the first sub-slot of the epoch (since the sub-slot iterations change).

Also, overflow blocks do not change the deficit unless they are based on a current-slot challenge, since overflow blocks are responses to the previous sub-slot's challenge. Overflow blocks are not challenge blocks unless they are based on a current-slot challenge. Note that it is rare for overflow blocks to decrease the deficit, since the deficit will almost always be decreased to zero, and a new slot will be started on every sub-slot.

Minimum Block Requirement

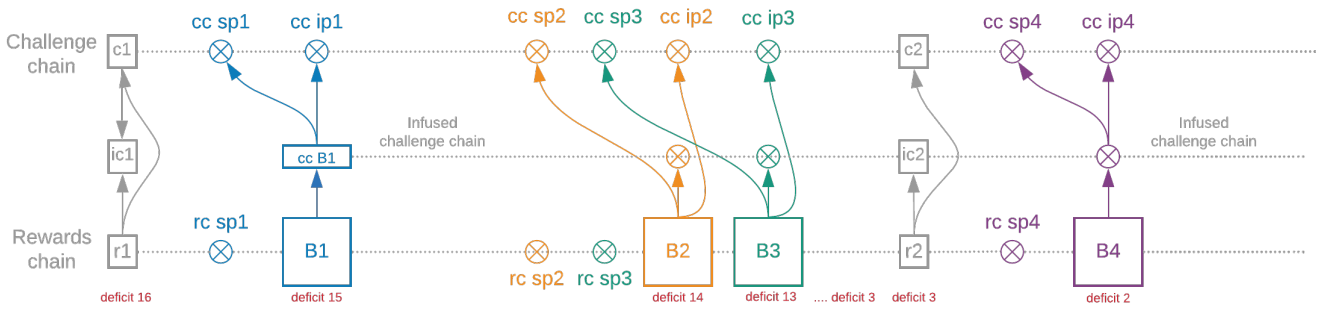
A minimum of 16 current-slot challenge blocks must be infused into the rewards chain in order for a slot to be finished. (Recall that a sub-slot has no such requirement, so a slot could span multiple sub-slots.)

The deficit is a number between 0 and 16 that is present at the start of a sub-slot, and is present for each finished block. This is defined as the number of reward chain blocks that we need to infuse in order to finish a slot. It is reset to 16 whenever we start a slot (so there must be at least 16 total blocks per challenge chain infusion). The deficit goes down for each reward chain infusion that is based on a current-slot challenge.

The block with deficit 15 is a challenge block.

The normal case is where the deficit starts at 16, and goes down to zero within the sub-slot, and resets back to 16 as we finish the slot and start a new one. In the case that we don't manage to reduce it to 0 within the end of the sub-slot, the challenge chain and infused challenge chain (if present) continue, and the deficit does not reset to 16. Blocks (including overflow blocks now), keep subtracting from the deficit until we reach 0. When we finish a sub-slot with a zero deficit, the infused challenge chain is included into the challenge chain, and the deficit is reset to 16.

This requirement was added to discourage long-range attacks. The vast majority of sub-slots will have more than 16 blocks (recall that the average number is targeted to be 32), therefore the minimum-block requirement will not have much of an affect on normal operation.



Weight

The **weight** of a block is the sum of the difficulty of this block, plus all previous blocks that are ancestors of this block. Honest beacon clients must choose the peak of the beacon chain such that the peak is the block with the heaviest weight that they know of. This is a crucial requirement, and is identical to Bitcoin's heaviest chain rule. Due to this rule, an attacker with less than 50% of the space and no VDF advantage will have trouble earning more than their fair share, since they must get lucky and create more reward chain blocks than the honest chain. Furthermore, farmers only farm on the challenges that correspond to the heaviest chain.

Both VDF speed and total amount of space are important for weight, and changes in these can trigger difficulty adjustments. If the amount of space increases, more than 32 blocks per slot will be created, so the difficulty has to be increased. If the network VDF speed increases, more than 32 blocks are created every 2.5 minutes, and thus the difficulty (and the sub-slot iterations) has to be increased.

