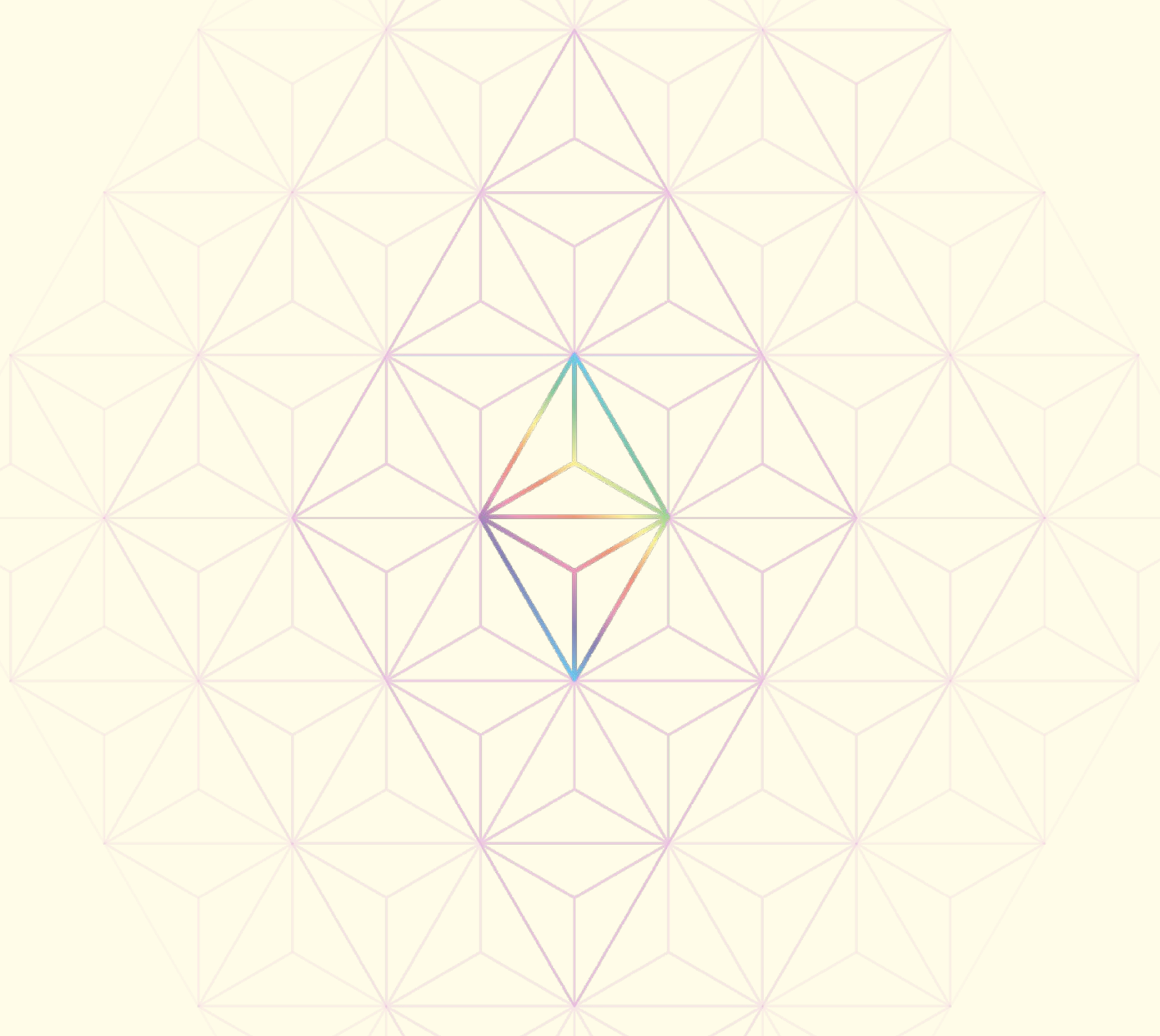
Phase 0 Deep Dive

EF Research Team

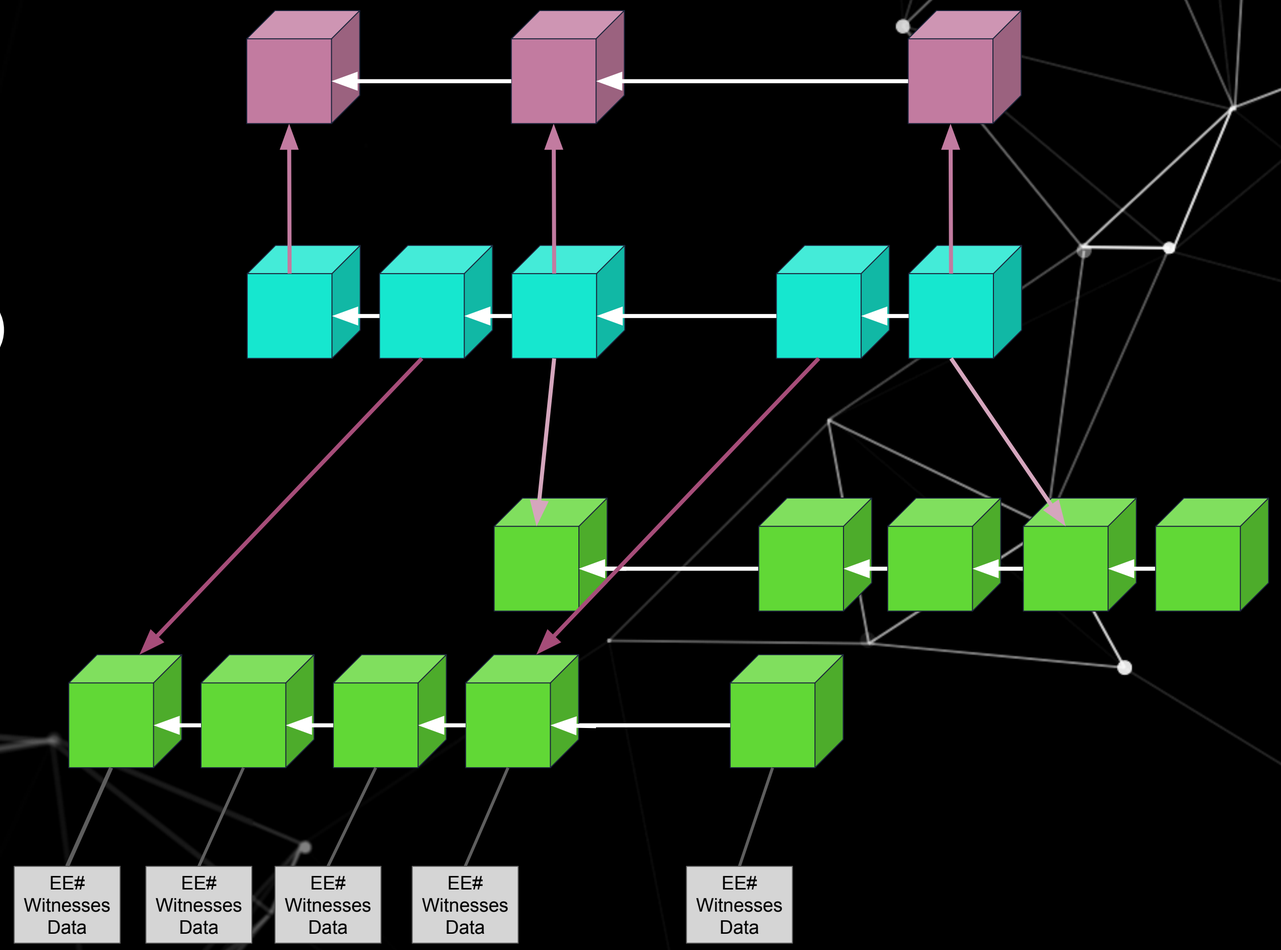


PoW 1.0 Chain
economic seed

Beacon Chain
(PoS System Chain)

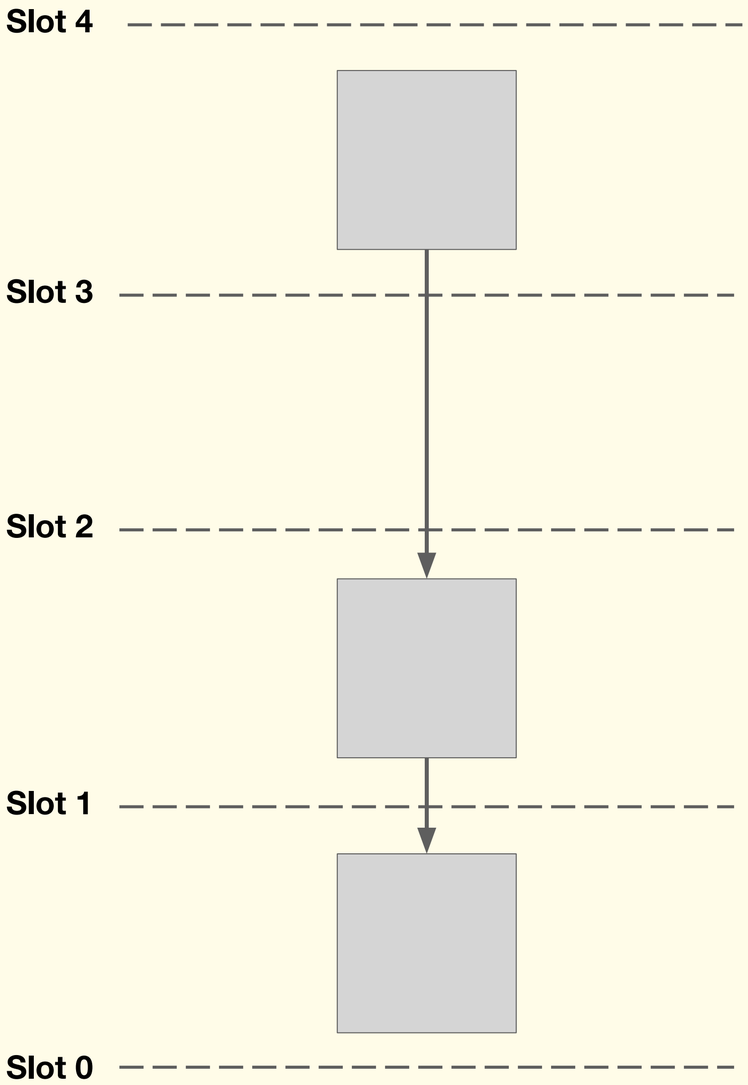
Shard Chains

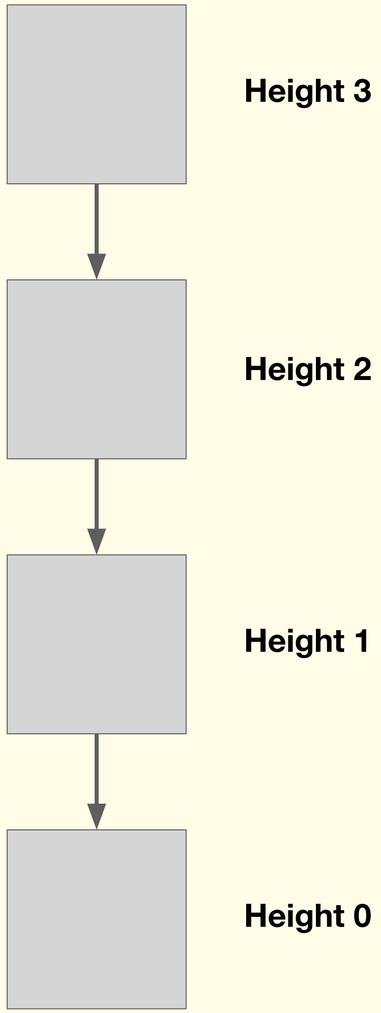
Stateless Execution



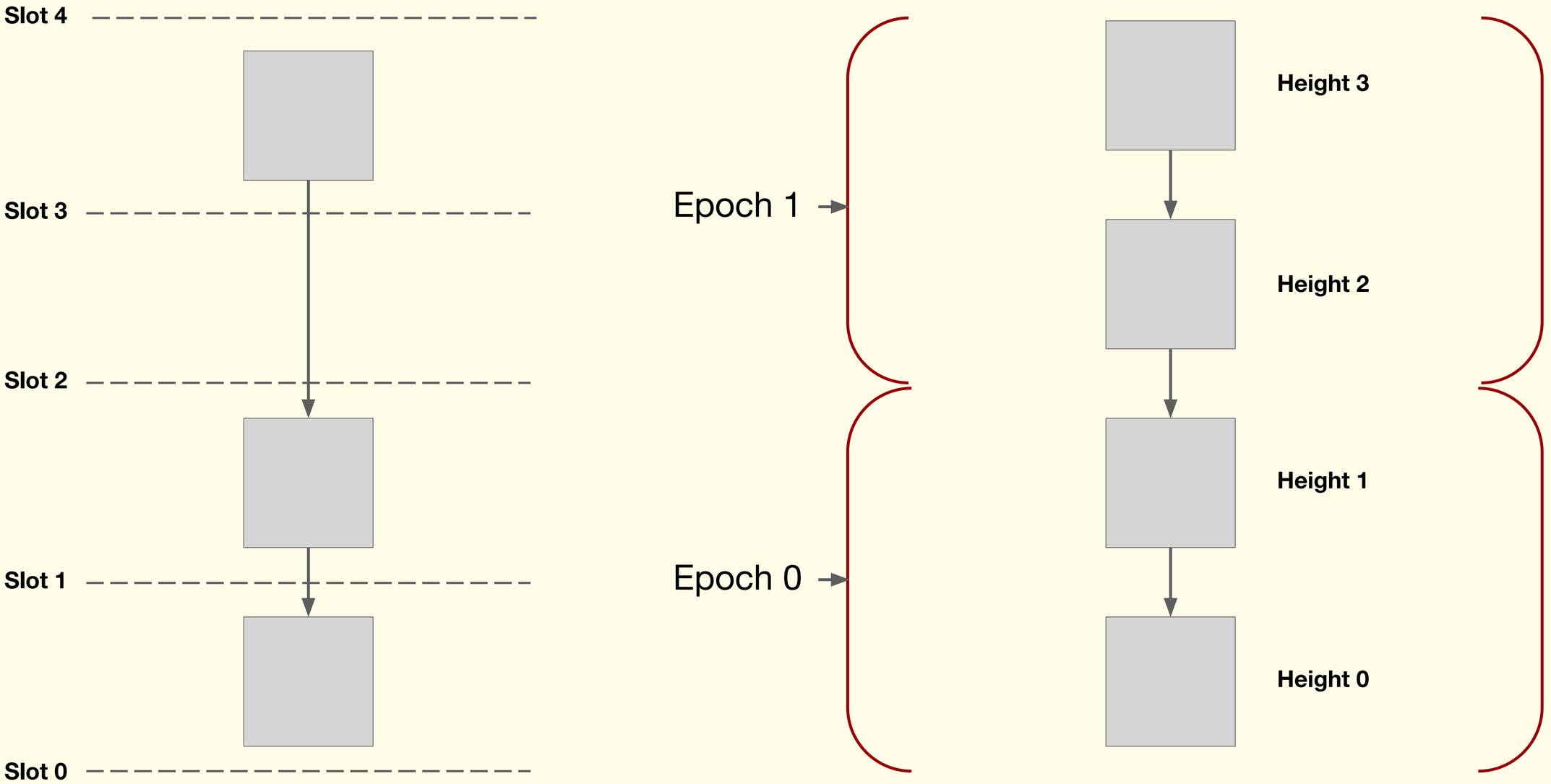
FFG applied to PoS

Slots vs Blockheight

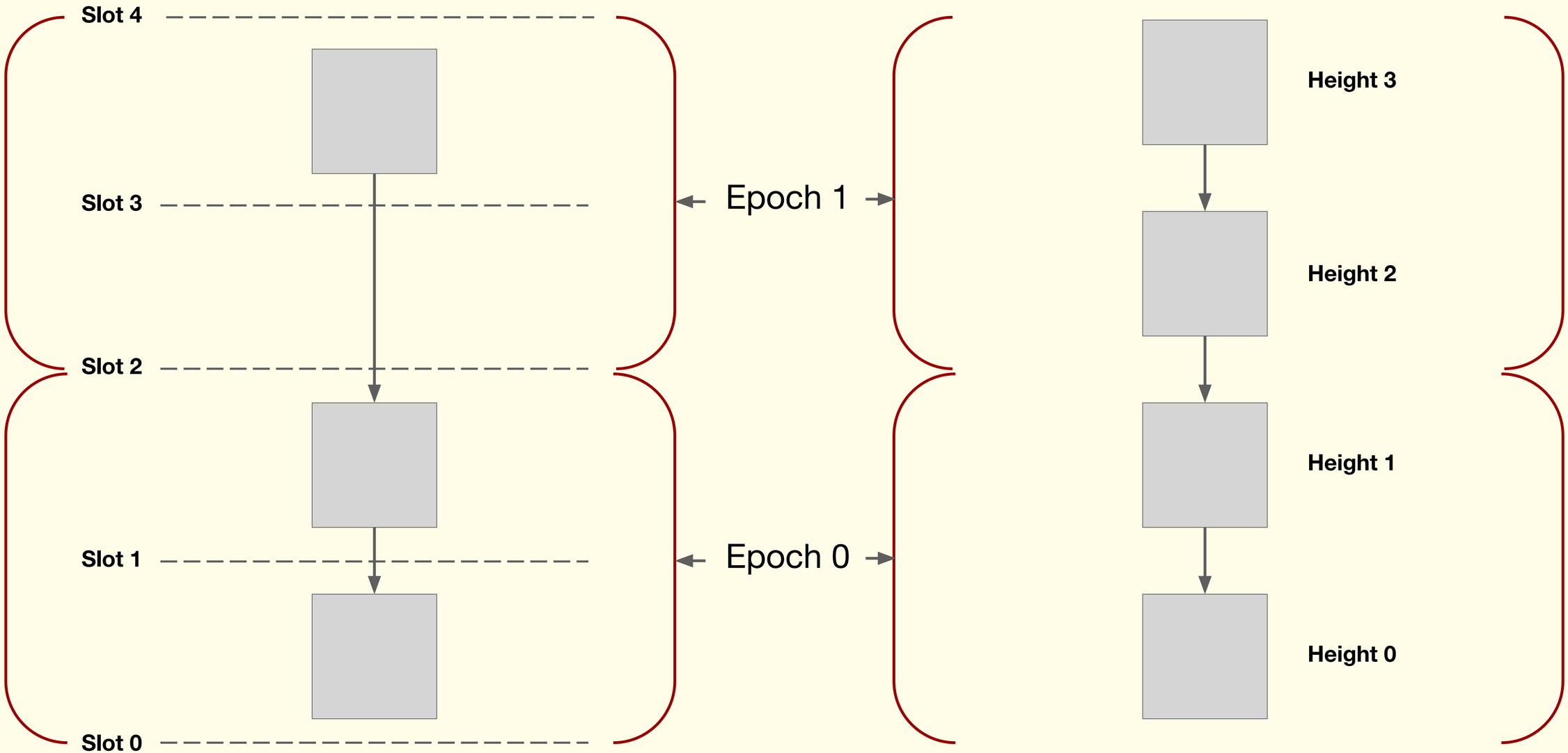




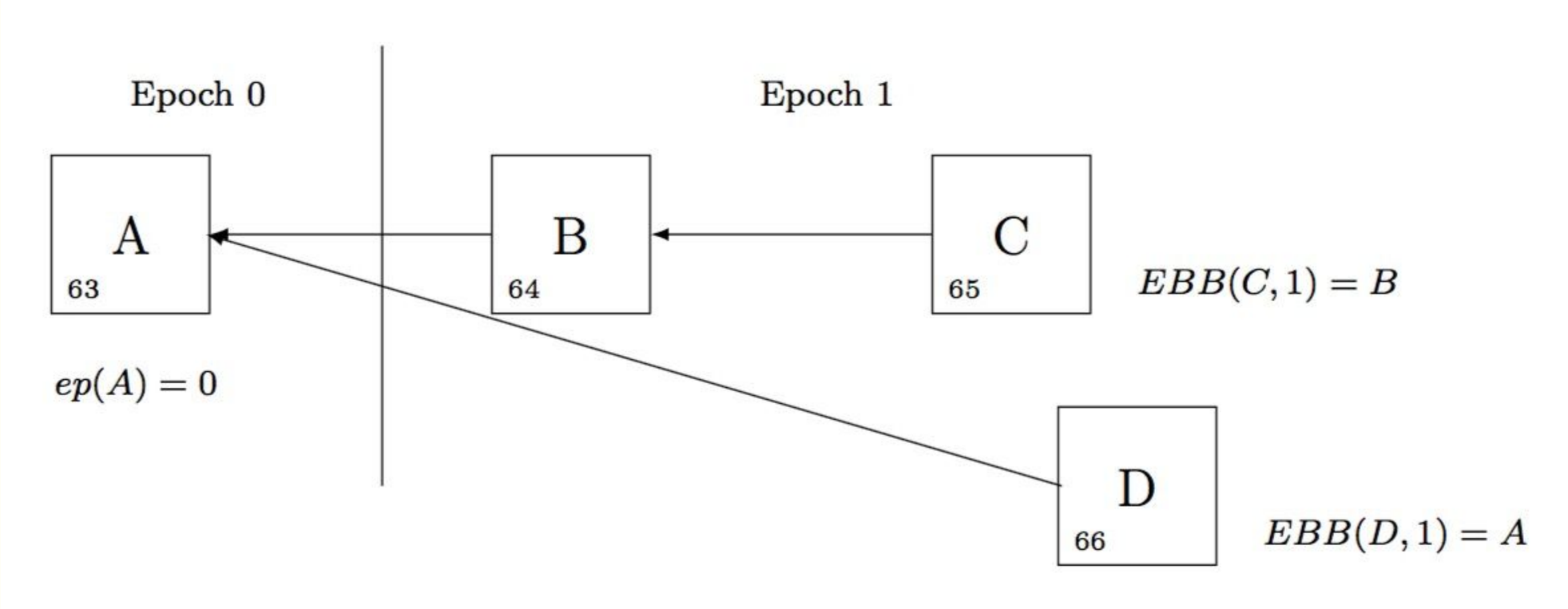
Slots vs Blockheight



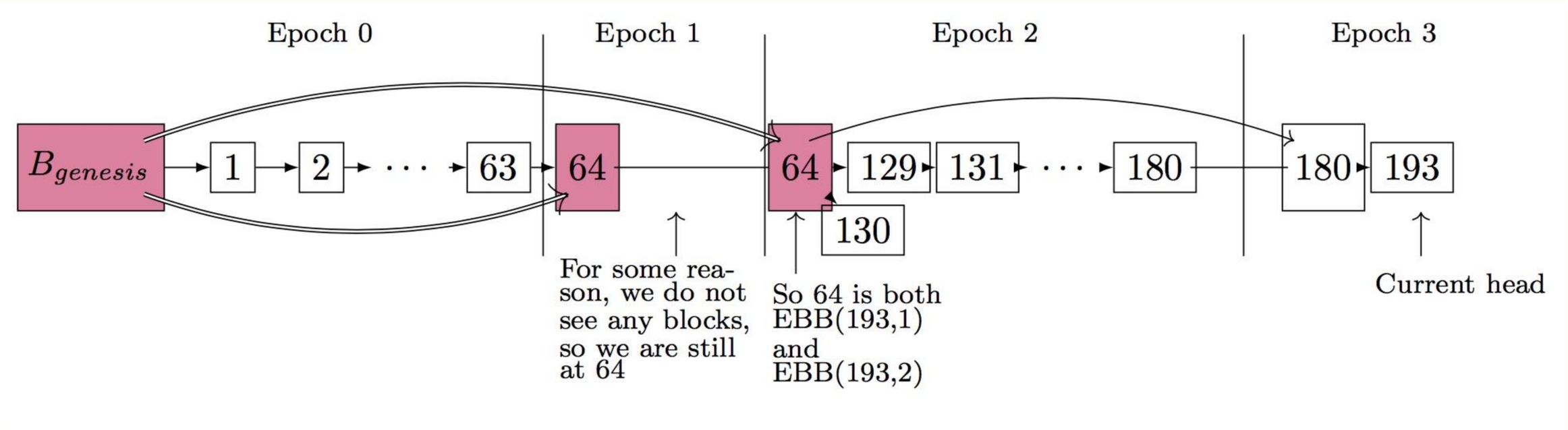
Slots vs Blockheight



Checkpoints



(Block, Epoch) -- Paired Justification



Justification Rules

Definition 4.2. We say that there is a *supermajority link* from pair (A, j') to pair (B, j) if the attestations with checkpoint edge $(A, j') \overset{\mathsf{V}}{\to} (B, j)$ have total stake more than $\frac{2N}{3}$. In this case, we write $(A, j') \overset{\mathsf{J}}{\to} (B, j)$.

