Fork Choice Compliance for Ethereum Clients

Alex Vlasov, Mikhail Kalinin

TxRx team @ Consensys

Problem statement

Fork Choice rule decides on the **head** of the canonical chain.

There is a risk that a client implementation that **isn't compliant** with the Fork Choice specification can cause **security issues**.

We can not expect all Fork Choice implementations are formally verified since this is a challenging and costly endeavor.

The approach we take in replacement to formal verification is **extensive testing.**

Fork Choice Rule

The **head** is computed from a node state (**blocks**, **votes**, etc.) in three steps:

- filter **viable block** sub-tree
- calculate sub-tree **weights**
- recursively choose the **heaviest** sub-tree until **the leaf block** is reached

Complexities:

- many interacting nodes result in a huge variety of states
- fork choice implementations are different due to optimisations

Testing strategy

Our goal is to reveal **differences** between the Fork Choice spec and implementations.
Generating lots of interesting test scenarios is challenging.

The strategy we employ is based on the following principles:

- **differential** testing
- **property based** testing
- **model based** testing
- **fuzzing** and **randomization**

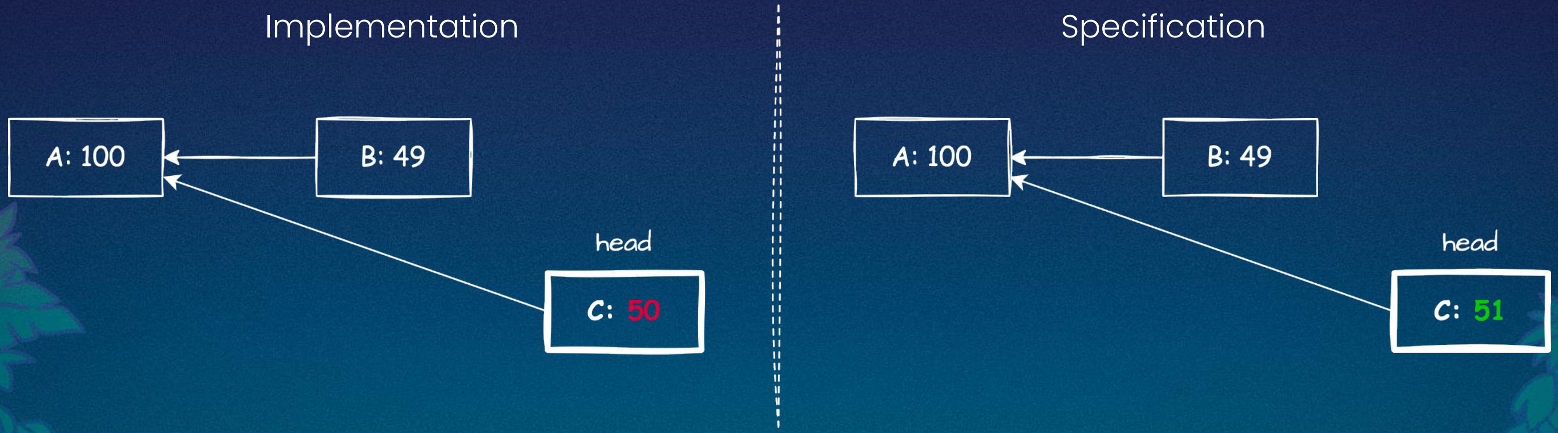
Differential property based testing

Fork choice spec is executable, so implementation behavior can be directly compared against it (**differential** testing).

The ultimate property to check is the **head** calculated by an implementation.

We can check other **properties** to reveal more bugs, using the same test suite.

Block weights



The **head** is the same in **both cases**. Direct **comparing** of the **weights** allows to find the bug without creating a test where the bug results in different **heads**.

Set of Fork Choice Properties

Our work extends the existing fork choice testing framework, which already follows differential property based approach:

- `head: Block`
- `time: Second`
- `justified_checkpoint: (Block, Epoch)`
- `finalized_checkpoint: (Block, Epoch)`
- `proposer_boost_root: Block`
- **`viable_for_head_roots_and_weights: Set[(Block, Weight)]`**

Model based testing

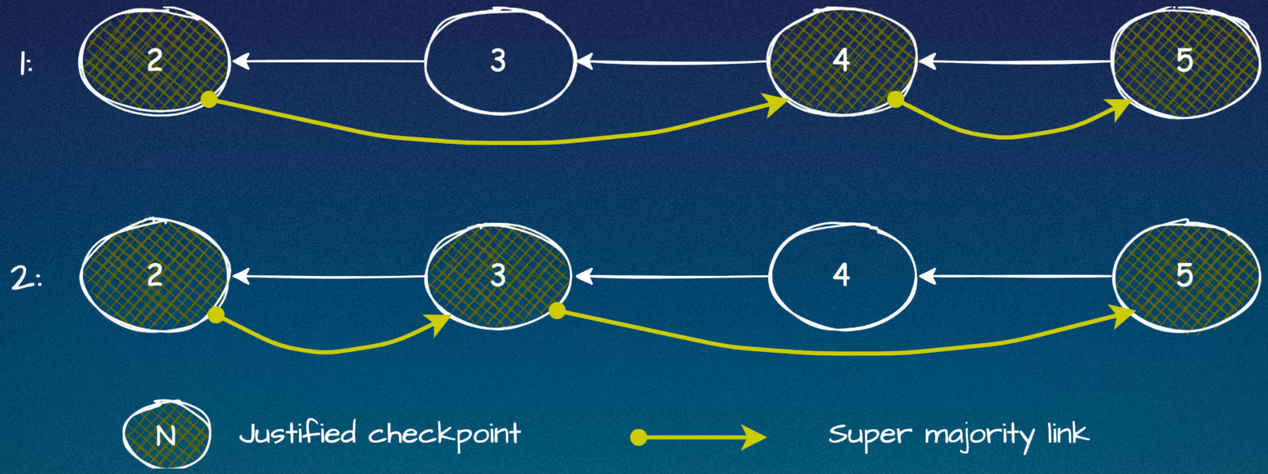
Fork Choice is executed on many nodes. Code coverage metric does not capture the complexity arising from their interaction.

