Blockchains & Cryptocurrencies

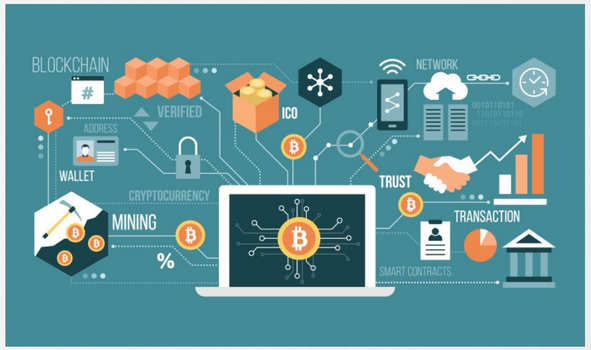
Smart Contracts III / Ethereum & Solidity

Instructor: Matthew Green & Abhishek Jain
Spring 2023

News?

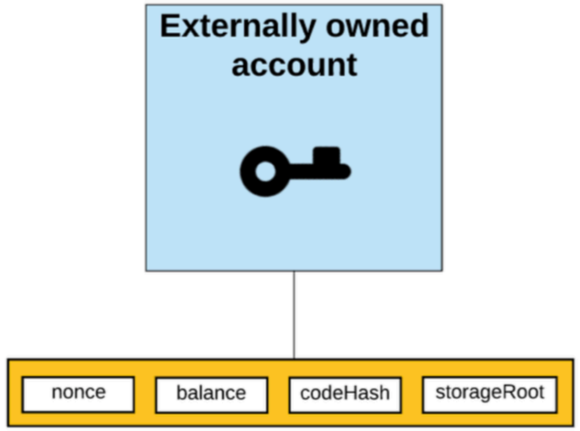
Today

- Finish Ethereum, finally
- Talk about how some actual smart contracts work
  - Next time: the magic of DeFi

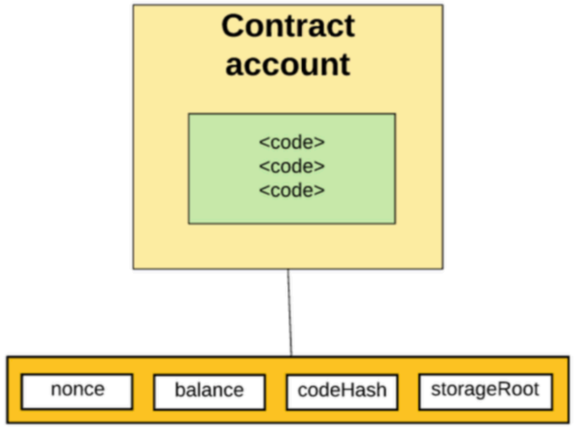


Review: Ethereum accounts

- Two types of account:
  - External (like Bitcoin), Contract accounts

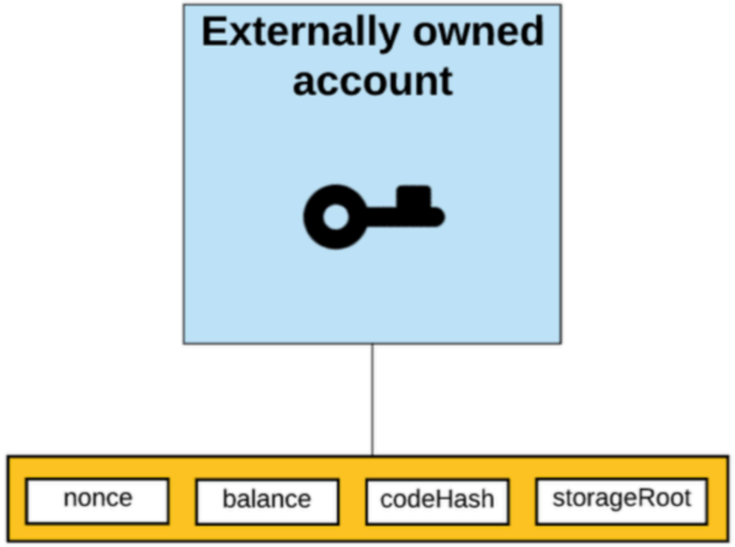


Like Bitcoin, updates require an signature by an external private key



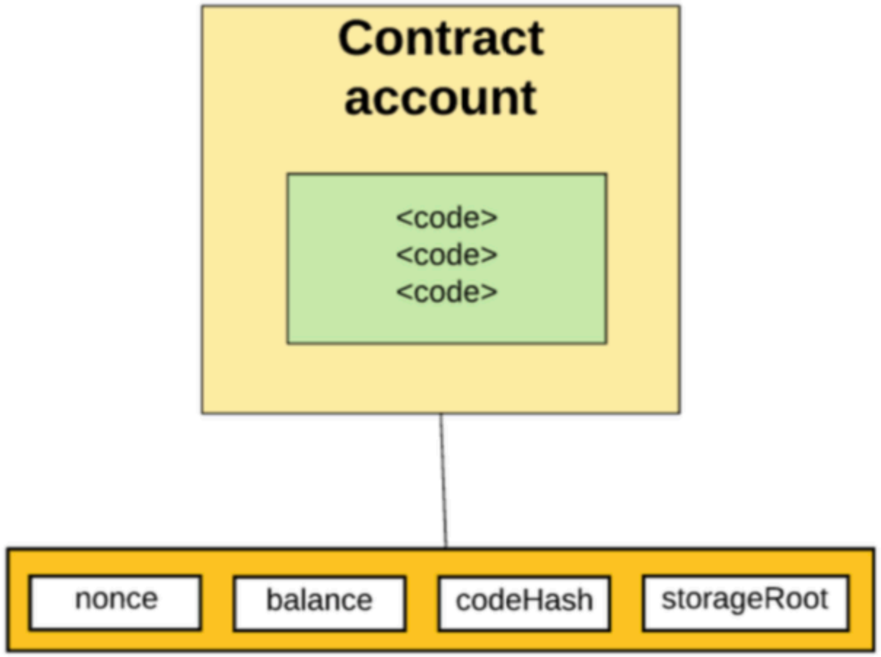
Anyone can call "methods" in the code, which trigger updates. Anyone can create.