Given a view G, a block B, and an epoch j, we say that the pair (B, j) is *justified in* G, or equivalently that B *is justified in* G *during epoch* j, if:

- $B = B_{\text{genesis}}$, $j = 0$; or
- for some justified pair (A, j'), $(A, j') \overset{\mathsf{J}}{\to} (B, j)$.

We use $J(G)$ to denote the set of justified pairs in G.

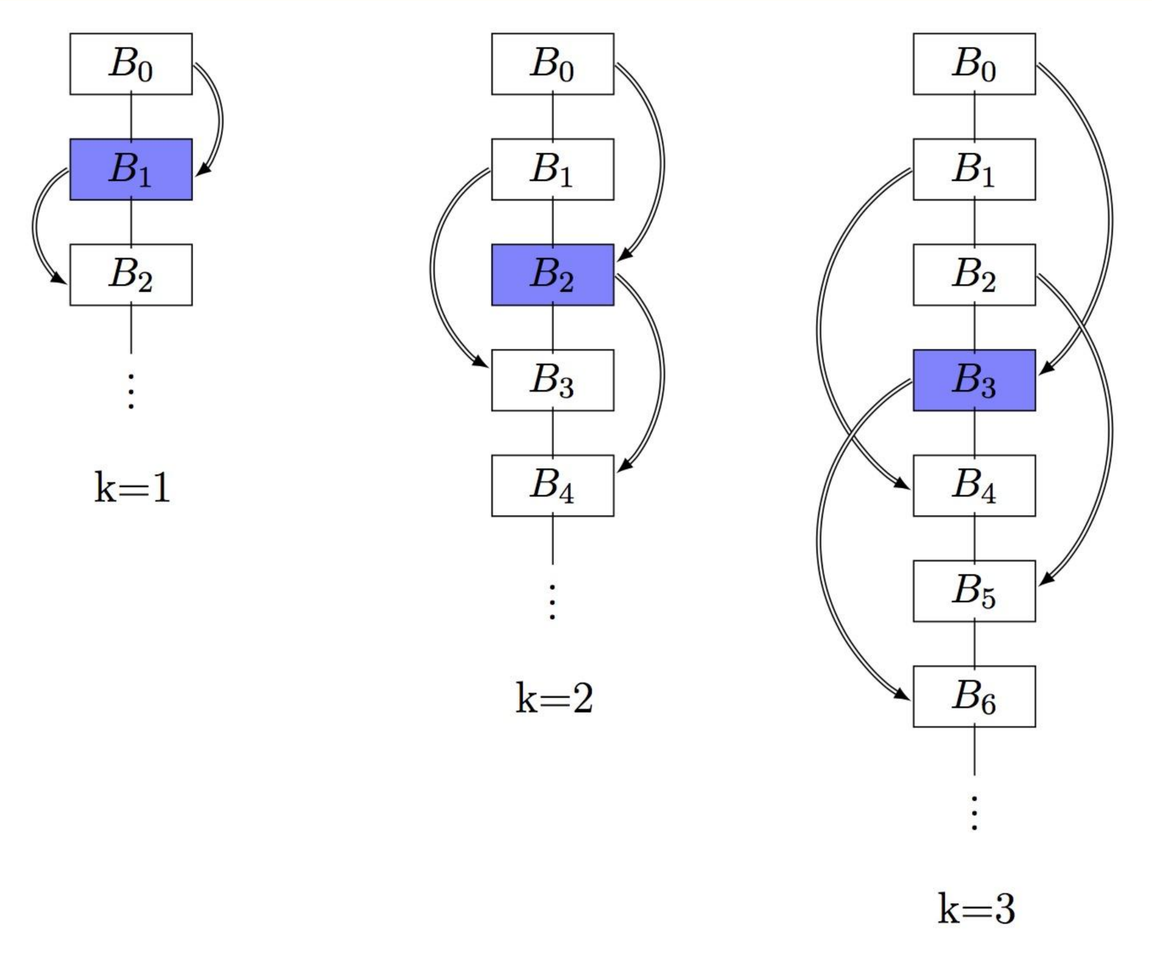
Finalization Rules

Definition 4.4. For a view G, we say that (B_0, j) is *finalized*[6] if there is an integer $k \geq 1$ and blocks $B_1, \ldots, B_k$ such that:

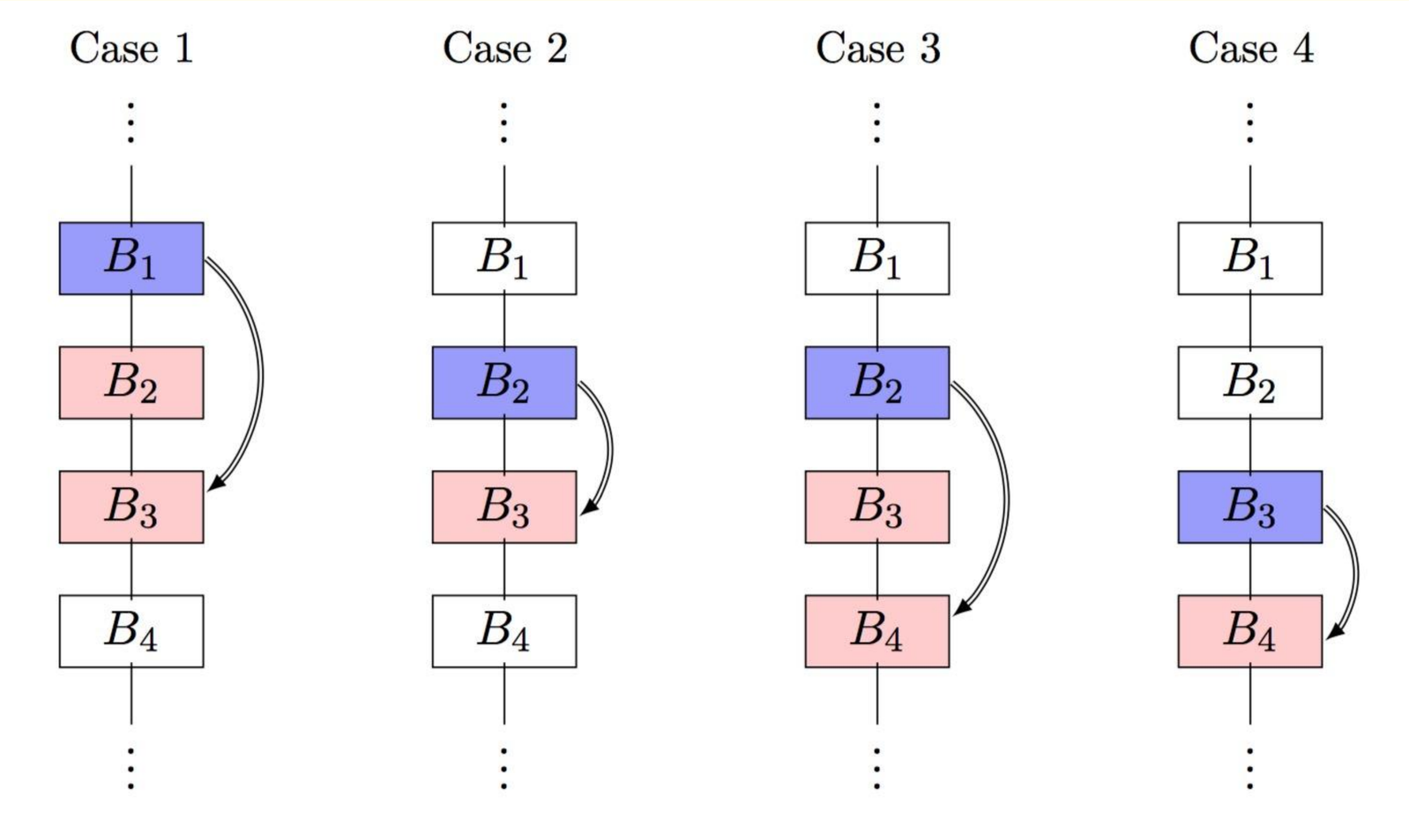
- $(B_0, j), (B_1, j+1), \ldots, (B_k, j+k)$ are adjacent epoch boundary pairs in $\mathsf{chain}(B_k)$;
- $(B_0, j), (B_1, j+1), \ldots, (B_k, j+k)$ are all in $J(G)$;
- $(B_0, j) \xrightarrow{\mathsf{J}} (B_k, j+k)$.

We define $F(G)$ to be the set of finalized pairs in the view G; we also say that a block B is *finalized* if $(B, j) \in F(G)$ for any epoch j.

k-finality rule



finality cases in eth2 (k=2)



Finality related fields in state

BeaconState

```
class BeaconState(Container):
    .
    .
    .
    # Finality
    justification_bits: Bitvector[JUSTIFICATION_BITS_LENGTH]  # Bit set for every recent justified epoch
    previous_justified_checkpoint: Checkpoint  # Previous epoch snapshot
    current_justified_checkpoint: Checkpoint
    finalized_checkpoint: Checkpoint
```

```
class Checkpoint(Container):
    epoch: Epoch
    root: Hash
```

```
def process_justification_and_finalization(state: BeaconState) -> None:
    if get_current_epoch(state) <= GENESIS_EPOCH + 1:
        return

    previous_epoch = get_previous_epoch(state)
    current_epoch = get_current_epoch(state)
    old_previous_justified_checkpoint = state.previous_justified_checkpoint
    old_current_justified_checkpoint = state.current_justified_checkpoint

    # Process justifications
    state.previous_justified_checkpoint = state.current_justified_checkpoint
    state.justification_bits[1:] = state.justification_bits[:-1]
    state.justification_bits[0] = 0b0
    matching_target_attestations = get_matching_target_attestations(state, previous_epoch)  # Previous e
    if get_attesting_balance(state, matching_target_attestations) * 3 >= get_total_active_balance(state)
        state.current_justified_checkpoint = Checkpoint(epoch=previous_epoch,
                                                        root=get_block_root(state, previous_epoch))
        state.justification_bits[1] = 0b1
    matching_target_attestations = get_matching_target_attestations(state, current_epoch)  # Current epo
    if get_attesting_balance(state, matching_target_attestations) * 3 >= get_total_active_balance(state)
        state.current_justified_checkpoint = Checkpoint(epoch=current_epoch,
                                                        root=get_block_root(state, current_epoch))
        state.justification_bits[0] = 0b1

    # Process finalizations
    bits = state.justification_bits
    # The 2nd/3rd/4th most recent epochs are justified, the 2nd using the 4th as source
    if all(bits[1:4]) and old_previous_justified_checkpoint.epoch + 3 == current_epoch:
        state.finalized_checkpoint = old_previous_justified_checkpoint
    # The 2nd/3rd most recent epochs are justified, the 2nd using the 3rd as source
    if all(bits[1:3]) and old_previous_justified_checkpoint.epoch + 2 == current_epoch:
        state.finalized_checkpoint = old_previous_justified_checkpoint
    # The 1st/2nd/3rd most recent epochs are justified, the 1st using the 3rd as source
    if all(bits[0:3]) and old_current_justified_checkpoint.epoch + 2 == current_epoch:
        state.finalized_checkpoint = old_current_justified_checkpoint
    # The 1st/2nd most recent epochs are justified, the 1st using the 2nd as source
    if all(bits[0:2]) and old_current_justified_checkpoint.epoch + 1 == current_epoch:
        state.finalized_checkpoint = old_current_justified_checkpoint
```

Slashing Conditions

Definition 4.5. We define the following *slashing conditions*:

(S1) No validator makes two distinct attestations α_1, α_2 with $\mathsf{aep}(\mathsf{LE}(\alpha_1)) = \mathsf{aep}(\mathsf{LE}(\alpha_2))$. Note this condition is equivalent to $\mathsf{ep}(\alpha_1) = \mathsf{ep}(\alpha_2)$.

(S2) No validator makes two distinct attestations α_1, α_2 with

$$\mathsf{aep}(\mathsf{LJ}(\alpha_1)) < \mathsf{aep}(\mathsf{LJ}(\alpha_2)) < \mathsf{aep}(\mathsf{LE}(\alpha_2)) < \mathsf{aep}(\mathsf{LE}(\alpha_1)).$$

Slashing Conditions

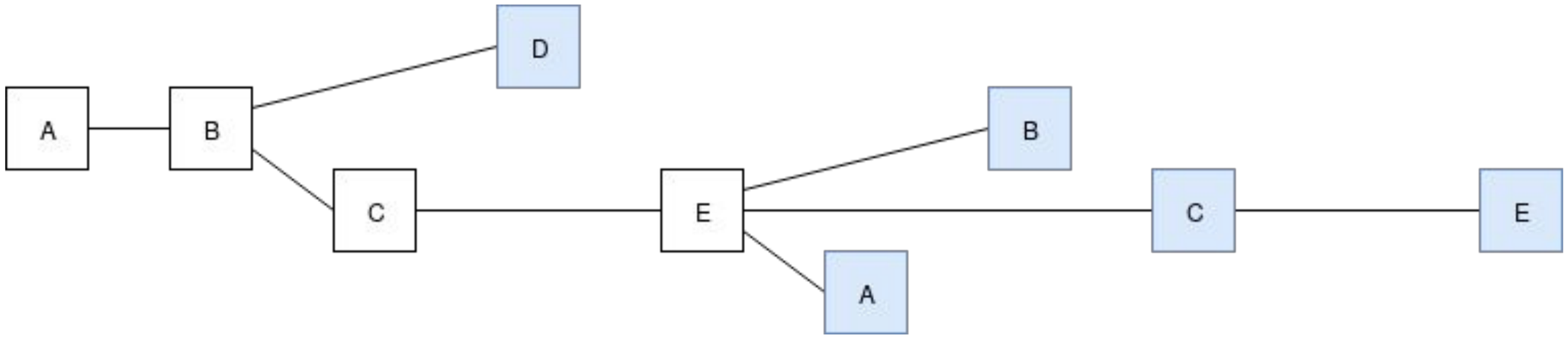
is_slashable_attestation_data

```python
def is_slashable_attestation_data(data_1: AttestationData, data_2: AttestationData) -> bool:
    """
    Check if ``data_1`` and ``data_2`` are slashable according to Casper FFG rules.
    """
    return (
        # Double vote
        (data_1 != data_2 and data_1.target.epoch == data_2.target.epoch) or
        # Surround vote
        (data_1.source.epoch < data_2.source.epoch and data_2.target.epoch < data_1.target.epoch)
    )
```

What is LMD GHOST?

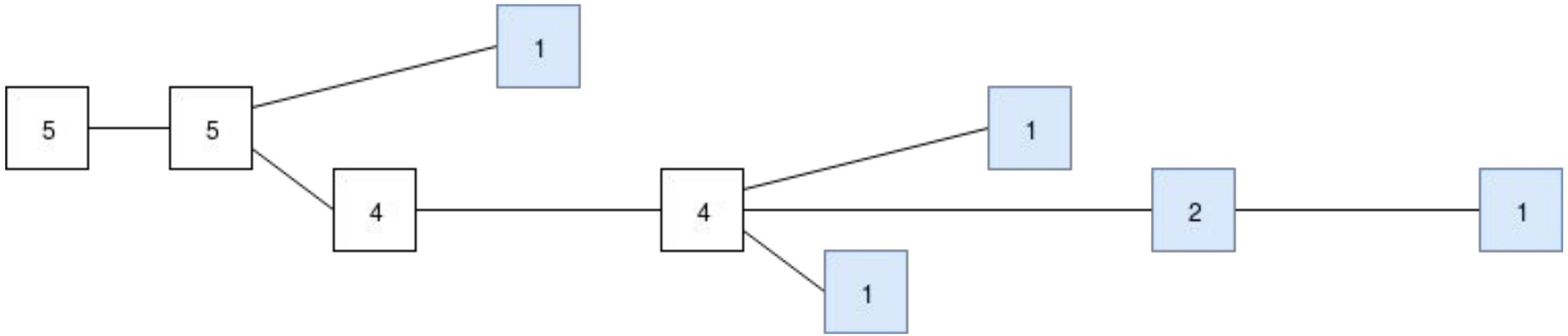
LMD GHOST fork choice rule

- Adaptation of GHOST (Sompolinsky et al, 2013) to a proof of stake context
- Idea: look only at latest messages of each node as evidence of what chain that node favors
- Assume for now “simple model” where blocks and messages are the same thing

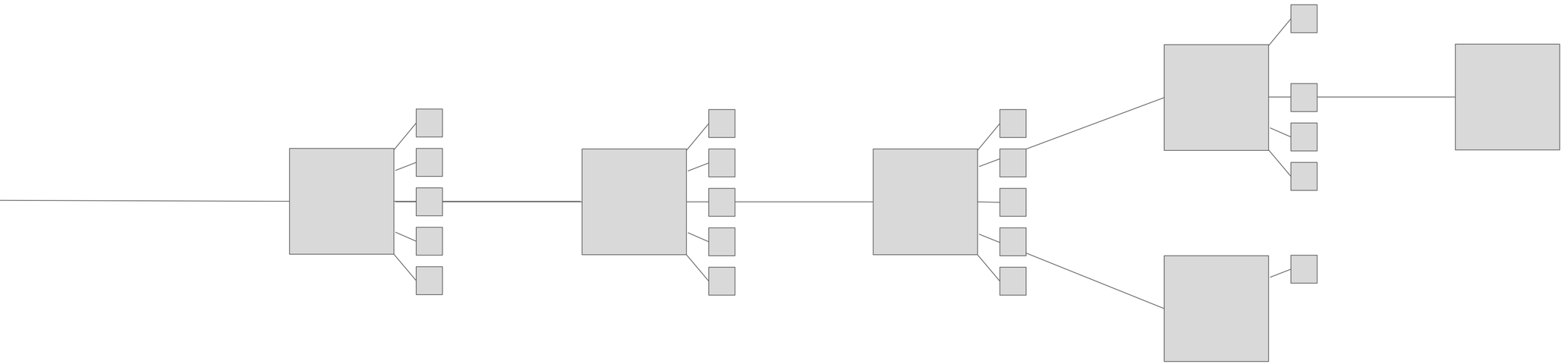


LMD GHOST fork choice rule

- As in GHOST, start from genesis, walk up the tree, at each branch choose the child that has more latest-messages supporting it (directly or indirectly)

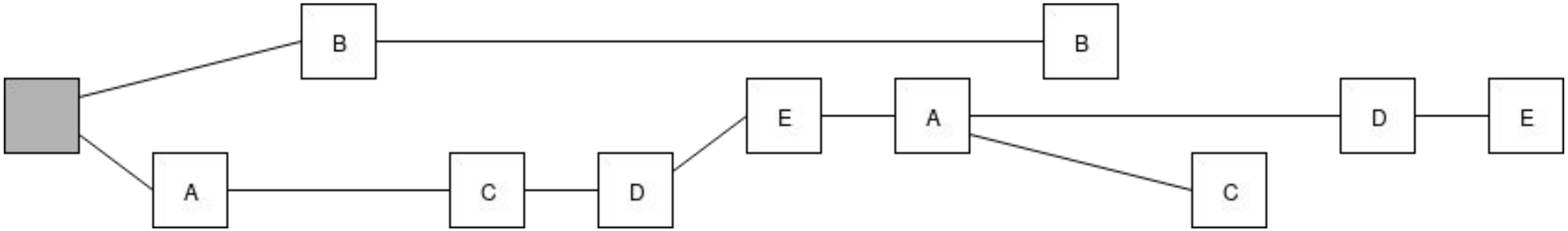


Why LMD GHOST: Parallel confirmations



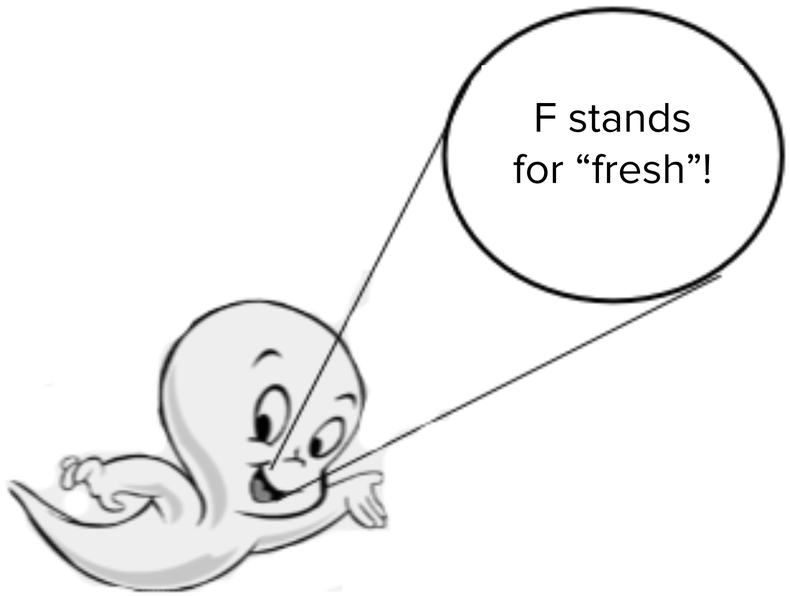
Why LMD GHOST?

- Longest chain rules cannot take into account info from frequent parallel attestations
- Minority can never “beat” majority, regardless of how many messages they send
- This property is useful for CBC Casper



Saved message attacks

- A validator can make max one vote per epoch (if more they get slashed)
- Problem: a validator can not vote for N epochs, then suddenly make N votes all at once



- Worst case: traditional GHOST (can influence fork choice by N)
- Middle case: LMD GHOST (can influence fork choice by 1 but go back-and-forth N times; can allow some anti-liveness attacks)
- Best case: **F**MD GHOST (clients only look at messages tagged with the current or previous epoch; prevents saving more than 1)

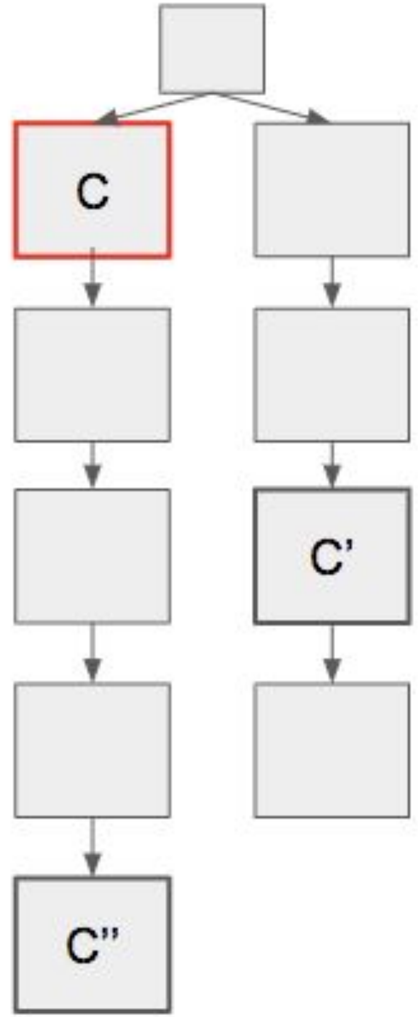
Interaction between LMD GHOST and FFG

The “real” fork choice rule:

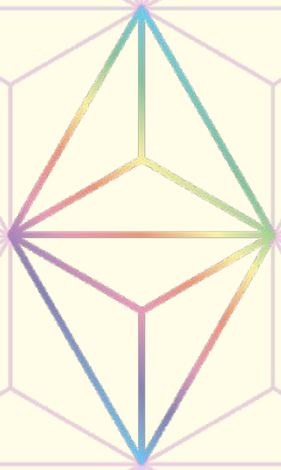
- First select the last finalized block $\mathbf{B_1}$ you were aware of (maybe the genesis)
- Then select the highest-epoch justified block $\mathbf{B_2}$ that is a descendant of $\mathbf{B_1}$
- Then use LMD GHOST starting from $\mathbf{B_2}$ to find the head

Interaction between LMD GHOST and FFG

- This does open room for some kinds of “bounce” attacks
- General solution: if a client sees a checkpoint that is NOT in the current canonical chain become justified, it delays when it switches chains
  - Idea 1: usually switch only on epoch boundaries (credit Ryuya Nakamura)
  - Idea 2: $h(B_1) + N$’th checkpoint can be used only from height $h(B_1) + 3N$; this ensures that there are periods of 3 epochs in which to finalize a new checkpoint

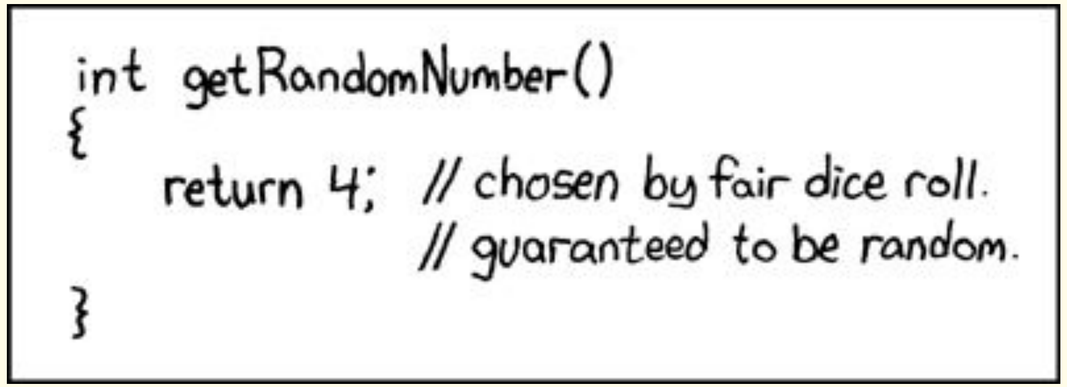


Randomness in Ethereum 2.0



Dankrad Feist
Ethereum Foundation (Eth2.0 Research)

Randomness in Ethereum 2.0



- Why randomness is critical in Proof of Stake
- RANDAO
- RANDAO issues
- Randomness using verifiable delay functions

Randomness in Proof of Stake

- Select proposers
- Select committees
- Provide randomness on-chain

Why is unpredictable, unbiased randomness required?