A farmer with exclusive access to a slightly faster VDF, however, cannot easily get more rewards than a farmer with the normal speed VDF. If an attacker tries to orphan one of the blocks on the chain, having a faster VDF will not help, since the attacker's chain will have fewer blocks (and thus a lower weight). Farmers must sign the block which they are building on top of, and they will only build on top of the highest weight chain.

The VDF speed comes into play when the attacker wishes to launch a 51% attack, however. In this case, an attacking farmer can use the VDF to create a completely alternate chain with no honest blocks, and overtake the honest chain. This requires 42.7% of the total netspace, since the faster VDF chain can obtain weight at a faster rate than the honest chain.

Foliage

In the previous diagrams, there is no place for farmers to specify their rewards, since all blocks are canonical. There is also no place to include execution layer data. Everything we have talked about so far is the trunk of the blockchain.

Farmers have no say in how their block is constructed in the trunk, since they must use the exact proof of space, VDFs, and signatures that are specified. In order to include farming rewards, as well as execution payload with transactions, in the system, we must introduce an additional component of beacon blocks called *foliage*.

Trunk: The component of beacon blocks and the beacon chain which includes VDFs, proofs of space, PoSpace signatures, challenges, and previous trunk blocks, and is completely canonical. The trunk never refers to the foliage chain.

Foliage: The component of beacon blocks and the beacon chain which includes specification of where rewards should go, execution chain payload with user transactions, and what the previous foliage block is. This is up to the farmer to decide and is grindable, so it can never be used as input to the challenges.

Re-org: A re-org (or reorganization) is when a node's view of the peak changes, such that the old view contains a block that is not included in the new view (some block has been reversed). Both trunk and foliage re-orgs are possible, but should be rare in practice, and low in depth.

In the diagram below we can see that the foliage is added to blocks to produce an additional chain. This foliage includes a hash of the previous foliage, a reward block hash, and a signature. These foliage pointers are separate from the trunk chain, and are not canonical. That is, farmers could theoretically create a foliage re-org where foliage is replaced, but the exact same trunk (proofs of space and time) are used.

To prevent a foliage re-org, honest farmers only create one foliage block per block. As soon as one honest farmer has added a foliage block, the foliage becomes impossible to re-org beyond that height with the same PoSpace, since that farmer will not sign again with the same PoSpace.

Furthermore, blocks like B3, which come parallel with another foliage block (B2), do not have to sign the previous foliage block, since they do not necessarily have enough time to see it.

> By "coming in parallel", we mean that the second block's signage point occurs before the first block's infusion point.

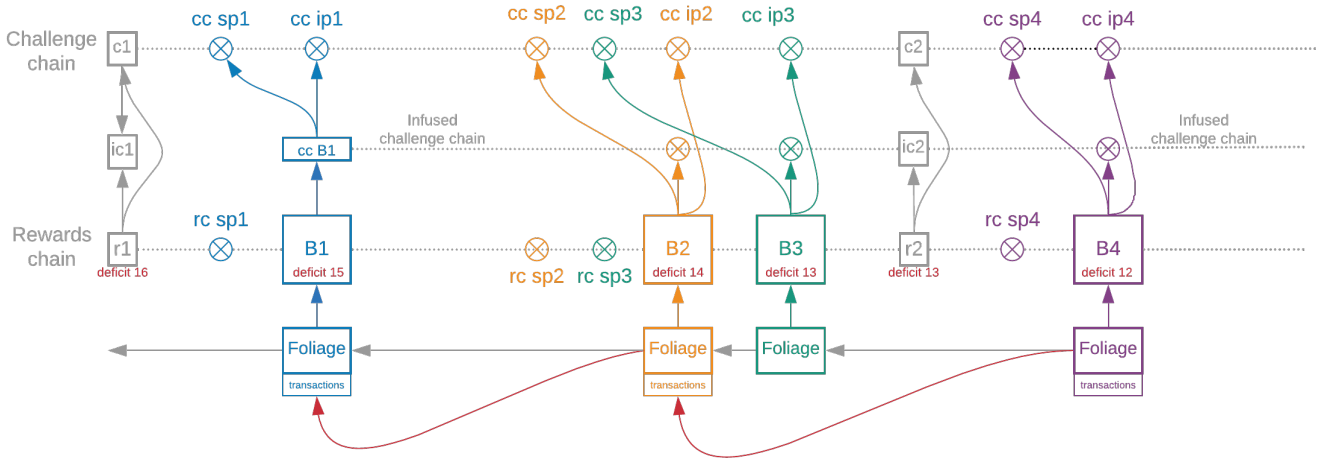
The red arrows in the diagram represent a foliage pointer that is signed by the plot key for the proof of space in that block. The gray arrows represent a hash pointer which is not signed by the plot key (therefore the gray arrow in B3 can be replaced if B2 changes or is withheld). This prevents attacks where B2 modifies their block and forces B3 to re-org.