External accounts



Note that these accounts have a "codeHash" and "storageRoot", but these fields are empty.

Like Bitcoin: the **address** is the hash of a public key.



From pyethereum:

```
def mk_contract_address(sender, nonce):
    return sha3(rlp.encode([normalize_address(sender), nonce]))[12:]
```

Note: does not include constructor code!

nonce

• bala

• stora

• code

this ac

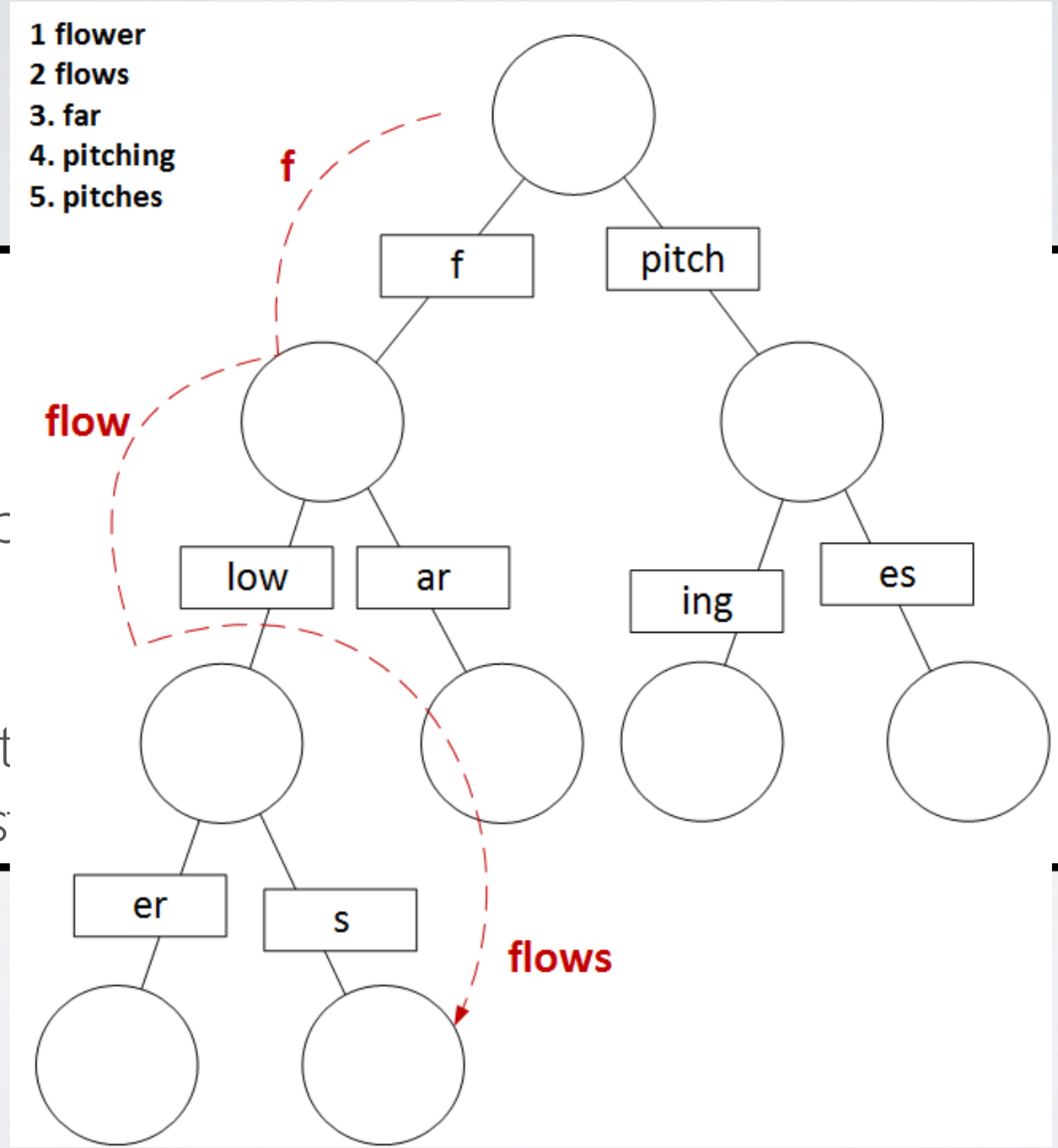
(I.e., t

