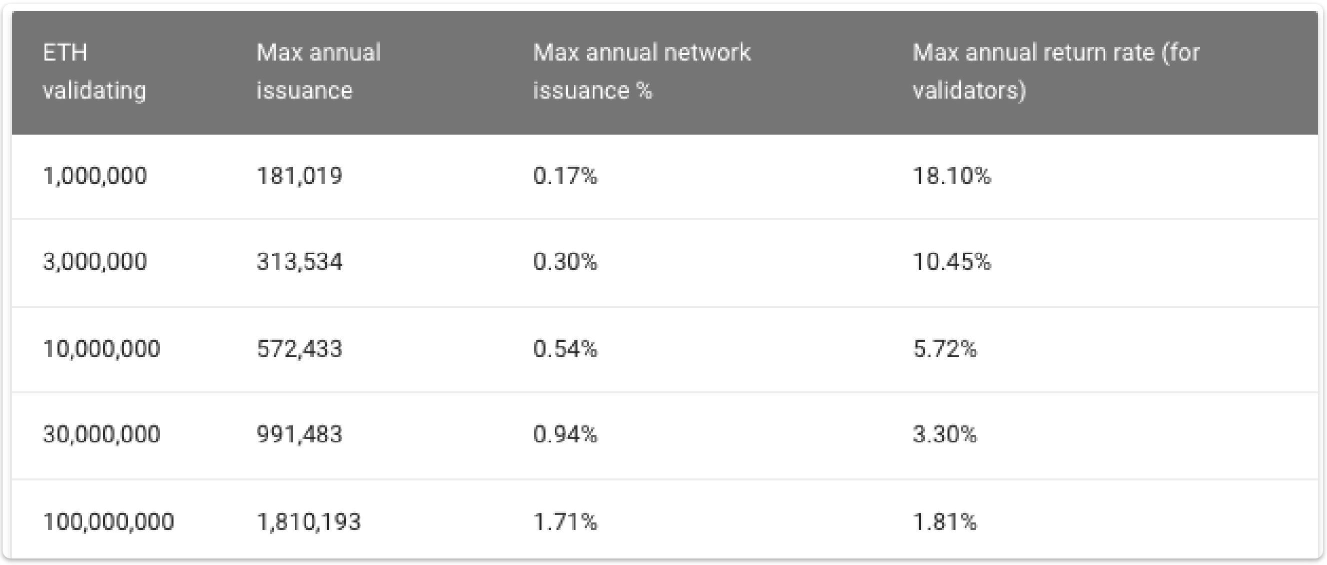
Lesson 15 - ETH 2.0 continued

"There are two ways of constructing a software design:
One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies.
The first method is far more difficult." -C.A.R. Hoare

