Week 4 EPFsg Test & Security Notes

Guest Speaker - Mario Vegas

- Mario Vega - Ethereum Foundation Testing & Security team

Summary notes

- Edited by Chloe Zhu
- Online version: https://ab9jvcjkej.feishu.cn/docx/R1IOdEkDxoiNNHxvA6sclfITnLB

Execution Layer Testing

EVM testing

- **Main purpose**: To verify that every execution client adheres to the specification, otherwise it would cause a positive potential fork in the chain
- **Setup**: To give same input for every clients, and expect to get the same output from each client, given the same environment, prestate, hard fork activation rules
- Important characteristics of a test:
  - **Pre-state:** Entire composition of an Ethereum chain, composed of accounts with balances, nonces, code, and storage
  - **Environment:** Depending on the type of the test, the environment can specify things such as the timestamp, previous RANDAO, block number, previous block hashs, total gas limit, base fee, and hard fork activation times etc.
  - **Transaction(s):** Messages to the blockchain that perform actions on the blockchain, which contains the origin and destination accounts, ether value, gas limit, and data
  - **Post-state:** Resulting state composed of the modified or created accounts

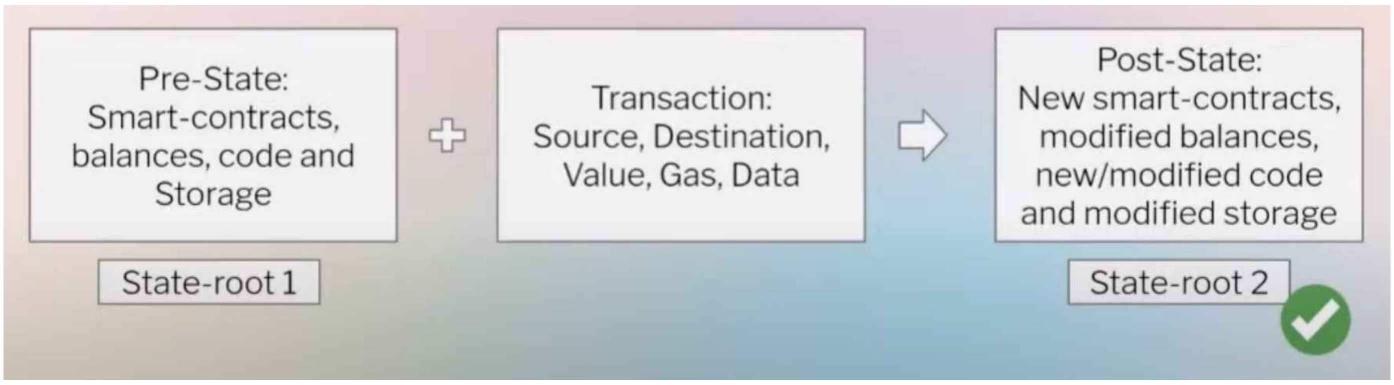
EVM testing - Tests Filling

- **Definition of Test filling:** Process of compiling a test source code into a fixture that can be consumed by any execution client
- **Test filling vs Client unit testing:** For test filling, the exact same test can be executed in any client implementation. While for client unit testing, it might vary between different client teams.