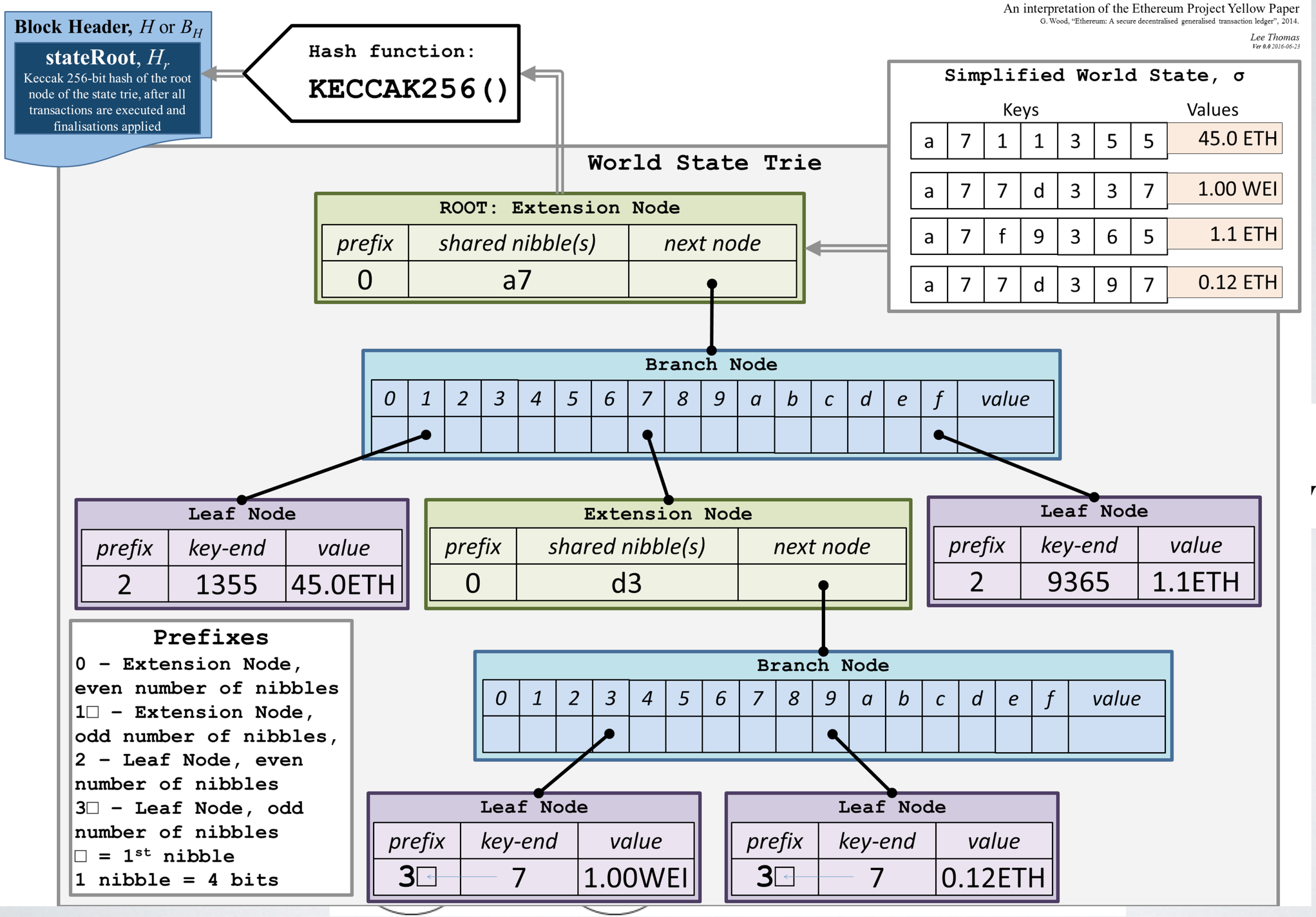
s

. The tree is empty

um Virtual Machir

the



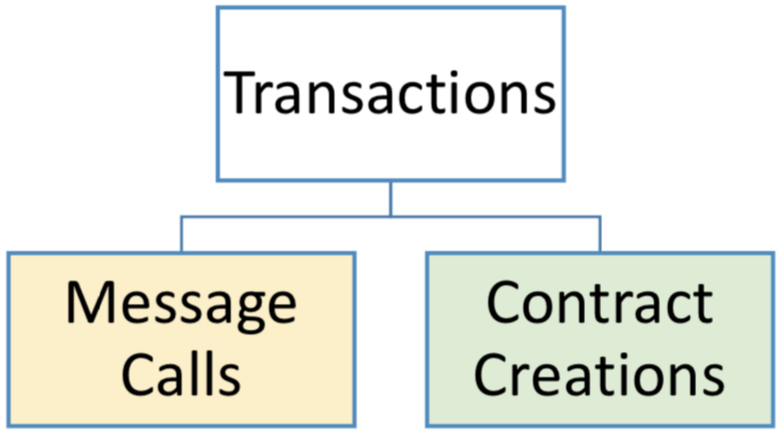


tree is empty

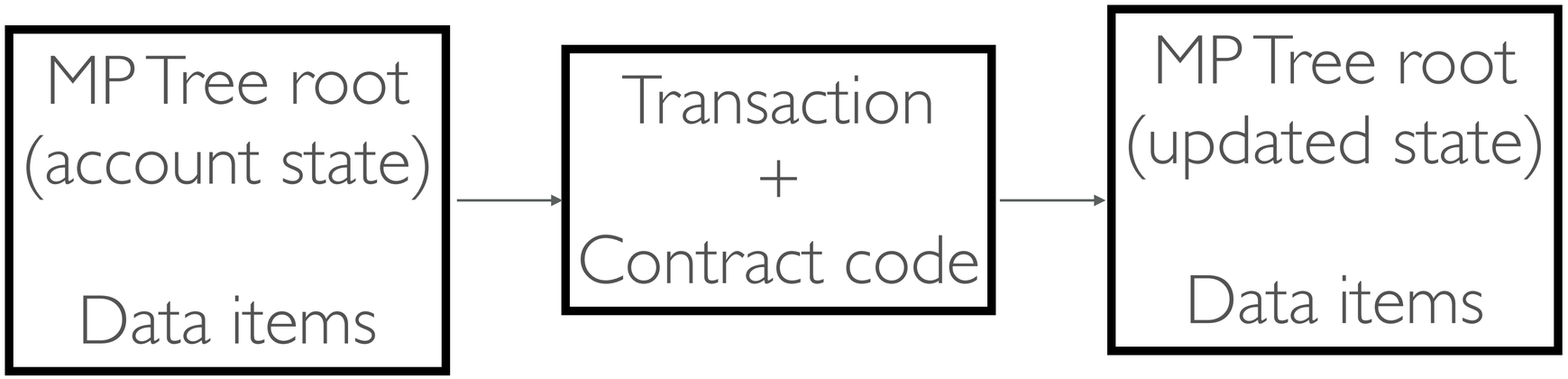
'irtual Machir

Ethereum Transactions

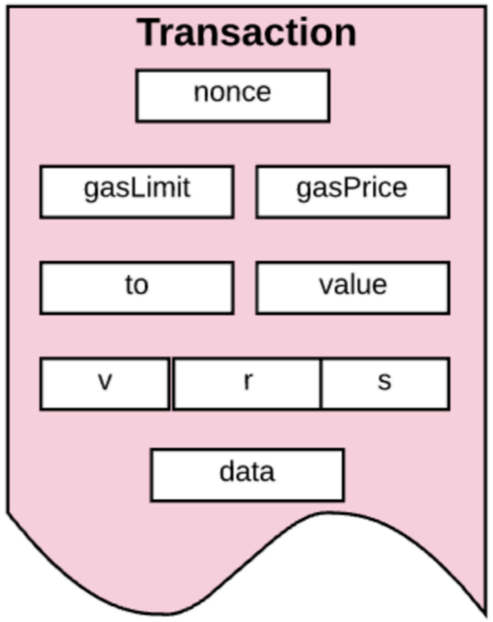
- Two types:
  - Message calls: update state in a given contract, by executing code (or simply transferring money)
  - Contract creations: instantiate a new contract account, with new state and code



