Lesson 18:
CONTRACT VULNERABILITIES, THE ETHEREUM BLOCKCHAIN

05/05/2021
DECENTRALIZED AUTONOMOUS ORGANIZATIONS

- a term introduced by Vitalik Buterin in 2013
- DAO: like a venture capital fund for cryptos
  - operates through smart contracts
  - financial transactions and trading rules encoded on a blockchain
  - remove the need for a central governing authority
  - reduce costs
  - in theory provides more control and access to investors
“THE DAO”

- created in May 2016
- first phase: investors from anywhere in the world can send Ether to a unique wallet in exchange for DAO tokens, in 1-100 scale
  - funds are pooled
- most of the participants were notable members of the Ethereum community

<table>
<thead>
<tr>
<th>Investor</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>100 Tokens</td>
</tr>
<tr>
<td>Bob</td>
<td>10 Tokens</td>
</tr>
<tr>
<td>Total</td>
<td>110 Tokens</td>
</tr>
</tbody>
</table>
“THE DAO”

- the platform allows everyone with a project to pitch their idea to the community for funding
- if a proposal passes a preliminary's curator check, owners of DAO tokens can vote it, in proportion to their tokens
- if a proposal is approved by a quorum of 20% of all tokens
  - the DAO automatically transfer Ether to the smart contract that represents the proposal
  - stakeholders receive rewards if the project returns a profit.
SLOCK.IT: “THE DAO” LAUNCHER

- A Germany company tied to Ethereum foundation
- Some products:
  1. Airbnb user submits payment to the Ethereum blockchain
  2. Slock Home Server (Ethereum client) receives the transaction
  3. Power switch connected to Home Server receives “unlock” command, unlocks the door
• **Slack.it** launched the project of DAO: a custom fundraising tool
  • went on live on 30 April 2016
  • the largest crowdfunding in history, raising over $150 million USD from more than 11,000 members.
  • was hacked in

• “the DAO”: a complex Smart Contract with many features to reflect laws
  • respecting the stakeholders's rights
  • appraisal right
  • right of a stakeholder to require his/her shares to have back his/her funds when a proposal they do not want to be a part of gets approved despite their objection
THE DAO: VULNERABILITIES

- security issues were raised during the first weeks
  - a big community call for a moratorium
  - it was not implemented and most of the security issues were not addressed fast enough.

- on 18th June 2016, members of the Ethereum community noticed that funds were being drained from the DAO
  - the overall ETH balance of the smart contract was going down
  - a total of 3.6m Ether (about $70M at the time) was drained by the hacker in the first few hours.
SECURITY BUG: REENTRANCY

User Contract

```javascript
function moveBalance() {
    wallet.withdraw();
}
```

Wallet

```javascript
uint balance = 10;

function withdraw() {
    if (balance > 0)
        msg.sender.call.value(balance());
    balance = 0;
}
```
SECURITY BUG: REENTRANCY

User Contract

```javascript
function moveBalance() {
    wallet.withdraw();
}
```

Wallet

```javascript
uint balance = 10;

function withdraw() {
    if(balance > 0)
        msg.sender.call.value(balance());
    balance = 0;
}
```

10 ether
SECURITY BUG: REENTRANCY

User Contract

```solidity
function moveBalance() {
    wallet.withdraw();
}
```

Wallet

```solidity
uint balance = 10;

function withdraw()
{
    if(balance > 0)
        msg.sender.call.value(balance)();
    balance = 0;
}
```

10 ether

Later...

withdraw()
SECURITY BUG: REENTRANCY

User Contract

```solidity
function moveBalance() {
    wallet.withdraw();
    ...
}
```

Wallet

```solidity
uint balance = 10;

function withdraw()
{
    if (balance > 0)
        msg.sender.call.value(balance)();
    balance = 0;
}
```

10 ether

Later...

withdraw()
Security Bug: Reentrancy

User Contract

```solidity
function moveBalance() {
    wallet.withdraw();
}
...```

Later...

Wallet

```solidity
uint balance = 10;

function withdraw()
if(balance > 0)
    msg.sender.call.value(balance)();
balance = 0;
```

Can the user contract withdraw more than 10 ether?
**SECURITY BUG: REENTRANCY**

User Contract

