Ethereum Blockchain Workshop

Decentralized applications

Tuesday, February 27, 2018
John Marquez

Part 1: 1h 30min

Table of Contents

1. **Ethereum Introduction**
    a. The Ethereum Blockchain
        i. EVM - Execution Cycle
        ii. State Transition Cycle
        iii. Block Finalization
        iv. Compute a Valid State
        v. Ethereum Network (PoW)
        vi. Mining Network
2. **DApps Introduction**
    a. Decentralized Applications
    b. Contract Oriented Programming in Solidity
        i. Smart Contracts
3. **DApp Development - Getting Started**
    a. DApp Architecture
        i. Dev Environment Setup
        ii. Solidity Compiler

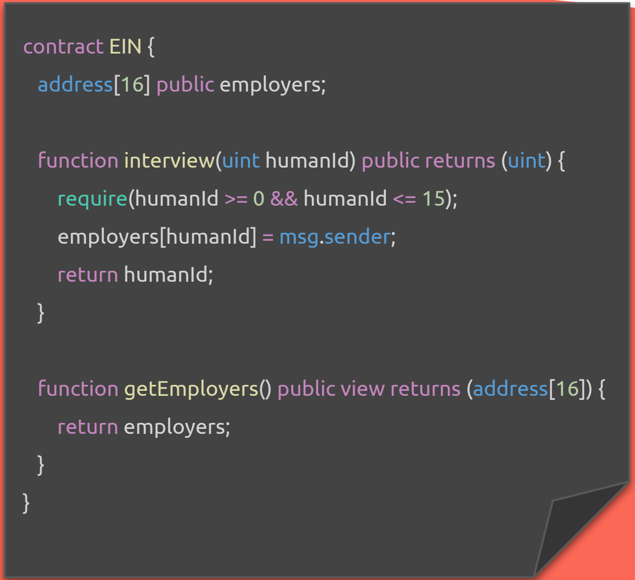
Ethereum

Introduction

The Ethereum Blockchain

- Released 2015
- Supports Turing-complete smart contracts
- Ether (ETH) currency

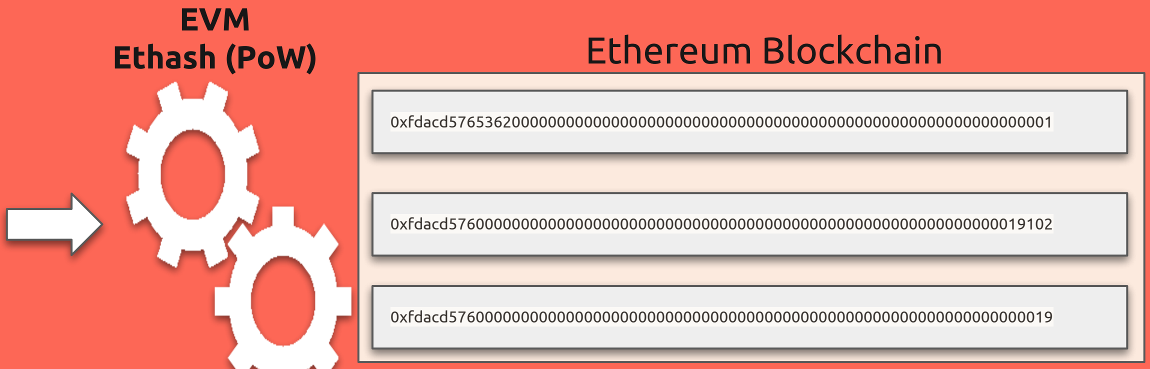
- Founded by Vitalik Buterin
- ‘Ethash’ - hash algorithm
- Uses the Proof-of-work (PoW) protocol



```
contract EIN {
  address[16] public employers;

  function interview(uint humanId) public returns (uint) {
    require(humanId >= 0 && humanId <= 15);
    employers[humanId] = msg.sender;
    return humanId;
  }

  function getEmployers() public view returns (address[16]) {
    return employers;
  }
}
```



EVM
Ethash (PoW)

Ethereum Blockchain

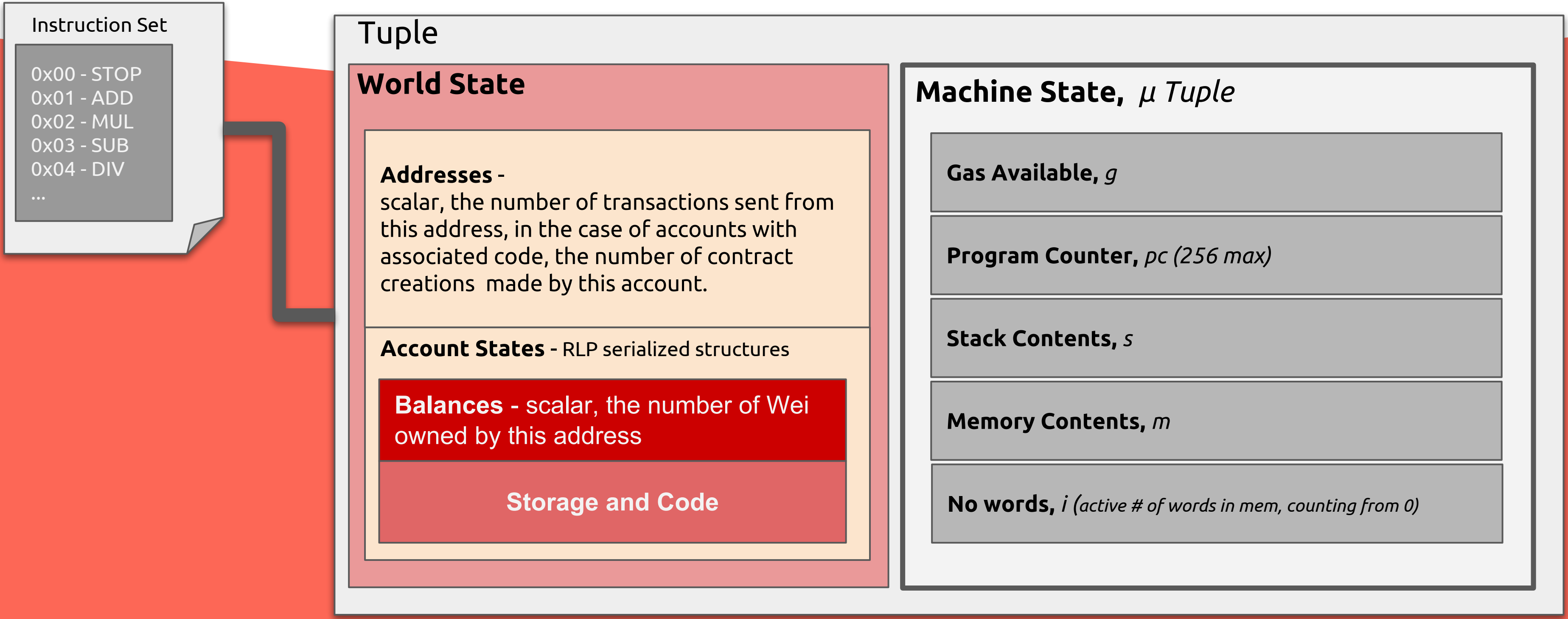
0xfdacd5765362000000000000000000000000000000000000000000000000000001
0xfdacd576000000000000000000000000000000000000000000000000000019102
0xfdacd576000000000000000000000000000000000000000000000000000000019

Block Header, Transaction List, Ommers List

Smart contracts use SHA3 for block headers

Ethereum Virtual Machine (EVM) - Execution Lifecycle

- Defines the result of a single cycle, of the state machine.