Ethereum: Accounts



Ethereum Transactions

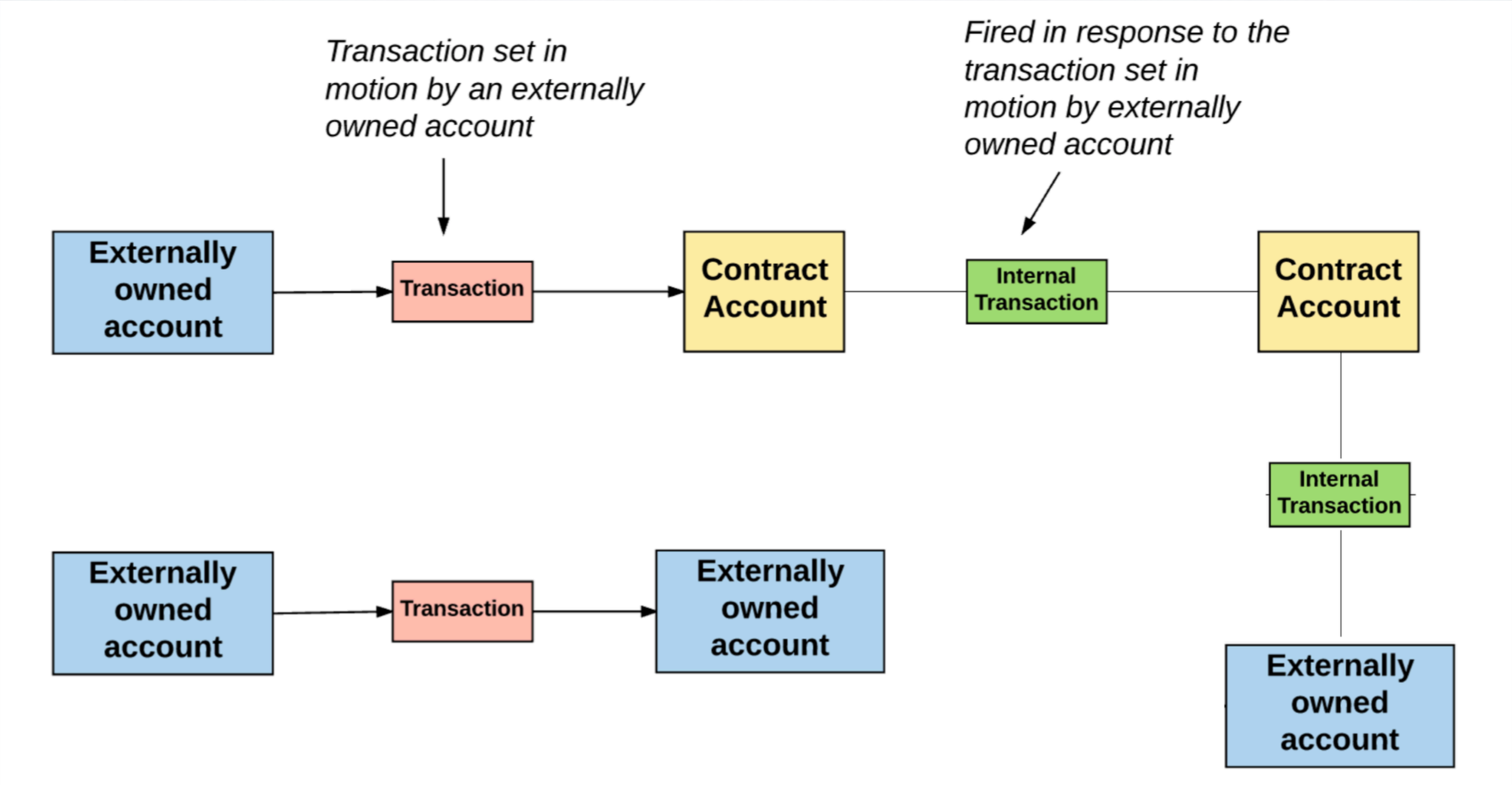


- nonce: A count of the number of transactions sent by the sender.
- gasPrice
- gasLimit
- to: Recepient's address
- value: Amount of Wei Transferred from sender to recipient.
- v,r,s: Used to generate the signature that identifies the sender of the transaction.
- init: EVM code used to initialize the new contract account.
- data: Optional field that only exists for message calls.