- **Select proposers**
  - Fairness
  - Protection against DOS attacks
- **Select committees**
  - Need honest committees or else will suffer from too many fraud proofs
- **Provide randomness on-chain**
  - Applications might attach huge value to random numbers
  - If attacks are possible, this will degrade the randomness for everyone

Why is unpredictable, unbiased randomness required?

- **Select committees**
  - We want to minimize the probability for dishonest committees

- **Crosslinks** to shards in eth2.0 are made based on crosslink committee votes
- A bad committee can create a link to an invalid or non-existent block
  - Fraud proof potentially means reverting beacon chain
- E.g. probability of bad committee of size 128 is $5*10^{-15}$ (⅔ honesty assumption)
  - But what if attacker can bias committee sampling?

Idea

n people want to generate a random number

- They all go into a room
- Each person contributes one value x_i
- Compute $\mathrm{XOR}(x_1, x_2, \ldots)$

Problem: The last player can just change their value after having seen the other choices, until they get a favourable output (grinding)

Commit-reveal — basis for RANDAO

n people want to generate a random number

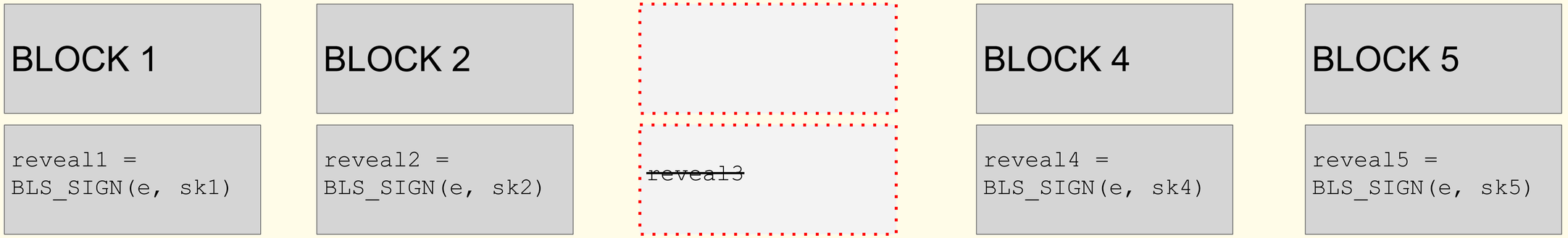
- They all go into a room
- Each person commits to one value x_i, commits by telling everyone hash(x_i)
- Compute `XOR(`x_1`, `x_2`, …)`

Nice! This cannot be manipulated

But: Anyone can stop the process by not revealing their preimage x_i

RANDAO in practice

e = epoch




```
mix = hash(hash(reveal1) XOR hash(reveal2)
      ~~XOR hash(reveal3)~~ XOR hash(reveal4)
      XOR hash(reveal5) …)
```

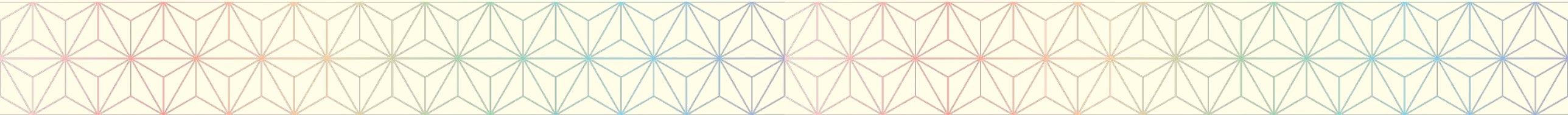
Problems with RANDAO

- Last revealer can choose to not produce a block
  - “Get another roll if I don’t like the result”
  - If I control validators with several slots in a row, more bias is possible
- A chain that uses RANDAO + Longest chain as fork choice can be completely taken over by an attacker with 36% stake (ethresear.ch analysis by Vitalik)

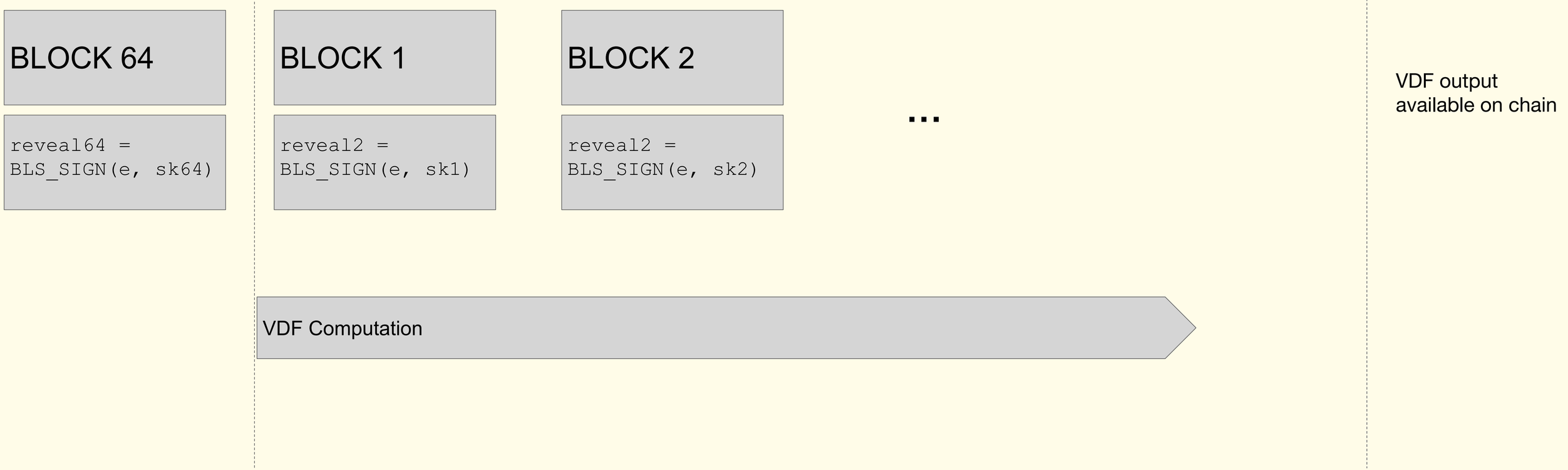
Outlook: Verifiable delay functions

- Verifiable delay function (VDF): A function `y, π = f(x)`, such that
  - Computing `f(x)` takes a long serial time
  - Checking the result y using the proof `π` is fast
- Example: Computing `x^(2^T) mod m` where `m = p*q` is an RSA modulus of unknown factorization

By using the VDF output on RANDAO, the last revealer loses the advantage



Outlook: Verifiable delay functions



Last revealer has to make decision before being able to know VDF result

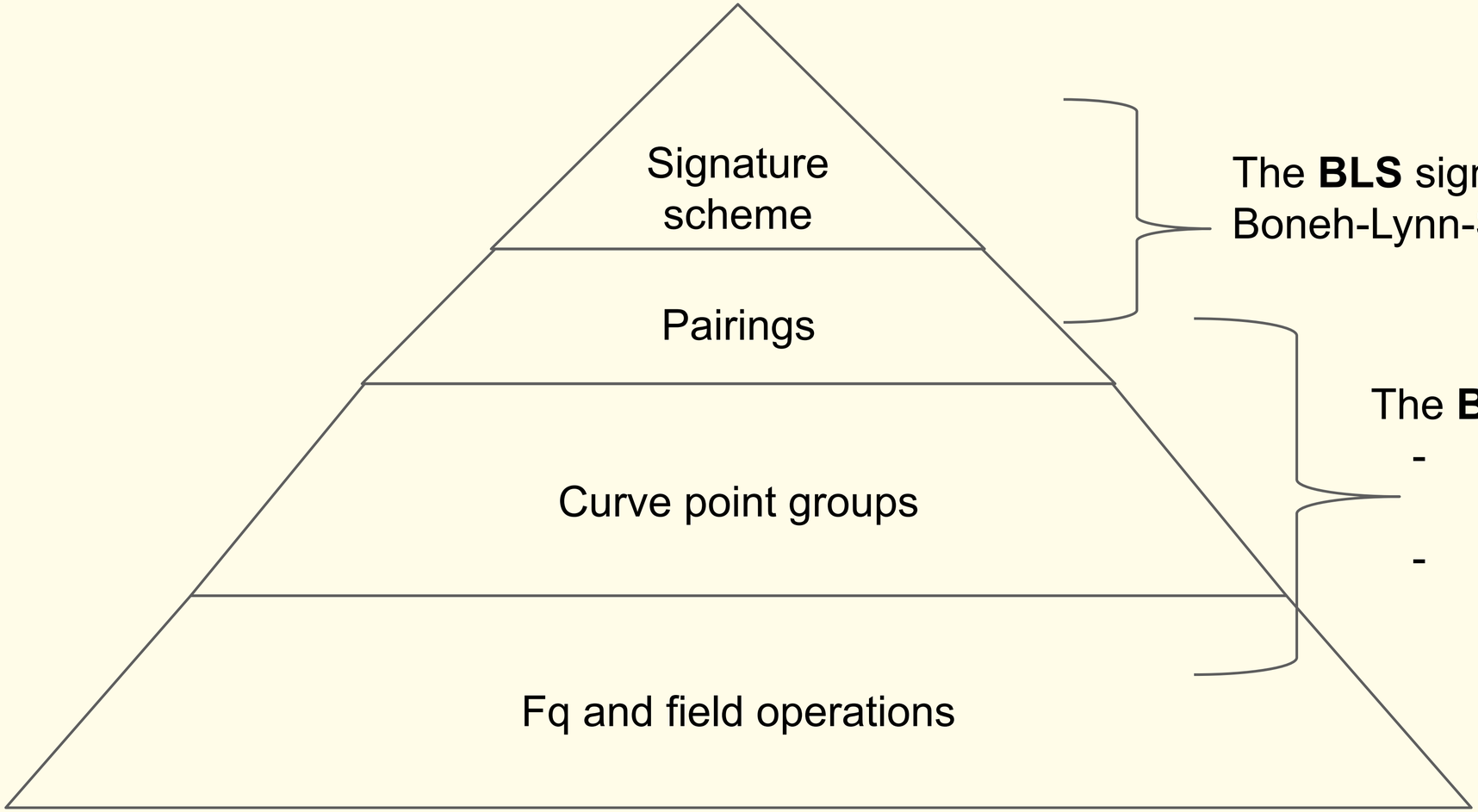
YOU FOUND A STAR

Speaker Track

dropparty.tech

BLS Signatures and Aggregation

Chih-Cheng Liang



The **BLS** signature aggregation scheme: Boneh-Lynn-Shacham (2003)

The **BLS12-381** Curve:

- It's a Barreto-Lynn-Scott(2002) curve
- Chosen by Zcash to have an embedding degree of 12 and a field modulus of 381 bits.

Operations of Prime Fields

Consider F_{13}, i.e. q=13

```
>>> from py_ecc.fields.field_elements import FQ
>>> class F13(FQ):
...     field_modulus = 13
...
>>> F13(10)/ F13(3)
12
```

Addition

$11 + 12 = 10 \mod 13$

Subtraction

$5 - 7 = 11 \mod 13$

Multiplication

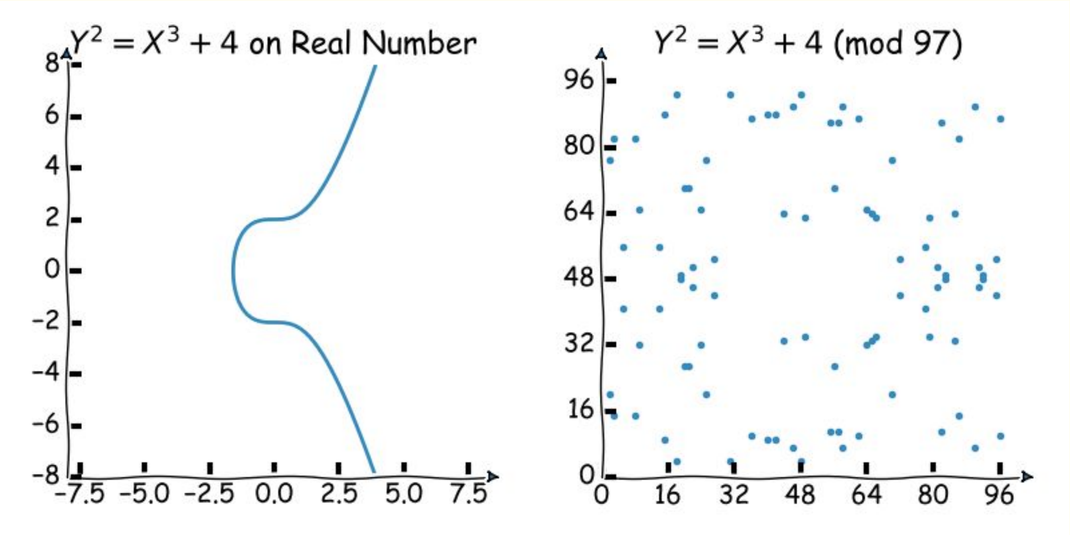
$2 * 8 = 3 \mod 13$

Division

$10 / 3 = 10 * (3)^{-1} = 10 * 3^{11} = 12 \mod 13$

A **curve** is an equation of the form:

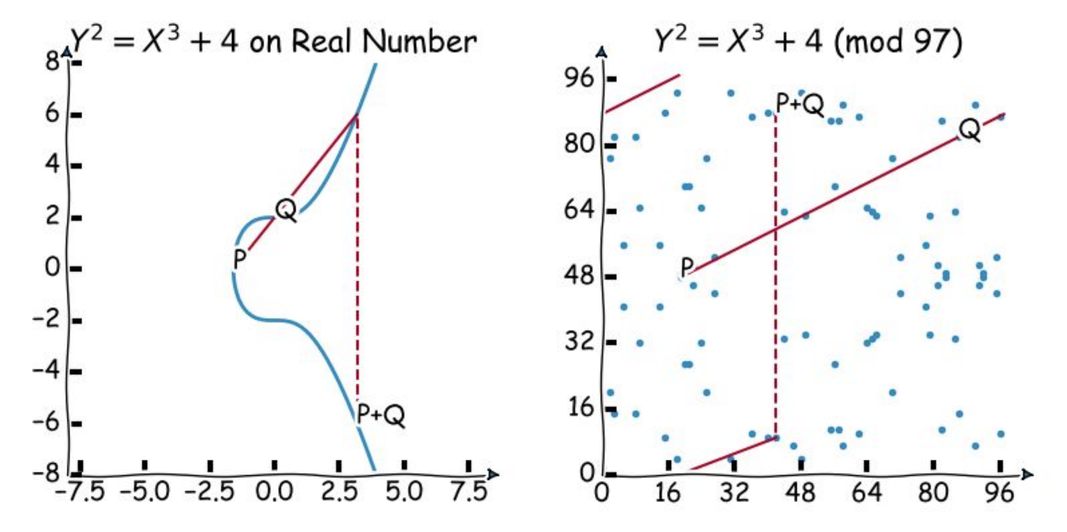
$$y^2 = x^3 + ax + b$$



A **point** on the curve has x and y coordinates. It can be *compressed* by specifying only x plus 1 bit to infer y.

P: (x, y)

You can **add** a point to another point P+Q



You can **multiply** a point with an integer

10P = P + P + P + P + P + P + P + P + P + P

When people see this...

Q := 10P

The integer can be used as a secret

They can't figure out this.

The pairing function 💸💸💸

$e: G_1 \times G_2 \rightarrow F_{q12}$

$e(aP, bQ) = e(P, Q)^{ab}$

$= e(P, abQ)$

$= e(abP, Q)$

$e(P + R, Q + S)$

$= e(P, Q)e(P, S)e(R, Q)e(R, S)$

48 bytes ✂️

96 bytes ✂️

$<G_1>$

$G_1, 2G_1, \ldots, (r-1)G_1$

$r \approx 2^{255}$

Curve point subgroup generated by the generator

$<G_2>$

$G_2, 2G_2, \ldots, (r-1)G_2$

$E(Fq) := y^2 = x^3 + 4$

Curves

$E'(F_{q^2}) := y^2 = x^3 + 4(i + 1)$

F_q 48 bytes

$q \approx 2^{381}$

Field elements

F_{q^2} 96 bytes

Private Key k 32 bytes

Public Key $P = k \cdot G_1$ 48 bytes

Sign $S = k \cdot H(m)$ 96 bytes

Verify $e(G_1, S) == e(P, H(m))$

Aggregate multiple signatures

$S_{123} = S_1 + S_2 + S_3$ $P_{123} = P_1 + P_2 + P_3$

Verify multiple signatures

$e(G_1, S_{123}) == e(P_{123}, H(m))$

Hash to G_2

$H(m)$

$$
\begin{aligned}
& e(G_1, S) \\
&= e(G_1, k \cdot H(m)) \\
&= e(k \cdot G_1, H(m)) \\
&= e(P, H(m))
\end{aligned}
$$

Attestation: An aggregation example

This is `'0b11000111110001100000001'` in binary.
The i'th position records if the i'th validator has voted.

A client uses this information to lookup the public keys of the voting validators.

```
aggregationBits: '0x63e301'
data:
  ...
signature: >-

0x63e7c50e0894483e0e7974b2a22dd5c0facb
853ae8cffcae1657301e36b55299438c760e6b
18a7e85b74ffce0697f028de9c4b633dcde21e
3a0caf3d686b98f565d2ea80a3aa63588515a8
d46e2af8a84144953dd1abe4b7f586291e1490
b1d9
```

Always 96 bytes, no matter how many votes it aggregated.

example credit: https://simpleserialize.com/

Break

Life of An Ethereum Beacon Chain Validator

Spec v.0.8.3 - 2019 August 24th

Ethereum Research

Hsiao-Wei Wang

hwwhww

icebearhww

Life of An Ethereum Beacon Chain Validator

1. Two main factors of validator status transition
    - Status epoch
    - Slashed
2. Rate-limiting entry/exits queues
3. Lifecycle

Validator Status

1. Activation Eligibility
2. Activation
3. Exit
4. Withdraw**able**

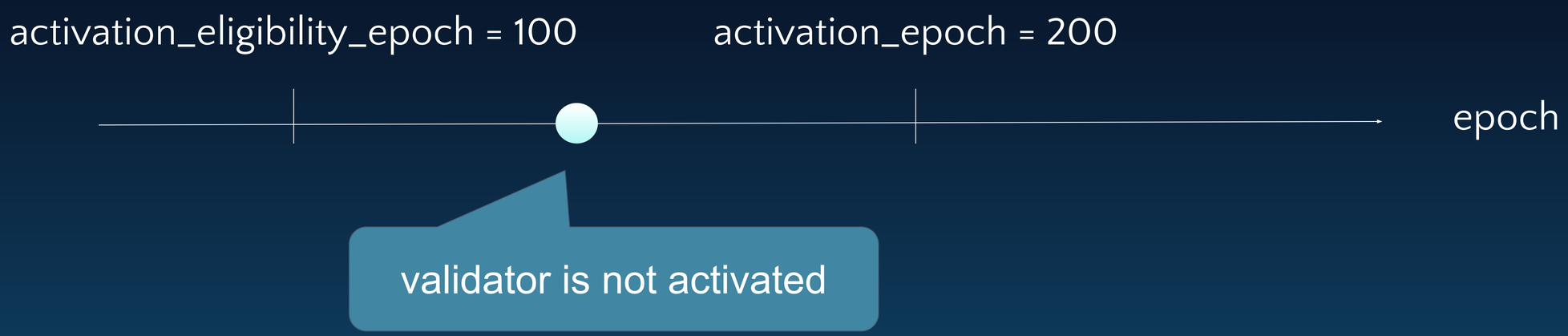
the validator will be able to withdraw to EEs in phase 2

Validator Status Epoch

```
class Validator(Container):
    pubkey: BLSPubkey
    withdrawal_credentials: Hash  # Commitment to pubkey for withdrawals and transfers
    effective_balance: Gwei  # Balance at stake
    slashed: boolean
    # Status epochs
    activation_eligibility_epoch: Epoch  # When criteria for activation were met
    activation_epoch: Epoch
    exit_epoch: Epoch
    withdrawable_epoch: Epoch  # When validator can withdraw or transfer funds
```

status epoch default value = FAR_FUTURE_EPOCH = Epoch(2**64 - 1)

Validator Status Epoch: example



Validator Status Epoch: example

`validator.slashed` Boolean Flag

```
class Validator(Container):
    pubkey: BLSPubkey
    withdrawal_credentials: Hash  # Commitment to pubkey for withdrawals and transfers
    effective_balance: Gwei  # Balance at stake
    slashed: boolean
    # Status epochs
    activation_eligibility_epoch: Epoch  # When criteria for activation were met
    activation_epoch: Epoch
    exit_epoch: Epoch
    withdrawable_epoch: Epoch  # When validator can withdraw or transfer funds
```

A slashed validator will be forced exited.