Instruction Set

0x00 - STOP
0x01 - ADD
0x02 - MUL
0x03 - SUB
0x04 - DIV
...

Tuple

World State

Addresses -
scalar, the number of transactions sent from this address, in the case of accounts with associated code, the number of contract creations made by this account.

Account States - RLP serialized structures

Balances - scalar, the number of Wei owned by this address

Storage and Code

Machine State, *μ Tuple*

Gas Available, *g*

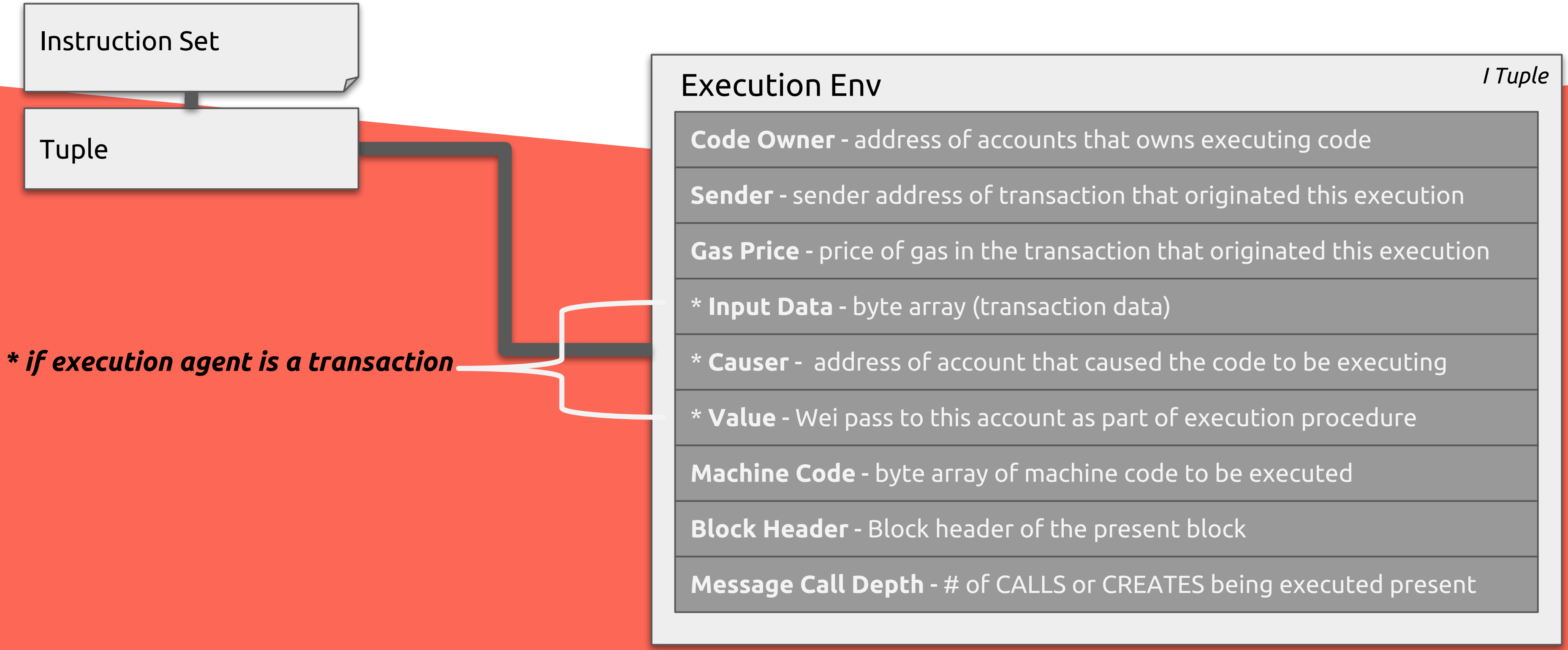
Program Counter, *pc (256 max)*

Stack Contents, *s*

Memory Contents, *m*

No words, *i (active # of words in mem, counting from 0)*

EVM - Execution Lifecycle



Instruction Set

Tuple

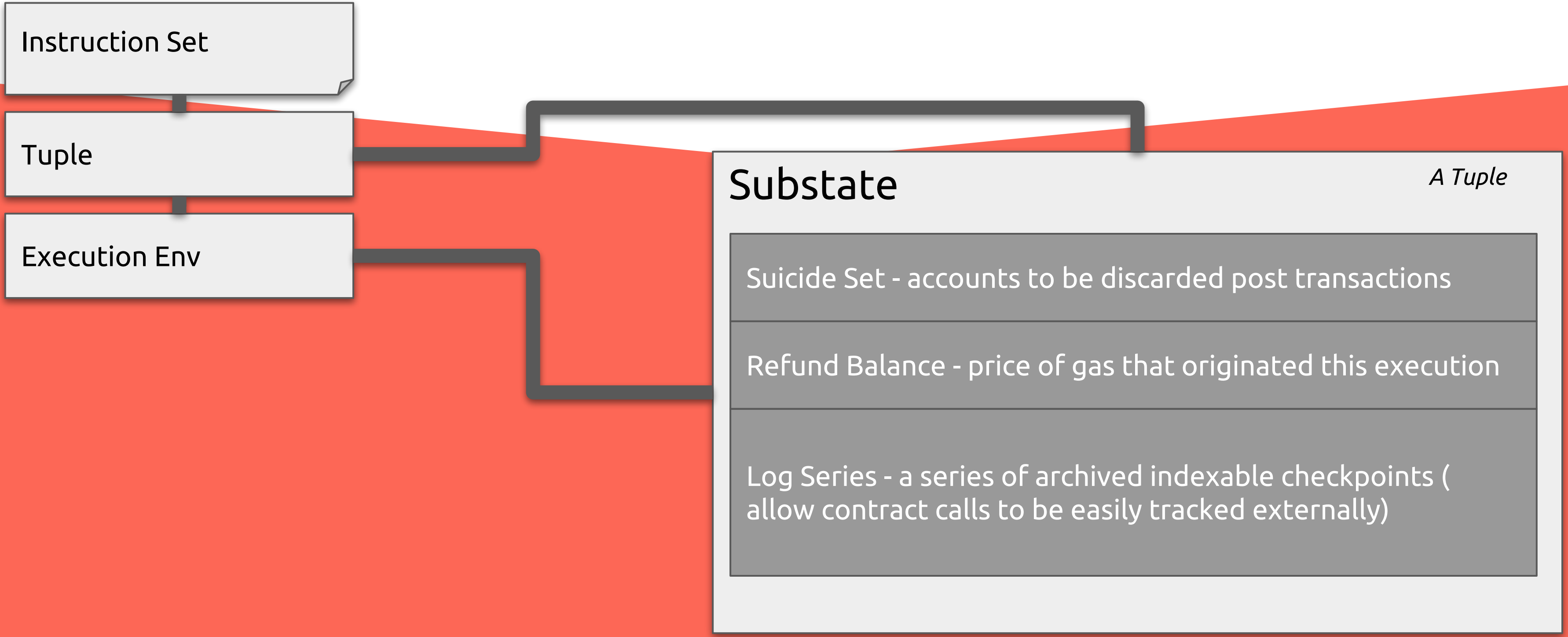
Execution Env

/ Tuple

- **Code Owner -** address of accounts that owns executing code
- **Sender -** sender address of transaction that originated this execution
- **Gas Price -** price of gas in the transaction that originated this execution
- * **Input Data -** byte array (transaction data)
- * **Causer -** address of account that caused the code to be executing
- * **Value -** Wei pass to this account as part of execution procedure
- **Machine Code -** byte array of machine code to be executed
- **Block Header -** Block header of the present block
- **Message Call Depth -** # of CALLS or CREATES being executed present