Blocks which have red pointers are also eligible to create execution chain block, and are therefore called transaction blocks.

A block is a transaction block if and only if it is the first block whose signage point occurs after the infusion of the previous transaction block.

In the diagram, sp3 comes before B2, (a transaction block, and the previous block of B3), so B3 cannot be a transaction block.

The red arrows provide security by burying foliage blocks, but the gray arrows do not. The purpose of the gray arrows is to maintain a linked list in the foliage, and to reduce complexity in implementations. However, blocks with gray arrows pointing to them do get buried in the next-next block.



The block hash is a hash of the entire foliage and trunk block. Re-orgs work on block hashes. Even if we see a chain with the same proofs of space and time, as long as the foliages are different, the blocks will have different hashes.

Note that the farmers of blocks B2 and B3 might both have a chance to create the block, so they must both provide the signed pointer and execution payload. However, any transaction block can be included as a normal block as well, and since B2 and B3 are in parallel, only one of them can make a transaction block.

While all blocks still choose the account address of where their rewards go, those execution chain block do not get withdrawn into the execution chain until the next transaction block.

Transaction Block Time

The average time between transaction blocks is 52 seconds. Several values are required to calculate this average:

- Sub-slot time = 300 seconds
- Signage point time = 64 per sub-slot, or 300/64 = 4.69 seconds
- Average block time = 32 per sub-slot, or 300/32 = 9.38 seconds
- Minimum signage points from current signage point until infusion_iterations is reached = 3
- Minimum time for infusion_iterations to be reached (and therefore, minimum time between transaction blocks) = 3 * (300/64) = 14.06 seconds
- Average signage points until infusion_iterations is reached is slightly more than 3.5 (must wait 3 signage points, plus an average wait of about 50% of the next signage point), or around 3.5 * 4.69 = 16.42 seconds.
- To create a transaction block, infusion_iterations first must be met, and then the next block some seconds afterwards will be a transaction block. The total average time for this to happen is around 26 seconds.

The time between transaction blocks was deliberately chosen for a specific game-theoretic reason: If transaction blocks occurred at the same rate but there were no empty blocks between them, re-orgs and bribery attacks would be easier to pull off.

Additionally, the fact that there are empty blocks between transaction blocks provides several benefits:

- If blocks were created at the same rate and all of them contained transactions, low-power machines such as the Raspberry Pi wouldn't be able to keep up with the chain and therefore wouldn't be supported.
- Empty blocks can also help dampen the effect of the chain slowing down, for example during a dust storm.
- Finally, empty blocks help to smooth farmers' rewards.

Epoch and Difficulty

Sub-epoch: Sub-epoch N starts when sub-epoch N-1 ends (except for 0th sub-epoch), and it ends at the end of the first slot where 384 * (N+1) blocks have been included since genesis.

Epoch: Epoch N starts when epoch N-1 ends (except for 0th epoch), and it ends at the end of the first slot where 4608 * (N + 1) blocks have been included since genesis.

Difficulty: A constant that scales the number of iterations for a given proof of space. Iterations are computed as difficulty / quality.