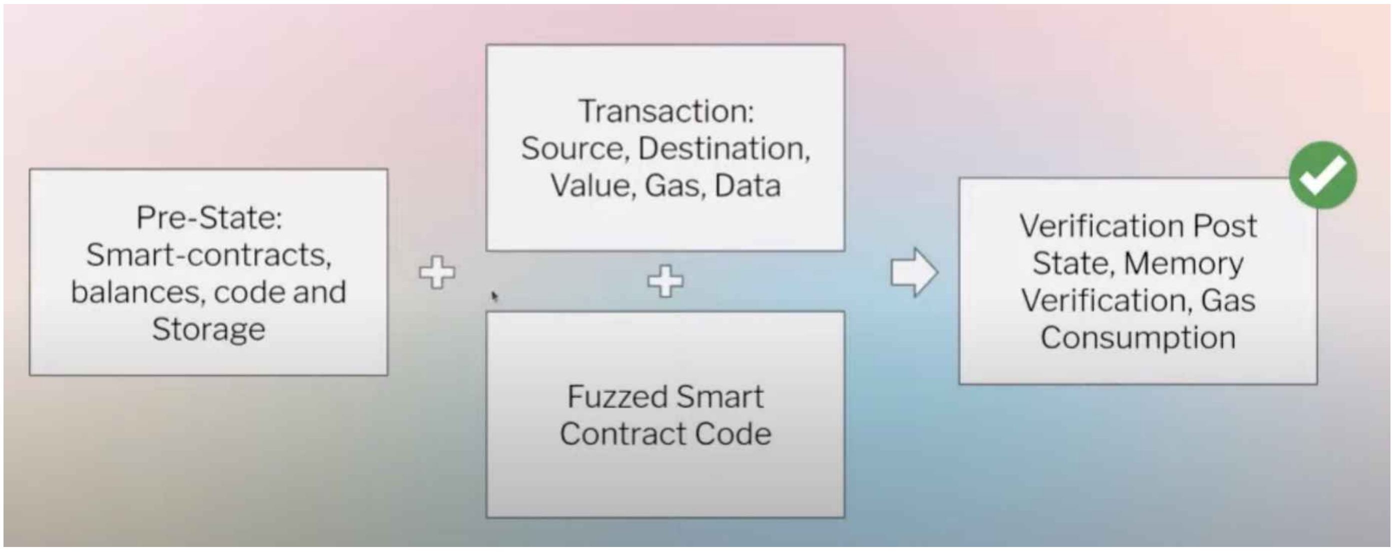
- **Feature:** All test fixtures in their different formats are single JSON files, consumable for every client

EVM testing formats

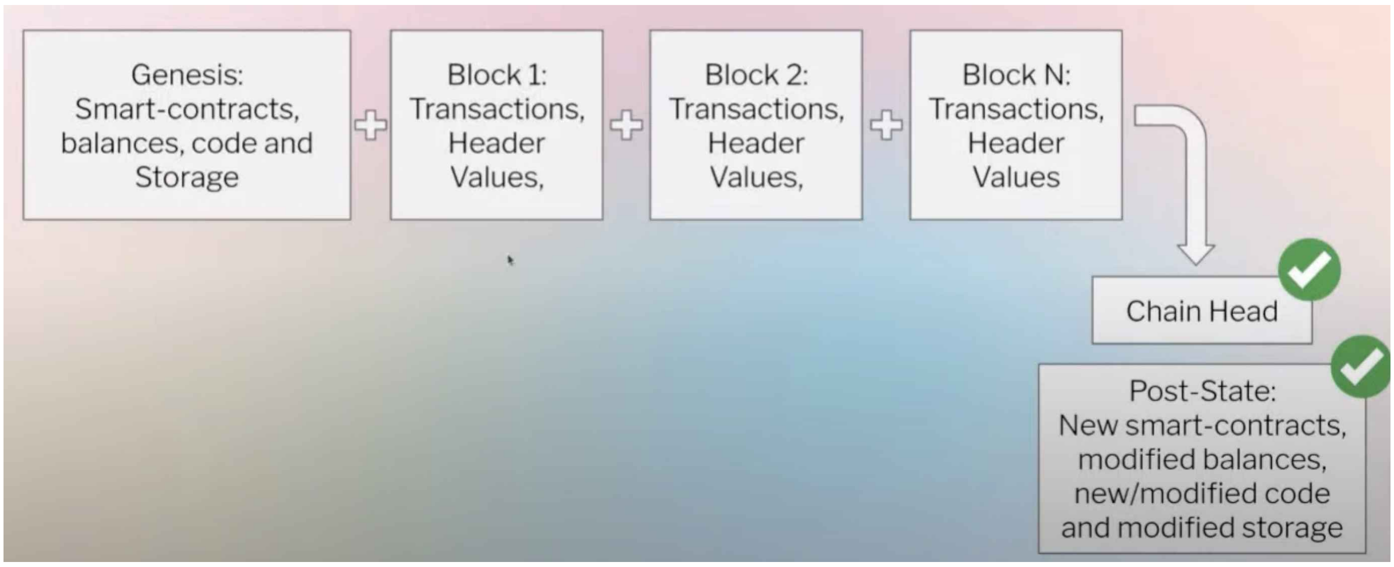
- State testing


Pre-State: Smart-contracts, balances, code and Storage
State-root 1
Transaction: Source, Destination, Value, Gas, Data
Post-State: New smart-contracts, modified balances, new/modified code and modified storage
State-root 2