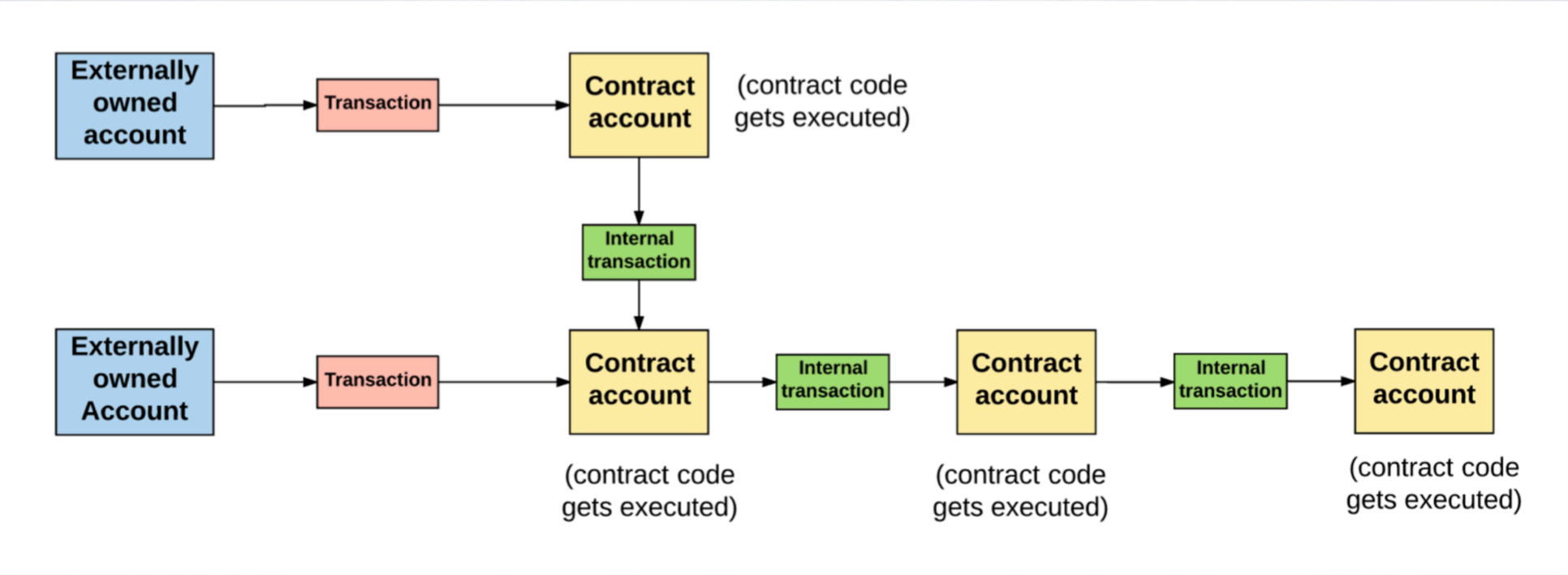
Transaction Types

- External transactions
  - Generated by a user, serialized and sent to the Ethereum network, put on the blockchain (just like Bitcoin)
  - Includes contract creation, payment, contract calls
- Internal transactions
  - Contracts A can make a transaction for Contract B (aka, contract can "call a method" for Contract B)
  - These are not serialized and put on the blockchain!

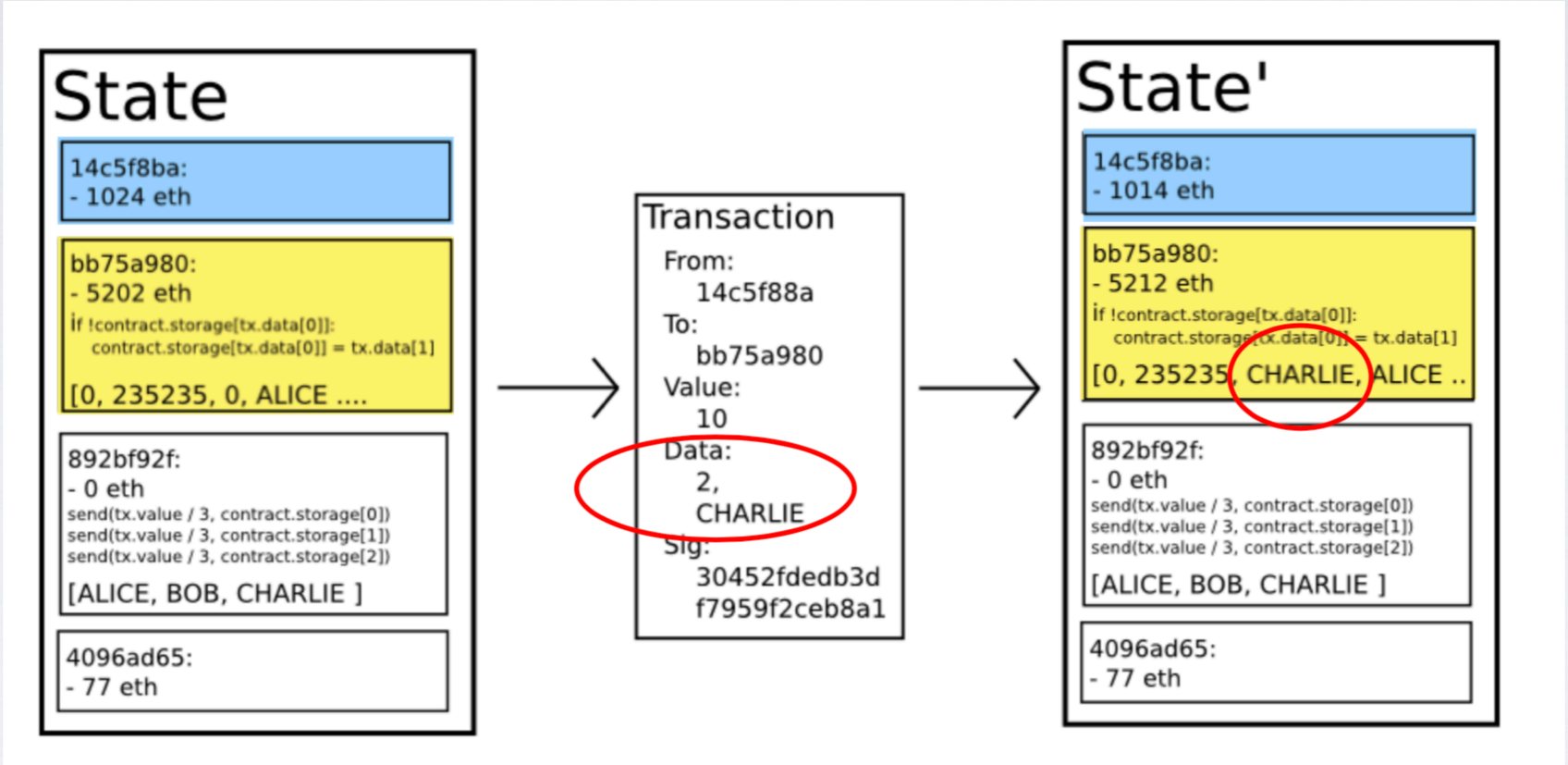
Transaction Types



Transaction Flow



Transaction Execution (Pay)



Contracts

- **At a low level:**
  - Smart contracts are defined by two blobs of EVM "bytecode"
  - One implements our "stateUpdate" function
    - This is the "core" smart contract
  - Another initializes the smart contract
    - This runs only during a deploy transaction
    - The constructor code is discarded after initialization

