Formal Specification and Verification of the Distributed Validator Technology Protocol

Roberto Saltini
Lead Researcher, ConsenSys R&D
consensys.net/research-and-development/
The 4 questions for today

Why do we need a Distributed Validator?

Why do we need to Formally Specify and Verify the Distributed Validator Technology protocol?

How does our Formal Specification look like?

What have we achieved so far and what is left to do?
Acknowledgements

Grantors

Distributed Systems Formal Verification

Roberto Saltini  Thanh-Hai Tran
Why do we need a Distributed Validator?
What can go wrong with an Ethereum Validator?

Node Failure  →  Miss Rewards  →  Incur Penalties

P  A  · · ·  P  A

Validator Key Compromisation  →  Slashing

Validator Key

SLASHING PROOF

100 → 101  99 → 102

CONSENSYS
The solution is not as straightforward as one may think

Let’s minimise the risk of validators failing by running multiple validators

---

I have just talked to the slashed validator and we found the issue at hand. They were running another instance of their validator. Let this be a warning to you:

Do NOT run your validators in more than one place and validator instance.

---

A validator got slashed for a block equivocation, losing ~0.25 ETH. Do not try to run fancy validator redundancy that could bypass the slashing protections.

beaconcha.in/validator/20075
Nodes must be coordinated

https://github.com/ethereum/distributed-validator-specs
Why do we need to Formally Specify and Verify the Distributed Validator Technology protocol?
Now we have a distributed system to deal with

- Removed any single point of failure
- Increased the complexity
- Higher resiliency
- Higher chances of design bugs
### How can Formal Verification help us?

<table>
<thead>
<tr>
<th></th>
<th>Formal Verification</th>
<th>Testing</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Exhaustive</strong></td>
<td>Any size</td>
<td>Only small sizes</td>
</tr>
<tr>
<td><strong>Non-exhaustive</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Network size considered</strong></td>
<td>● Any size</td>
<td>● Only small sizes</td>
</tr>
<tr>
<td><strong>Byzantine behaviours considered</strong></td>
<td>● Any</td>
<td>● Only simplistic</td>
</tr>
<tr>
<td><strong>If property X is not true</strong></td>
<td>● It will be detected</td>
<td>● It may be detected</td>
</tr>
<tr>
<td><strong>If property X is true</strong></td>
<td>● Can prove it</td>
<td>● Cannot prove it</td>
</tr>
</tbody>
</table>
What does Formal Verification consist of?

**Formal Specification**
- Mathematical definition of the behaviour of a distributed protocol.
  - When message X is received, message Y should be sent

**Property Definition**
- Mathematical definition of the properties that a protocol is expected to guarantee.
  - Never commit a slashable offence

**Formal Proof**
- Mathematical Proof that the protocol specification guarantees the properties defined

**Machine-checked Formal Proof**
- Proof checkable for correctness by a computer.
  - Higher degree of confidence in the correctness of the proof

*We target this!*
How does our Formal Specification look like?
Our Formal Specification is Modular

- All written in Dafny
  - Programming Language
  - Python-like syntax
  - Formal verification aware
  - [GitHub repository](https://github.com/dafny-lang/dafny)

---

**Specification**

- Distributed Validator (Distributed System)
- Network
- Consensus
- Adversary (Byzantine Nodes)
- Single Node

**Implements**

- Single Node Executable Reference Implementation

**Network x N**
Let's start from the Executable Reference Implementation

```java
method att_consensus_decided(
    id: Slot,
    decided_attestation_data: AttestationData
) returns (r: Status)

  requires ValidRepr()
  modifies getRepr()

  if & current_attestation_duty.isPresent()
    & current_attestation_duty.safe_get().slot == id
  {
    var local_current_attestation_duty := current_attestation_duty.safe_get();

    update_attestation_slashing_db(decided_attestation_data);

    var fork_version := bn.get_fork_version(compute_start_slot_at_epoch(decided_attestation_data.target.epoch));
    var attestation_signing_root := compute_attestation_signing_root(decided_attestation_data, fork_version);
    var attestation_signature_share := rs.sign_attestation(decided_attestation_data, fork_version, attestation_signing_root);

    var attestation_with_signature_share := AttestationShare(
      aggregation_bits := get_aggregation_bits(local_current_attestation_duty.validator_index),
      data := decided_attestation_data,
      signature := attestation_signature_share
    );

    attestation_shares_to_broadcast := attestation_shares_to_broadcast[local_current_attestation_duty.slot := attestation_with_signature_share];
    network.send att share(attestation_with_signature_share, peers);
    current_attestation_duty := None;
  }

  { - check_for_next_queued_duty(); }

  return success;
```

"Exception propagation"
The Distribute Validator Formal Specification defines how the system moves from one state to the next.