  - Use the state root for verification: Given the same pre-state, and the same transaction, different clients should return the same state root
    - State root: cryptographic computation that securely commits all the contents of the state
- Fuzzy differential state testing

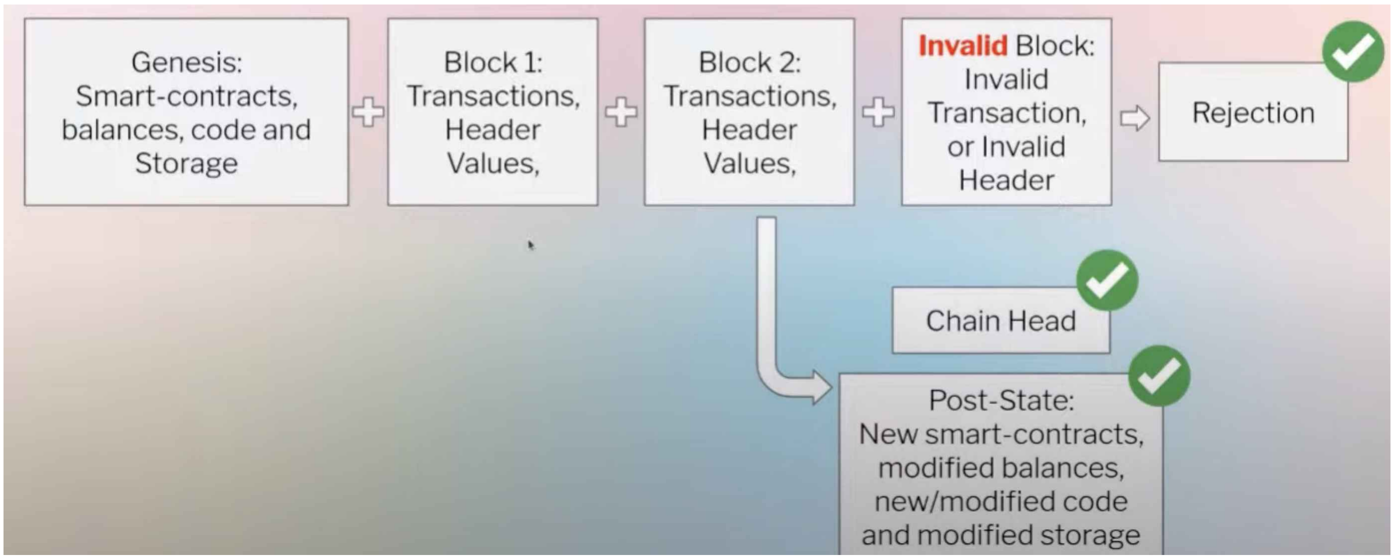

Pre-State: Smart-contracts, balances, code and Storage
Transaction: Source, Destination, Value, Gas, Data
Fuzzed Smart Contract Code
Verification Post State, Memory Verification, Gas Consumption

  - On top of the designed transaction, fuzzed smart contract code will be added using the tool FuzzyVM. Different clients are still expected to return the same state root.
- Blockchain testing


Genesis: Smart-contracts, balances, code and Storage
Block 1: Transactions, Header Values,
Block 2: Transactions, Header Values,
Block N: Transactions, Header Values
Chain Head
Post-State: New smart-contracts, modified balances, new/modified code and modified storage

- As not everything we check on the execution clients is part of the EVM execution, such as the execution result of the previous block, 1559 base fee etc., full block testing is also needed to verify clients' execution.