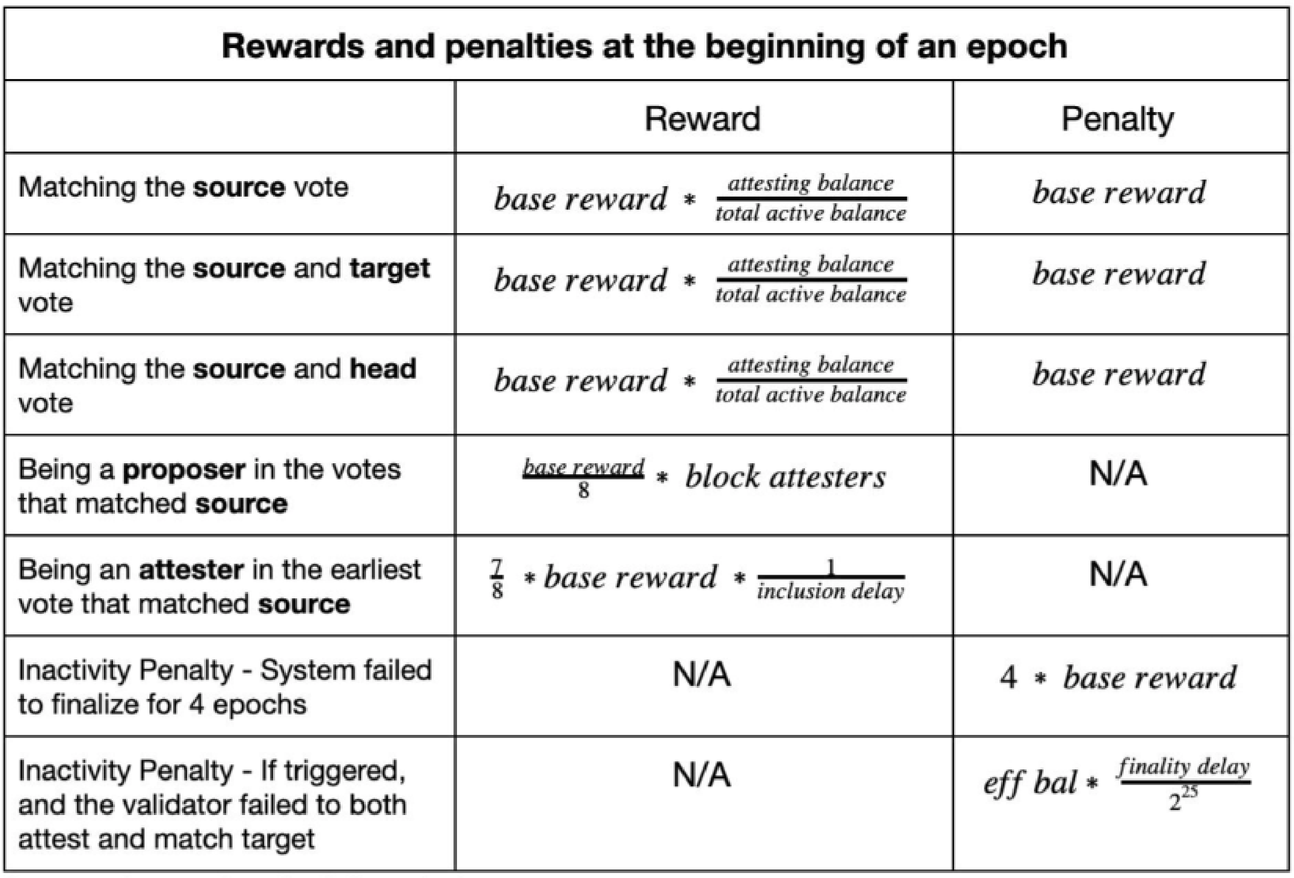
ETH 2 Validator rewards

Staking Rewards

In order to incentivize those that have ETH to stake in the network, validators will be rewarded for performing their assigned duties. Every 6 minutes, a validator is assigned a duty and is rewarded if it is performed. This reward is a sliding scale based on total network stake. So if total ETH staked is very low, the return rate per validator increases, but as stake rises, total annual issuance increases to fund those validators, while they individually will receive less rewards. The current suggested payouts are as follows:



ETH validating	Max annual issuance	Max annual network issuance %	Max annual return rate (for validators)
1,000,000	181,019	0.17%	18.10%
3,000,000	313,534	0.30%	10.45%
10,000,000	572,433	0.54%	5.72%
30,000,000	991,483	0.94%	3.30%
100,000,000	1,810,193	1.71%	1.81%



Rewards and penalties at the beginning of an epoch		
	Reward	Penalty
Matching the **source** vote	$base\ reward * \frac{attesting\ balance}{total\ active\ balance}$	$base\ reward$
Matching the **source** and **target** vote	$base\ reward * \frac{attesting\ balance}{total\ active\ balance}$	$base\ reward$
Matching the **source** and **head** vote	$base\ reward * \frac{attesting\ balance}{total\ active\ balance}$	$base\ reward$
Being a **proposer** in the votes that matched **source**	$\frac{base\ reward}{8} * block\ attesters$	N/A
Being an **attester** in the earliest vote that matched **source**	$\frac{7}{8} * base\ reward * \frac{1}{inclusion\ delay}$	N/A
Inactivity Penalty - System failed to finalize for 4 epochs	N/A	$4 * base\ reward$
Inactivity Penalty - If triggered, and the validator failed to both attest and match target	N/A	$eff\ bal * \frac{finality\ delay}{2^{25}}$

$$base\ reward = effective\ balance * \frac{BASE\_REWARD\_FACTOR}{BASE\_REWARDS\_PER\_EPOCH\ \sqrt{\Sigma\ active\ balance}}$$

- *Finality Delay = Previous epoch − Finalized epoch*
- *Attesting Balance = Sum of unslashed attester balance*
- Constant BASE_REWARD_FACTOR = 64
- Constant BASE_REWARDS_PER_EPOCH = 4
- Constant PROPOSER_REWARD_QUOTIENT = 8
- Constant MIN_EPOCHS_TO_INACTIVITY_PENALTY = 4
- Constant INACTIVITY_PENALTY_QUOTIENT = 2**25

Upcoming changes to Solidity

- Event definition at file level
- Modifiers jump rather than in lining
- Try catch for custom errors
- Have selectors for events and errors
- Immutable reference types
- Generics

Is upgrading a contract an anti pattern ?