Weak Subjectivity

Only need to trust when

a. You join the first time
b. You were offline for a long time
   - How long? It depends on how long it takes for attacker to withdraw their stake.
   - “weak subjectivity period”

Rate-limiting entry/exits queues

1. Ensure the validator set is stable enough between two points
2. Ensure that **finality guarantees** still remain between two chains as long as a validator logs on often enough

Reference: https://notes.ethereum.org/@vbuterin/rkhCgQteN?type=view#Exiting

Churn rate

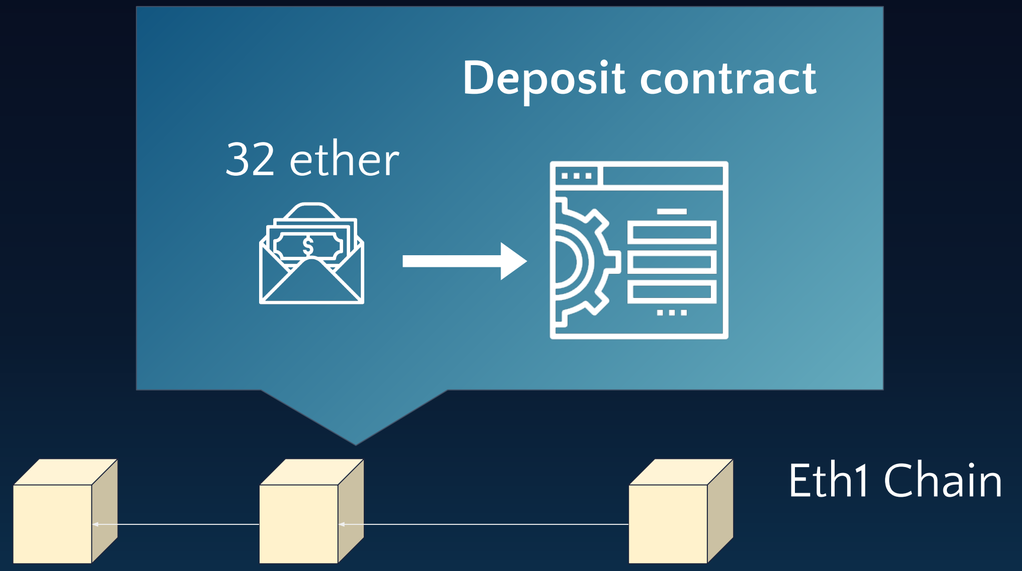
- The churn rate is based on current **active validators count**
- Maximum **`max(4, |V|/65535)`** valdators can join per epoch

```
def get_validator_churn_limit(state: BeaconState) -> uint64:
    """
    Return the validator churn limit for the current epoch.
    """
    active_validator_indices = get_active_validator_indices(state, get_current_epoch(state))
    return max(MIN_PER_EPOCH_CHURN_LIMIT, len(active_validator_indices) // CHURN_LIMIT_QUOTIENT)
```

Lifecycle

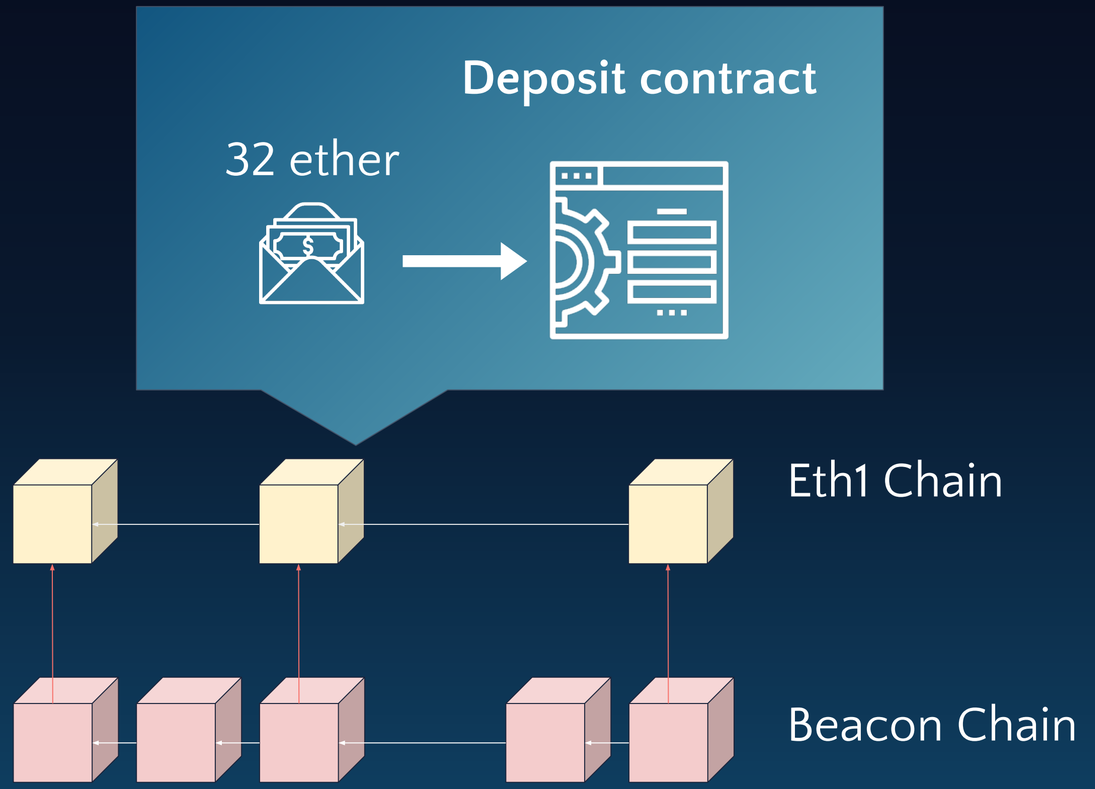
Join the Staking

1. Deposit `MAX_DEPOSIT_AMOUNT` (32 ether) to a special **deposit contract**

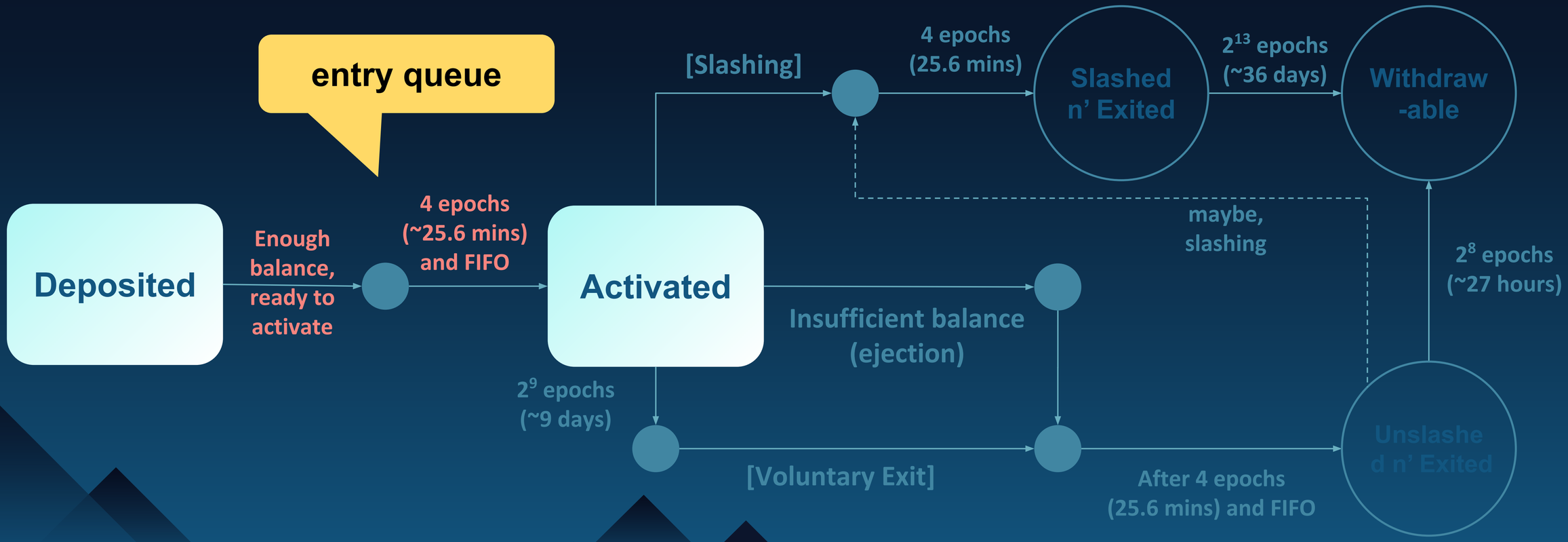


Join the Staking

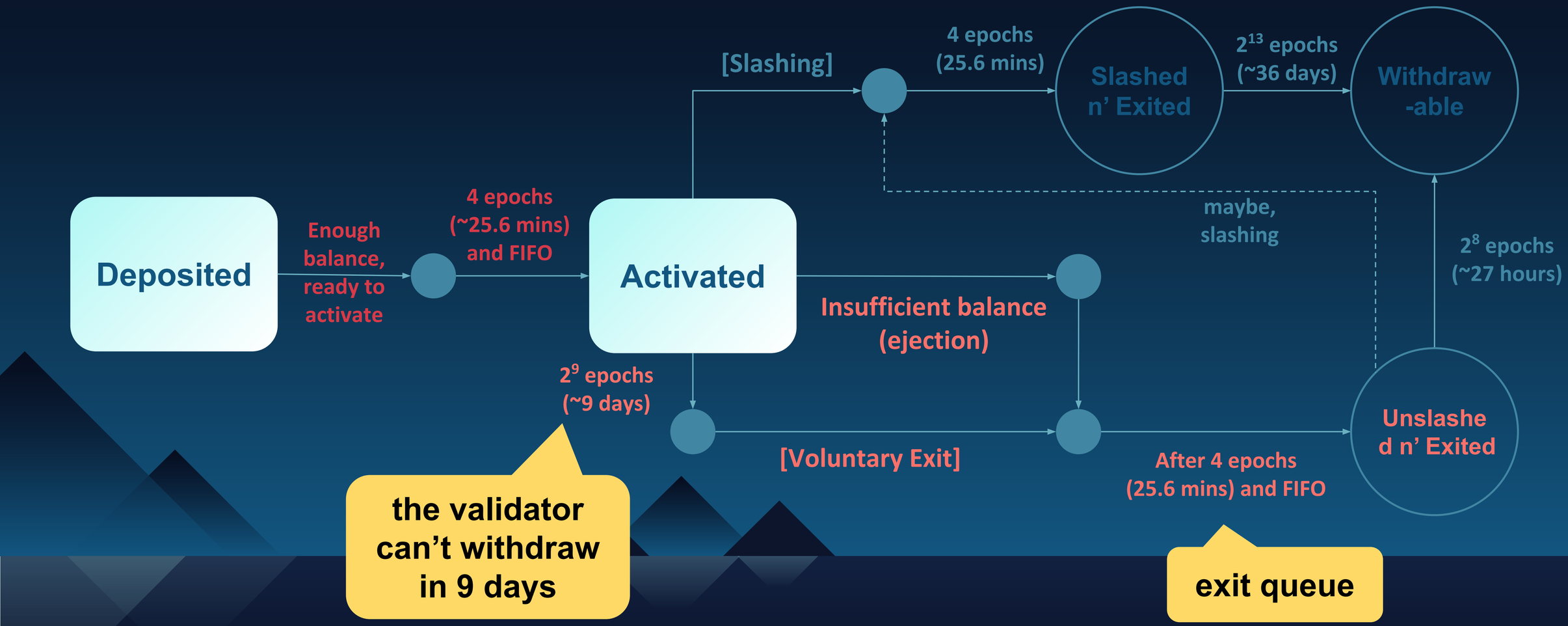
1. Deposit `MAX_DEPOSIT_AMOUNT` (32 ether) to a special **deposit contract**
2. Watch the deposit contract status



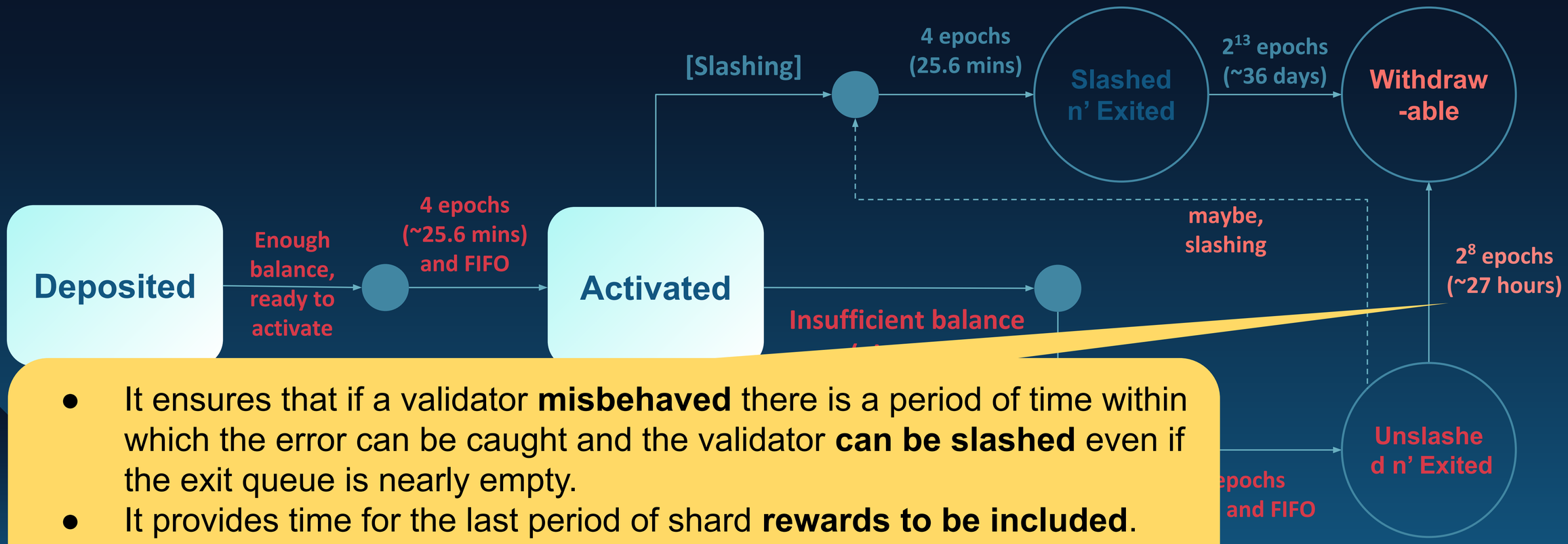
Validator Status (v0.8.3)



Validator Status (v0.8.3)

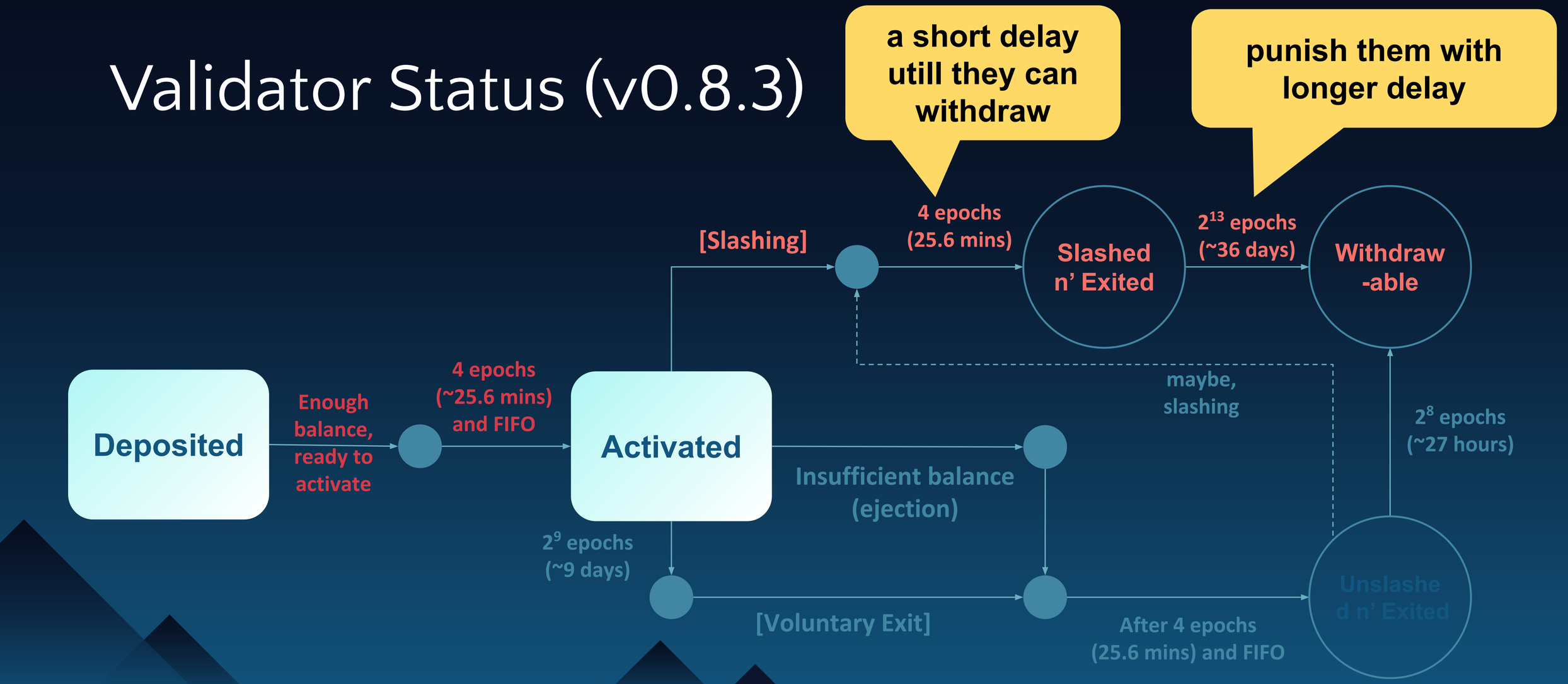


Validator Status (v0.8.3)

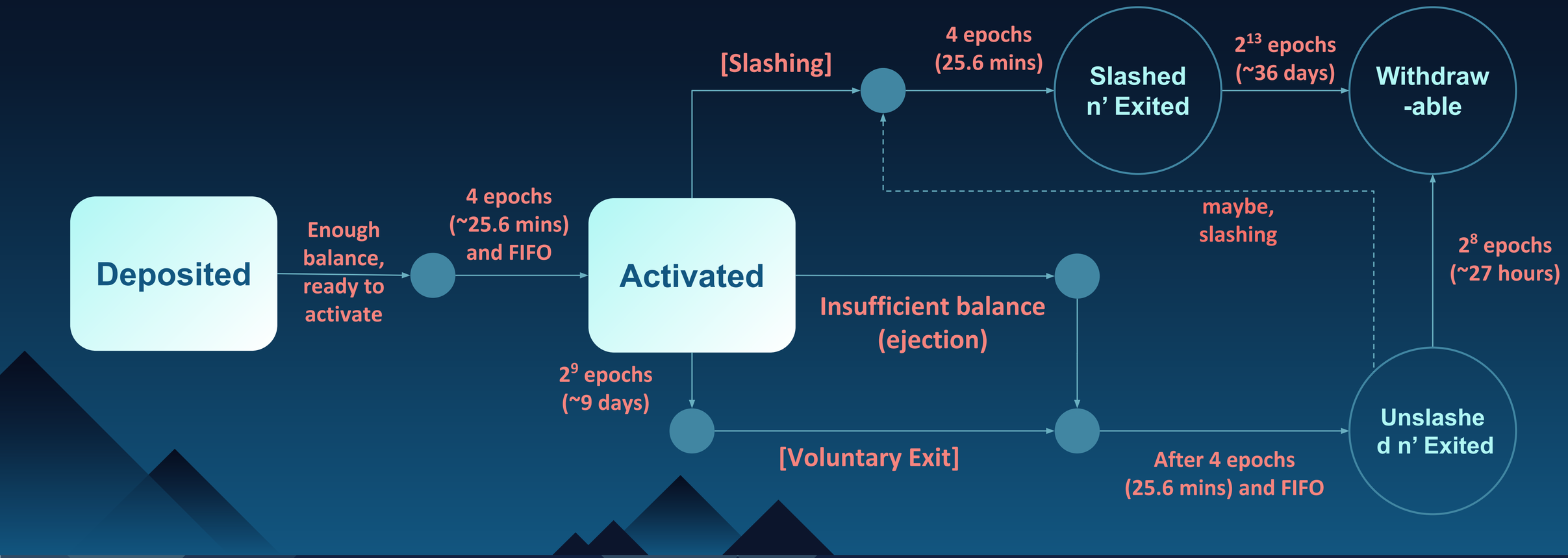


- It ensures that if a validator **misbehaved** there is a period of time within which the error can be caught and the validator **can be slashed** even if the exit queue is nearly empty.
- It provides time for the last period of shard **rewards to be included**.
- It provides **time for proof of custody challenges** to be made.

Validator Status (v0.8.3)



Validator Status (v0.8.3)



Resource credit

- Slide template: 24Slides.com
- Icons: designed by Freepik from www.flaticon.com

Thank you!

beacon chain

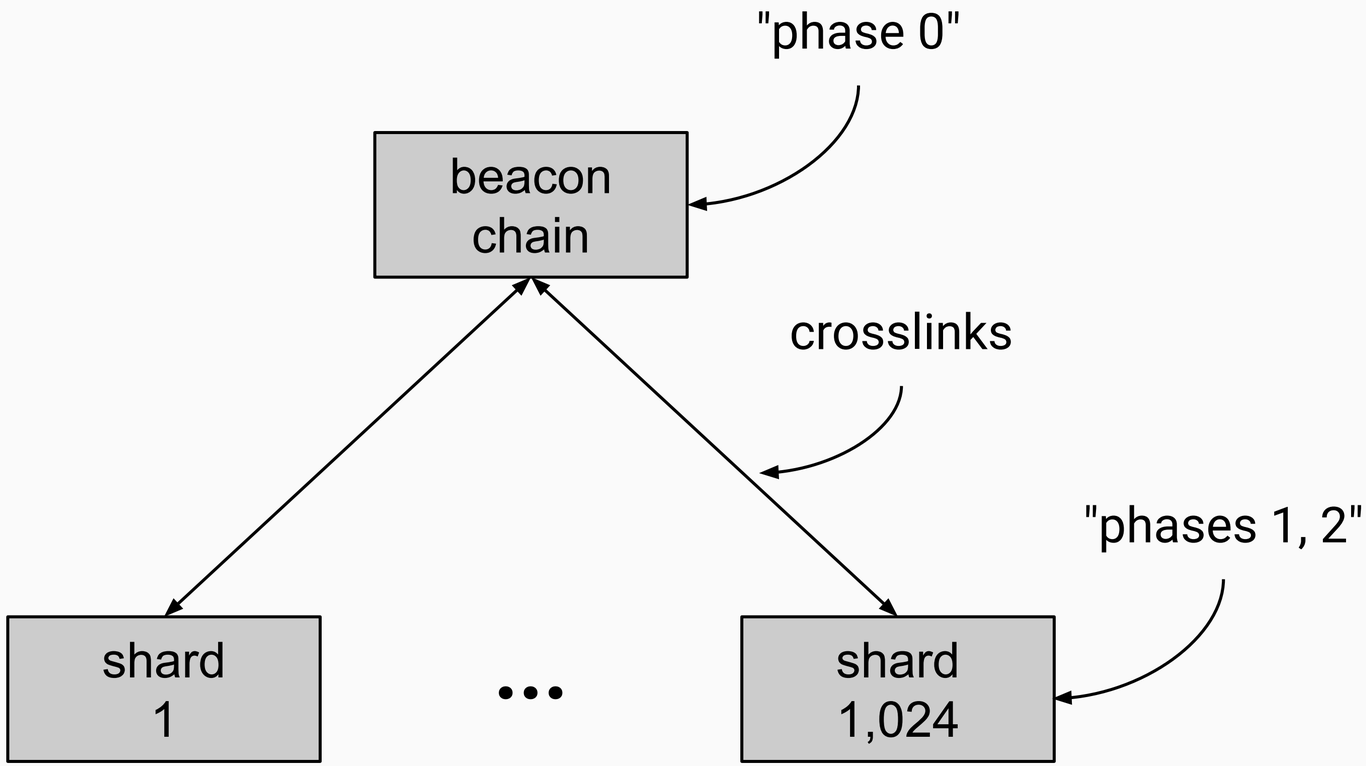
state transition function

read the specs

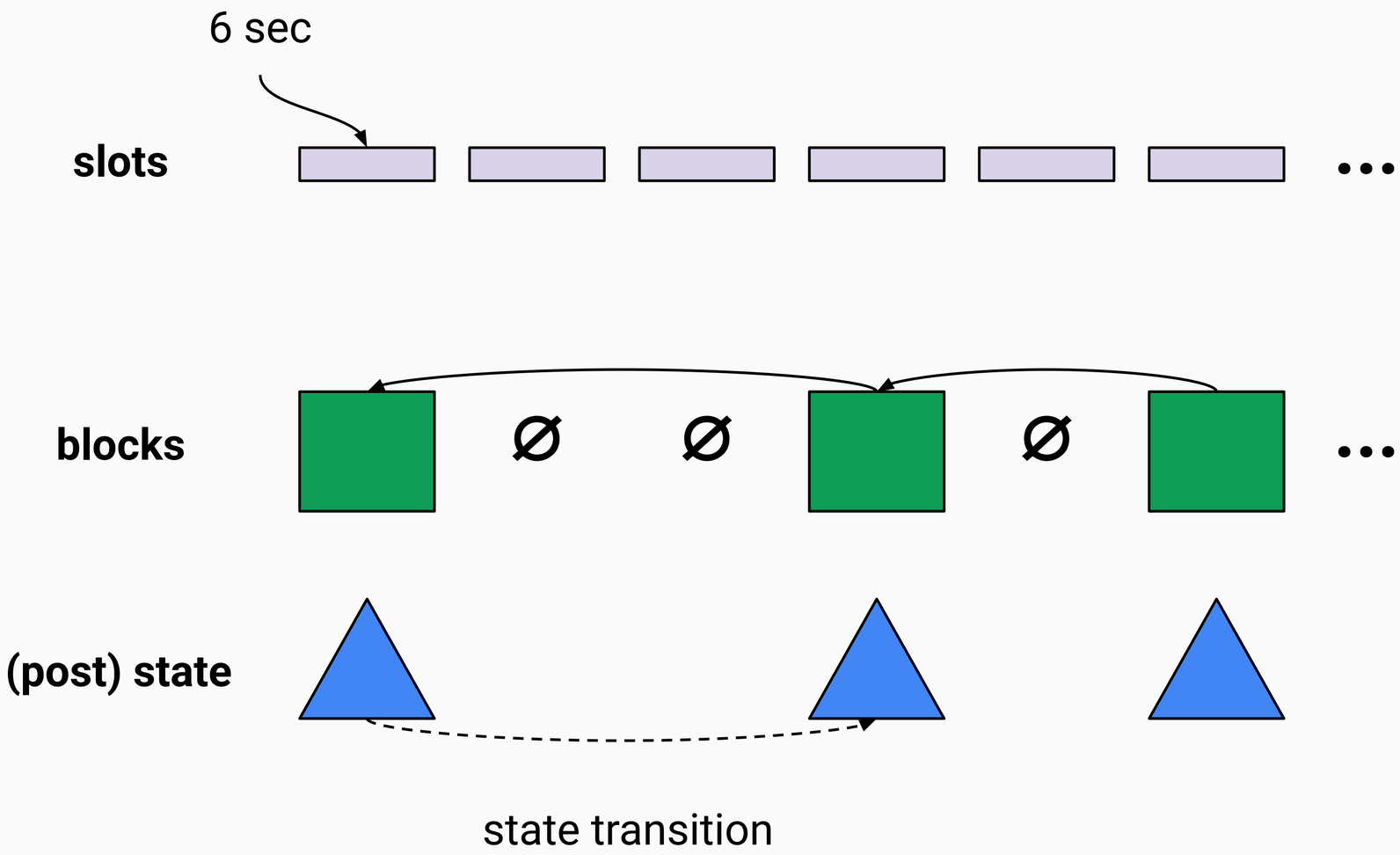
github.com/ethereum/eth2.0-specs/blob/dev/specs/core/0_beacon-chain.md