**** if execution agent is a transaction***

EVM - Execution Lifecycle



Instruction Set

Tuple

Execution Env

Substate

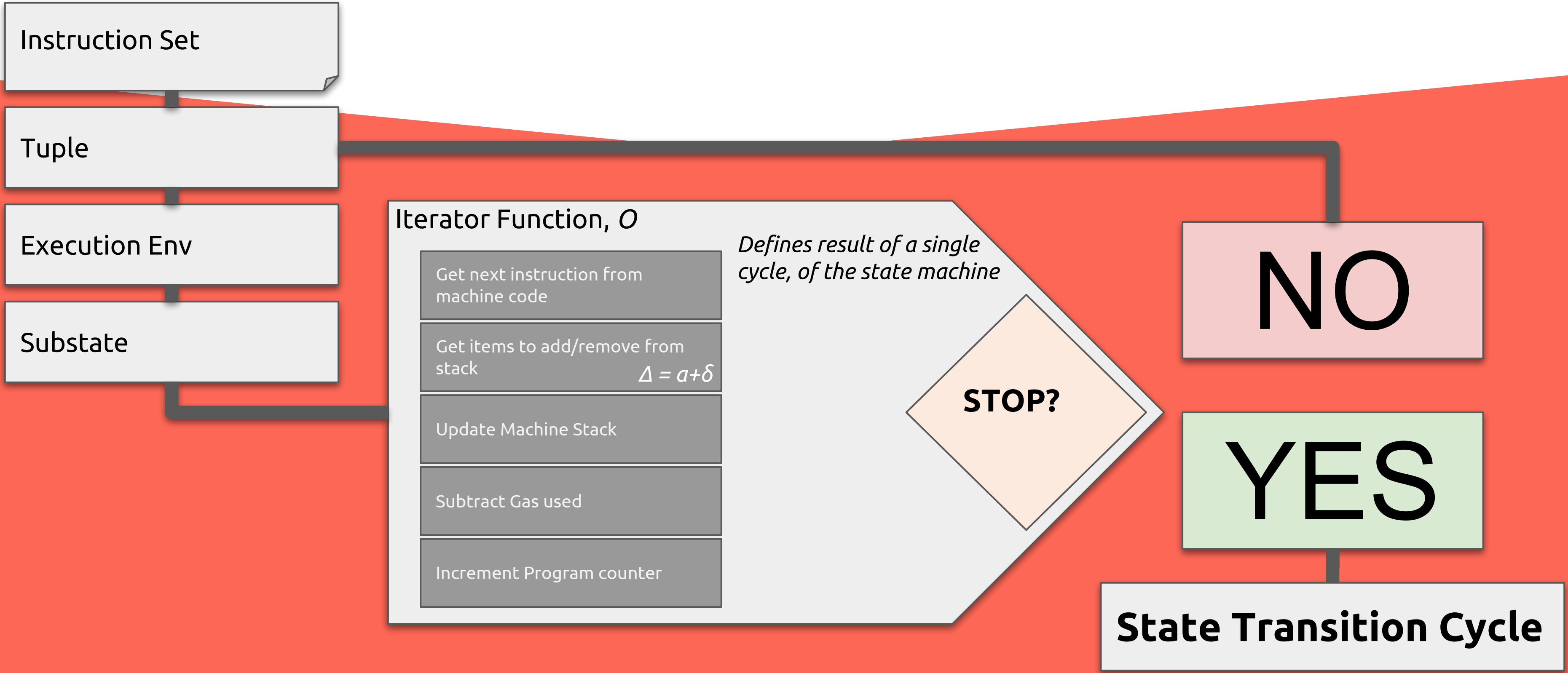
A Tuple

Suicide Set - accounts to be discarded post transactions

Refund Balance - price of gas that originated this execution

Log Series - a series of archived indexable checkpoints (allow contract calls to be easily tracked externally)

EVM - Execution Lifecycle



Instruction Set

Tuple

Execution Env

Substate

Iterator Function, *O*

Defines result of a single cycle, of the state machine

Get next instruction from machine code

Get items to add/remove from stack $\Delta = a+\delta$

Update Machine Stack

Subtract Gas used

Increment Program counter

STOP?