- Blockchain negative testing


Genesis: Smart-contracts, balances, code and Storage
Block 1: Transactions, Header Values,
Block 2: Transactions, Header Values,
Invalid Block: Invalid Transaction, or Invalid Header
Rejection
Chain Head
Post-State: New smart-contracts, modified balances, new/modified code and modified storage

  - Add an invalid block at some point to check if the clients can reject the invalid block for the designed purpose, go back to the previous valid block and claim it as the chain head

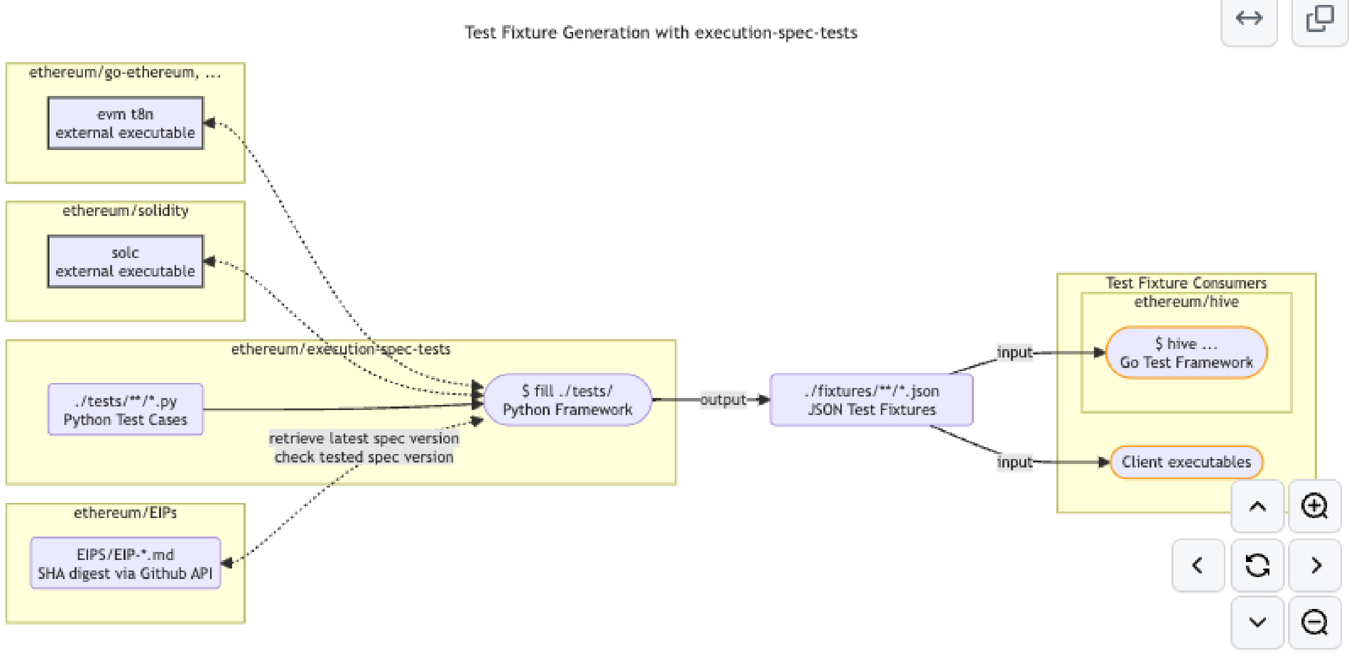
Tests Filling in details

ethereum/tests

- Repo link: https://github.com/ethereum/tests
- Features
  - Simple JSON and YAML source codes
  - Provide simple parametrization
  - Filled by Retesteth (written in C++)

ethereum/execution-spec-tests

- Repo link: https://github.com/ethereum/execution-spec-tests
- Features
    - Python source codes
    - Powered by pytest, and provide simple to complex parametrization
    - Still require an actual client implementation to fill due to the transition function
- Execution spec testing



Test Fixture Generation with execution-spec-tests
ethereum/go-ethereum, ...
evm t8n external executable
ethereum/solidity
solc external executable
ethereum/execution-spec-tests
./tests/**/*.py Python Test Cases
$ fill ./tests/ Python Framework
retrieve latest spec version
check tested spec version
output
./fixtures/**/*.json JSON Test Fixtures
input
input
Test Fixture Consumers
ethereum/hive
$ hive ... Go Test Framework
Client executables
ethereum/EIPs
EIPS/EIP-*.md SHA digest via Github API

