Testing the Merge

Testing Ethereum with Parallel Universes

David Searle
Head of EMEA
Doctor Strange
Primary protector of Earth
....adventures in bizarre worlds
and twisting dimensions

Cher
Goddess of Pop
....If I could find a way

David
25 years in tech industry
....now professional Bug Hunter!
My house
Things I’m selling on eBay

NVIDIA GeForce RTX 3090
Founders Edition 24GB GDDR6X GPU - FE
Pre-owned
£389.00
1 bid · 6d 12h
+ £20.00 postage

MSI NVIDIA GeForce GTX 1660 SUPER VENTUS XS OC - Graphics card GPU
Refurbished
£170.00
or Best Offer
Free next day postage
Get it by Tue 4 Oct
27 watchers
115 mi

Nvidia GTX Titan Xp 12GB GDDR5X GPU
Pre-owned
£100.00
1 bid · 6d 1h
Free postage
Click & Collect
125 mi
The Day of the Merge
Or this?
In an imperfect IT world, software testing is difficult due to the **unfair expectation**...

... Fundamentally, this is **impossible requirement** because “absence of evidence is not the evidence of absence (of error, bugs, breach, etc)”...

... Software Testing is always never easy because we can't have 100% test coverage. We couldn't test **every cases possible**. And we will need to deal with time and resources constraints.

... The real issue is really **reporting bugs nicely**. Let me tell you a story where Wife is the Developer and the Husband is the QA...

... So, testers have to think about **all possible scenarios** where issues may arise and ensure they are handled by the code.

... features which have been implemented without **considering testability** in advance...
Ethereum Testing

● **Unit testing**

● **Testnets**
  ○ Shadow-forks and testnet merges
  ○ Hive, Kurtosis, Antithesis

● **Fuzzing**
  ○ Coverage guided fuzzing on multiple portions of all CLs
  ○ Modified CL and EL clients for local-testnets and attack-nets.

● **Static Analysis**
  ○ manual code audits of clients, code queries (semgrep, codeql..)
Distributed Systems...
So what did we do…

- **Auto-generate** network, container, and other faults
- Fuzz **large systems**, not just the individual applications
- Ran all software **deterministically, across all clients at the same time**
- Used strategies to seek **rare events** (code coverage, events, log messages, etc.) and explored them as much as possible – better than random
- Used a **great exploration tool** set
Individually Tested

Consensus Clients: Lighthouse, Teku, Nimbus, Prysmatic Labs, Lodestar

Execution Clients: Go Ethereum, Nethermind, Hyperledger Besu, Erigon
One Simulation, many Parallel Universes
Test Branches: 858
Test Hours (hrs): 536
Wall Clock (hrs): 13
Edges Seen (unique): 180,802
Reducing the risk of exploits at every step
Simulating the Merge

Ethereum Chain (EL + CL)

Faults:
- Network: Partitions, Delays, Drops
- Nodes: Stops, Pause, Kills
- Threads: Pause, Release
31 years of 24x7 nonstop testing
Over 50 million edges
45 validated errors
33 logged bugs
How about an example?
Report on test suite “Ethereum - Multi-client--kilnv2__inst Fault Tolerance Experiment”

Conducted on Wednesday, July 27th, 2022 at 5:23 PM

Nightly experiment expected to run for 16 hours with the latest images in the registry matching the given tags. Client images are expected to have been instrumented as the experiment is set up to use a coverage strategy for exploring the states.

- Experiment configuration details

Test Results

- **Pass** At some point workload started successfully
- **Pass** At some point there is at least one phase 3 success
- **Pass** At some point geth reports leaving PoW stage
- **Pass** Ultimately no panics
- **Fail** Ultimately no seg faults

<table>
<thead>
<tr>
<th>Test Branches</th>
<th>Test Hours (hrs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>858</td>
<td>536</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Wall Clock (hrs)</th>
<th>Edges Seen (unique)</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>180,802</td>
</tr>
</tbody>
</table>

Counterexamples

```
timestamp: 2402.57
source: service_nimbus-client-geth-0
rollout_id: 608
state_id: 119806
vtime: 1031895033345
text: SIGSEGV: Illegal storage access. (Attempt to read from nil?)
```

- **Pass** Ultimately no impossible re-orgs
- **Pass** Ultimately no nimbus panics
- **Pass** Ultimately no lighthouse panics
- **Pass** Ultimately no prysm panics
- **Pass** Ultimately no lighthouse segfaults
- **Pass** Ultimately no lodestar segfaults
- **Fail** Ultimately no nimbus segfaults
- **Pass** Ultimately no prysm segfaults
- **Pass** Ultimately no teku segfaults

Graph showing edges seen over time.
SIGSEGV: Illegal storage access. (Attempt to read from nil?)
panic: runtime error: invalid memory address or nil pointer dereference

goroutine 266 [running]:
github.com/prysmaticlabs/prysm/beacon-chain/blockchain.(Service).notifyForkchoiceUpdate

github.com/prysmaticlabs/prysm/beacon-chain/blockchain/execution_engine.go:193

goroutine 266 [running]:
github.com/prysmaticlabs/prysm/beacon-chain/blockchain.(Service).notifyEngineIfChangedHead
If I could turn back time
Look back \textbf{n seconds} ... and turn on packet capture
But how far should you look back?
What's Next

- Post Merge Testing on Stable Branches
- EIP 4844 testing
  - Dank Sharding
  - Withdrawals
- Networking Testing Simulations (Gossip)
- API Fuzzing
- Byzantine / Malicious Client
- Merge code cleanup
Thank you!

David Searle
European Field Operations at Antithesis
david.searle@antithesis.com
Fault Injection

- Network Faults
  - Events will either partition the network, or bring down individual nodes
  - Disruptions include latency, queuing packets until the end of event, or dropping all packets
- Thread pausing
  - Short pauses randomly inserted at any block of code
- Container faults
  - Pause (and resume) or kill (and restart) individual containers at randomized intervals
- User-defined scripts
  - The Fault Injector can be instructed to run scripts contained in the software or workload container
1. Specifications & Guidance
   - Intelligent Fuzzer: Drives testing and uses the output to learn and explore

2. Software Under Test
   - 1. Start containers
   - 2. Reach ready state
   - 3. Activate fault injector
   - 4. Intelligent Fuzzer provides input to generate new testing actions
   - 5. Keep searching for what's interesting

3. Deterministic Hypervisor
   - State transitions:
     - root state
     - state 1.1
     - state 1.2
     - state 1.3
     - state 2.1
     - state 2.2
     - state 2.3
     - state 3.1
     - state 3.N
     - state X.Y
     - state 3.N
     - state 2.M
     - state 1.L

4. Reporting
   - Quickly identify assertion and test property failures

5. Analysis & State Replay
   - Perform ad-hoc analysis on logs and metadata

6. Iteration
   - Engineers use the information to fix issues, create new builds and quickly validate