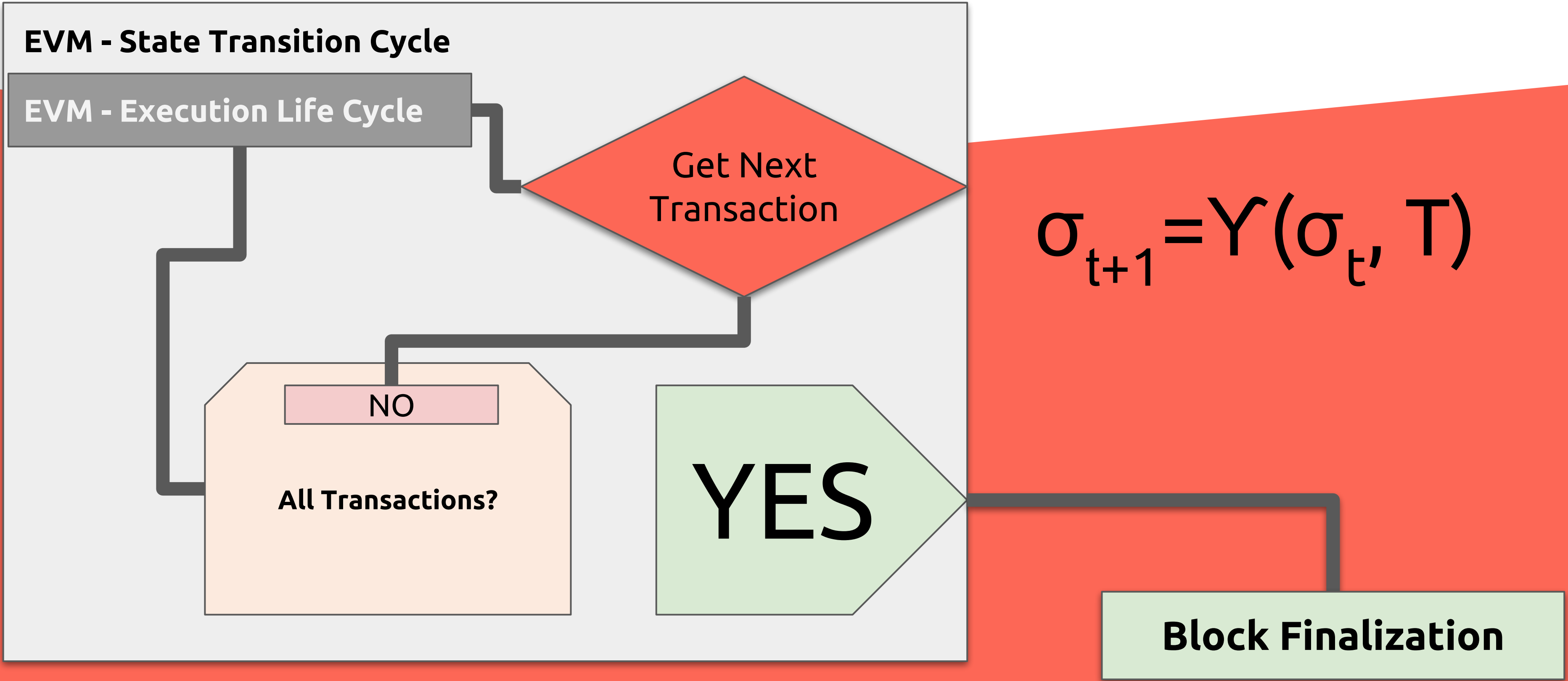
NO

YES

State Transition Cycle

EVM - State Transition Cycle

- Defines the result of a single transaction.



EVM - State Transition Cycle

EVM - Execution Life Cycle

Get Next Transaction

NO

All Transactions?

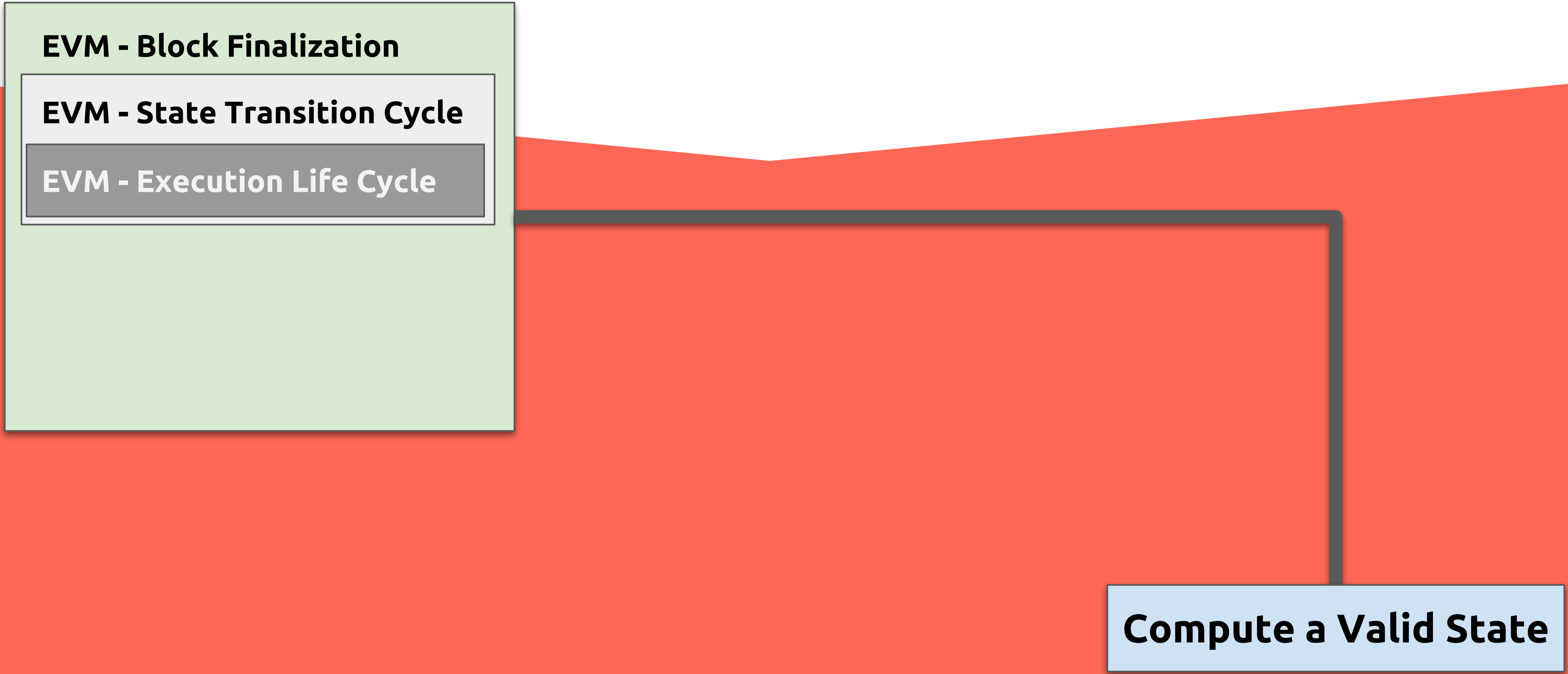
YES

Block Finalization

$$\sigma_{t+1} = \Upsilon(\sigma_t, T)$$

EVM - Block Finalization

- Defines the result of all selected state transitions.