Pic source: https://github.com/ethereum/execution-spec-tests

- Start with the **Python test** containing all the written parameterization
    - Currently there is one python test for each fork, from frontier to cancun
- **3 dependencies**
    - **Evm t8n:** Geth sub-command **evm t8n**
        - Rationale: As the test case is written in python, dev needs execution client to provide actual client implementation
        - Input to the command: All the pre-state, transactions and the environment
        - Output: The result of the execution
    - **Solidity:** (less and less used by the dev in testing)
        - Only use solidity when there is very complex code that dev cannot do with bycode writing
    - **EIP/ EIPs**: primary source of the specification when dev write a test
- **Fill command**: $ fill ./tests/

        - Input: The python test case, and the 3 dependencies above
        - Output: JSON test fixtures consumable by clients for 3 status, incl. State testing, Block testing, and Block testing in Hive
- Two main sub repos: Source & Tests
    - Src: https://github.com/ethereum/execution-spec-tests/tree/main/src
        - Source code of the framework, i.e. The code dev use to fill the tests
        - There is no test inside src
    - Tests: https://github.com/ethereum/execution-spec-tests/tree/main/tests
        - Incl. hardfork tests, from frontier to cancun
        - Since the ethereum/execution-spec-tests repo is activated from Shanghai upgrade, Shanghai & Cancun have tests for all the EIPs, while for the previous ones the full tests will be in the ethereum/tests repo

FuzzyVM

- Repo link: https://github.com/MariusVanDerWijden/FuzzyVM
- A framework to fuzz EVM implementations
    - FuzzyVM creates state tests that can be used to differential fuzz EVM implementations against each other
    - It only focuses on the test generation part, the test execution part is handled by Go evmlab

Execution APIs testing

- Repo link: https://github.com/ethereum/execution-apis/tree/main/tests
- To test all the execution APIs used to query the execution clients

Consensus Layer Testing

- Repo link: https://github.com/ethereum/consensus-specs/tree/dev/tests

Features

- Similar idea in terms of generating test fixtures in different formats that all clients can consume
- Self-contained within the spec, hence the tests can be written & filled in the same repo, and not dependent on any CL client
- Written in Python

CL testing formats

- Repo: https://github.com/ethereum/consensus-specs/tree/dev/tests/formats
- There are more CL testing formats than that of EVM's. And it's useful for dev to test granularly every aspect of the CL.