We designed custom coverage criteria, which we fulfill using **model-based** test generation:

- FFG checkpoint tree shapes
- block tree shapes
- `filter_block_tree` predicate coverage

Currently, we use a **thousand** of solutions for these **three** models, but it can be arbitrarily larger (e.g. a **million** or more)

FFG Checkpoints




```
input: {anchor_epoch: 2, epochs_count: 4, super_majority_links_count: 2}
```

Fuzzing and randomization

Randomized instantiation

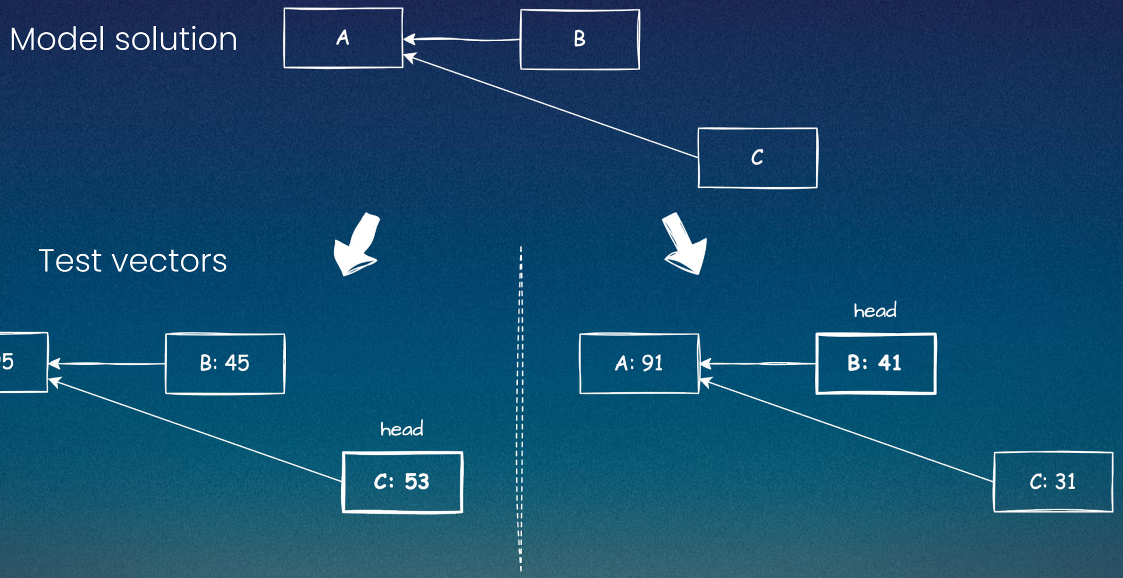
- Each model solution can be turned into a test vector in different ways

Fuzzing

- Test vector mutations (e.g. shuffling, duplication, drops, etc)
- Coverage guided fuzzing of implementations (future work)

This allows us to **increase** a number of tests by **orders of magnitude.**

Randomizing weights



One model **solution** is used to produce **multiple test vectors**

Current Status

The initial phase of the strategy is implemented.
The work was supported by a grant from EF and Consensys.

The results:

- The test generator implementation in the consensus-specs repo
- Several test suites: tiny, small and standard
  - The *standard* one contains ~13k tests (a reasonable number for the initial phase)
- Integration of the suite in Teku
  - New property check
  - 6 issues found (2 optimisations and 4 edge case scenarios)

Test suite details

Test group	size (standard suite)	parameters (solutions + variations + mutations)	description
Block tree	4096 tests	1024*2*(1+1)	focus on trees of varying shapes
Block weight	2048 tests	8*64*(1+3)	focus on producing block trees with varying weights
Shuffling	2048 tests	8*4*(1+63)	focus on shuffling/mutation operators
Attester slashing	1024 tests	8*16*(1+7)	focus on attester slashing
Invalid messages	1024 tests	8*32*(1+3)	focus on invalid messages
Block cover	3000 tests	60*5*(1+9)	cover combinations of filter_block_tree predicates

Problems

- Advanced mutations are challenging to implement
- Slow test generation (~10s per test)
  - Python is interpreted, slow spec components
  - Limits test suite size
- Some client optimisations aren’t compliant with the spec
  - e.g. non valuable attestations are discarded to protect from DoS attacks

Next steps

- Integrate the test suite into other Consensus Layer clients
- Speed up test generation
  - transpile the specification to C (Nim, Rust, etc)
  - optimise slow components
- Coverage guided fuzzing

Thank you!

Alex Vlasov

Github @ericsson49

Discord @alex.vlasov

Mikhail Kalinin

Github @mkalinin

Discord @m.kalinin