EVM - Block Finalization

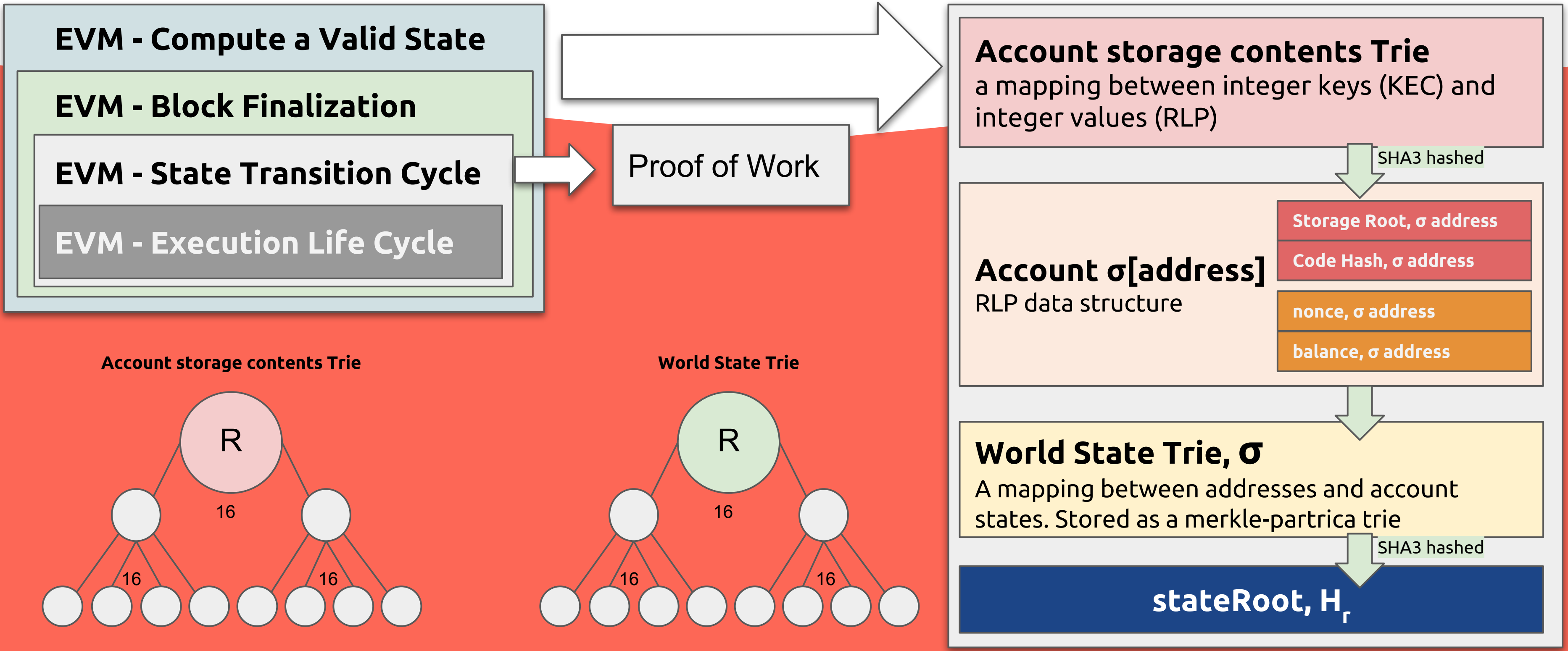
EVM - State Transition Cycle

EVM - Execution Life Cycle

Compute a Valid State

EVM - Compute a Valid State

- Information required to derive a block header



EVM - Compute a Valid State

EVM - Block Finalization

EVM - State Transition Cycle

EVM - Execution Life Cycle

Proof of Work

Account storage contents Trie
a mapping between integer keys (KEC) and integer values (RLP)

SHA3 hashed

Account σ[address]
RLP data structure

Storage Root, σ address

Code Hash, σ address

nonce, σ address

balance, σ address

World State Trie, σ
A mapping between addresses and account states. Stored as a merkle-partrica trie

SHA3 hashed

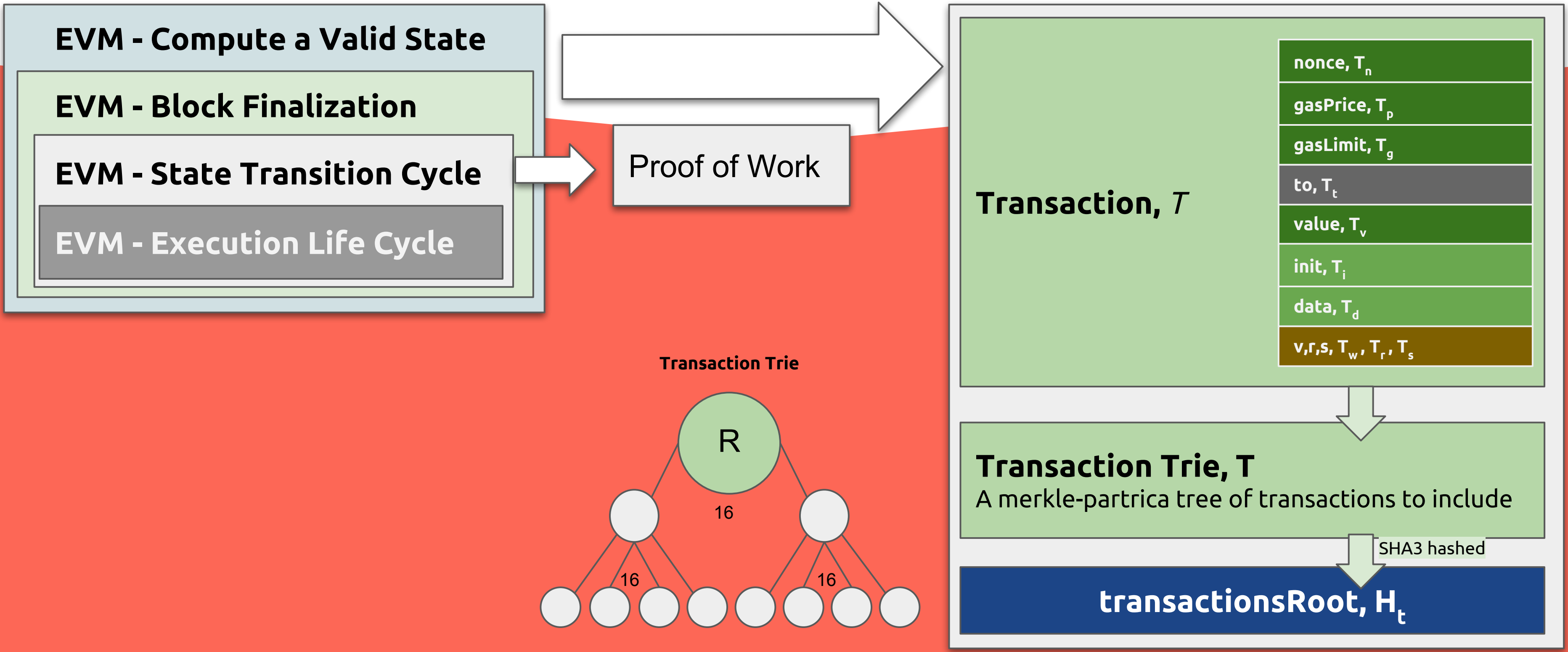
stateRoot, H_r

Account storage contents Trie

World State Trie

EVM - Compute a Valid State

- Information required to derive a block header



EVM - Compute a Valid State

EVM - Block Finalization

EVM - State Transition Cycle

EVM - Execution Life Cycle

Proof of Work

Transaction Trie

Transaction, *T*

nonce, T_n
gasPrice, T_p
gasLimit, T_g
to, T_t
value, T_v
init, T_i
data, T_d
v,r,s, T_w, T_r, T_s