The great advantage to smart contract is that they're immutable, no one can hack them or change their terms once they are deployed

**The great drawback to smart contracts is that they're immutable, you can't fix them once they're deployed.

See article
Upgradability is a bug

1. Smart contracts are useful because they’re **trustless**.
2. **Immutability** is a critical feature to achieve trustlessness.
3. **Upgradeability** undermines a contract’s immutability.
4. Therefore, upgradeability is a **bug**. (But there are mitigations!)

Contract upgrade anti patterns
"We strongly advise against the use of these patterns for upgradable smart contracts. Both strategies have the potential for flaws, significantly increase complexity, and introduce bugs, and ultimately decrease trust in your smart contract. Strive for simple, immutable, and secure contracts rather than importing a significant amount of code to postpone feature and security issues."

Open Zeppelin universal upgradeable proxy standard (UUPS)

In this pattern, the upgrade logic is placed in the implementation contract.

```
contract UUPSProxy {
    address implementation;
    fallback() external payable {
       // delegate here
    }
}

abstract contract UUPSProxiable {
    address implementation;
    address admin;

    function upgrade(address newImplementation) external {
        require(msg.sender == admin);
        implementation = newImplementation;
    }
}
```

There are plugins available for Hardhat and truffle

Diamond pattern

Based on EIP 2535
Trail of Bits Audit - Good idea, bad design
"The code is over-engineered, with lots of unnecessary complexities, and we can’t recommend it at this time."
But.. projects are using it
For example Aavegotchi

Foundry template

See this template from Paul Berg

You can click the 'use this template' button
or install

```
forge init my-project --template https://github.com/paulrberg/foundry-template
cd my-project
yarn install # install solhint and prettier and other goodies
```

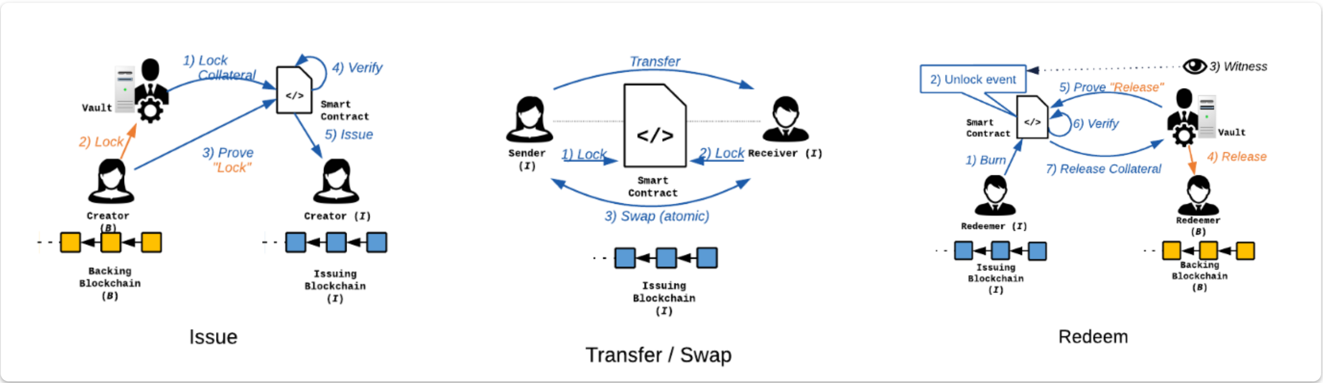
Cross chain bridges

The general idea behind a bridge is that

- The users deposits TokenA in a contract on chain A
- The contract signals a contract on chain B to verify that a deposit has been made
- The contract in chain B mints TokenB and transfers this to the users address on chain B.

XClaim

Focussed on implementing Bitcoin-backed tokens on Ethereum,



Features

- Secure audit logs: Logs are constructed to record actions of all users both on Bitcoin and Ethereum.
- Transaction inclusion proofs: Chain relays are used to prove correct behavior on Bitcoin to the smart contract on Ethereum.
- Proof-or-Punishment: Instead of relying on timely fraud proofs (reactive), XCLAIM requires correct behavior to be proven proactively.
- Over-collateralization: Non-trusted intermediaries are bound by collateral, with mechanisms in place to mitigate exchange rate fluctuations.