```javascript
function moveBalance() {
    wallet.withdraw();
}

... function () payable {
    // log payment
}
```

Wallet

```javascript
uint balance;

function withdraw()
    if(balance > 0)
        msg.sender.call.value(balance)();
    balance = 0;
```

- calls the default “payable” function
- balance is zeroed after ether transfer
SECURITY BUG: RE_ENTRANCY

- this vulnerability resides in the fact that function withdraw is not reentrant
- it may misbehave if invoked before its termination
RE-ENTRANCY VULNERABILITY

- A procedure is re-entrant if its execution
  - Can be interrupted in the middle
  - Initiated over (re-entered)
  - Both runs complete without errors

- A contract, non-recursive function, re-entered before its termination
  - The attacker exploited the fallback mechanism to re-enter the function
  - The atomicity and sequentiality of transactions may induce programmers to believe that it cannot be reentered

- In general, re-entrancy may result in
  - Unexpected behaviours
  - Loops of invocations which eventually consume all the gas
RE-ENTRANCY PREVENTION

• instead of `call.value()`, use the functions `send()` or `transfer()`

• manually limit the amount of gas passed to `call.value()`
  • would not allow for recursive withdrawal calls due to the low gas stipend

• simply update the user balance prior to the transfer
  • any recursive withdraw call would attempt to transfer a balance of 0 Ether
  • a generally appliable principle: the method is safe from the re-entrancy vulnerability.

  • if no internal state updates happen after an Ether transfer or an external function call inside a method
DAO: THE HARD FORK

- a soft fork was proposed first:
  - black-listing all the transaction coming from the child DAO of the attacker

- then the Ethereum community decided for a hard fork
  - the main miners agreed to come back in the history before the attack transaction
  - they renounced to all the rewards gained after the attack
    - otherwise the Ethereum prize would have been collapsed

- Ethereum Classic
  - a set of miners considered this centralized event not suitable for a cryptocurrency
  - they went on mining on the chain containing the attack transaction, giving birth to the Ethereum Classic chain.
TIMELINE AND AFTERMATH OF THE DAO ATTACK

• June 12 2016: slock.it developers announce that a bug is found, but no funds at risk

• June 17 (Morning): attacker drains 1/3 of the DAO’s Ether ($50M) over 24 hrs
  Attacker’s funds were trapped in a subcontract for 40 days (July 27)

• June 17 (Evening): Eth Foundation proposes a “Soft Fork” to freeze the funds

• June 28: researchers publish a flaw in the Soft Fork Proposal

• July 15 (Morning): Eth Foundation proposes a “Hard Fork” to recover funds

• July 15 (Evening): “Ethereum Classic” manifesto published on github

• July 19: “Hard Fork” moves funds from attacker’s contract to recovery contract
  Ethereum Classic blockchain survives and is traded on exchanges
  Both Ethereum and Ethereum Classic are both around, reached new peaks
How $800k Evaporated from the PoWH Coin Ponzi Scheme Overnight

Three days ago, 4chan’s /biz/ banded together and created a cryptocurrency Ponzi scheme: Proof of Weak Hands Coin. They advertised it as a legitimate pyramid scheme, and surprisingly its value quickly grew to over a million dollars, and over a thousand Ethereum. A few hours ago, 866 Ethereum vanished from the contract, however, due to a flaw in the code.

- a Ponzi scheme
- attacker managed to subtract 1 to a 0-balance account
- leaving balance with a huge amount of Ether
- managed to escape with 866 Ether
pragma solidity ^0.8.0;

contract Token {
    mapping(address => uint) balances;
    uint public totalSupply;

    constructor (uint _initialSupply) {
        balances[msg.sender] = totalSupply = _initialSupply;
    }

    function transfer(address _to, uint _value) public returns (bool) {
        require(balances[msg.sender] - _value >= 0);
        balances[msg.sender] -= balances[msg.sender] - _value;
        balances[_to] = balances[msg.sender] + _value;
        return true;
    }

    function balanceOf (address _owner) public view returns (uint balance) {
        return balances[_owner];
    }
}
ARITHMETIC OVER/UNDER FLOW VULNERABILITY

• created by an operation requiring a fixed-size variable to store a number that is outside the range of the variable’s data type.