tiny.cc/beaconchain

hub-and-spoke

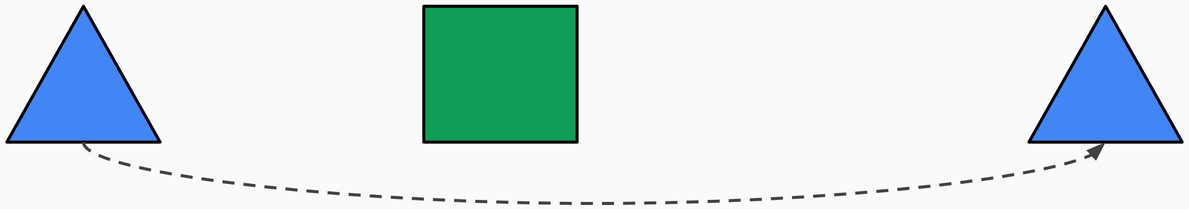


slots, blocks, state

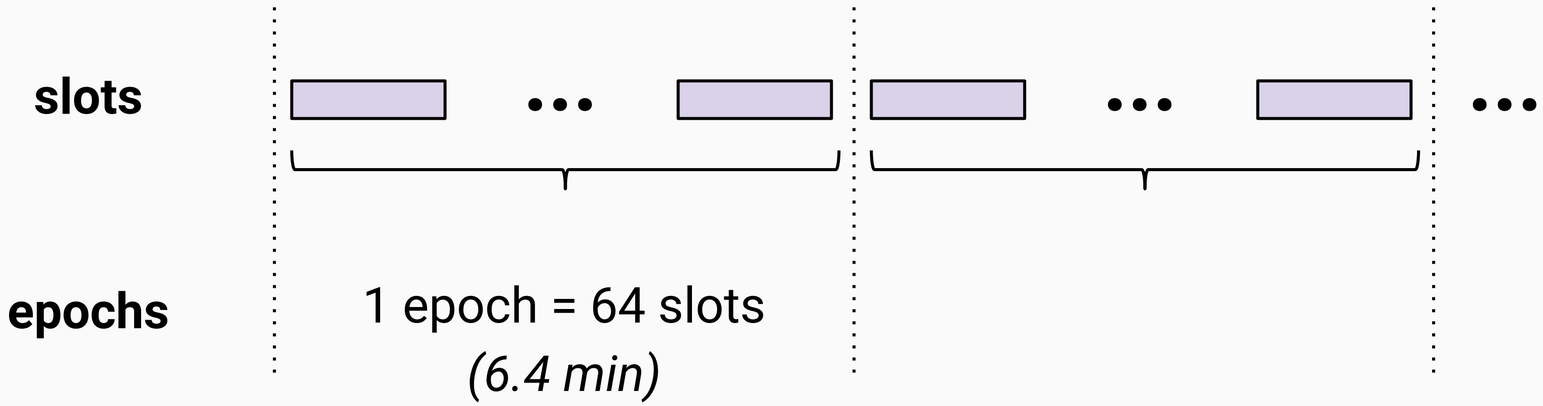


state transition function

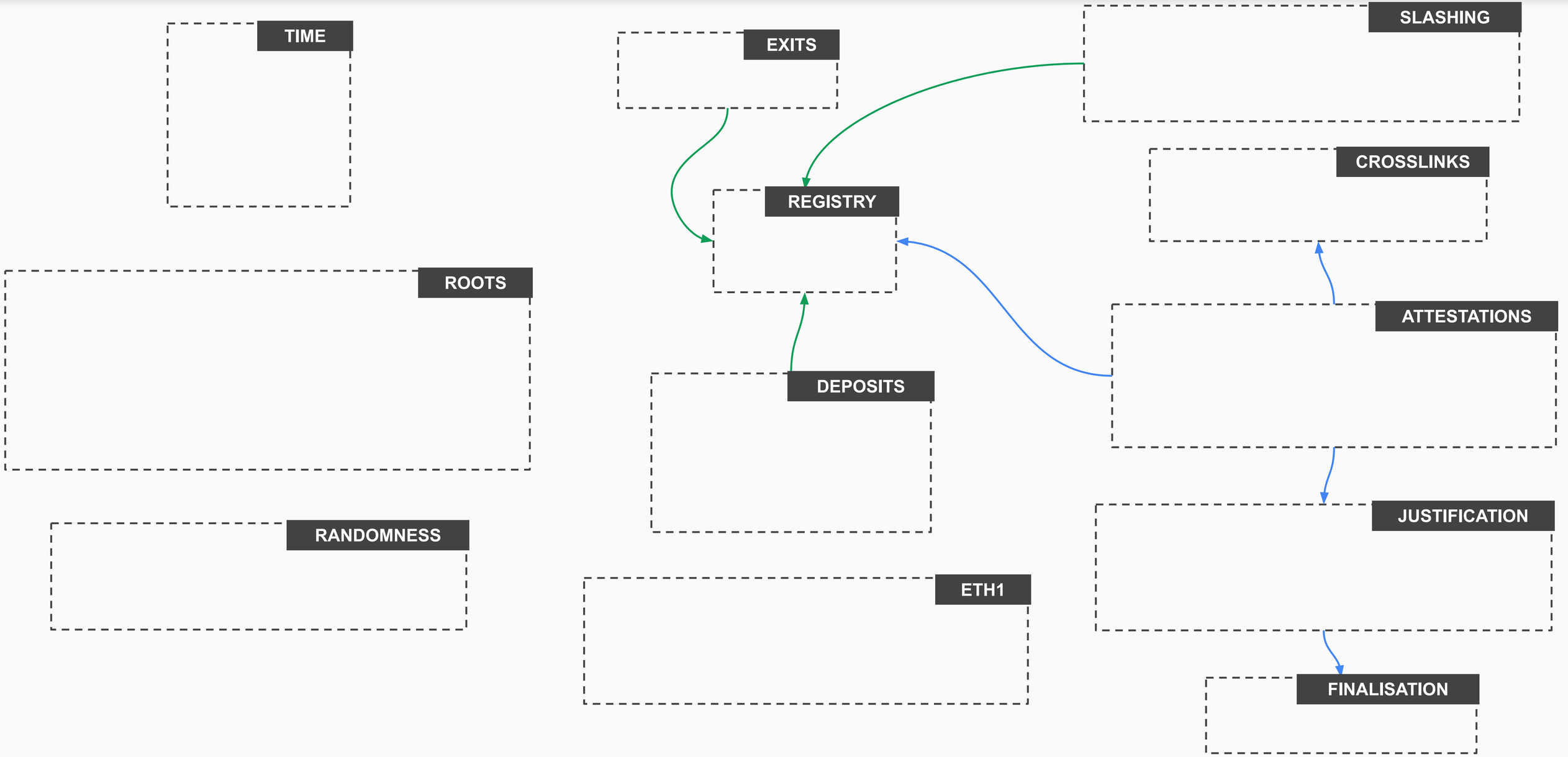
```
(pre_state, block) -> post_state (or error)
```



epochs

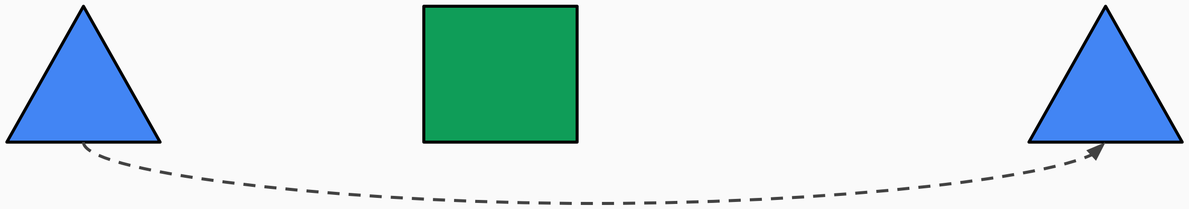


modularity


TIME
EXITS
SLASHING
CROSSLINKS
REGISTRY
ROOTS
ATTESTATIONS
DEPOSITS
JUSTIFICATION
RANDOMNESS
ETH1
FINALISATION

state transition function

```
(pre_state, block) -> post_state (or error)
```



BeaconState

```
class BeaconState(Container):
    # Versioning
    genesis_time: uint64
    slot: Slot
    fork: Fork
```

BeaconState

```python
class BeaconState(Container):
    # Versioning
    genesis_time: uint64
    slot: Slot
    fork: Fork
    # History
    latest_block_header: BeaconBlockHeader
    block_roots: Vector[Hash, SLOTS_PER_HISTORICAL_ROOT]
    state_roots: Vector[Hash, SLOTS_PER_HISTORICAL_ROOT]
    historical_roots: List[Hash, HISTORICAL_ROOTS_LIMIT]
```

BeaconState

```python
class BeaconState(Container):
    # Versioning
    genesis_time: uint64
    slot: Slot
    fork: Fork
    # History
    latest_block_header: BeaconBlockHeader
    block_roots: Vector[Hash, SLOTS_PER_HISTORICAL_ROOT]
    state_roots: Vector[Hash, SLOTS_PER_HISTORICAL_ROOT]
    historical_roots: List[Hash, HISTORICAL_ROOTS_LIMIT]
    # Eth1
    eth1_data: Eth1Data
    eth1_data_votes: List[Eth1Data, SLOTS_PER_ETH1_VOTING_PERIOD]
    eth1_deposit_index: uint64
```

BeaconState

```python
class BeaconState(Container):
    # Versioning
    genesis_time: uint64
    slot: Slot
    fork: Fork
    # History
    latest_block_header: BeaconBlockHeader
    block_roots: Vector[Hash, SLOTS_PER_HISTORICAL_ROOT]
    state_roots: Vector[Hash, SLOTS_PER_HISTORICAL_ROOT]
    historical_roots: List[Hash, HISTORICAL_ROOTS_LIMIT]
    # Eth1
    eth1_data: Eth1Data
    eth1_data_votes: List[Eth1Data, SLOTS_PER_ETH1_VOTING_PERIOD]
    eth1_deposit_index: uint64
    # Registry
    validators: List[Validator, VALIDATOR_REGISTRY_LIMIT]
    balances: List[Gwei, VALIDATOR_REGISTRY_LIMIT]
```

BeaconState

```python
class BeaconState(Container):
    # Versioning
    genesis_time: uint64
    slot: Slot
    fork: Fork
    # History
    latest_block_header: BeaconBlockHeader
    block_roots: Vector[Hash, SLOTS_PER_HISTORICAL_ROOT]
    state_roots: Vector[Hash, SLOTS_PER_HISTORICAL_ROOT]
    historical_roots: List[Hash, HISTORICAL_ROOTS_LIMIT]
    # Eth1
    eth1_data: Eth1Data
    eth1_data_votes: List[Eth1Data, SLOTS_PER_ETH1_VOTING_PERIOD]
    eth1_deposit_index: uint64
    # Registry
    validators: List[Validator, VALIDATOR_REGISTRY_LIMIT]
    balances: List[Gwei, VALIDATOR_REGISTRY_LIMIT]
    # Shuffling
    start_shard: Shard
    randao_mixes: Vector[Hash, EPOCHS_PER_HISTORICAL_VECTOR]
```

BeaconState

```python
class BeaconState(Container):
    # Versioning
    genesis_time: uint64
    slot: Slot
    fork: Fork
    # History
    latest_block_header: BeaconBlockHeader
    block_roots: Vector[Hash, SLOTS_PER_HISTORICAL_ROOT]
    state_roots: Vector[Hash, SLOTS_PER_HISTORICAL_ROOT]
    historical_roots: List[Hash, HISTORICAL_ROOTS_LIMIT]
    # Eth1
    eth1_data: Eth1Data
    eth1_data_votes: List[Eth1Data, SLOTS_PER_ETH1_VOTING_PERIOD]
    eth1_deposit_index: uint64
    # Registry
    validators: List[Validator, VALIDATOR_REGISTRY_LIMIT]
    balances: List[Gwei, VALIDATOR_REGISTRY_LIMIT]
    # Shuffling
    start_shard: Shard
    randao_mixes: Vector[Hash, EPOCHS_PER_HISTORICAL_VECTOR]
    # Slashings
    slashings: Vector[Gwei, EPOCHS_PER_SLASHINGS_VECTOR]
```

BeaconState

```python
class BeaconState(Container):
    # Versioning
    genesis_time: uint64
    slot: Slot
    fork: Fork
    # History
    latest_block_header: BeaconBlockHeader
    block_roots: Vector[Hash, SLOTS_PER_HISTORICAL_ROOT]
    state_roots: Vector[Hash, SLOTS_PER_HISTORICAL_ROOT]
    historical_roots: List[Hash, HISTORICAL_ROOTS_LIMIT]
    # Eth1
    eth1_data: Eth1Data
    eth1_data_votes: List[Eth1Data, SLOTS_PER_ETH1_VOTING_PERIOD]
    eth1_deposit_index: uint64
    # Registry
    validators: List[Validator, VALIDATOR_REGISTRY_LIMIT]
    balances: List[Gwei, VALIDATOR_REGISTRY_LIMIT]
    # Shuffling
    start_shard: Shard
    randao_mixes: Vector[Hash, EPOCHS_PER_HISTORICAL_VECTOR]
    # Slashings
    slashings: Vector[Gwei, EPOCHS_PER_SLASHINGS_VECTOR]
    # Attestations
    previous_epoch_attestations: List[PendingAttestation, MAX_ATTESTATIONS * SLOTS_PER_EPOCH]
    current_epoch_attestations: List[PendingAttestation, MAX_ATTESTATIONS * SLOTS_PER_EPOCH]
```

BeaconState

```python
class BeaconState(Container):
    # Versioning
    genesis_time: uint64
    slot: Slot
    fork: Fork
    # History
    latest_block_header: BeaconBlockHeader
    block_roots: Vector[Hash, SLOTS_PER_HISTORICAL_ROOT]
    state_roots: Vector[Hash, SLOTS_PER_HISTORICAL_ROOT]
    historical_roots: List[Hash, HISTORICAL_ROOTS_LIMIT]
    # Eth1
    eth1_data: Eth1Data
    eth1_data_votes: List[Eth1Data, SLOTS_PER_ETH1_VOTING_PERIOD]
    eth1_deposit_index: uint64
    # Registry
    validators: List[Validator, VALIDATOR_REGISTRY_LIMIT]
    balances: List[Gwei, VALIDATOR_REGISTRY_LIMIT]
    # Shuffling
    start_shard: Shard
    randao_mixes: Vector[Hash, EPOCHS_PER_HISTORICAL_VECTOR]
    # Slashings
    slashings: Vector[Gwei, EPOCHS_PER_SLASHINGS_VECTOR]
    # Attestations
    previous_epoch_attestations: List[PendingAttestation, MAX_ATTESTATIONS * SLOTS_PER_EPOCH]
    current_epoch_attestations: List[PendingAttestation, MAX_ATTESTATIONS * SLOTS_PER_EPOCH]
    # Crosslinks
    previous_crosslinks: Vector[Crosslink, SHARD_COUNT]
    current_crosslinks: Vector[Crosslink, SHARD_COUNT]
```

BeaconState

- SHA256
- tree hashing

- RANDAO
- swap-or-not

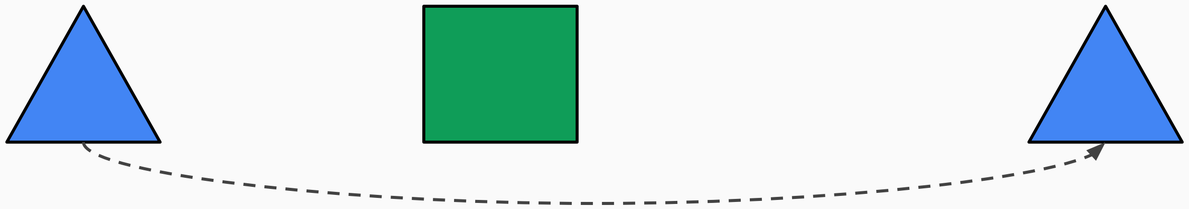
- BLS
- GHOST

- FFG

```python
class BeaconState(Container):
    # Versioning
    genesis_time: uint64
    slot: Slot
    fork: Fork
    # History
    latest_block_header: BeaconBlockHeader
    block_roots: Vector[Hash, SLOTS_PER_HISTORICAL_ROOT]
    state_roots: Vector[Hash, SLOTS_PER_HISTORICAL_ROOT]
    historical_roots: List[Hash, HISTORICAL_ROOTS_LIMIT]
    # Eth1
    eth1_data: Eth1Data
    eth1_data_votes: List[Eth1Data, SLOTS_PER_ETH1_VOTING_PERIOD]
    eth1_deposit_index: uint64
    # Registry
    validators: List[Validator, VALIDATOR_REGISTRY_LIMIT]
    balances: List[Gwei, VALIDATOR_REGISTRY_LIMIT]
    # Shuffling
    start_shard: Shard
    randao_mixes: Vector[Hash, EPOCHS_PER_HISTORICAL_VECTOR]
    # Slashings
    slashings: Vector[Gwei, EPOCHS_PER_SLASHINGS_VECTOR]
    # Attestations
    previous_epoch_attestations: List[PendingAttestation, MAX_ATTESTATIONS * SLOTS_PER_EPOCH]
    current_epoch_attestations: List[PendingAttestation, MAX_ATTESTATIONS * SLOTS_PER_EPOCH]
    # Crosslinks
    previous_crosslinks: Vector[Crosslink, SHARD_COUNT]
    current_crosslinks: Vector[Crosslink, SHARD_COUNT]
    # Finality
    justification_bits: Bitvector[JUSTIFICATION_BITS_LENGTH]
    previous_justified_checkpoint: Checkpoint
    current_justified_checkpoint: Checkpoint
    finalized_checkpoint: Checkpoint
```

state transition function

```
(pre_state, block) -> post_state (or error)
```



BeaconBlock

```
class BeaconBlock(Container):
    slot: Slot
    parent_root: Hash
    state_root: Hash
    body: BeaconBlockBody
    signature: BLSSignature
```

BeaconBlock

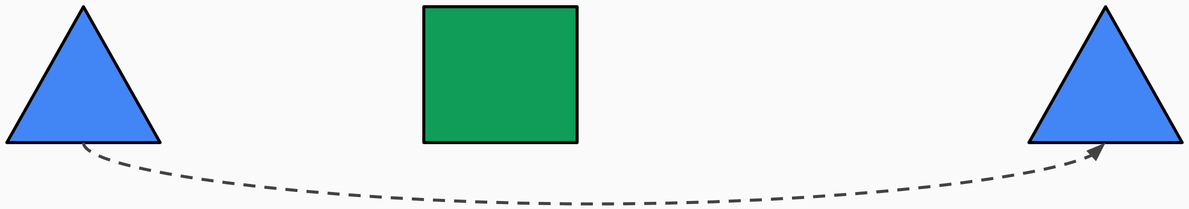
```
class BeaconBlock(Container):
    slot: Slot
    parent_root: Hash
    state_root: Hash
    body: BeaconBlockBody
    signature: BLSSignature

class BeaconBlockBody(Container):
    randao_reveal: BLSSignature
    eth1_data: Eth1Data
    graffiti: Bytes32
```

BeaconBlock

```python
class BeaconBlock(Container):
    slot: Slot
    parent_root: Hash
    state_root: Hash
    body: BeaconBlockBody
    signature: BLSSignature

class BeaconBlockBody(Container):
    randao_reveal: BLSSignature
    eth1_data: Eth1Data
    graffiti: Bytes32
    # Operations
    proposer_slashings: List[ProposerSlashing, MAX_PROPOSER_SLASHINGS]
    attester_slashings: List[AttesterSlashing, MAX_ATTESTER_SLASHINGS]
    attestations: List[Attestation, MAX_ATTESTATIONS]
    deposits: List[Deposit, MAX_DEPOSITS]
    voluntary_exits: List[VoluntaryExit, MAX_VOLUNTARY_EXITS]
    transfers: List[Transfer, MAX_TRANSFERS]
```

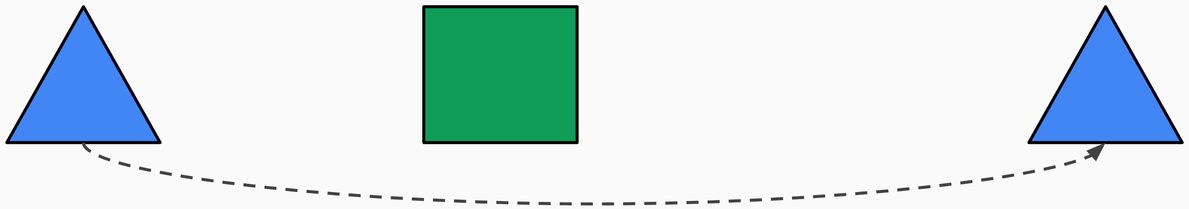
honest state transition function

```
(pre_state, block) -> post_state (or error)
```



honest state transition function

```
(pre_state, block) -> post_state (or error)
```