- Smart contr
  object-orien
  - The "contr
    public or p
  - Public met
    can only b
  - External tr
    (argument

thods

- the data

```
contract SimpleAuction {
    // Parameters of the auction. Times are either
    // absolute unix timestamps (seconds since 1970-01-01)
    // or time periods in seconds.
    address public beneficiary;
    uint public auctionEnd;

    // Current state of the auction.
    address public highestBidder;
    uint public highestBid;

    // Allowed withdrawals of previous bids
    mapping(address => uint) pendingReturns;

    // Set to true at the end, disallows any change
    bool ended;

    // Events that will be fired on changes.
    event HighestBidIncreased(address bidder, uint amount);
    event AuctionEnded(address winner, uint amount);

    // The following is a so-called natspec comment,
    // recognizable by the three slashes.
    // It will be shown when the user is asked to
    // confirm a transaction.

    /// Create a simple auction with `_biddingTime`
    /// seconds bidding time on behalf of the
    /// beneficiary address `_beneficiary`.
    constructor(
        uint _biddingTime,
        address _beneficiary
    ) public {
        beneficiary = _beneficiary;
        auctionEnd = now + _biddingTime;
    }

    /// Bid on the auction with the value sent
    /// together with this transaction.
    /// The value will only be refunded if the
    /// auction is not won.
    function bid() public payable {
```

Contract Transactions

- Smart c
  object-c
  - The "c
    public
  - Public
    can or
  - Extern
    (argum

```
contract SimpleAuction {
    // Parameters of the auction. Times are either
    // absolute unix timestamps (seconds since 1970-01-01)
    // or time periods in seconds.
    address public beneficiary;
    uint public auctionEnd;

    // Current state of the auction.
    address public highestBidder;
    uint public highestBid;

    // Allowed withdrawals of previous bids
    mapping(address => uint) pendingReturns;

    // Set to true at the end, disallows any change
    bool ended;

    // Events that will be fired on changes.
    event HighestBidIncreased(address bidder, uint amount);
    event AuctionEnded(address winner, uint amount);

    // The following is a so-called natspec comment,
    // recognizable by the three slashes.
    // It will be shown when the user is asked to
    // confirm a transaction.

    /// Create a simple auction with `_biddingTime`
    /// seconds bidding time on behalf of the
    /// beneficiary address `_beneficiary`.
    constructor(
        uint _biddingTime,
```

- Smart co
  object-o
  - The "co
    public
  - Public
    can onl
  - Externa
    (argum

```
// confirm a transaction.

/// Create a simple auction with `_biddingTime`
/// seconds bidding time on behalf of the
/// beneficiary address `_beneficiary`.
constructor(
    uint _biddingTime,
    address _beneficiary
) public {
    beneficiary = _beneficiary;
    auctionEnd = now + _biddingTime;
}

/// Bid on the auction with the value sent
/// together with this transaction.
/// The value will only be refunded if the
/// auction is not won.
function bid() public payable {
    // No arguments are necessary, all
    // information is already part of
    // the transaction. The keyword payable
    // is required for the function to
    // be able to receive Ether.

    // Revert the call if the bidding
    // period is over.
    require(
        now <= auctionEnd,
        "Auction already ended."
    );
```

s

data

Contract Creation

- The same piece of code ("contract") can be deployed multiple times, by different people
  - Contract "addresses" refer to a specific instance of a contract (combines contract code and a "nonce")
  - Contracts are compiled into EVM byte code and sent to the network
  - To deploy the code, you send the code (as data) to the special Ethereum address ("0")
    - (Contracts have a specialized opcode for this function…)

EVM

- Contracts are compiled into a type of Bytecode and run on a VM

To prevent cheating, the network works like Bitcoin:

Every single node in the network must also run the EVM machine instructions and inputs for each transaction in a received block, and only accepts the block if the EVM outputs (in the block) match their local computations.

Verification through repeated computing:

Each contract execution is "replicated" across the entire Ethereum network!

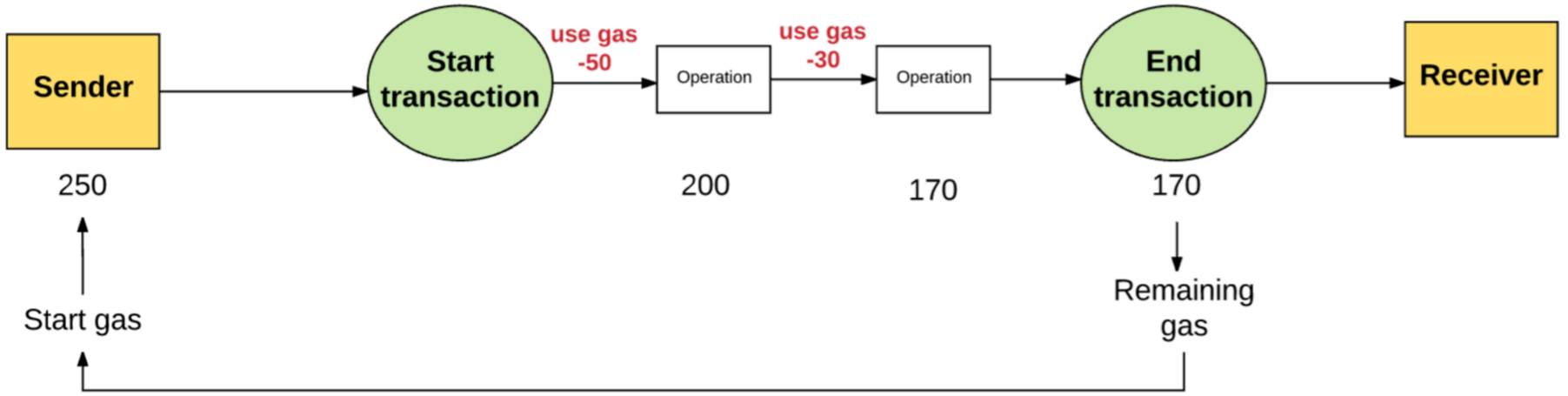
- What if that node cheats?


Gas limits/price

- Gas limit: Max no. of computational steps the transaction is allowed.
- Gas Price: Max fee the sender is willing to pay per computation step.