```cpp
uint8 x = 0;

uint8 y = x - 1; // y is 255!
```

• **uint**: unsigned integer of 8 bits, non negative numbers

• subtracting 1 from x results in an underflow
  • wraps around and gives as a result 255

• how can be this exploited by an attacker?
THE UNDERFLOW ATTACK

function transfer(address _to, uint _value) public returns (bool) {
    require(balances[msg.sender] - _value >= 0);
    balances[msg.sender] -= balances[msg.sender] - _value;
    balances[_to] = balances[msg.sender] + _value;
    return true;
}

• the attacker may exploit this underflow vulnerability
  • the attacker has a 0 balance
  • call the transfer function with a non-zero _value and pass the requirement
    • an underflow is generated, so the resulting value is positive
    • his/her balance, that was 0, will be credited a positive number

• prevention
  • do not use Solidity arithmetic directly!
  • use SafeMath library always
library SafeMath {

  ...

  function (sum uint256 a, uint256 b)
    public view returns (uint256) {
      assert(b <= a);
      return a - b;
    }

}
FRONT RUNNING VULNERABILITIES

- Ethereum nodes pool transactions and form them into blocks
- all the transactions are visible in the mempool for a short before they are executed, i.e. inserted in a block by a miner
  - can be “observed” by attackers
- the miner who solves the block also chooses which transactions from the pool
  - typically order them by the gasPrice of each transaction
- the attacker
  - can get the data from transactions before they are inserted in a block
  - can create a transaction of their own with a higher gas price
  - the attacker's transaction is included in a block before the original.
- “front running” comes form the finance field
  - an illegal practice of purchasing a security based on advance non-public information regarding an expected large transaction that will affect the price
pragma solidity ^0.8.0;

contract FindThisHash {
    bytes32 constant public hash =
        0xb5b5b97fafd9855eec9b41f74dfb6c38f5951141f9a3ecd7f44d5479b630ee0a;
    constructor() payable {} // load with ether
    function solve(string memory solution) public {
        // If you can find the pre-image of the hash, receive 1000 ether
        require(hash == keccak256(abi.encode(solution)));
        address payable ap = payable(msg.sender);
        ap.transfer(1000 ether);
    }
}
“FIND THIS HASH GAME” VULNERABILITY

- Alice realizes that the solution is “Hello!” and calls
  - `FindThisHash.solve (“Hello!”)` to receive 1,000 Ether
- the attacker, Bob, may be clever enough to watch the transaction pool looking for anyone submitting a solution
  - he/she sees the transaction and validates it
  - submits an equivalent transaction with a much higher gasPrice
- most likely miners will order Bob transaction before Alice's transaction
  - the attacker will take the 1,000 ether
  - Alice, who solved the problem, will get nothing.
FRONT RUNNING VULNERABILITIES

• prevention
  • upper bound on gasPrice
    • prevents users from getting preferential transactions ordering
    • does not work if the miners are the attackers
  • use a commit-reveal scheme
    • transaction sent with hidden information
    • after the transaction is included in a block, the user sends another transaction revealing the data that was sent
Security Alert – Smart Contract Wallets created in frontier are vulnerable to phishing attacks

Affected configurations: All smart contract wallets created using Ethereum Wallet Frontier, version 0.4.0 (Beta 7) or earlier. Wallets created with Ethereum Wallet all later versions released after March 3, 2016, are not affected.

Likelihood: Low
A PHISHING ATTACK

- is the fraudulent attempt to obtain sensitive information
  - usernames, passwords and credit card details
  - by appearing as a trustworthy entity in an electronic communication

- a phishing attack may be performed by exploiting the global variable `tx.origin`
  - address of the account that generated the transaction
  - refers to the original external account that started the transaction
  - `msg.sender` refers to the immediate account, external or contract account that invokes the function.
  - do not use `tx.origin` variable for authentication!
pragma solidity ^0.8.0;

contract Phishable {