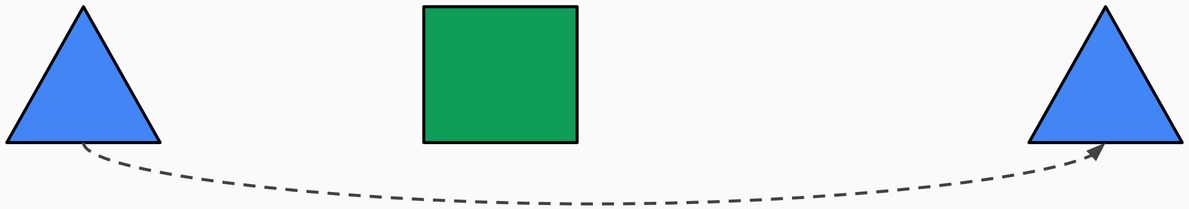


✓

```
state.some_field = some_value
```

honest state transition function

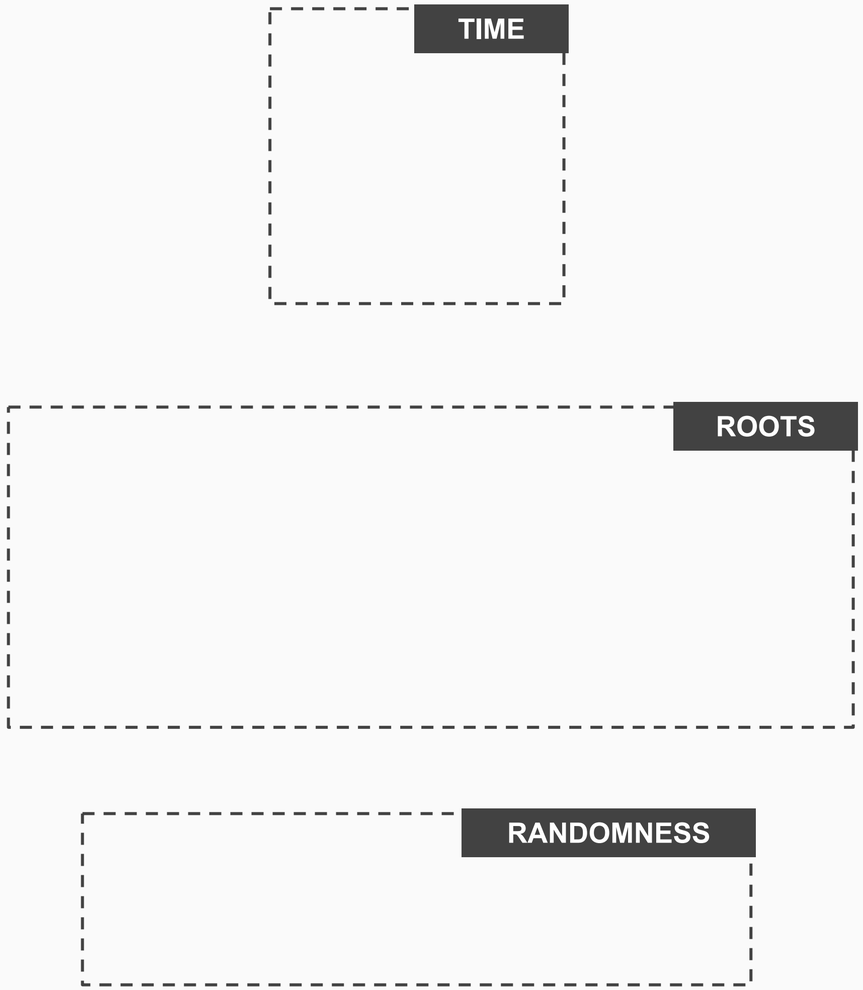
```
(pre_state, block) -> post_state (or error)
```



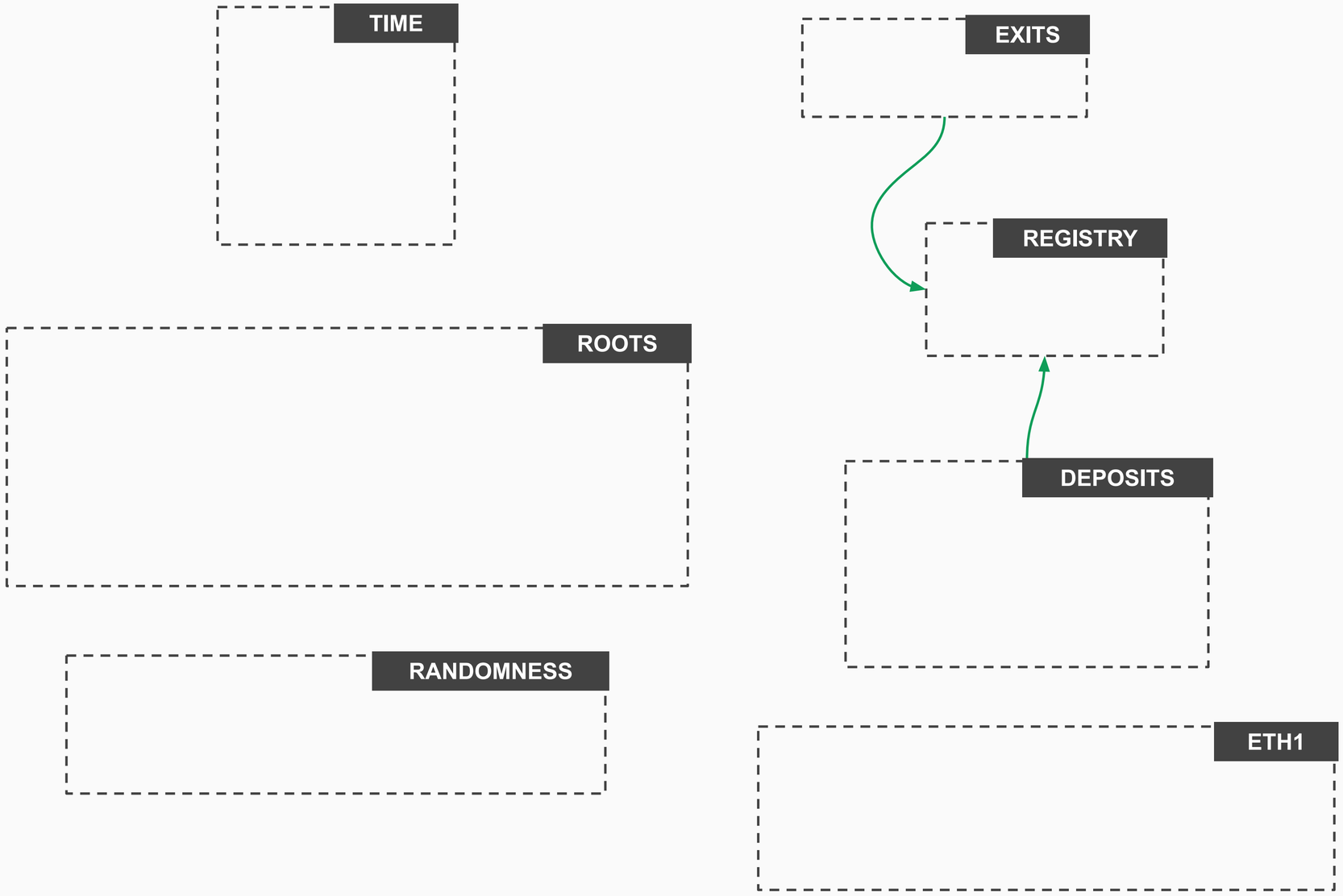
```
state.some_field = some_value
```

```
assert some_invariant(block, state)
```

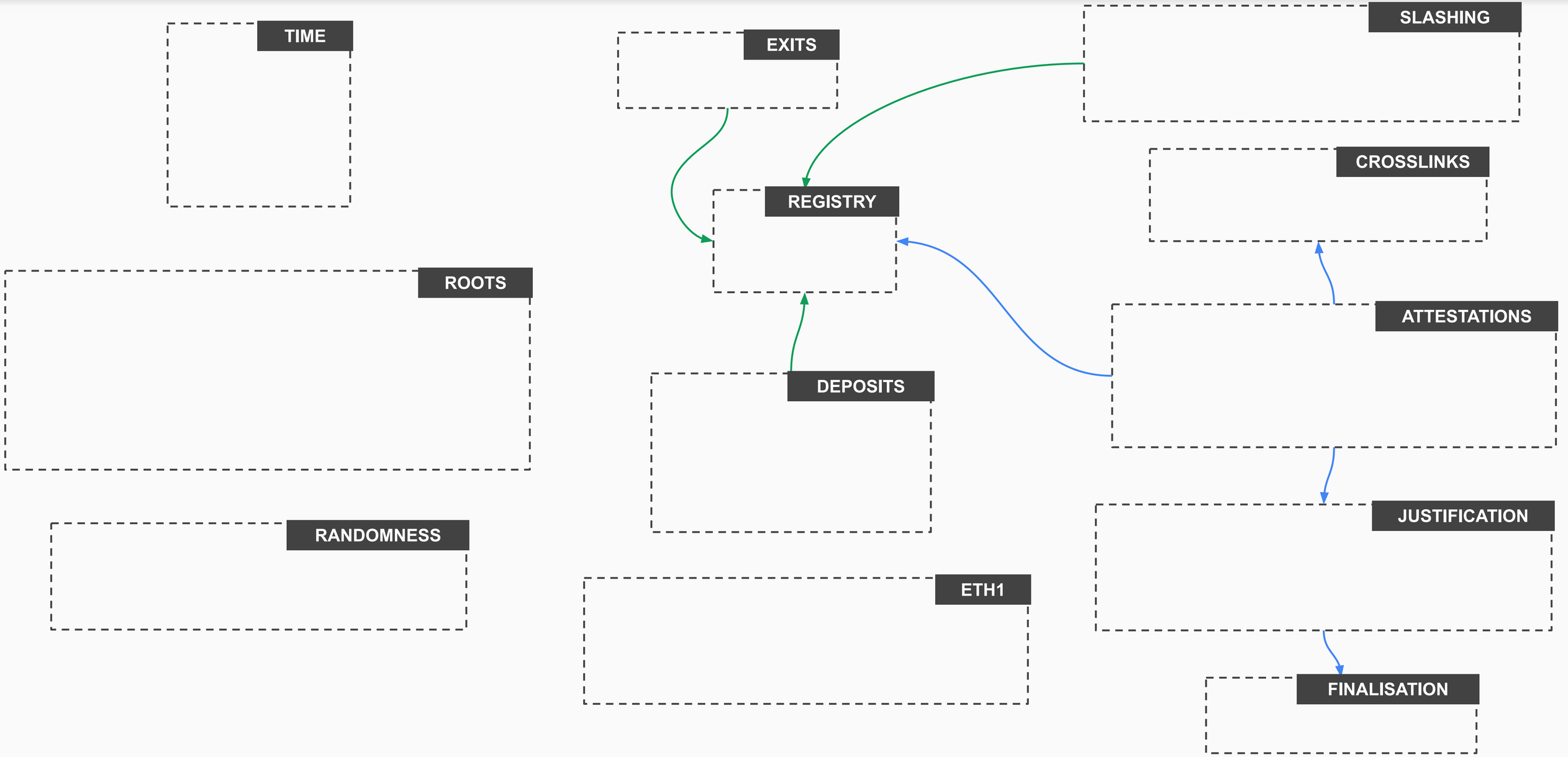
state transition function



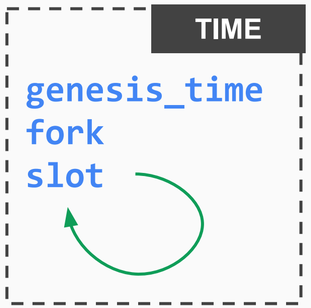
state transition function



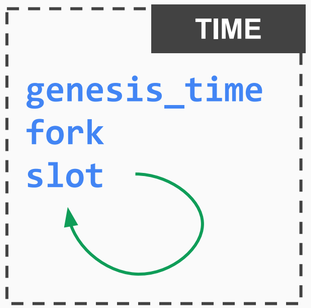
state transition function

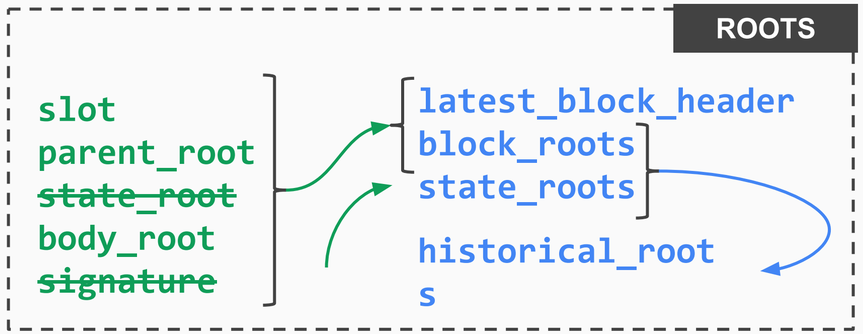


state transition function

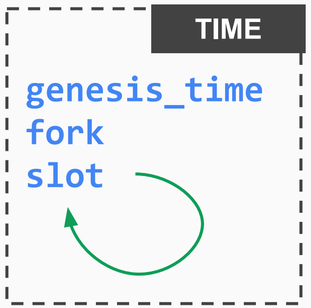


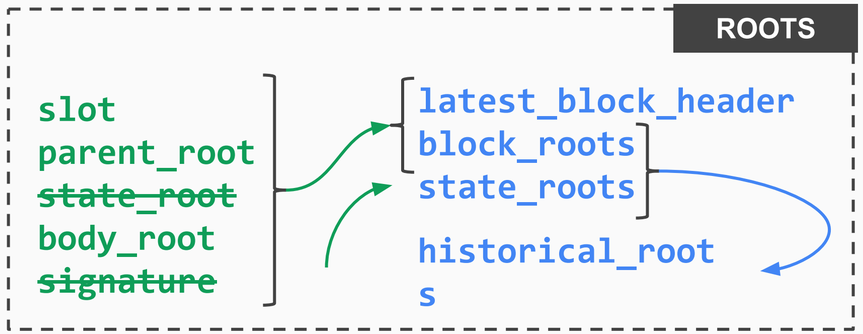
state transition function

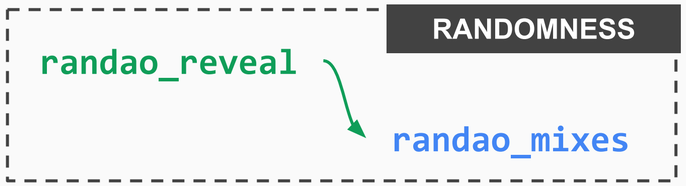




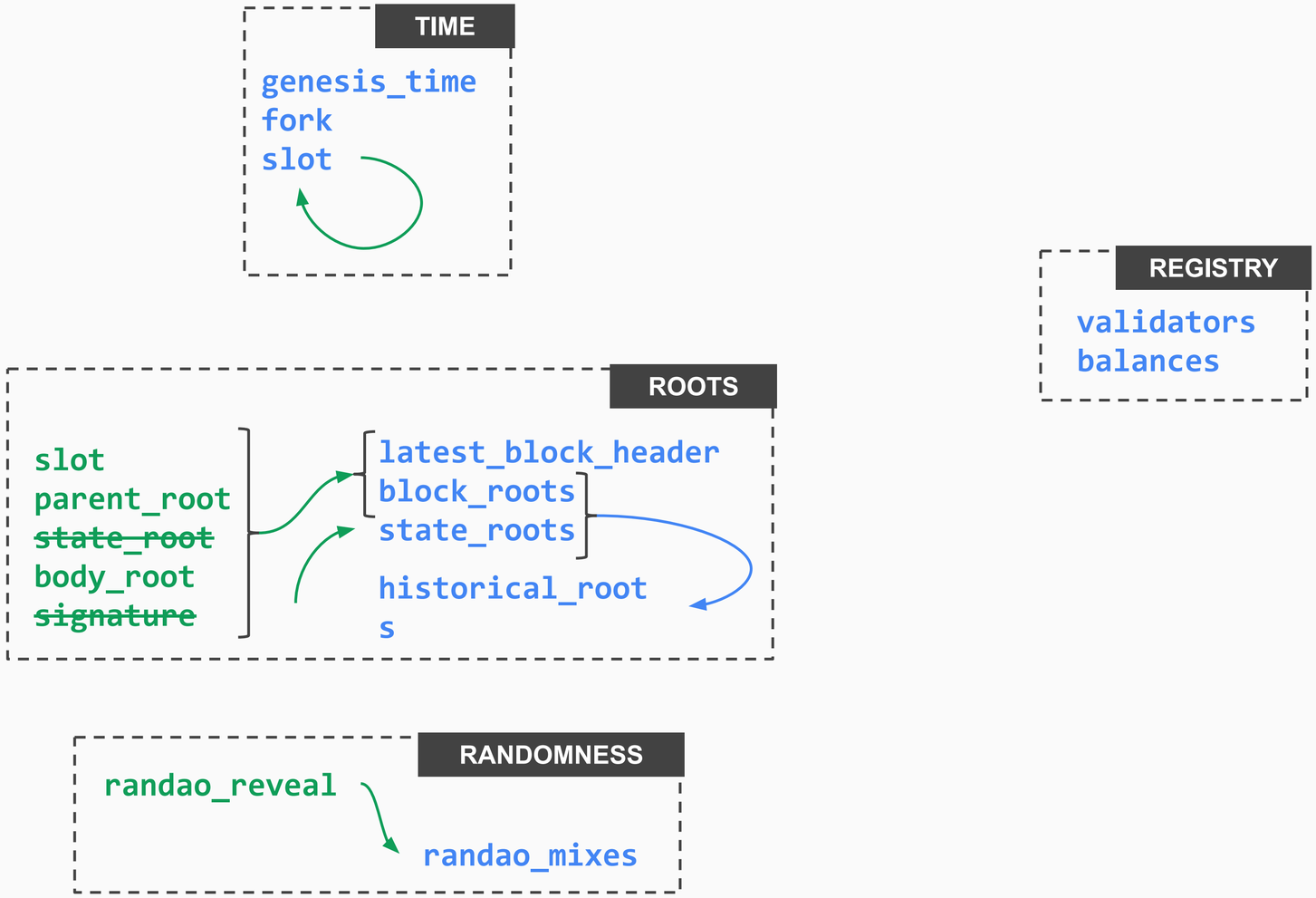
state transition function







state transition function



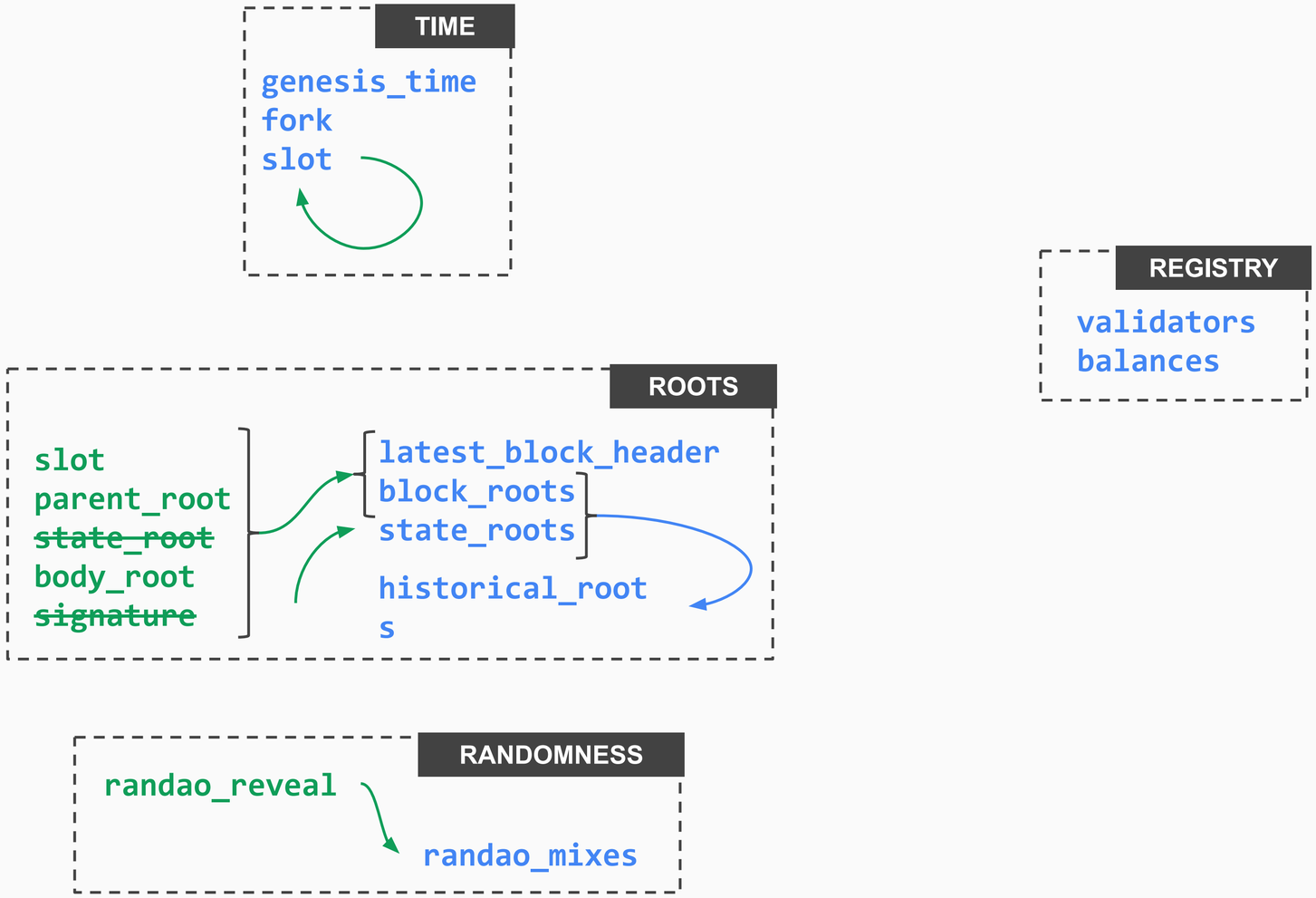
Validator

```
class Validator(Container):
    pubkey: BLSPubkey
    withdrawal_credentials: Hash
    effective_balance: Gwei
    slashed: boolean
```