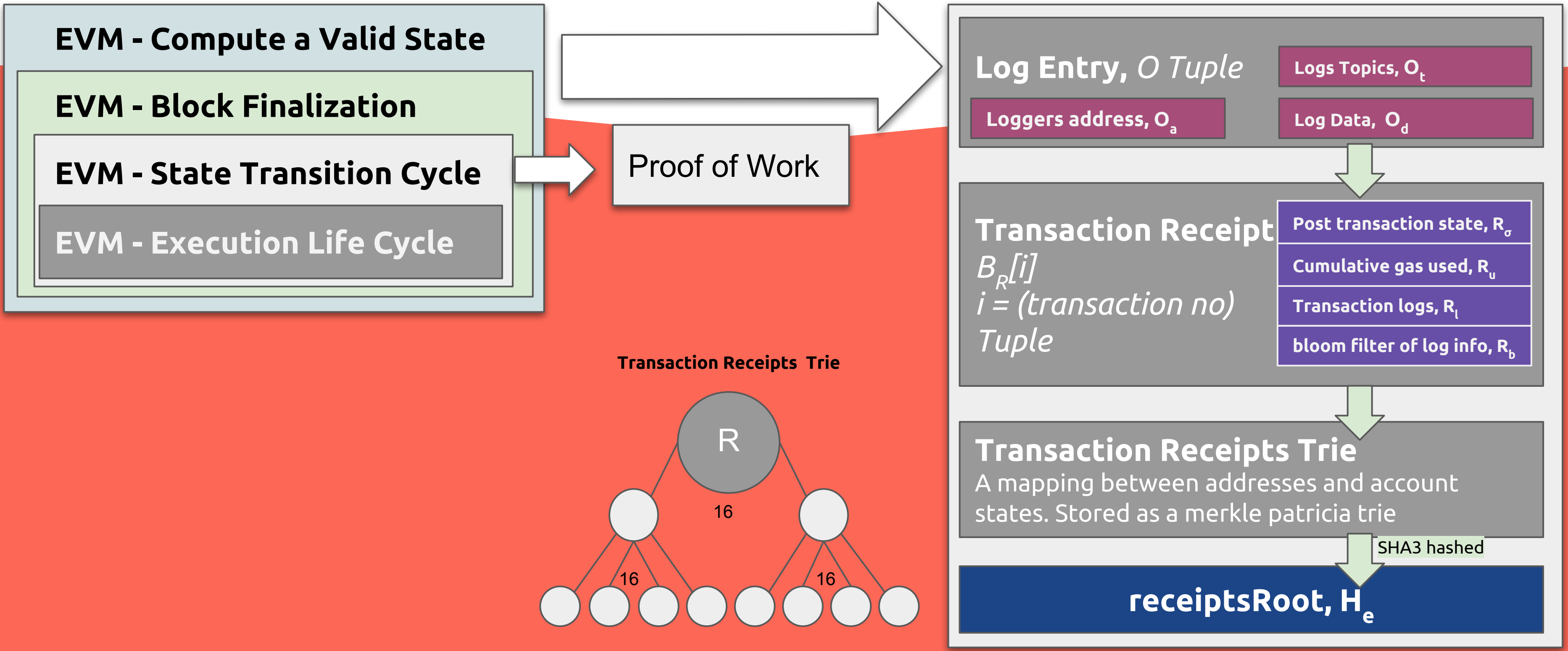
Transaction Trie, T
A merkle-partrica tree of transactions to include

SHA3 hashed

transactionsRoot, H_t

EVM - Compute a Valid State

- Information required to derive a block header



EVM - Compute a Valid State

EVM - Block Finalization

EVM - State Transition Cycle

EVM - Execution Life Cycle

Proof of Work

Transaction Receipts Trie

Log Entry, *O Tuple*

- Logs Topics, O_t
- Loggers address, O_a
- Log Data, O_d

Transaction Receipt

$B_R[i]$
i = (transaction no)
Tuple

- Post transaction state, R_σ
- Cumulative gas used, R_u
- Transaction logs, R_l
- bloom filter of log info, R_b

Transaction Receipts Trie

A mapping between addresses and account states. Stored as a merkle patricia trie

SHA3 hashed

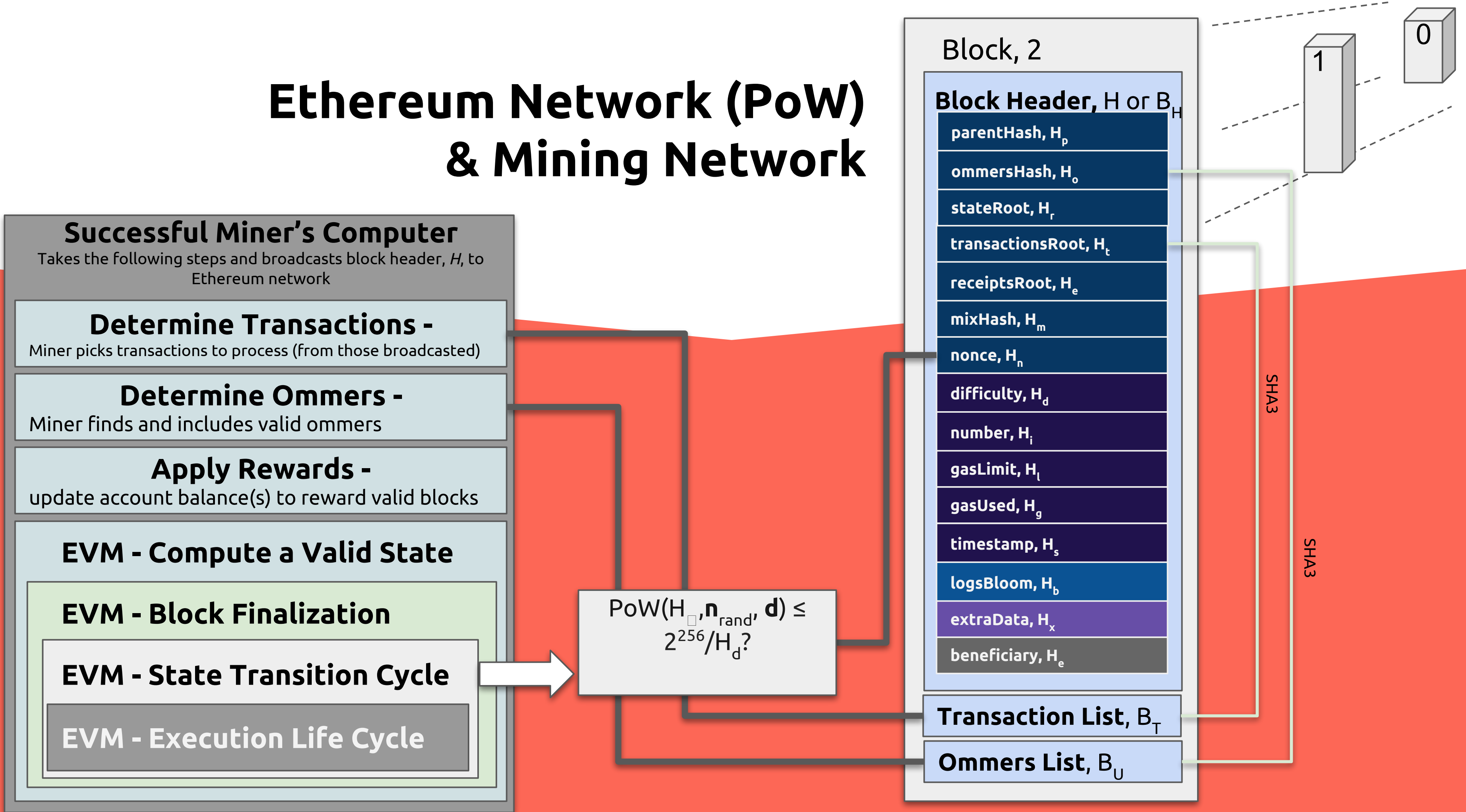
receiptsRoot, H_e

What Is Proof of Work?

$$\mathrm{PoW}(H_{\square}, H_{n}, \mathbf{d}) =$$

$$\{\mathbf{m}_{c}(\mathbf{KEC}(\mathbf{RLP}(L_{H}(H_{\square}))), H_{n}, \mathbf{d}), \mathbf{KEC}(\mathbf{s}_{h}(\mathbf{KEC}(\mathbf{RLP}(L_{H}(H_{\square}))), H_{n}) + \mathbf{m}_{c}(\mathbf{KEC}(\mathbf{RLP}(L_{H}(H_{\square}))), H_{n}, \mathbf{d}))\ \}$$

Ethereum Network (PoW) & Mining Network



Successful Miner's Computer

Takes the following steps and broadcasts block header, *H*, to Ethereum network

Determine Transactions -
Miner picks transactions to process (from those broadcasted)

Determine Ommers -
Miner finds and includes valid ommers

Apply Rewards -
update account balance(s) to reward valid blocks

EVM - Compute a Valid State

EVM - Block Finalization

EVM - State Transition Cycle

EVM - Execution Life Cycle

PoW($H_{\square}$,$\mathbf{n}_{rand}$, **d**) ≤ $2^{256}/H_d$?