    address payable public owner;

    constructor (address payable _owner) {
        owner = _owner;
    }

    fallback() external payable {} // collect ether

    function withdrawAll(address payable _recipient) public {
        require (tx.origin == owner);
        _recipient.transfer(address(this).balance);
    }
}

import "Phishable.sol";

contract AttackContract {

    Phishable phishableContract;

    address payable attacker; // The attacker's address to receive funds

    constructor(Phishable _phishableContract, address payable _attackerAddress)
    {
        phishableContract = _phishableContract;

        attacker = _attackerAddress;
    }

    fallback () external payable {

        phishableContract.withdrawAll(attacker);
    }
}
A PHISHING ATTACK

• needs social engineering

• the attacker socially engineers the victim
  • the owner of the Phishable contract
  • convince him/her to send some some amount of ether.
  • the victim, unless careful, may not notice that there is code at the attacker’s address,
A PHISHING ATTACK

- the attacker
  - deploys the AttackContract
  - convince the owner of the *Phishable* contract to send the *Attacker Contract* some amount of ether
  - the fallback function is invoked
  - in turn, the *withdrawnAll* of the victim is invoked
- the victim
  - receives a call to *withdrawnAll*
  - the address that first initialised the call was the victim, that is the owner of the Phishable contract.
  - therefore, *tx.origin* will be equal to owner and the require of the Phishable contract will pass.
  - the victim sends all the funds to the attacker
Block Timestamps used for:

- randomness
- escrowing funds
- time dependent state-changes

the victim, unless careful, may not notice that there is code at the attacker’s address,
Attack #3. Also, in this case, the attacker is a miner impersonating a player. A miner manages to join the scheme. To be the last player in the round for a minute, she uses a block timestamp. More specifically, the attacker sets the timestamp of the new block to be later the timestamp of the current block. As discussed along with the "played timestamp, she will be the last player in the round, and will win the
pragma solidity ^0.8.0;

contract Roulette {
    uint public pastBlockTime; // forces one bet per block

    constructor() payable {} // initially fund contract

    // fallback function used to make a bet
    fallback () external payable {
        require(msg.value == 10 ether); // must send 10 ether to play
        require(block.timestamp != pastBlockTime);
        // only 1 transaction per block
        pastBlockTime = block.timestamp;
        if (block.timestamp % 15 == 0) { // winner
            address payable ap = payable(msg.sender);
            ap.transfer(address(this).balance);
        }
    }
}
BLOCK TIMESTAMP MANIPULATION ATTACK

- like a simple lottery.
  - one transaction per block can bet 10 ether for a chance to win all the balance of the contract

- basic assumption:
  - `block.timestamp`’s last two digits are uniformly distributed
  - there would be a 1 in 15 chance of winning this lottery

- the attack: the miners can adjust the timestamp
  - choose a timestamp such that `block.timestamp` (or now) modulo 15 is 0.
  - in doing so they may win both
    - the Ether locked in this contract
    - the block reward.
• in practice, block timestamps

  • miners cannot choose arbitrary block timestamps

  • must be are monotonically increasing

  • block times cannot be set too far in the future

• otherwise, the block will likely be rejected by the network
BLOCK TIMESTAMP MANIPULATION ATTACK

- do not use timestamps for entropy
  - dangerous to generate random numbers
- avoid time sensitive decisions based on small timestamp differences
  - enforcing expire dates
- for time sensitive-logic use something like
  - \((\text{block.number} \times \text{average block time})\)
  - 10 seconds block time, one week corresponds to 60480 blocks
THE ETHEREUM BLOCKCHAIN

- some differences with respect to Bitcoin in the structure of the blocks of the blockchain and in the consensus algorithm

- PoW mining algorithm
  - Ethash, goal increase ASIC resistance,
  - difficulty of PoW is recomputer for each block

- consensus