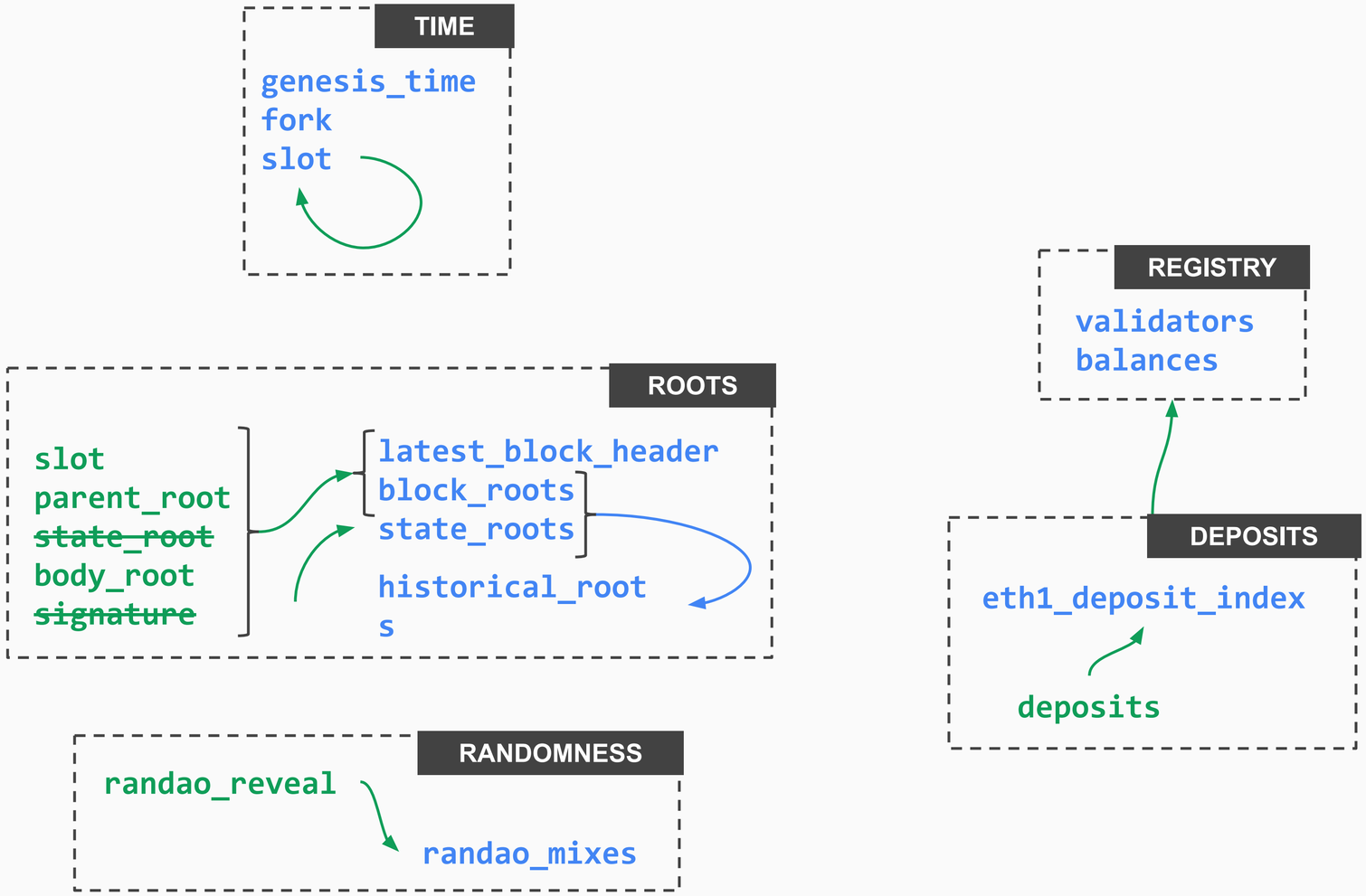
Validator

```
class Validator(Container):
    pubkey: BLSPubkey
    withdrawal_credentials: Hash
    effective_balance: Gwei
    slashed: boolean
    # Status epochs
    activation_eligibility_epoch: Epoch
    activation_epoch: Epoch
    exit_epoch: Epoch
    withdrawable_epoch: Epoch
```

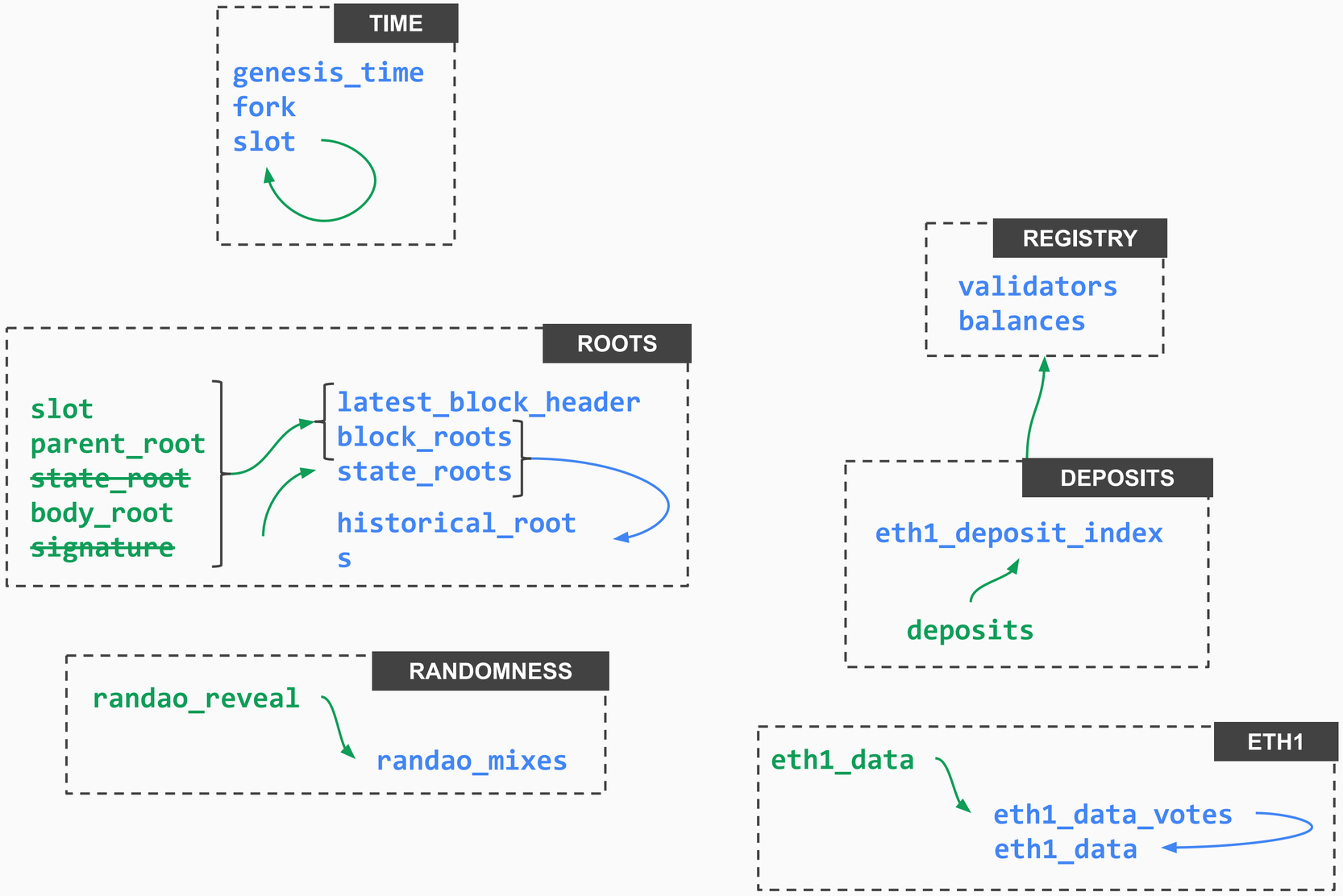
state transition function



state transition function

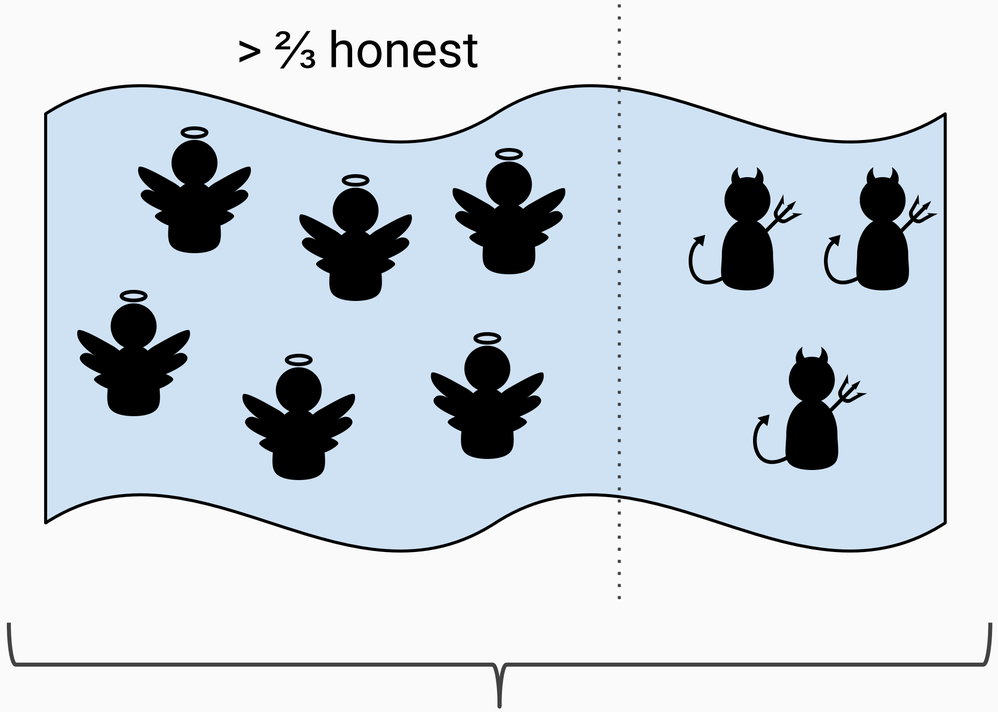


state transition function



honest majority assumption

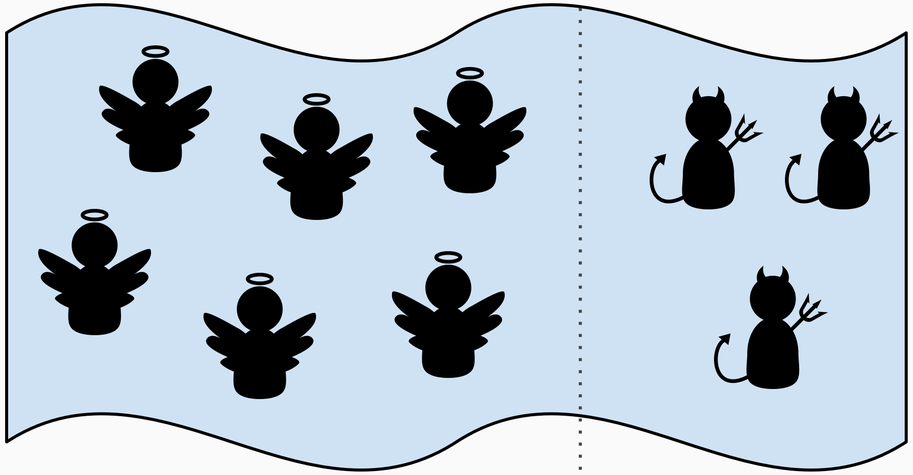
active validators



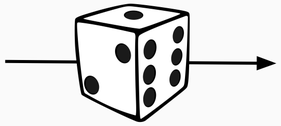
honest majority assumption

active validators

> ⅔ honest

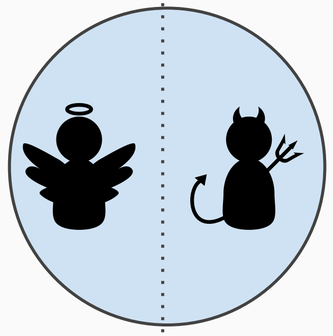


~1,000,000 validators



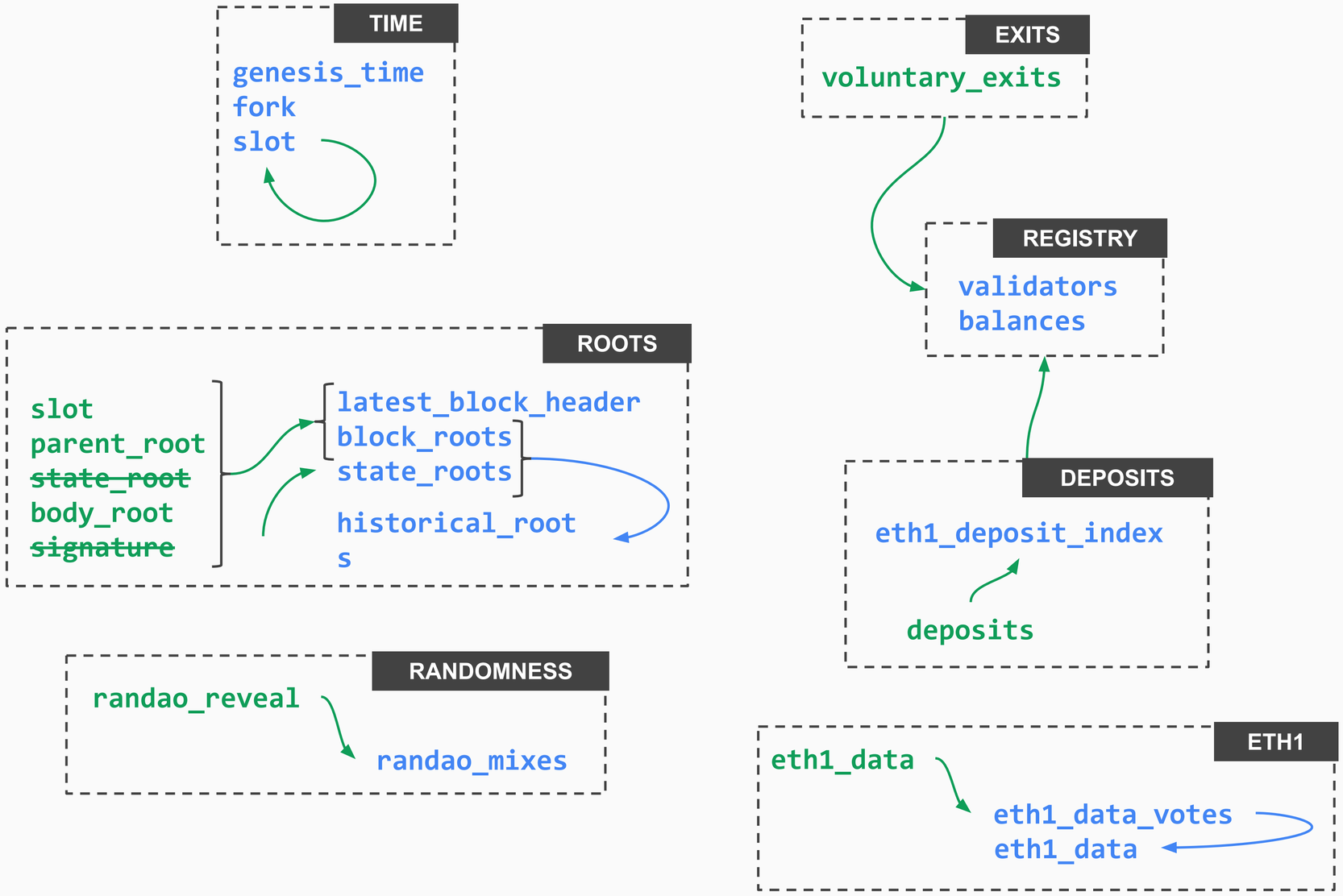
Eth1 committee

> ½ honest

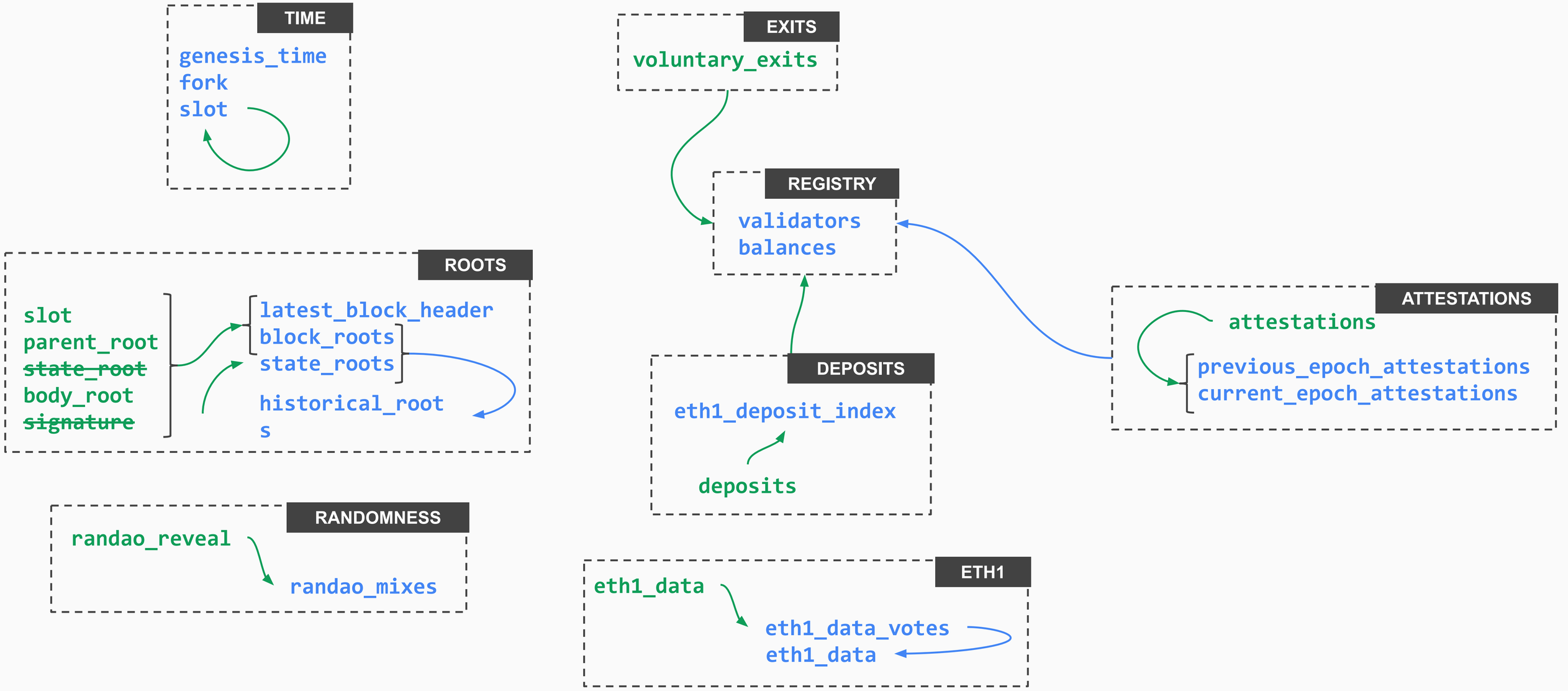


~1,000 validators

state transition function



state transition function



Attestation

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

Attestation

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

class AttestationData(Container):
    # LMD GHOST vote
    beacon_block_root: Hash
    # FFG vote
    source: Checkpoint
    target: Checkpoint
    # Crosslink vote
    crosslink: Crosslink
```

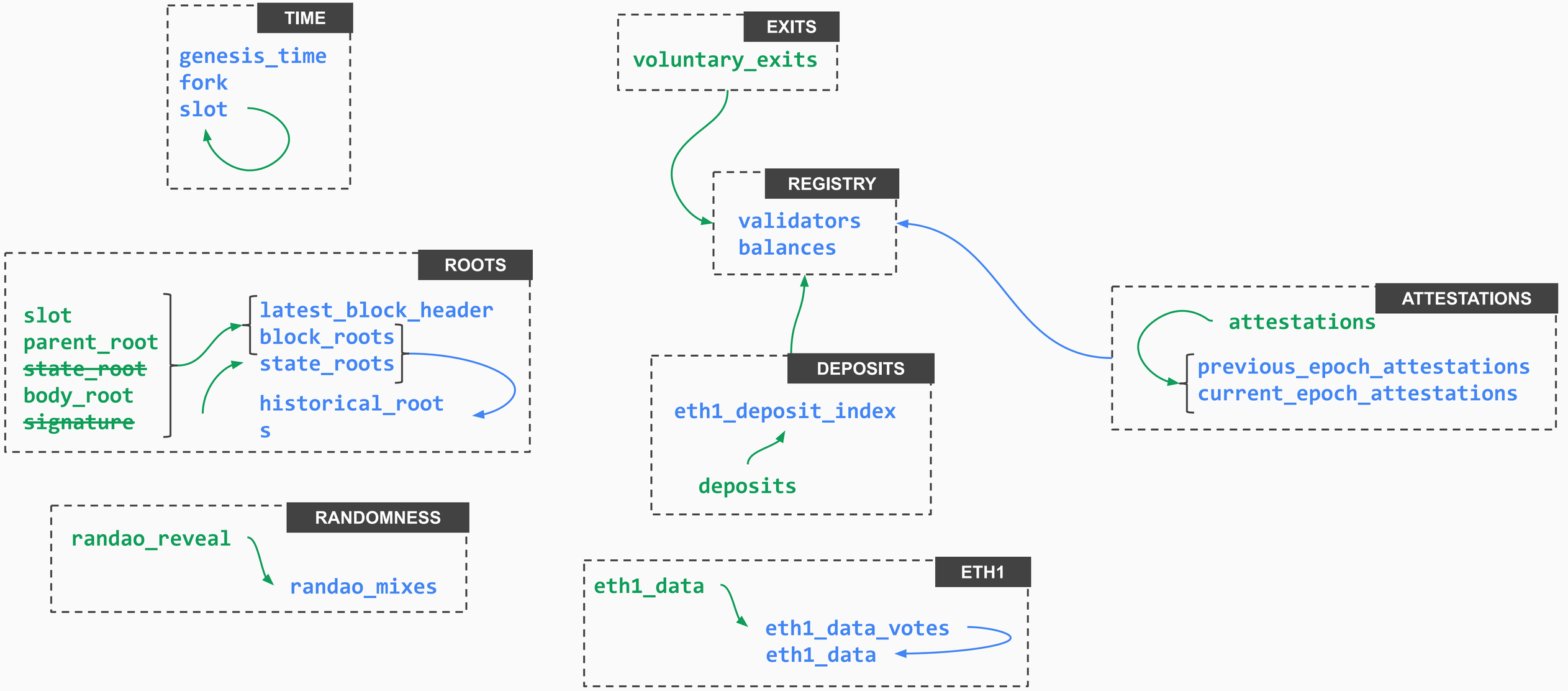
Attestation

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

class AttestationData(Container):
    # LMD GHOST vote
    beacon_block_root: Hash
    # FFG vote
    source: Checkpoint
    target: Checkpoint
    # Crosslink vote
    crosslink: Crosslink

class Checkpoint(Container):
    epoch: Epoch
    root: Hash
```