Every 4608 blocks, the difficulty adjustment is automatically performed. This modifies two parameters: The slot_iterations parameter, and the difficulty parameter.

The sub_slot_iterations parameter is reset so a 5-minute slot requires close to slot_iterations many iterations. The reset is done using the values from the last epoch to approximate the iterations-per-second ratio, concretely.

We'll define `epoch$` as the period beginning with the last block that was infused *prior* to the current epoch, and ending with the last block that was infused *in* the current epoch. Thus, `epoch$` is a slightly shifted period that occurs for each epoch.

The values `t1`,`i1` and `w1` denote the timestamp, iterations (since genesis), and weight (since genesis) at the beginning of `epoch$`. Along the same lines, `(t2,i2,w2)` are the values at the end of `epoch$`.

Here's how we calculate iterations per second:

```
iterations per second = floor(num iterations in last epoch / duration of last epoch)
                      = floor((i2 - i1) / (t2 - t1))
```

That is, the delta in total iterations from the start to the end of the epoch, divided by the delta in timestamps.

Sub-slot iterations is the total number of iterations per ten-minute sub-slot. Signage point interval iterations is sub-slot iterations divided by 64 (the number of signage points per sub-slot).

```
sub slot iterations = iterations per second * 300
sp interval iterations = floor(sub slot iterations / 64)
```

Note that we don't take the iterations and time exactly at the end of an epoch, but at the last infusion point of a block in an epoch (aka `epoch$`), the reason being simply that we only have timestamps available when blocks are infused.

```
weight/sec of last epoch = (new weight - old weight) / duration of last epoch
                         = (w2 - w1) / (t2 - t1)


new difficulty = (weight/sec * target seconds) / target number of blocks
               = ((w2 - w1) / (t2 - t1) * (4608/64) * 300) / 4608
```

The sub-slot iterations are adjusted such that each slot lasts around 300 seconds. The difficulty is adjusted such that every challenge gets 32 blocks on average with fewer iterations than slot_iterations.

It is important to note that the VDF iterations per slot is not material to the weight. That is, if there were two identical worlds where VDF speeds were equal and space was equal, but the sub-slot iterations parameter was twice as high in one world, then the blockchain with the higher sub-slot iterations would get twice as many blocks included per slot, but each slot would take twice as long. The weight per second added to the chain would be the same in both cases.

Another way to look at it is that increasing sub-slot iterations increases the number of blocks per slot, but it also makes slots last longer, and thus has no effect on weight per second.

Sub Epochs

The challenge chain is completely separate and does not refer to anything in the rewards chain. If these chains stayed separate forever, an attacker with a faster VDF would be able to look into the far future and predict challenges. The attacker could create one block per slot, with limited space, thus creating a whole challenge chain. This would allow them to create plots and instantly create proofs of space for these plots that will win in the future, and then delete the plots (a long range replotting attack). This would enable them to fill their reward chain and increase their weight.

The solution to this is to periodically (every 384 blocks, which is an average of 1 hour) infuse the reward chain into the challenge chain. This means that the attacker can only perform the replotting attack for a few hours into the future. Plotting takes some time, but even if the attacker could replot instantly, the cost of a replotting attack will outweigh the benefits. This is because we do not infuse the *current* reward chain output; instead we infuse the *previous* sub-epoch's reward chain output (two hours in the past).

The cost of creating a plot includes the electricity to calculate all of the tables, the RAM necessary while creating this plot, and the fixed infrastructure costs (space, power, cooling, etc). Assuming the worst case scenario of a super fast VDF, and instant ASIC plotting - the benefits would be equivalent to the benefits of storing that plot on a HDD for a few hours. Note that this would not

guarantee a winning plot; it would be the equivalent of storing a normal plot.

It is clear that this attack is not worthwhile, and that storing the plots is much cheaper.

The above explains why the sub-epoch interval should be kept relatively low. But why can't we further reduce it to lower than 30 minutes to further disincentivize replotting attacks? The reason is that whenever non-canonical data is infused into the challenge chain (which the reward chain contains), an opportunity for grinding occurs. This means an attacker can possibly choose to include or exclude blocks to manipulate what the challenge will be 1 hour into the future. If this time is too short, they can gain a small space advantage by doing this more often.