  - Ghost increase mining throughput, without decreasing security

- state of the blockchain
  - more complex: Patricia Merkle trie
• similar to Nakamoto Consensus
  • ...but more complex than the simple "hash twice" of Bitcoin

• ASIC resistance
  • make it difficult, or at least unprofitable to produce ASICs to increase the level of mining performance

• from the documentation

  “the algorithm for finding a valid proof of work includes, not only changing a nonce value but also fetching pieces of data from the DAG for each try in a pseudo-random way. Every 30000 blocks this data set is recalculated”
ETHASH: A MEMORY HARD PROOF OF WORK

- relies on a pseudorandom dataset, the DAG, stored in memory
- the DAG
  - is regenerated at every epoch, 30,000 blocks (every ~5 days).
  - is initially 1Giga and continues growing in size as the blockchain grows.
  - DAG slice: 128 consecutive bytes
ETHASH: A MEMORY HARD PROOF OF WORK

- at each epoch: the algorithm self-updates its parameters
  - computes a new seed
  - uses the new seed to generate the Memory Cache and the DAG dataset
  - initial size 16M and 1 G, but they grow in time

![Diagram of ETHASH process]

- Light nodes only need to store the cache for verification.
- They can efficiently verify a transaction without storing the entire blockchain dataset.

- Each item in the dataset depends on only a small number of items from the cache.
- The dataset DAG grows linearly with time.
- Miners need to store this entire dataset DAG.
ETHASH: THE MINING ALGORITHM

- hash block header and a nonce
  - generate a first mixHash $\text{Mix}_0$
- combine $\text{Mix}_0$ and a random slice of the DAG through a mathematical function
  - obtain $\text{Mix}_1$
- repeat 63 times the mixing process
  - a memory-hard loop
  - reads random slices of the DAG
- each $\text{Mix}_i$ is used as an index of the slice to read
ETHASH: THE MINING ALGORITHM

after 63 iterations

• compare the last mixHash against the Target Threshold.

• if less than or equal
  • current nonce is successful and PoW has been solved

• otherwise
  • current nonce is invalid
  • the entire algorithm is re-run with a different nonce
WHY IS ETHASH MEMORY HARD?

- every mixing operation requires a 128 byte read from the DAG.
- hashing a single nonce requires 64 mixes, resulting in
  
  \[128 \text{ Bytes} \times 64 = 8 \text{ KB of memory read}\]
- the reads are random accesses, so putting a small chunk of the DAG in an L1 or L2 cache isn’t going to help much.
- fetching the DAG pages from memory is much slower than the mixing computation
- the best way to speed up the Ethash hashing algorithm is to speed up the 128 byte DAG page fetches from memory.
- thus, we consider the Ethash algorithm to be memory hard.
only the system's memory bandwidth limits its performance

most regular personal computers are already optimized for I/O operations

it is hard or expensive to create hardware to speed up the mining process

DAG page fetches hits the memory bandwidth limits of modern day hardware, limiting their theoretical maximum hashrate.

possibility of custom Ethereum miners?

not ASIC or FPGA based.

likely may be based on off-the-shelf chips graphic cards (mobile GPUs or VPU's),
HOW TO INCREASE THROUGHPUT?

• the solutions for Ethereum consensus

  • **block creation rate**: lower the difficulty of the computational problem to accelerate the block creation process
    • ETH reduces block confirmation time to 10 sec.
  
  • **block size**: build larger blocks to propagate
ACCELERATE BLOCK CREATION

- while the most recent block is propagated in the blockchain,
  - more blocks are being created by the honest network
- brings to more forks
  - due to the high mining frequency, not to double spending attacks
  - block are mined but discarded because they belong to forks
- lot of mining effort not contributing to extend the longest chain

Assume block 3E is the new block. While 3E is propagated in the network from its miner to other nodes, block 3C and 3D are created simultaneously by another two miners, who did not receive 3E yet. As a result, all three blocks are appended to block 2C.
• as the block size is increased, blocks naturally take longer time to propagate through the network

• hence, more forks occur