MINA - ETH Bridge

See video
Documentation 1
Documentation 2

Other Bridges

Rainbow bridge (Near <> ETH)
tBTC (BTC <> ETH)
and many more

Useful Maths Libraries

https://www.smartcontractresearch.org/t/deep-diving-into-prbmath-a-library-for-advanced-fixed-point-math/686
You get these functions

- Absolute
- Arithmetic and geometric average
- Exponentials (binary and natural)
- Floor and ceil
- Fractional
- Inverse
- Logarithms (binary, common and natural)
- Powers (fractional number and basic integers as exponents)
- Multiplication and division
- Square root

In addition, there are getters for mathematical constants:

- Euler’s number
- Pi
- Scale (1e18, which is 1 in fixed-point representation)

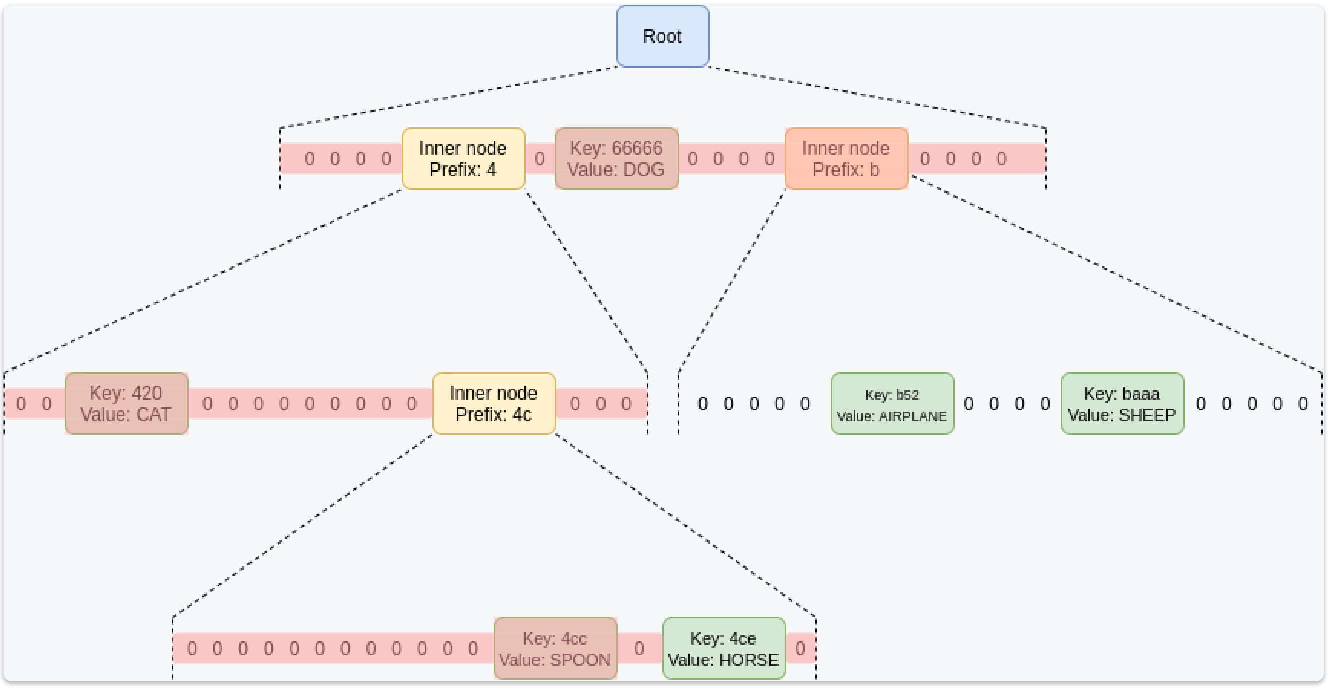
Verkle trees

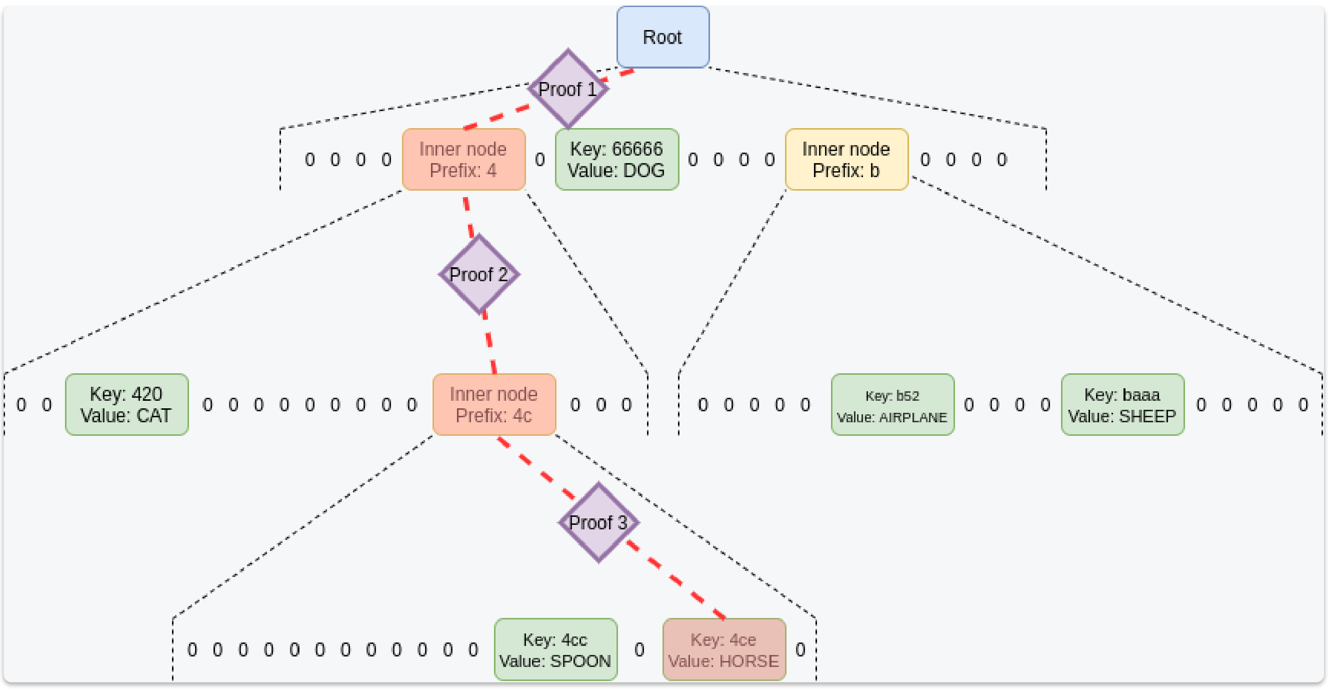
https://vitalik.ca/general/2021/06/18/verkle.html

See article
and Ethereum Cat Herders Videos