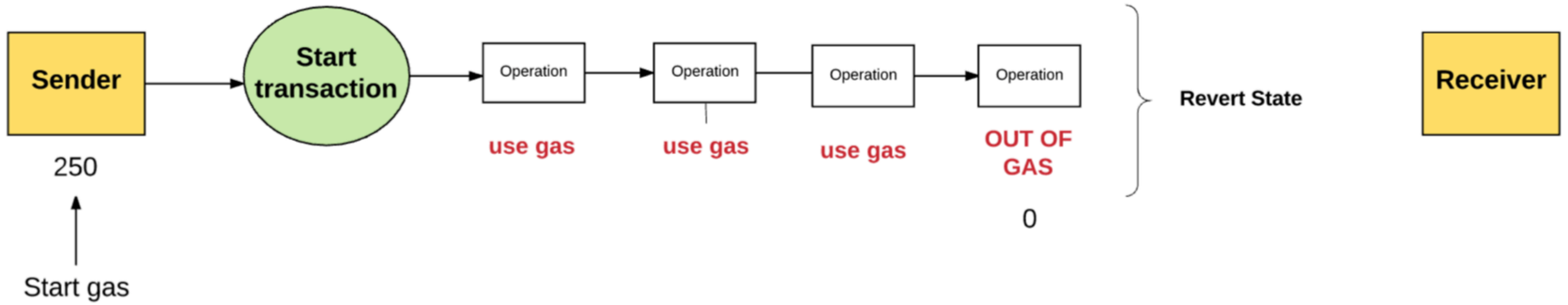
Gas limits/price

The sender is refunded for any unused gas at the end of the transaction.



Gas limits/price

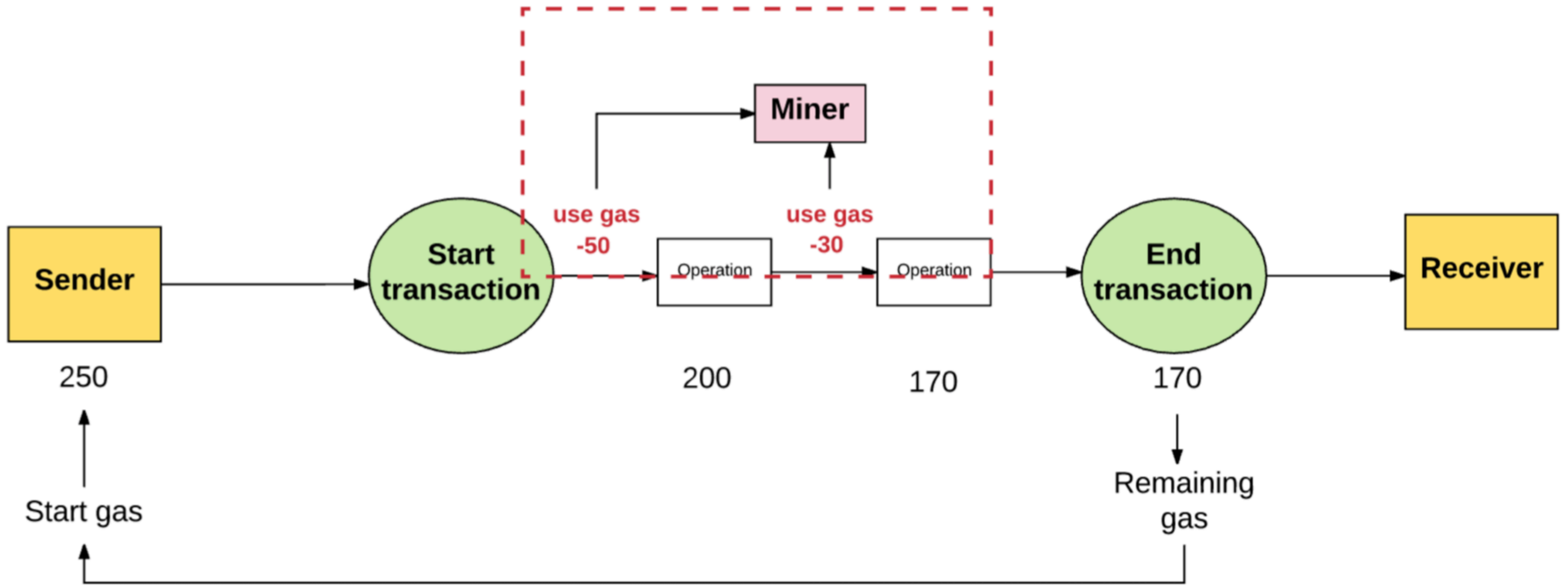
If sender does not provide the necessary gas to execute the transaction, the transaction runs “out of gas” and is considered invalid.



- The changes are reverted.
- None of the gas is refunded to the sender.

Gas limits/price

All the money spent on gas by the sender is sent to the miner's address.