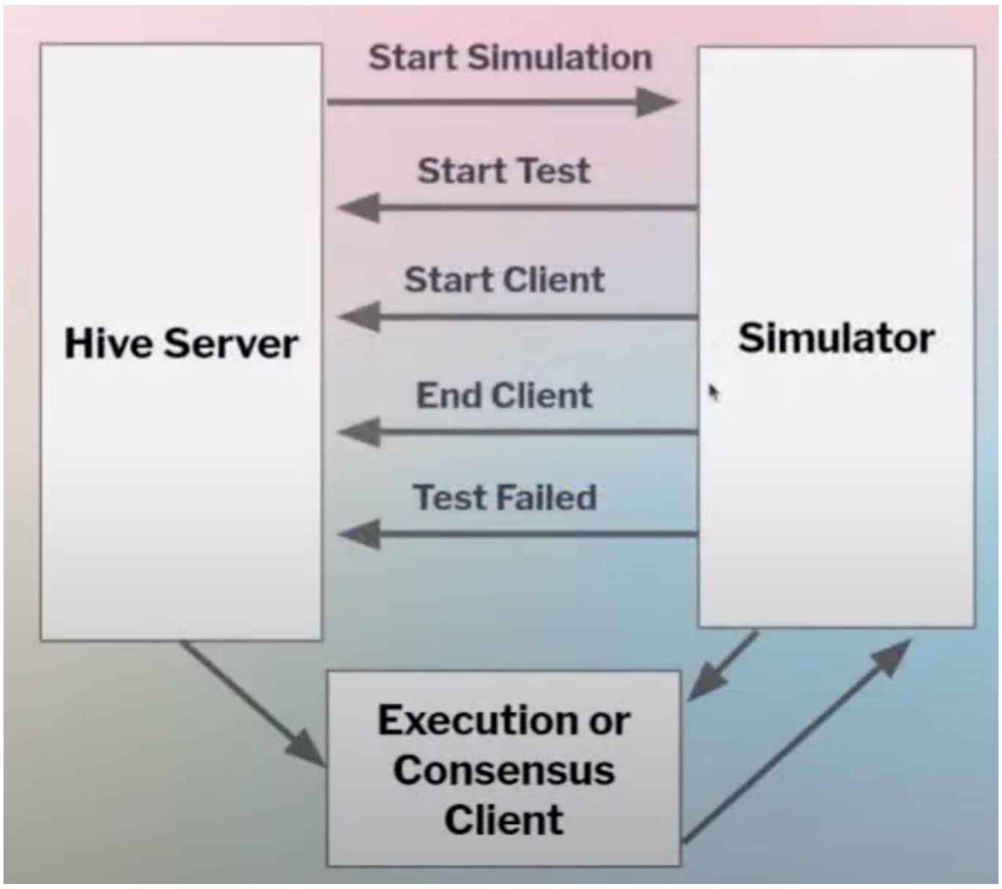
Cross-Layer (Interop) Testing

Feature

- Involve testing a fully instantiated client, feeding info to it and verifying the correctness of its behavior
- Basically, it's building the chain from Genesis to some point, then verify if all the interactions have happened correctly between EL and CL

Hive

- Repo link: https://github.com/ethereum/hive
- Hive is a system for running integration tests against Ethereum clients
    - What makes hive different from other, generic CI infrastructure is the tight integration of Ethereum clients and their features.
    - Work flow of Hive
        - Build the Hive server and start it
        - The Hive server will start a given simulator, which contains all the instructions on how to run the test. A simulator's job is to know when to start & end the test, how, when to start& end a client, etc.



- Different Hive simulators
    - Repo link: https://github.com/ethereum/hive/tree/master/simulators

Devnets

- Limited node count chains that are used to verify proof of concept or early stages of hardforks

Shawdow forks

- Limited node count forks that are configured to follow Ethereum mainnet, but have an early hardfork configuration time to test real network activity

Public Testnets

- Goerli testnet (Deprecated)
- Sepolia testnet (Launched in Oct 23rd, 2021)
- Holesky testnet (Launched in Sep 28th, 2023)

Security

Potential issues

EL side

- Valid invalidation: Execution client invalidates a block that fully complies with the Ethereum specification
- Invalid validation: Execution client validates a block that doesn't comply with the Ethereum specification
- DoS during block execution: A client takes too much time to process a block due to a transaction

CL side

- Faulty clients and finalization
    - <33% faulty node majority: can cause missed slots but chain will still finalize
    - 33%+ faulty node majority: can cause delayed finality
    - 50%+ faulty node majority: can disrupt forkchoice
    - 66%+ faulty node majority: can finalize an incorrect chain

Bug bounties

- Link: https://ethereum.org/en/bug-bounty/

Public disclosures

- Link: https://github.com/ethereum/public-disclosures

Q&A

- As the EVM testing needs Geth implementation to fill the test, what if there is a bug in Geth's code or how to ensure there is no bug?
    - Ideally, devs don't want to depend on Geth's implementation. Devs are currently working on another specs-oriented repo, so that in the future the test filing will not be dependent on Geth.
- Which is the most complex part to test in EL/ CL? How much time it takes to run all these tests?
    - Interopperability is the most complex part.
    - EVM is also complex in its own way as there are lots of nuances during execution, which needs multiple back and forth testing.
    - Regarding the time, it depends on the hardware running the test. Currently, it takes 5-10 mins to run the execution spec test in parallel
- How to communicate bugs to the client teams?

    - It depends on the severity of the bug. If the bug affects any live network, it's gonna be handled with caution such as communicate to the specific dev or through special communication.