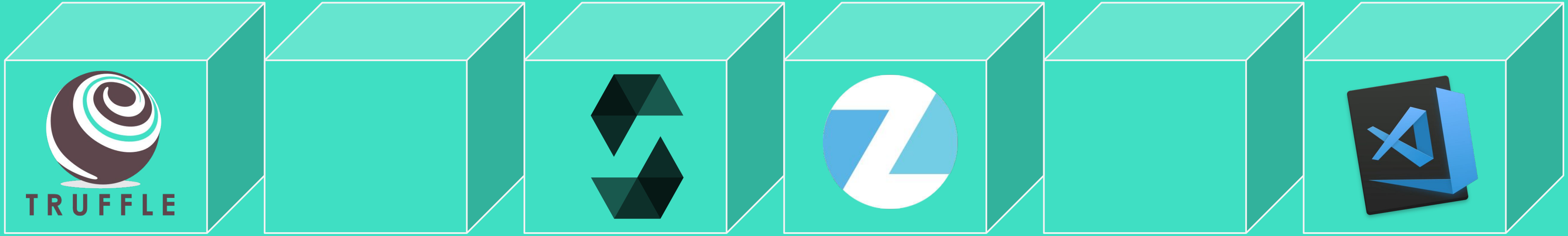
Block, 2

Block Header, H or B_H

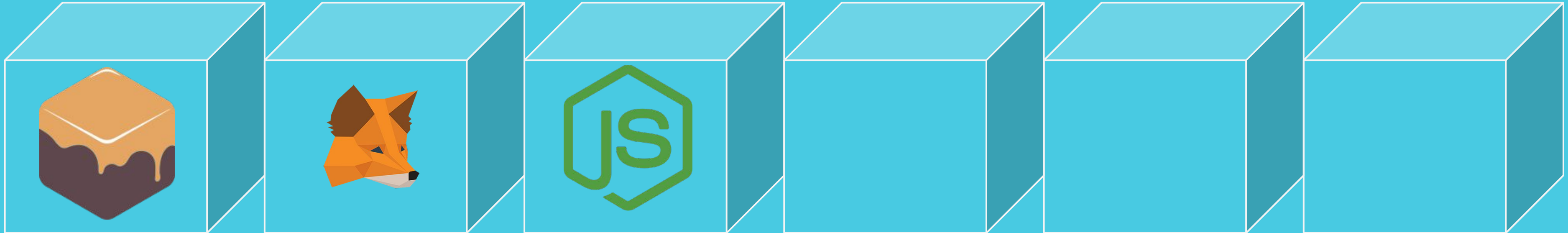
- **parentHash, H_p**
- **ommersHash, H_o**
- **stateRoot, H_r**
- **transactionsRoot, H_t**
- **receiptsRoot, H_e**
- **mixHash, H_m**
- **nonce, H_n**
- **difficulty, H_d**
- **number, H_i**
- **gasLimit, H_l**
- **gasUsed, H_g**
- **timestamp, H_s**
- **logsBloom, H_b**
- **extraData, H_x**
- **beneficiary, H_e**

Transaction List, B_T

Ommers List, B_U



DApps Introduction



Decentralized Applications

What are Decentralized Applications (Dapps)?

- A Dapp is an application which serves some specific purpose to its users, but which has the important property that the application itself does not depend on any specific party existing.

bitcoin steemit Dether. district0x

car|Vertical ETHERISC augur ARAGON

Contract-oriented programming in Solidity

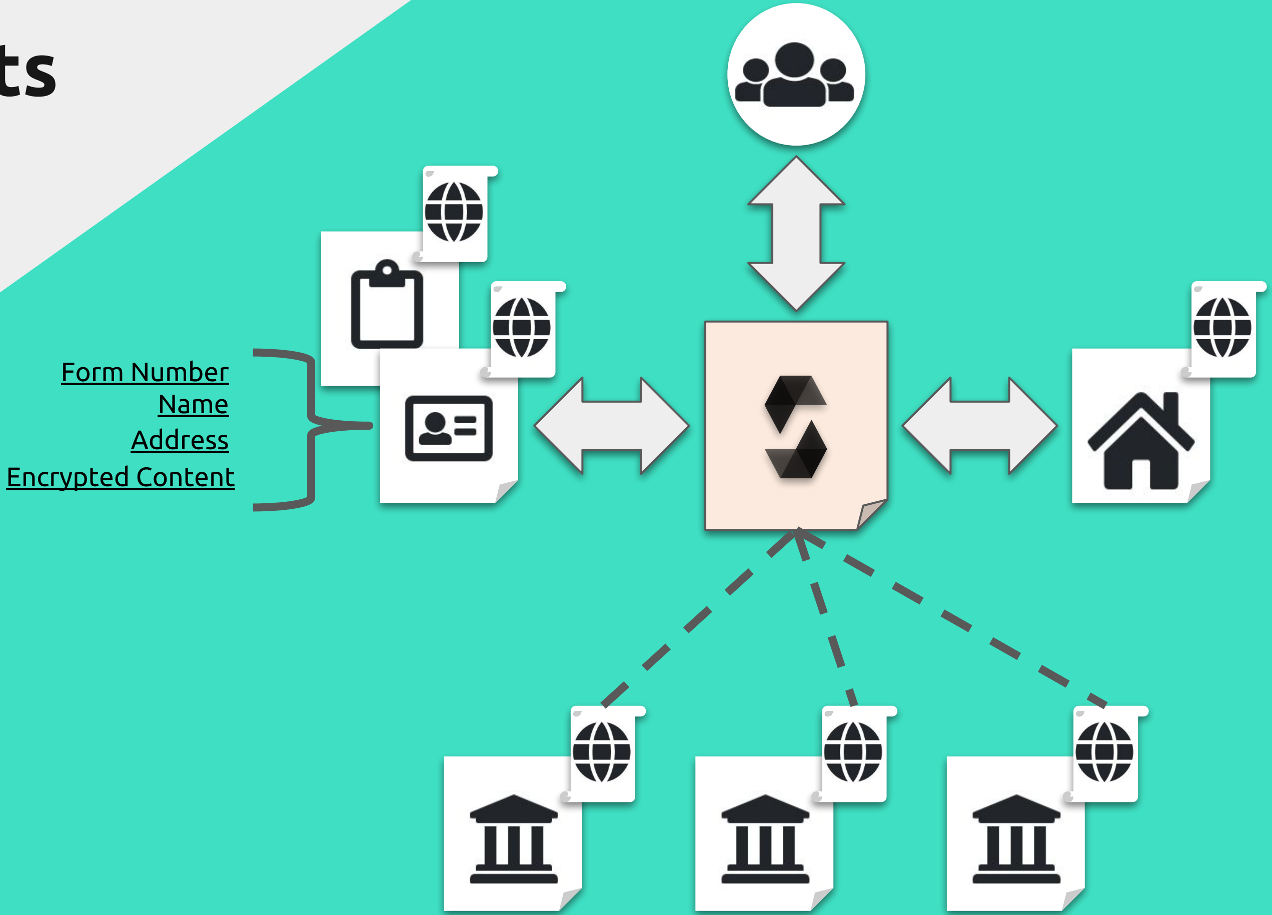
- Solidity is a contract-oriented, high-level language for implementing smart contracts. It was influenced by C++, Python and JavaScript and is designed to target the Ethereum Virtual Machine (EVM)