Fig. 1. The relation between the block size and the time it took to reach 25% (red), 50% (green), and 75% (blue) of monitored nodes, based on data provided by Decker and Wattenhofer [7].
FORKS: REDUCTION IN SECURITY

- more forks due to larger blocks and faster creation
- assume that the attacker is creating long chains efficiently: its blocks are always built on top of one another
- more forks implies less effort for an attacker to secretly create a longer chain
- honest miners’ efforts spread over multiple forks
  - the computational power of honest miners is not able to defeat the attacker
• **GHOST**: "Greedy Heaviest Observed Subtree"
  • a new consensus protocol
  • goal: to solve the issue of network security loss
THE GHOST CONSENSUS

- observation: blocks that are off the main chain still contribute to its weight
  - no longer the longest chain wins, but rather the heaviest sub-tree
  - “heaviest” the hardest combined PoW performed
THE GHOST CONSENSUS

- follow a path from the root of the tree (the genesis block)
- chooses at each fork the block leading to the heaviest subtree
- properties
  - convergence: each block is eventually fully abandoned or fully adopted
  - resilience to 50% attacks

![Diagram showing the Ghost Consensus process](image)

The subtree of block 1B contains 12 blocks, whereas that of 1A contains only 6. The algorithm will thus pick 1B as belonging to the main chain, and proceed to resolve the forks inside subtree (1B). This will result the choice of blocks 0, 1B, 2C, 3D, 4B as the main chain of the tree (and not the longest chain, ending in block 5B).
blocks that are not part of the best chain (stale blocks) can become uncles or ommers (gender neutral aunt/uncle) when they are referenced by blocks in the best chain.

- the references to ommers are stored in a new field in the each block header.
- a regular block may reference up to two ommers

Bitcoin: longest branch prevails

Ethereum: heaviest branch prevails
CONSENSUS WITH GHOST

- when an uncle is referenced, the branch that referenced it increases its weight as the uncle would be part of the best chain.
- the best chain is selected as the branch with higher weight.
- ormons have no only the function of contributing to the heaviest branch
  - their transaction content is ignored, a valid header, but ignored payload

---

Bitcoin: longest branch prevails

Ethereum: heaviest branch prevails
ETHEREUM REWARDS

- currently, block reward is 2 ETH (before 5 and 3 ether) + gas fees + 1/32 block reward for each uncle block referenced
- however, there is no limit an established limit of the number of ether in circulation, like in Bitcoin
- the reductions in block rewards are in an effort to reduce inflation by reducing the newly-available supply of ETH.
REWARDING OMMERS

- work of ommers should be rewarded, even if the reward should be smaller
- reward for ommers:
  - ranges from a 7/8 block reward (7/8 x 3 ether) for an immediate inclusion to a 2/8 block reward for inclusion 6 blocks later
  - transaction fees are not awarded to omens
  - no rewards for blocks that whose parent does not belong to the main chain

the miner of block C1 receives 7/8 of the reward of block B2.
DIFFICULTY ADJUSTMENT: FRONTIER RELEASE

MIN_DIFF = 131072

def calc_difficulty(parent, timestamp):
    offset = parent.difficulty // 2048
    sign = 1 if timestamp - parent.timestamp < 13 else -1
    return int(max(parent.difficulty + offset * sign, MIN_DIFF))

• if the difference between current timestamp and parent’s timestamp is less than 13 seconds, the difficulty is increased
  • otherwise, the difficulty is decreased
• quantum of change is 1/2048 of parent block’s difficulty
• difficulty is not allowed to go below a fixed minimum
DIFFICULTY ADJUSTMENT: HOMESTEAD RELEASE

- looks at the time distance between the generation of the parent block and of the current block
  - <10 seconds the difficulty goes up
  - >10 seconds and <19 seconds remains the same
  - >20 seconds or more difficulty decreases and the decrease is proportional to the time distance

- in addition: difficulty *time bomb* or *ice age*
  - spike (increase) in mining difficulty
  - incentive to reduce the number of miners
    - aimed to pre-date shift of algorithm from PoW to Proof-of-Stake (Casper)
DIFFICULTY ADJUSTMENT: BYZANTIUM RELEASE