Like merkle trees, you can put a large amount of data into a Verkle tree, and make a short proof ("witness") of any single piece, or set of pieces, of that data that can be verified by someone who only has the root of the tree.
What Verkle trees provide, however, is that they are much more efficient in proof size. If a tree contains a billion pieces of data, making a proof in a traditional binary Merkle tree would require about 1 kilobyte, but in a Verkle tree the proof would be less than 150 bytes.
Verkle trees replace hash commitments with vector commitments or better still a polynomial commitment.
Polynomial commitments give us more flexibility that lets us improve efficiency, and the simplest and most efficient vector commitments available are polynomial commitments.

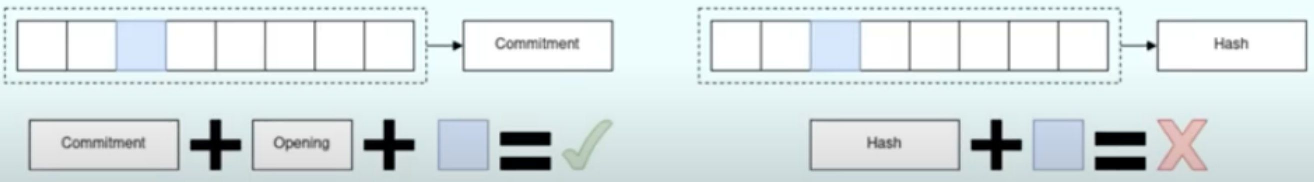
The number of nodes needed in a merkle proof is much greater than in a verkle proof