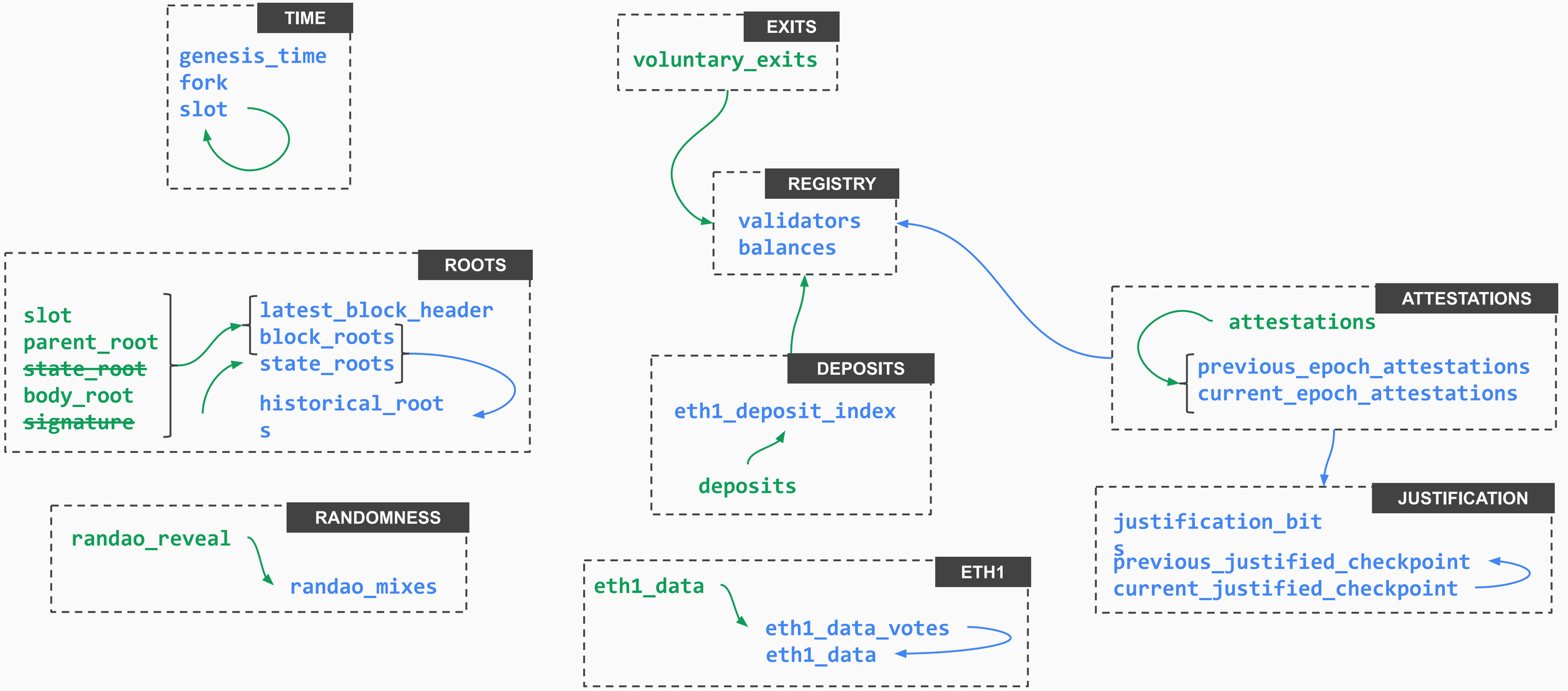
Attestation

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

class AttestationData(Container):
    # LMD GHOST vote
    beacon_block_root: Hash
    # FFG vote
    source: Checkpoint
    target: Checkpoint
    # Crosslink vote
    crosslink: Crosslink

class Checkpoint(Container):
    epoch: Epoch
    root: Hash

class Crosslink(Container):
    shard: Shard
    parent_root: Hash
    # Crosslinking data
    start_epoch: Epoch
    end_epoch: Epoch
    data_root: Hash
```

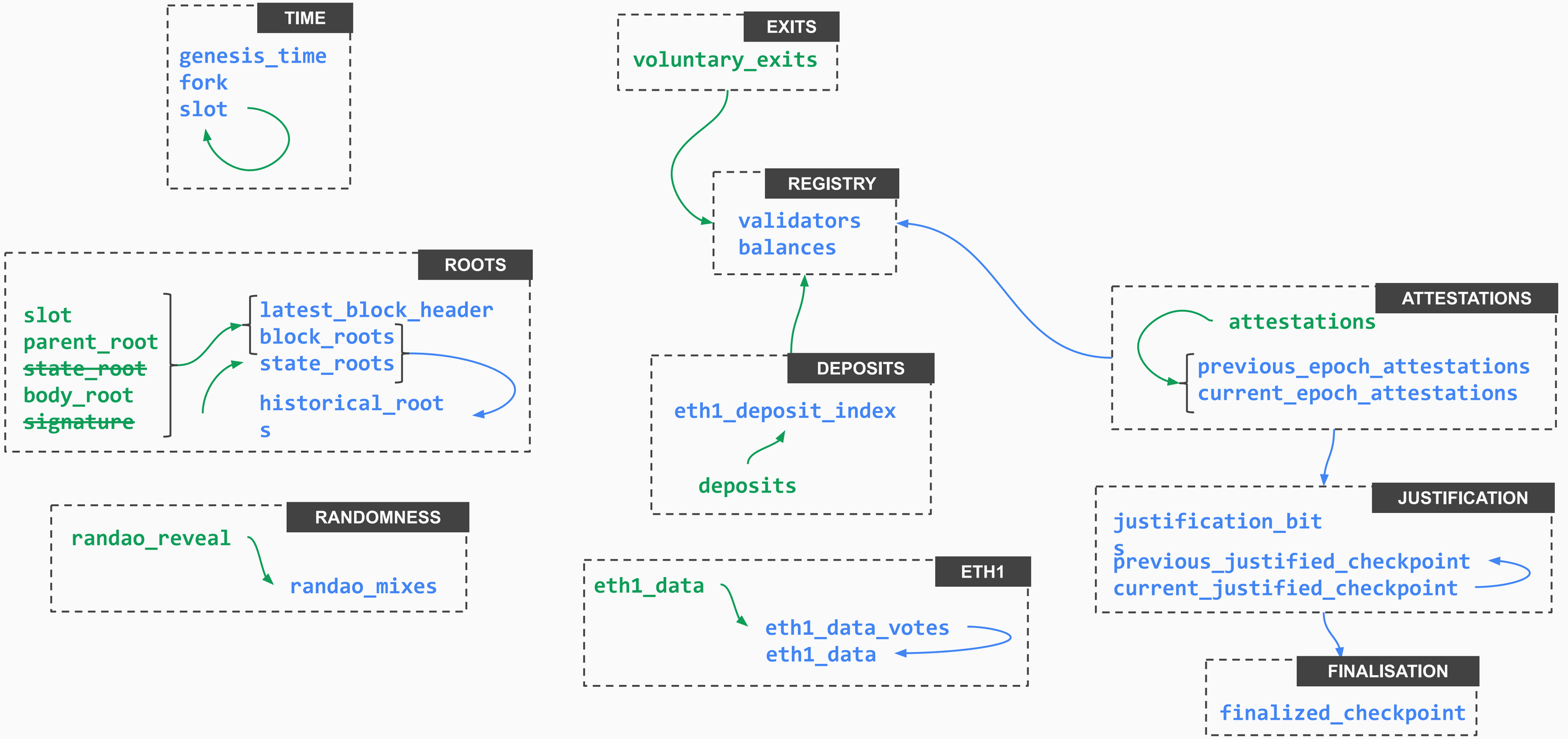
state transition function



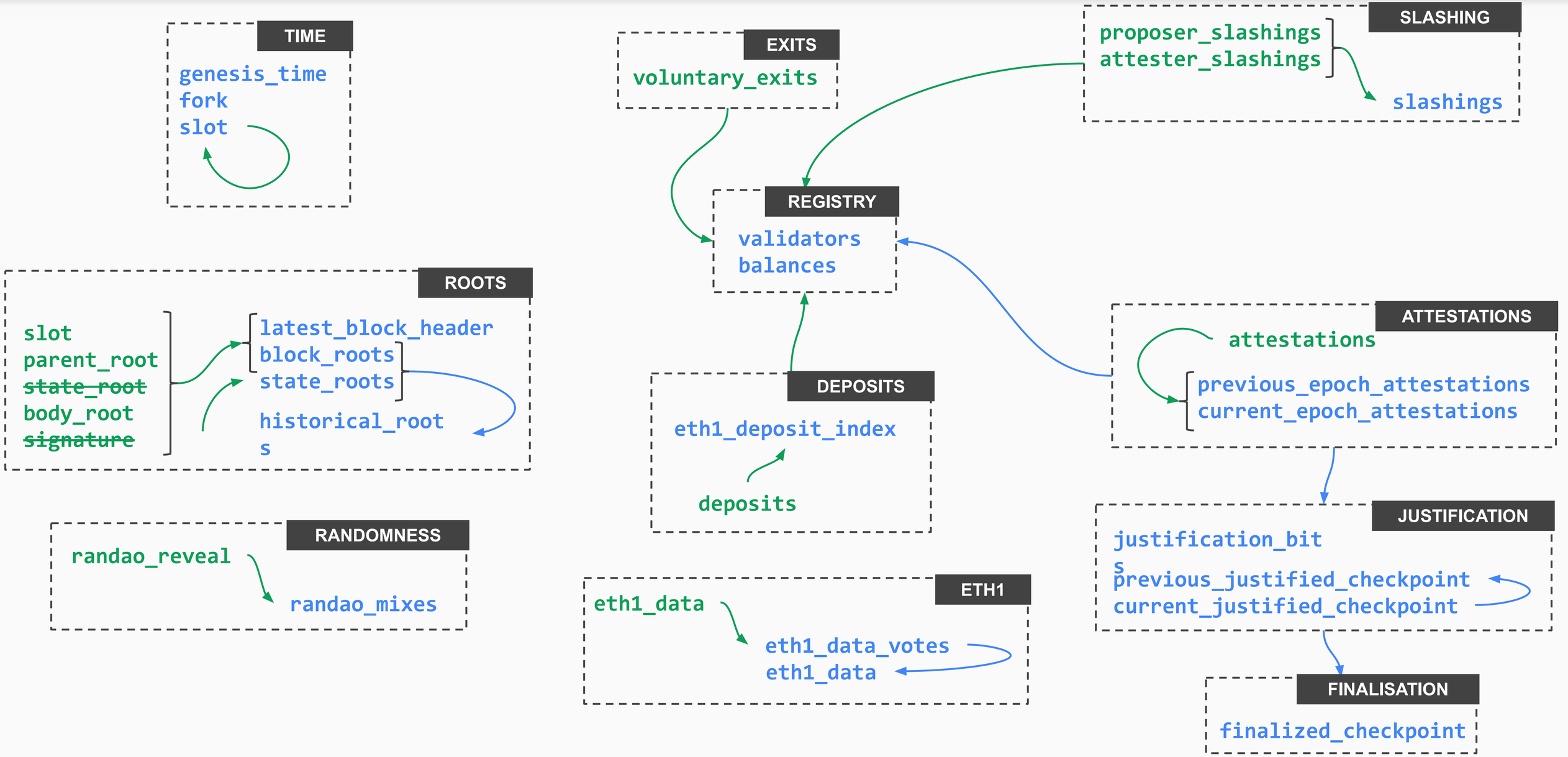
state transition function



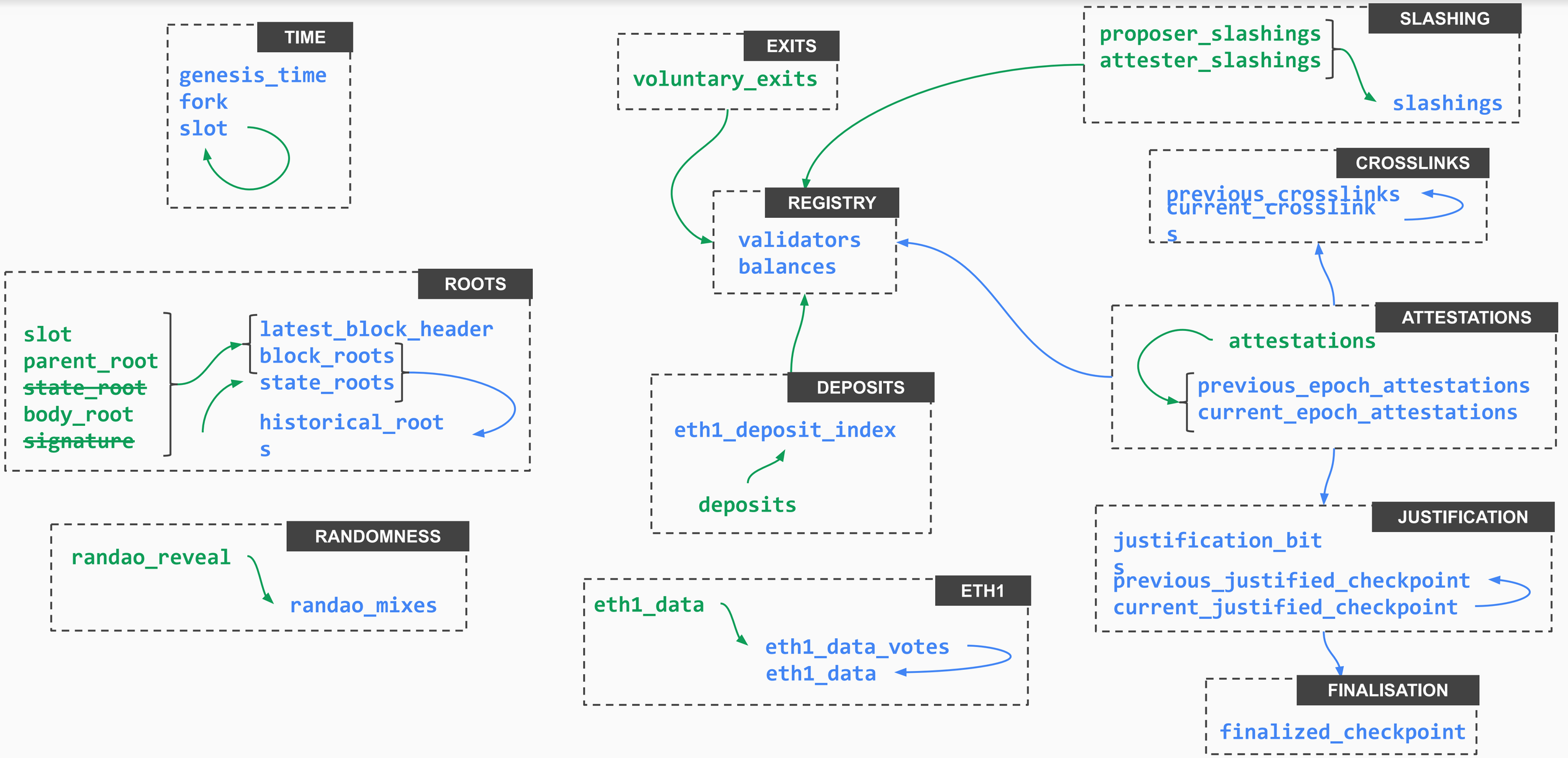
state transition function



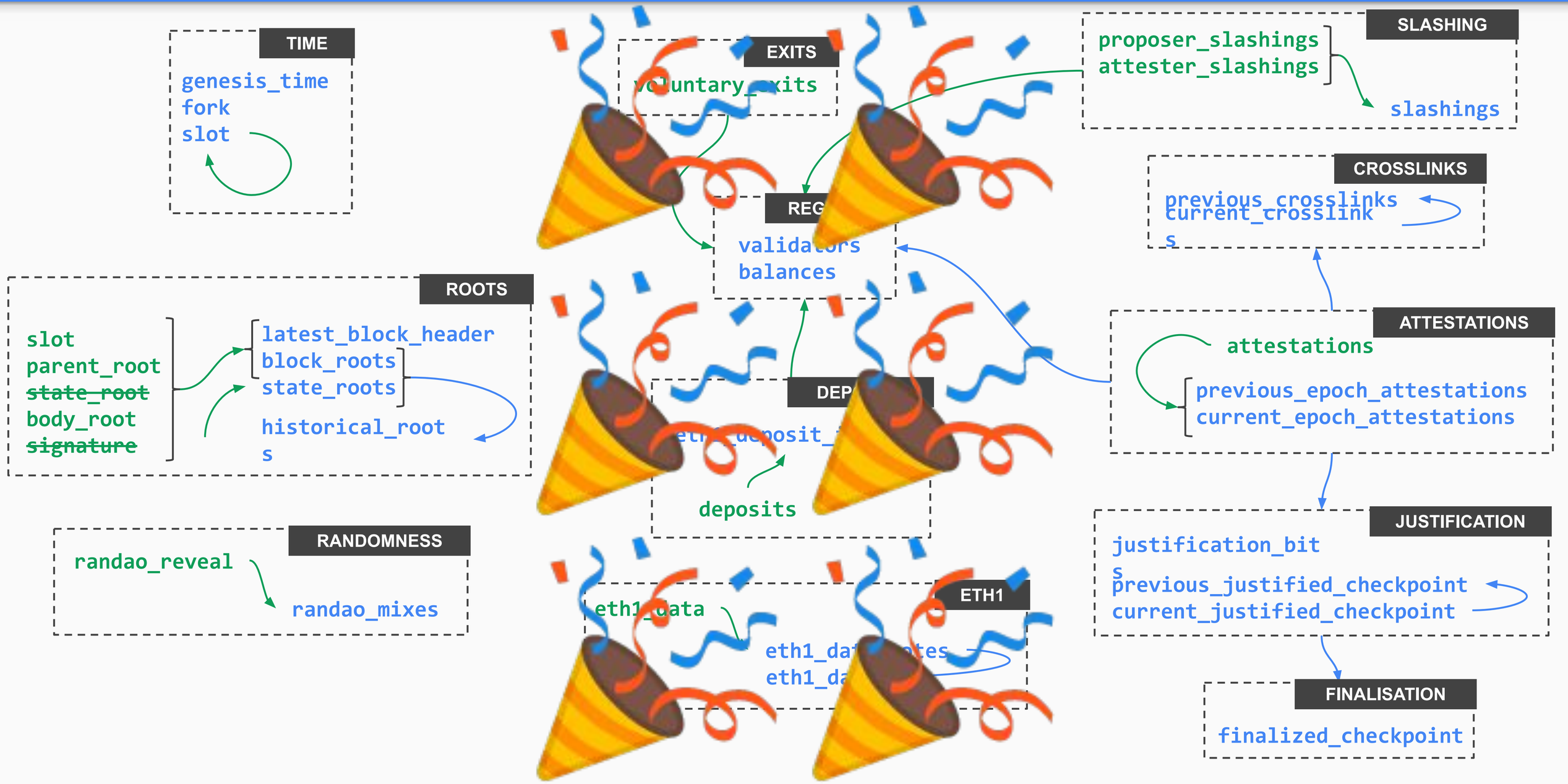
state transition function



state transition function



state transition function



after phase 0

- transfers

after phase 0

- transfers
- custody proofs
- secret proposers
- VDF randomness
- availability proofs

security upgrades
(optional; fancy crypto)

after phase 0

- transfers
- custody proofs
- secret proposers
- VDF randomness
- availability proofs
- multi hashing
- quantum-security

security upgrades
(optional; fancy crypto)

STARKs, Eth3

Validator Duties

read the spec

Ethereum 2.0 Phase 0 -- Honest Validator

https://github.com/ethereum/eth2.0-specs/blob/dev/specs/validator/0_beacon-chain-validator.md

tiny.cc/phase0validator

Proposer Assignment

```
def is_proposer(state: BeaconState,
                validator_index: ValidatorIndex) -> bool:
    return get_beacon_proposer_index(state) == validator_index
```

```
def compute_proposer_index(state: BeaconState, indices: Sequence[ValidatorIndex], seed: Hash) -> Validat
    """
    Return from ``indices`` a random index sampled by effective balance.
    """
    assert len(indices) > 0
    MAX_RANDOM_BYTE = 2**8 - 1
    i = 0
    while True:
        candidate_index = indices[compute_shuffled_index(ValidatorIndex(i % len(indices)), len(indices),
        random_byte = hash(seed + int_to_bytes(i // 32, length=8))[i % 32]
        effective_balance = state.validators[candidate_index].effective_balance
        if effective_balance * MAX_RANDOM_BYTE >= MAX_EFFECTIVE_BALANCE * random_byte:
            return ValidatorIndex(candidate_index)
        i += 1
```

- Independent of committee assignment
- Knowable *within* the epoch of proposal
- Action performed at `slot_start_time`
- Publicly known and DOS concern

Block Proposal

```
class BeaconBlock(Container):
    slot: Slot
    parent_root: Hash
    state_root: Hash
    body: BeaconBlockBody
    signature: BLSSignature
```

```
class BeaconBlockBody(Container):
    randao_reveal: BLSSignature
    eth1_data: Eth1Data  # Eth1 data vote
    graffiti: Bytes32  # Arbitrary data
    # Operations
    proposer_slashings: List[ProposerSlashing, MAX_PROPOSER_SLASHINGS]
    attester_slashings: List[AttesterSlashing, MAX_ATTESTER_SLASHINGS]
    attestations: List[Attestation, MAX_ATTESTATIONS]
    deposits: List[Deposit, MAX_DEPOSITS]
    voluntary_exits: List[VoluntaryExit, MAX_VOLUNTARY_EXITS]
```

Eth1Data

```
def get_eth1_vote(state: BeaconState, previous_eth1_distance: uint64) -> Eth1Data:
    new_eth1_data = [get_eth1_data(distance) for distance in range(ETH1_FOLLOW_DISTANCE, 2 * ETH1_FOLLOW
    all_eth1_data = [get_eth1_data(distance) for distance in range(ETH1_FOLLOW_DISTANCE, previous_eth1_d

    valid_votes = []
    for slot, vote in enumerate(state.eth1_data_votes):
        period_tail = slot % SLOTS_PER_ETH1_VOTING_PERIOD >= integer_squareroot(SLOTS_PER_ETH1_VOTING_PE
        if vote in new_eth1_data or (period_tail and vote in all_eth1_data):
            valid_votes.append(vote)

    return max(
        valid_votes,
        key=lambda v: (valid_votes.count(v), -all_eth1_data.index(v)),  # Tiebreak by smallest distance
        default=get_eth1_data(ETH1_FOLLOW_DISTANCE),
    )
```

- Mechanism to come to consensus on recent eth1 data
- Mechanism to avoid stale data winning

Slashability of Block Proposals

```
def process_proposer_slashing(state: BeaconState, proposer_slashing: ProposerSlashing) -> None:
    proposer = state.validators[proposer_slashing.proposer_index]
    # Verify that the epoch is the same
    assert (compute_epoch_of_slot(proposer_slashing.header_1.slot)
            == compute_epoch_of_slot(proposer_slashing.header_2.slot))
    # But the headers are different
    assert proposer_slashing.header_1 != proposer_slashing.header_2
```

Committee Assignment

```
def get_committee_assignment(state: BeaconState,
                             epoch: Epoch,
                             validator_index: ValidatorIndex) -> Optional[Tuple[Sequence[ValidatorIndex]
    """
    Return the committee assignment in the ``epoch`` for ``validator_index``.
    ``assignment`` returned is a tuple of the following form:
        * ``assignment[0]`` is the list of validators in the committee
        * ``assignment[1]`` is the shard to which the committee is assigned
        * ``assignment[2]`` is the slot at which the committee is assigned
    Return None if no assignment.
    """
    next_epoch = get_current_epoch(state) + 1
    assert epoch <= next_epoch

    committees_per_slot = get_committee_count(state, epoch) // SLOTS_PER_EPOCH
    start_slot = compute_start_slot_of_epoch(epoch)
    for slot in range(start_slot, start_slot + SLOTS_PER_EPOCH):
        offset = committees_per_slot * (slot % SLOTS_PER_EPOCH)
        slot_start_shard = (get_start_shard(state, epoch) + offset) % SHARD_COUNT
        for i in range(committees_per_slot):
            shard = Shard((slot_start_shard + i) % SHARD_COUNT)
            committee = get_crosslink_committee(state, epoch, shard)
            if validator_index in committee:
                return committee, shard, Slot(slot)
    return None
```

- Assigned one per epoch
- At least one epoch of lookahead
- Tunable via `MIN_SEED_LOOKAHEAD`
- Action performed at `slot_start + 0.5*SECONDS_PER_SLOT`

Attestation Data

```
class AttestationData(Container):
    # LMD GHOST vote
    beacon_block_root: Hash
    # FFG vote
    source: Checkpoint
    target: Checkpoint
    # Crosslink vote
    crosslink: Crosslink
```

Crosslink within AttestationData

Construct `attestation_data.crosslink` via the following.

- Set `attestation_data.crosslink.shard = shard` where `shard` is the shard associated with the validator's committee.
- Let `parent_crosslink = head_state.current_crosslinks[shard]`.
- Set `attestation_data.crosslink.start_epoch = parent_crosslink.end_epoch`.
- Set `attestation_data.crosslink.end_epoch = min(attestation_data.target.epoch, parent_crosslink.end_epoch + MAX_EPOCHS_PER_CROSSLINK)`.
- Set `attestation_data.crosslink.parent_root = hash_tree_root(head_state.current_crosslinks[shard])`.
- Set `attestation_data.crosslink.data_root = ZERO_HASH`. *Note*: This is a stub for Phase 0.

Custody Bit (Phase 0)

```
class AttestationDataAndCustodyBit(Container):
    data: AttestationData
    custody_bit: bit  # Challengeable bit (SSZ-bool, 1 byte) for the custody of crosslink data
```

- The actual object signed
- Stubbed `custody_bit` as 0 for Phase 0

Outer Attestation Structure

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

Crosslink Committee Incentives

- Correct Head
- Correct Target
- Correct Source
- Correct Crosslink
- Fast Inclusion

See `process_rewards_and_penalties`

Slashability of Attestations

is_slashable_attestation_data

```python
def is_slashable_attestation_data(data_1: AttestationData, data_2: AttestationData) -> bool:
    """
    Check if ``data_1`` and ``data_2`` are slashable according to Casper FFG rules.
    """
    return (
        # Double vote
        (data_1 != data_2 and data_1.target.epoch == data_2.target.epoch) or
        # Surround vote
        (data_1.source.epoch < data_2.source.epoch and data_2.target.epoch < data_1.target.epoch)
    )
```