Block Validation

Block validation in BPX is composed of two parts: header validation and body validation.

The header validation performs consensus algorithm-related checks, such as proof of space and time, signage points and infusion points, previous block hashes, foliage hashes, and timestamps. Execution payload headers are also checked, if the block contains one. Notably, it does not validate the execution payload body.

Body validation entails sending execution payload to execution client to execute all transactions in EVM.

Validating a beacon block will require access to some blocks in the past, up to a maximum theoretical value of three times the max number of blocks in a slot (3x128=384), but usually only a few are needed. Also, information regarding previous sub-epochs and epochs is needed for validation, as well as the current system timestamp. Implementations can cache only some recent blocks instead of storing all blocks in memory. Beacon client maintains a database of BlockRecords, which contain only the important pieces of block information required for validating future blocks.

Full Sync vs Normal Operation

There are two cases when a beacon client might verify blocks.

1. During a full sync, where the beacon client is trying to catch up to the most recent block, starting from an old block height. In this case, the beacon client is able to download many blocks at once.
2. During normal operation, where the beacon client is caught up to the most recent block, and is only downloading one block every few seconds.

We'll cover both of these cases below.

Full Sync

Full sync is the process by which a beacon client downloads and validates all of the blocks in the blockchain and catches up to the most recent block. Full sync is important, because it allows new nodes to validate that a blockchain is the heaviest - and thus, the currently valid - beacon chain. It allows everyone to come to consensus on the current state, regardless of when they come online,

or for how long they go offline.

The method of full sync can vary between implementations, but the high level algorithm is the following:

1. Connect to other peers on the network, by querying the DNS introducer, and crawling the network.
2. Check the current weight of the peak of the peers, and select a few peers to sync from.
3. Download and validate a weight proof, to ensure that the given peak has real work behind it.
4. Download and validate all blocks in the beacon chain, in batches.

Weight proofs are important, because they prevent other peers from lying to us about what the heaviest peak is. They also prevent us from downloading potentially useless data. Once the beacon client is caught up to the beacon chain, it can properly farm, access the execution payloads, etc.

Normal Operation

Normal operation is the process by which a beacon client continuously gossips and receives blocks with other peers, always following the heaviest peak. If our beacon client is at weight 2000, and we see that a peer has a peak at weight 2100, then we fetch that block from the peer. Usually, this is done in two phases:

1. The unfinished block is propagated across the network, along with all information up to the signage point, execution payload, etc.
2. The finished block, which includes infusion point VDFs, is also propagated.

Normal operation is much less CPU-intensive than full sync. Low-power machines like the Raspberry PI 4 should be able to easily continue normal operation.

Block Validation Steps

1. Beacon client receives a beacon block (beacon clients p2p network)
2. Beacon client performs the header validation
3. If the block is transaction block, the transactions in the block are sent to execution client as an execution payload (local RPC connection)
4. Execution client executes the transactions in EVM and validates the state in the execution block header (i.e. checks hashes match)
5. Execution client passes validation data back to beacon client, block now considered to be validated (local RPC connection)
6. Beacon client adds beacon block to head of its own blockchain

Block Creation

As soon as the Execution Client receives a new execution block, it starts working on a candidate for the next block, whether it is ever used or not. The potential next execution block is always available for fetch by the beacon client using local RPC connection.

When beacon client receives notice from farmer that we have found the proof of space and we are able to create new block, the following procedure takes place:

- Beacon client grabs the transactions and block hash from the execution client and adds them to the beacon block as an execution payload (local RPC)
- Beacon client broadcasts the beacon block over the block gossip protocol (beacon clients p2p network)
- Other beacon clients receive the block via the block gossip protocol and validate it

Block Rewards

In most cryptocurrencies, the creator of a block pays themselves based on the *current* block reward. In BPX, there is a slight difference - block rewards are paid in a *future* block, depending on whether the farmer's block is a transaction block or not.