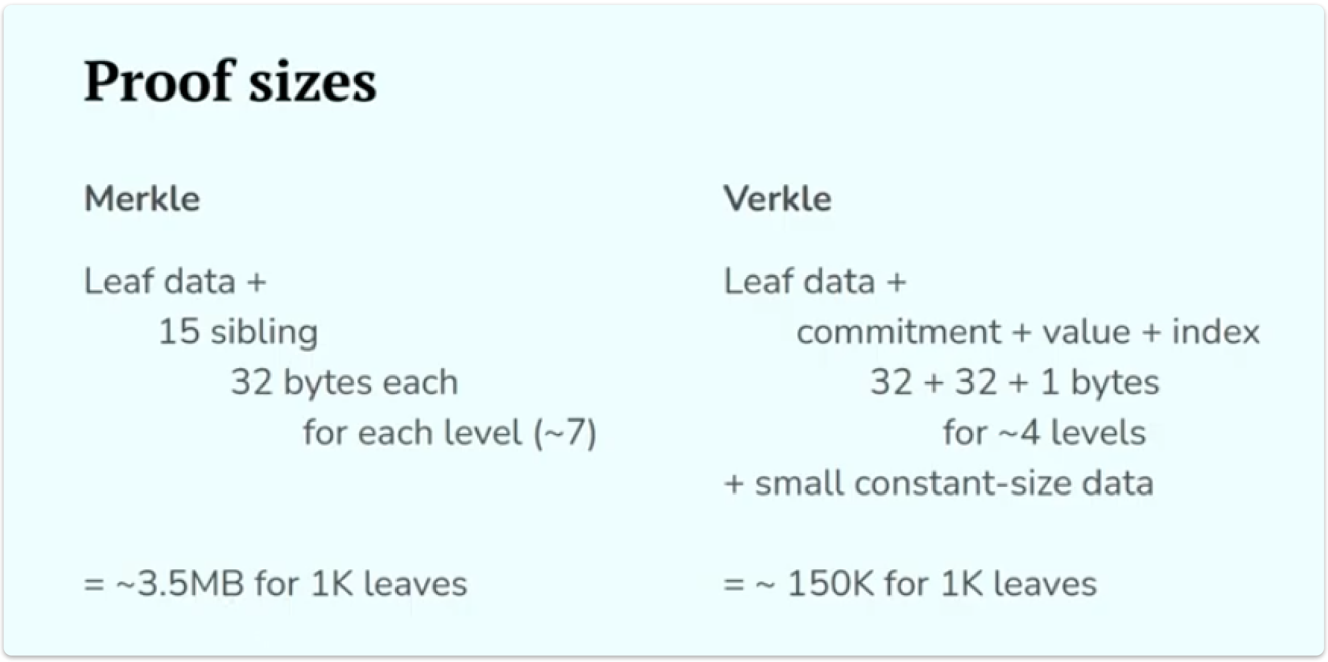




Vector commitments vs. Hash

- Vector commitments: existence of an "opening", a small payload that allow for the verification of a portion of the source data without revealing it all.
- Hash : verifying a portion of the data = revealing the whole data.





Proof sizes

Merkle

Leaf data +
  15 sibling
    32 bytes each
      for each level (~7)

= ~3.5MB for 1K leaves

Verkle

Leaf data +
  commitment + value + index
    32 + 32 + 1 bytes
      for ~4 levels
+ small constant-size data

= ~ 150K for 1K leaves

Stateless Ethereum

The Ethereum world state contains all Ethereum accounts, their balances, deployed smart contracts, and associated storage, it grows without bound.

The idea of stateless Ethereum was proposed in 2017, it was realised that unbounded state is problematic, especially in providing a barrier for entry to people wanting to run nodes. Increasing the hardware requirements for a node leads to centralisation.

The aim of Stateless Ethereum is to mitigate unbounded state growth.

Two paths were initially proposed : **weak statelessness** and **state expiry**:

- State expiry:
  remove state that has not been recently accessed from the state (think: accessed in the last year), and require witnesses to revive expired state. This would reduce the state that everyone needs to store to a flat ~20-50 GB.
- Weak statelessness:
  only require block proposers to store state, and allow all other nodes to verify blocks statelessly. Implementing this in practice requires a switch to Verkle trees to reduce witness sizes.

However according to this roadmap it may make sense to do both together.

State expiry without Verkle trees requires very large witness sizes for proving old state, and switching to Verkle trees without state expiry requires an in-place transition procedure (eg. EIP 2584) that is almost as complicated as just implementing state expiry.