Principles of contract oriented programming

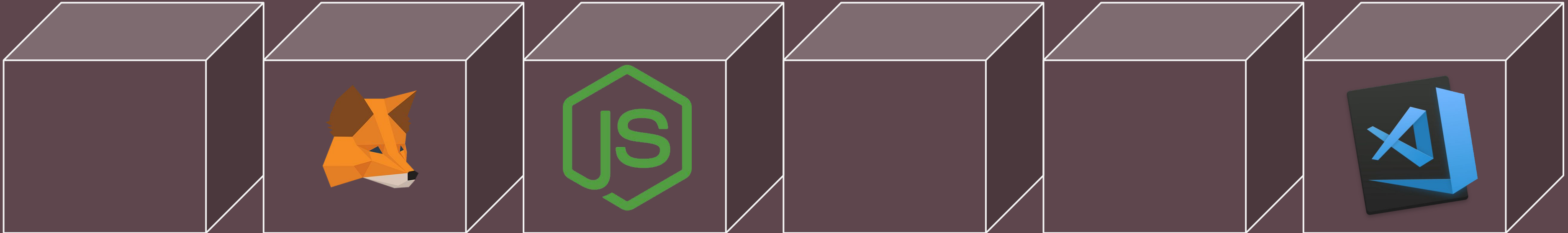
Exposing the smart contract data *(e.g. uint public)*	**Transparency**
Do not restrict interactions *(e.g. use msg.sender)*	**Scalability**
Keep smart contracts simple	**Security, Cost, Correctness**
Separation of logic and data *(e.g decentralized storage)*	**Revision**

Smart Contracts

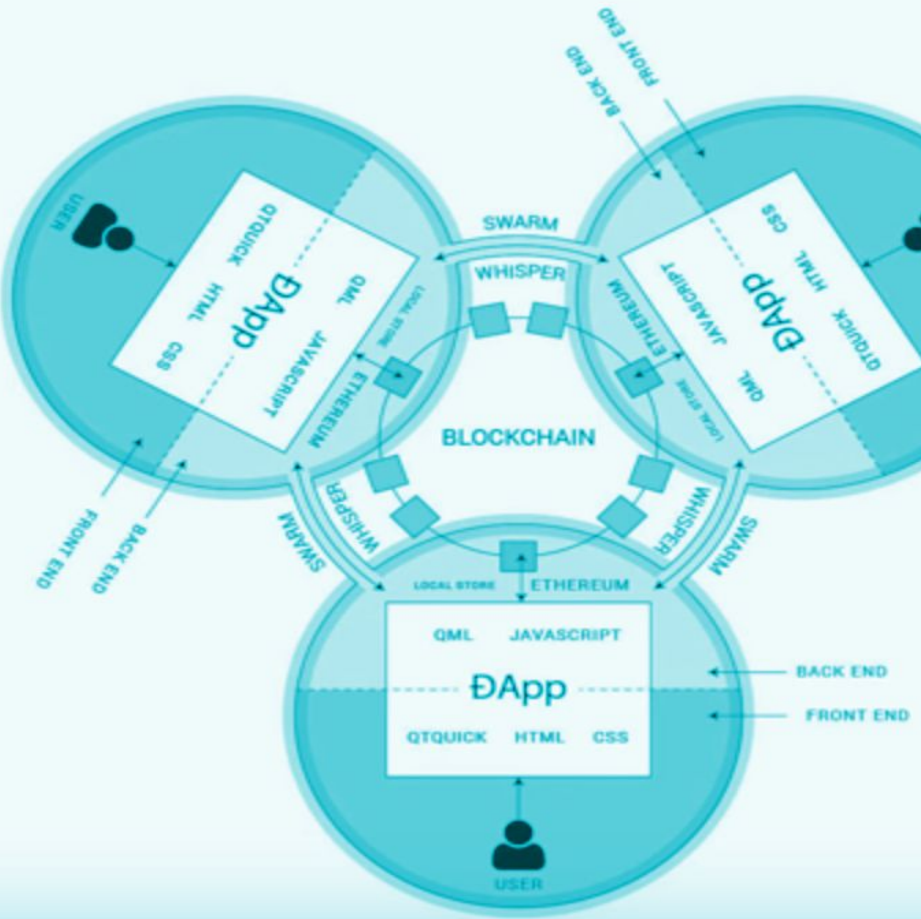
- An axiomatic system of logic that is to automatically perform steps of validation and encode resulting conditions of a physical contract



DApp Development



DApp Architecture



Dev Environment Setup

// Download and Install Node.js 8.9.4 LTS
// Download and Install an Editor or IDE *(preferably VSCode due to Solidity plugin)*
// Download and install Ganache from truffleframework.com *(i.e. local blockchain RPC client)*
// Download and install the MetaMask plugin for your browser
// In a terminal, make a directory for your workspace and make it your current directory

```
$ mkdir give-me-dapps
$ cd give-me-dapps
```

// Install the following node packages for Truffle Dev Framework and the Solidity Compiler

```
$ npm install -g truffle
$ npm install -g solc
```

// Truffle boxes are helpful boilerplates, let's unbox our dev environment

```
$ truffle unbox tutorialtoken
```

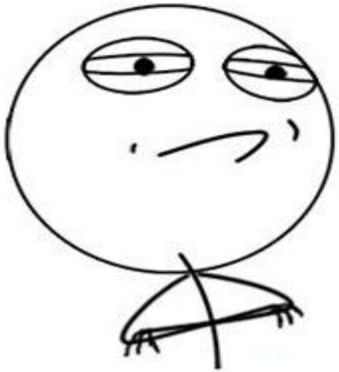
Dev Environment Setup

// Next we will be using a zepplin-solidity smart contract library that uses a security standard

```
$ npm install zeppelin-solidity
```

// Let’s now create, compile, deploy, and test our smart contract in our DApp

Note: This development is done LIVE



We just developed a decentralized application

Tuesday, February 27, 2018
John Marquez

END

Questions

?

Tuesday, February 27, 2018
John Marquez