- Option 1: If the farmer's block is a transaction block, the farmer will get paid on the next transaction block.
- Option 2: If the farmer's block is not a transaction block, the farmer will get paid on the next transaction block after the next transaction block (next next).

On the other hand, the priority part of transaction fees is paid in the current block. The base part is burned in accordance with the assumptions of EIP-1559.

Timelord algorithm

A timelord keeps track of the current peak of beacon chain, which includes an infused block at a certain height, and signage points from the peak onward. A timelord might receive new blocks to infuse, new peaks (blocks which are already infused), or new signage points.

How does a timelord decide which challenges to create proofs of time on, given a limited number of available processors? While ASICs are likely to develop in the future, at the moment the fastest classgroup VDF implementations are on general purpose hardware. Furthermore, even after the development of ASICs, it's important to emphasize that any user with a CPU can be a timelord, to provide fallbacks in the case that the ASIC timelords go down, or becomes malicious, etc.

In general, timelords work on the heaviest chain. They create proofs of time at the signage points, and broadcast these to the network as they are reached. Timelords also infuse blocks as often as they can. When the timelord receives an infused block which has a greater weight than the current peak, they switch to it immediately.

Timelords also run the three VDF chains in parallel. Therefore, at least three fast CPU cores are necessary to advance the blockchain at an efficient rate. Extra CPU cores will be necessary to create proofs, but they do not have to be as fast.

If the timelord receives a challenge with less weight than their current peak, they ignore it. If the timelord receives a challenge point later in the current chain, they do the safe thing: ignore it. The reason is that by switching to a point further in the future, the timelord might be skipping the infusion of blocks, and thus orphaning valid blocks.

If the timelord receives a block for infusion which is late (we have already reached the challenge point at which the block should have been infused), the timelord ignores the block, since infusing to it would allow attackers to instigate a withholding attack, in an attempt to re-org or DoS the chain.

Therefore, the main operation of the timelord involves keeping a cache of future blocks to infuse, broadcasting challenge points when they are reached, and infusing blocks when they reach their challenge points.

If the timelord receives a challenge with the same weight as the current peak, they choose the unfinished block which they saw first (that is, the block that has not been infused yet), as opposed to choosing the infused block (peak) which they saw first. This also disincentivizes the withholding of blocks.

Analysis

Safety

The safety of BPX consensus is similar to that of other Nakamoto consensus algorithms like Bitcoin. There is no guaranteed finality, but the more confirmations a transaction has, the safer it is.

A transaction needs a certain number of confirmations for a receiver to assume that it cannot be re-orged, under the < 42.7% (* vdf advantage) colluding assumption. Since farmers can theoretically sign multiple blocks at the same height, more *confirmations* should be used in BPX than in Bitcoin. However, BPX doesn't require anywhere near as much *clock time* as Bitcoin for a transaction to be considered safe.

In BPX, there are two main reasons to wait for a certain number of confirmations:

1. To be confident there won't be a chain re-org. A small re-org is a natural occurrence in blockchains, though rare in BPX.

To be confident that there won't be a chain re-org, you should wait for six beacon blocks to be created (around two minutes after the first confirmation).

2. Just in case there is a foliage re-org attack. This type of attack would require an attacker to discover the identity of - and successfully bribe - a large and consecutive number of anonymous block winners. This attack would be difficult to pull off, so it is expected to be extremely rare, if it is ever even attempted.

If you want to be nearly certain that even a successful foliage re-org attack won't reverse your transaction, you should wait for 32 beacon blocks to be created (around ten minutes after the first confirmation).

It's worth noting that the 54% requirement only pertains to *non-colluding* space, rather than *honest* farming space. Profit-seeking farmers gain very little by deviating from the protocol.

There is the added assumption that at least one fast timelord must be connected to the non-colluding portion of the network, and that the attacker's timelord is not significantly faster.

Liveness

The liveness of the BPX consensus system is one of its greatest strengths. Like Bitcoin, the BPX system continues advancing even when a majority of the space goes offline. Unlike bitcoin, though, the system does not slow down significantly when this happens, since not all blocks are transaction blocks. Therefore transaction throughput does not drop significantly if many participants go offline.