See the full proposal here

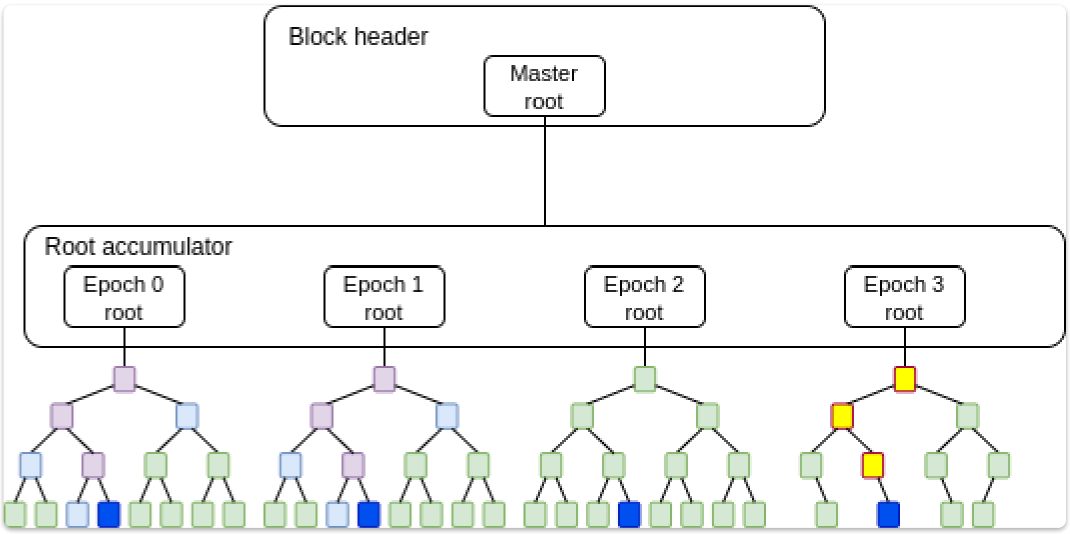
The core idea is that there would be a state tree per epoch (think: 1 epoch ~= 8 months), and when a new epoch begins, an empty state tree is initialized for that epoch and any state updates go into that tree.
Full nodes in the network would only be required to store the most recent two trees, so on average they would only be storing state that was read or written in the last ~1.5 epochs ~= 1 year.

Block producers will in addition to the block provide a 'witness' that the data is required to execute the transactions in the block.

There are two key principles:

- Only the most recent tree (ie. the tree corresponding to the current epoch) can be modified. All older trees are no longer modifiable; objects in older trees can only be modified by creating copies of them in newer trees, and these copies supersede the older copies.

- Full nodes (including block proposers) are expected to only hold the most recent two trees, so only objects in the most recent two trees can be read without a witness. Reading older objects requires providing witnesses.



Suppose the dark-blue object was last modified in epoch 0, and you want to read/write it in a transaction in epoch 3.
To prove that epoch 0 really was the last time the object was touched, we need to prove the dark-blue values in epochs 0, 1 and 2.
Full nodes still have the full epoch 2 state, so no witness is required.
For epochs 0 and 1, we do need witnesses: the light blue nodes, plus the purple nodes that can be regenerated during witness verification.
After this operation, a copy of the object is saved in the epoch 3 state.

EIP712 and EIP2612

EIP712 a standard for signing transactions and ensuring that they are securely used

DOMAIN SEPARATOR (FOR AN ERC-20)

```
bytes32 eip712DomainHash = keccak256(
    abi.encode(
        keccak256(
            "EIP712Domain(string name,string version,
            uint256 chainId,address verifyingContract)"
        ),
        keccak256(bytes(name())), // ERC-20 Name
        keccak256(bytes("1")),    // Version
        chainid(),
        address(this)
    )
);
```

Permit Function

"Arguably one of the main reasons for the success of ERC-20 tokens lies in the interplay between `approve` and `transferFrom`, which allows for tokens to not only be transferred between externally owned accounts (EOA), but to be used in other contracts under application specific conditions by abstracting away `msg.sender` as the defining mechanism for token access control.

However, a limiting factor in this design stems from the fact that the ERC-20 `approve` function itself is defined in terms of `msg.sender`. This means that user's *initial action* involving ERC-20 tokens must be performed by an EOA .
If the user needs to interact with a smart contract, then they need to make 2 transactions (`approve` and the smart contract call which will internally call `transferFrom`). Even in the simple use case of paying another person, they need to hold ETH to pay for transaction gas costs."

The permit function is used for any operation involving ERC-20 tokens to be paid for using the token itself, rather than using ETH.
EIP2612

EIP2612 adds the following to ERC20

```
function permit(address owner, address spender,
    uint value, uint deadline, uint8 v, bytes32 r, bytes32 s) external
function nonces(address owner) external view returns (uint)
function DOMAIN_SEPARATOR() external view returns (bytes32)
```

A call to `permit(owner, spender, value, deadline, v, r, s)` will set `approval[owner][spender]` to `value`,
increment `nonces[owner]` by 1, and
emit a corresponding `Approval` event,
if and only if the following conditions are met:

- The current blocktime is less than or equal to `deadline`.
- `owner` is not the zero address.
- `nonces[owner]` (before the state update) is equal to `nonce`.
- `r`, `s` and `v` is a valid `secp256k1` signature from `owner` of the message:

If any of these conditions are not met, the `permit` call must revert.