- October 2017, Block 4370000
  
  https://github.com/ethereum/EIPs/issues/100

- takes uncles into account while adjusting difficulty

- delays the ice age by approximately 42 million seconds (18 months)
  
  - wait for Casper protocol

- block reward reduced from 5 ETH to 3 ETH
  
  - a further disincentive to PoW
THE ETHEREUM STATE

- the state has to be stored permanently on the blockchain
- block header is more complex than for Bitcoin
  - Ommers list
  - parent hash + 3 Merkle Roots (transaction, state, receipts)
- other information related to Ethereum model (gas, gas price, ommers...)
MERKLE PATRICIA TRIES

- combines the functionality of a classic Merkle trees and a Patricia ties
  - integrity checking of data stored in the structure obtained from the Merkle tree,
  - optimizations for storage, insertion, and deletion operations obtained from the Patricia tree.

- Bitcoin’s state: key value mapping addresses to account balance
- Ethereum's state: key value mapping address to account objects

<table>
<thead>
<tr>
<th>Address</th>
<th>Balance (BTC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x123456</td>
<td>10</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0x1a2b3f</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0xab123d</td>
<td>1.1</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x123456</td>
<td>X</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0x1a2b3f</td>
<td>Y</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0xab123d</td>
<td>Z</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
TYPE OF NODES IN MERKLE PATRICIA TRIES

Merkle Patricia Tree Nodes
**THE WORLD STATE TRIE**

**Block Header, H or B_H**
- stateRoot, H:
  - Keccak 256-bit hash of the root node of the state trie, after all transactions are executed and stabilizations applied

**Hash function:**
- KECCAK256()

**World State Trie**
- **ROOT: Extension Node**
  - **prefix**: 0
  - **shared nibble(s)**: a7
  - **next node**:

**Branch Node**
- **Leaf Node**
  - **prefix**: 2
  - **key-end**: 1355
  - **value**: 45.0ETH

**Extension Node**
- **prefix**: 0
  - **shared nibble(s)**: d3
  - **next node**:

**Leaf Node**
- **prefix**: 2
  - **key-end**: 9365
  - **value**: 1.1ETH

**Prefixes**
- 0 - Extension Node, even number of nibbles
- 1 - Extension Node, odd number of nibbles
- 2 - Leaf Node, even number of nibbles
- 3 - Leaf Node, odd number of nibbles
- $\square$ = 1st nibble
- 1 nibble = 4 bits

**Merkle Patricia Tree**
- (Reference from i.stack.imgur.com)

---

**Contract vulnerabilities, the Ethereum Blockchain**

Laura Ricci

Dipartimento di Informatica
Università degli Studi di Pisa
Block header contains three Merkle trees

- state root
- transactions root
- Receipts root
TRANSACTION RECEIPTS

- a mechanism to store the state after a transaction has been executed.
- contains
  - **post-transaction state**: trie structure that holds the state after the transaction has been executed.
  - **gas used**: total amount of gas used in the block.
  - **set of logs**: the set of log entries created as a result of transaction execution.
    - logger's address
    - a series of log topics
    - log data.
  - **bloom filter**: of the information contained in the set of logs.
- all receipts are stored in a Merkle Patricia trie
  - the hash (Keccak 256-bit) of the root is the receipt the receipts root.
<table>
<thead>
<tr>
<th></th>
<th><strong>Bitcoin</strong></th>
<th><strong>Ethereum</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Specification</td>
<td>Bitcoin Core client</td>
<td>Ethereum yellow paper</td>
</tr>
<tr>
<td>Consensus</td>
<td>SHA256 PoW</td>
<td>Ethash PoW (later PoS)</td>
</tr>
<tr>
<td>Contract Language</td>
<td>Script</td>
<td>EVM bytecode</td>
</tr>
<tr>
<td>Block interval</td>
<td>10 minutes</td>
<td>14 to 15 seconds (^1)</td>
</tr>
<tr>
<td>Block size limit</td>
<td>approx 4 MB</td>
<td>11 KB to 34 KB (Aug 2017 to Aug 2018) (^2)</td>
</tr>
<tr>
<td>Difficulty adjustment</td>
<td>After 2016 blocks</td>
<td>After every block</td>
</tr>
<tr>
<td>Currency supply</td>
<td>Fixed to 21 million</td>
<td>Variable (101 million in Aug 2018) (^3)</td>
</tr>
<tr>
<td>Currency units</td>
<td>1 BTC = (10^8) satoshi</td>
<td>1 ETH = (10^{18}) Wei</td>
</tr>
</tbody>
</table>