**Initial State**

- \( S_0 \)

**Transition**

- **Event**
  - New set of duties
  - Time to execute the next duty
  - Message received
  -...

- \( S \) to \( S' \)
Our Formal Specification capture non deterministic behaviour

More than one event for each state

- Event: Node 1 receives message X sent by Node 2
- Event: Node 1 receives message Y sent by Node 3

Captures network asynchrony

The same event can lead to more than one state

- Event: Node 1 receives message X sent by Node 2

Allows to capture Byzantine behaviour
How is the Formal Specification written in Dafny?

```
function DVNextEvent(s: DVState, event: Event, s': DVState): bool {
    exists messagesReceived, messagesSent, decidedValue: {
        NodeNext(
            s.honest_nodes[n],
            event,
            messagesReceived,
            decidedValues,
            messagesSent,
            s'.honest_nodes[n])
    }
    NetworkNext(s.network, messagesReceived, messagesSent, s'.network)
    AdversaryNext(s.adversary, messagesReceived, messagesSent, s'.adversary)
    ConsensusNext(s.consensus, decidedValues, s'.consensus)}
```

Single node transitions

Network, Adversary and Consensus transitions

Source of non determinism
What is trusted and what is not?

How do we ensure that what we trust is correct?

- Keep trusted specification part as simple as possible
- Peer review
- Use a Formal Verification tool with strong support
- Formally proving both safety (nothing bad will ever happen) and liveness properties (something good will eventually happen) highly reduces the chances of errors in the assumptions
What have we achieved so far and what is left to do?
What have we been able to formally prove?

Assumptions

- Arbitrary message delay, including lost messages
- Beacon Nodes on completely different, but not conflicting, canonical chains
- The number of nodes that either are Byzantine or had their signing key compromised < 1/3
- Signature unforgeability and uniqueness
- Sound 2/3-threshold signature scheme

Proofs

- **No Slashable Attestations**
  Slashable attestations signed by the Distributed Validator signing key can never be created.
What have we been able to formally prove?

### Assumptions
- Arbitrary message delay, including lost messages
- Beacon Nodes on completely different, but not conflicting, canonical chains
- The number of nodes that either are Byzantine or had their signing key compromised < 1/3
- Signature unforgeability and uniqueness
- Sound ⅔-threshold signature scheme

### Proofs
- **No Slashable Attestations**
  Slashable attestations signed by the Distributed Validator signing key can never be created.

- **The reference implementation adheres to the specification**
  A Distributed Validator where each node runs the reference implementation ensures the “No Slahable Attestations” property defined above.
## What is left to do?

<table>
<thead>
<tr>
<th>Done ✔</th>
<th>To Do ⚒</th>
</tr>
</thead>
<tbody>
<tr>
<td>● Formal Specification of Distributed Attestation signing</td>
<td></td>
</tr>
<tr>
<td>● Reference Implementation for Distributed Attestation signing</td>
<td></td>
</tr>
<tr>
<td>● Formal Proof of No Slashable Attestations</td>
<td></td>
</tr>
<tr>
<td>● Simplify the trusted part of the specification</td>
<td></td>
</tr>
<tr>
<td>● Formal Specification of Distributed Block Proposing and Block Header Signing</td>
<td></td>
</tr>
<tr>
<td>● Formal Proof of No Slashable Blocks</td>
<td></td>
</tr>
<tr>
<td>● Formal Proof that the Distributed Validator eventually issues valid attestations, blocks and block header signatures</td>
<td></td>
</tr>
</tbody>
</table>
Thank You!

Roberto Saltini
Lead Researcher, ConsenSys R&D
roberto.saltini@consensys.net

www.linkedin.com/in/roberto-saltini/