The network will continue to advance even if only one farmer is online, although there will be many empty slots, since a transaction block can only be created if it's below the sub-slot iterations threshold.

Of course, in the event of a long-term network split, the effects are that one chain must be chosen, so there can be large re-orgs in this case. The network will automatically choose the heavier chain, similar to PoW.

Comparison to Nakamoto PoW

("+" means a pro for BPX)

- (+) Different resources. PoSpace is ASIC-resistant and therefore anyone can participate in farming.
- (+) Hopefully more decentralized. (Analysis in mainnet's first year shows this to be the case.)
- (+) Easy merge farming. Other cryptocurrencies can use the same format, and everyone can share the space (assuming their farming computers have sufficient disk space and memory).
- (+) Minimum energy used, since only a few nodes run VDFs, and these are not parallelized. Very low marginal cost to farm.
- (+) More consistent transaction block times.
- (+) Less susceptible to selfish mining attacks.
- (+) Smaller orphan rates and forks, since blocks can be included in parallel.
- (+) Continues to advance at nearly the same rate when space decreases, since only around 1/3 of blocks include transactions. PoW Nakamoto Consensus slows down linearly when hashrate drops.

- (-) Drawback of more potential attackers (large companies). Hardware is general purpose, and therefore attackers could switch between farming, attacking, and using for data storage.
- (-) If an attacker acquires a significantly faster VDF, they could gain a space advantage.

- (-) More complexity due to sub slots and VDFs, potentially more cryptographic assumptions.

Comparison to Proof of Stake

BPX consensus algorithm could also be used for Proof of Stake, where the space farmers are replaced by stakers who own coins in the system. The benefit would be the ability to slash (delete people's stake), and farmers would have "skin in the game", but there are some concerns if Proof of Stake is used. ("+" means benefit for using space vs stake).

- (+) An attacker can transfer their stake to someone else, but fork the chain right before their stake is transferred. In this alternate chain, the attacker still has all of their stake, and can therefore advance the chain. The "nothing at stake" issue is different in PoS than in PoSpace since creating a PoSpace requires a physical resource (hard drive space), while creating a PoS only requires a key.
- (+) An attacker can guarantee their share of the whole monetary supply, by staking their rewards (the rich get richer), since the total number of coins is limited.
- (+) There might be situations where the attacker can grind on many different ways to transfer stake. Perhaps this can be mitigated by requiring a long period before stake becomes active.
- (+) Registration is required, you cannot participate in proof of stake until you sign up. This reduces privacy and scalability (how many people can stake).
- (+) Higher barrier to entry: security deposits and slashing make it difficult for small users to participate. Slashing can be a huge risk for participants in the network. Centralized custodians lead to a less distributed set of participants.

- (-) Skin in the game: with PoS, the consensus can slash people's stake, and also requires some investment into the system (exposure to price). In Proof of Space, hard drives can be used for other purposes and there is no ability to "slash" people's hardware.

Comparison to BFT consensus algorithms

Proof of Space could also be used as a Sybil-resistant mechanism in order to bootstrap a Byzantine consensus (k-agreement) system. Filecoin, and many Proof of Stake systems use aspects of Byzantine consensus.

The pros and cons of using BPX consensus vs Byzantine consensus, which vary from algorithm to algorithm ("+" means a pro for BPX):

- (+) Much simpler.
- (+) No registration requirement.
- (+) No scalability requirement (scales to millions of farmers).
- (+) More censorship resistant. As long as a small portion of the farming space does not censor, eventually you can get into the blockchain.
- (+) No liveness requirements, potentially fewer network assumptions.
- (+) Fully objective (A node can compare chain 1 and chain 2, and immediately know which one is heavier). No need for checkpoints with ⅔ consensus.

- (-) No finality, only probabilistic.
- (-) Need to wait longer for transaction confirmations (related to no finality).
- (-) Less consistent block times and transaction throughput.