Traditional Process = Approve + TransferFrom

The user sends a transaction which will approve tokens to be used via the UI.
The user pays the gas fee for this transaction
The user submits a second transaction and pays gas again.

Permit Process

User signs the signature — via Permit message which will sign the approve function.
User submits signature. This signature does not require any gas — transaction fee.
User submits transaction for which the user pays gas, this transaction sends tokens.

Original introduced by Maker Dao

In more detail

Taken from permit article

1. Our contract needs a domain hash as above and to keep track of nonces for addresses
2. We need a permit struct

```
bytes32 hashStruct = keccak256(
    abi.encode(
        keccak256("Permit(address owner,address spender,
        uint256 value,uint256 nonce,uint256 deadline)"),
        owner,
        spender,
        amount,
        nonces[owner],
        deadline
    )
);
```

This struct will ensure that the signature can only used for

the permit function
to approve from owner
to approve for spender
to approve the given value
only valid before the given deadline
only valid for the given nonce
The nonce ensures someone can not replay a signature, i.e., use it multiple times on the same contract.

We can then put these together

```
bytes32 hash = keccak256(
    abi.encodePacked(uint16(0x1901), eip712DomainHash, hashStruct)
);
```

On receiving the signature we can verify with

```
address signer = ecrecover(hash, v, r, s);
require(signer == owner, "ERC20Permit: invalid signature");
require(signer != address(0), "ECDSA: invalid signature");
```

We can then increase the nonce and perform the approve

```
nonces[owner]++;
_approve(owner, spender, amount);
```

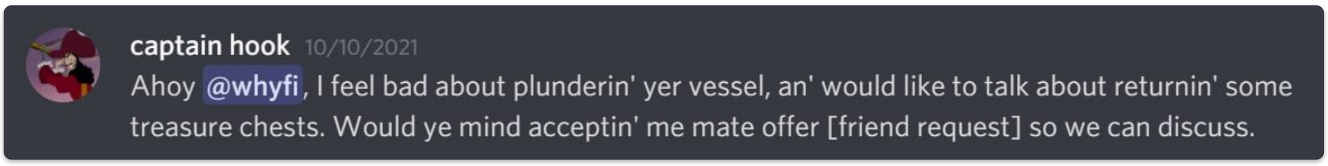
Example from Solmate

Draft Example from Open Zeppelin

```
abstract contract ERC20Permit is ERC20, IERC20Permit, EIP712 {
    using Counters for Counters.Counter;

    mapping(address => Counters.Counter) private _nonces;

    // solhint-disable-next-line var-name-mixedcase
    bytes32 private immutable _PERMIT_TYPEHASH =
        keccak256("Permit(address owner,address spender,uint256 value,
        uint256 nonce,uint256 deadline)");

    /**
     * @dev Initializes the {EIP712} domain separator using the `name`
parameter,
     * and setting `version` to `"1"`.
     *
     * It's a good idea to use the same `name` that is defined
     * as the ERC20 token name.
     */
    constructor(string memory name) EIP712(name, "1") {}

    /**
     * @dev See {IERC20Permit-permit}.
     */
    function permit(
        address owner,
        address spender,
        uint256 value,
        uint256 deadline,
        uint8 v,
        bytes32 r,
        bytes32 s
    ) public virtual override {
        require(block.timestamp <= deadline, "ERC20Permit: expired
deadline");

        bytes32 structHash = keccak256(abi.encode(_PERMIT_TYPEHASH, owner,
        spender, value, _useNonce(owner), deadline));

        bytes32 hash = _hashTypedDataV4(structHash);

        address signer = ECDSA.recover(hash, v, r, s);
        require(signer == owner, "ERC20Permit: invalid signature");

        _approve(owner, spender, value);
    }
```

Exploited by a pirate

https://twitter.com/bertcmiller/status/1505698980067434541
An exploit in an MEV bot allowed an attacker to take $1m in a single transaction
The attacker felt bad so they revealed themselves in the Flashbots Discord and offered to give the victim their money back... all while roleplaying as a pirate



captain hook 10/10/2021
Ahoy @whyfi, I feel bad about plunderin' yer vessel, an' would like to talk about returnin' some treasure chests. Would ye mind acceptin' me mate offer [friend request] so we can discuss.