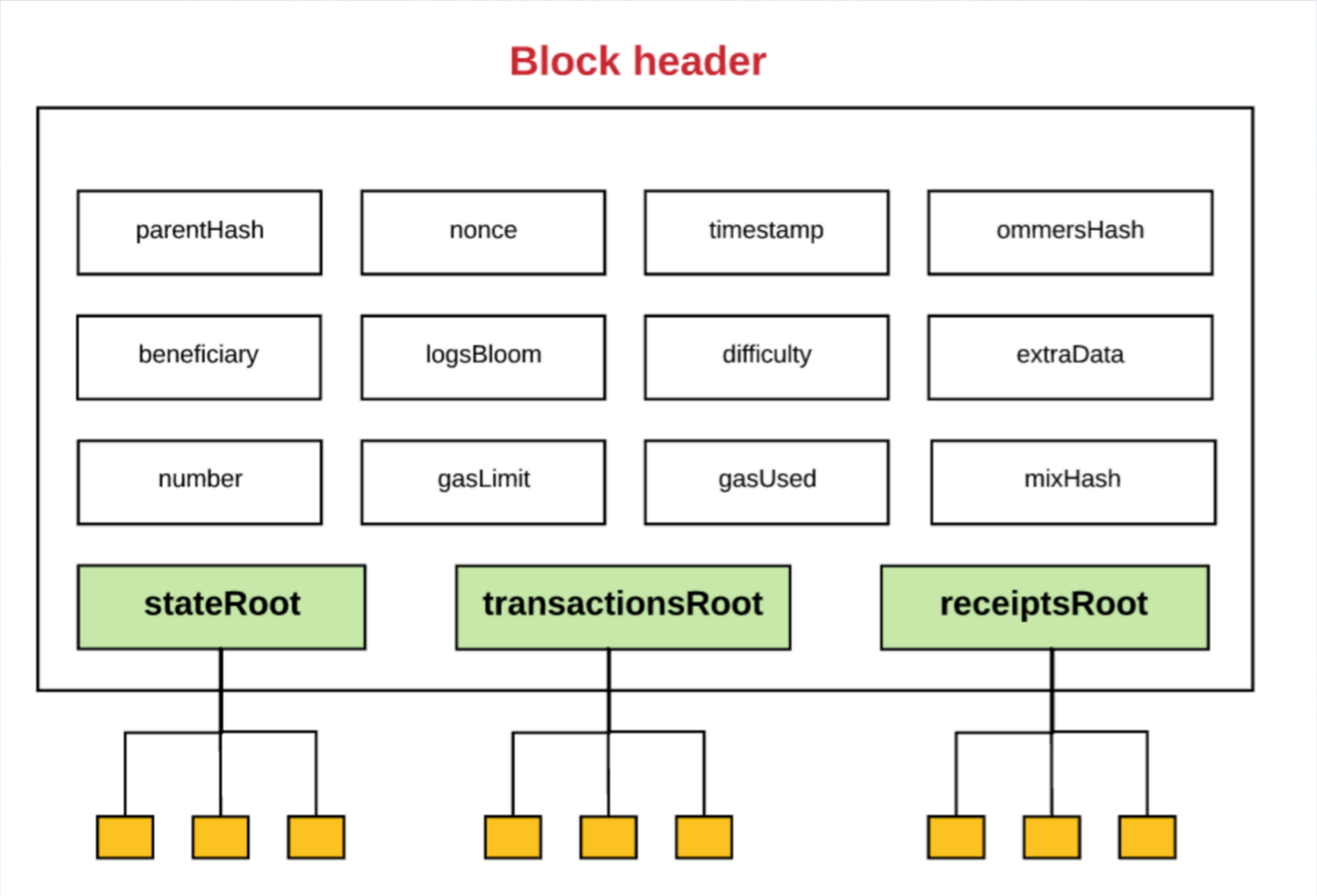
Incentive problems?

Does ETH need an internal currency?

Where do internal TXes happen?

- Contracts can call public methods in other contracts, by authoring internal ("virtual") transactions
- These transactions are not serialized and are not included on the blockchain
  - Instead, the miner formulates the internal transaction, executes the call locally, and serializes the result of the original call
  - The entire sequence of calls must be paid for within the calling transaction's gas limit

Merkle Trees



Binary Merkle Trees:

- Good data structure for authenticating information.
- Any edits/insertions/deletions are costly.

Merkle Patrica Trees:

- New tree root can be quickly calculated after an insert, update edit or delete operation in O(log n) time.
- Key-Value Pairs: Each value has a key associated with it.
- Key under which a value is stored is encoded into the path that you have to take down the tree.

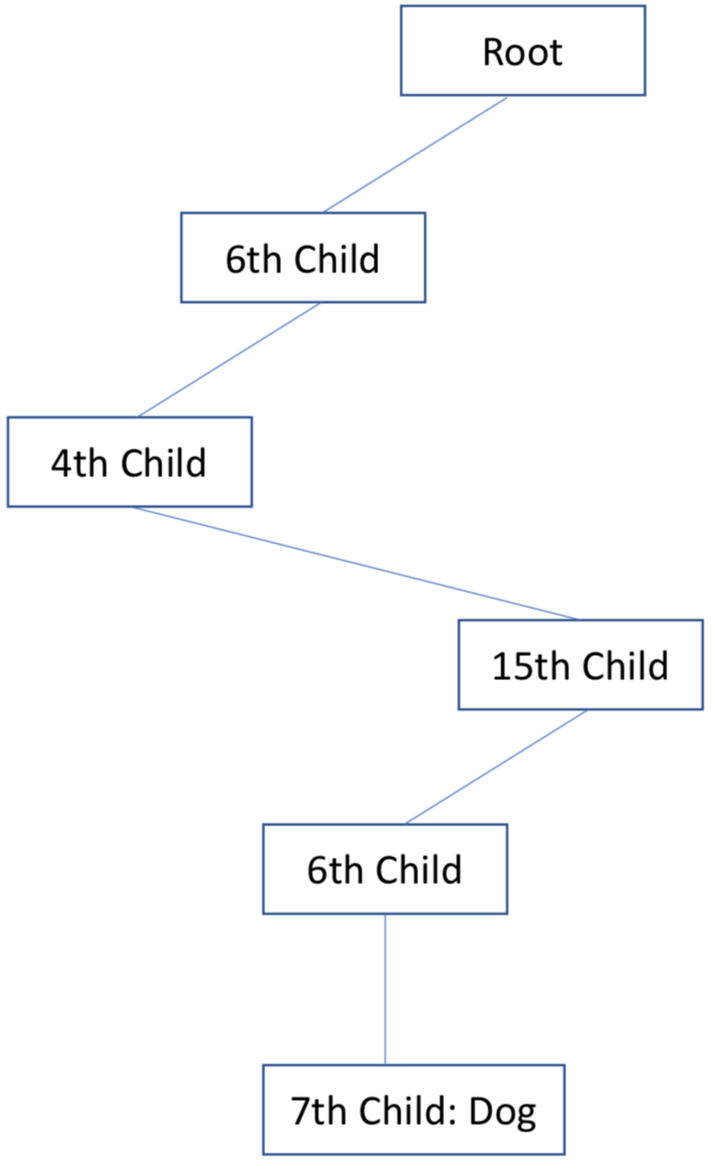
Why Trees?

Why Trees?

- A "light" client does not need to have the entire state of the world
  - A server can construct a Merkle proof that data exists in the state tree
  - Provided the state is encoded into the Merkle (Patricia) trees and we are given inclusion proofs, we can verify any block given just the previous block's tree root
  - Full nodes do need to keep current state data in order to work, but can "prune" (throw away) old states that are no longer valid

Merkle Patricia Trees

- Each node has 16 children.
- eg: Hex(dog)= 6 4 6 15 6 7

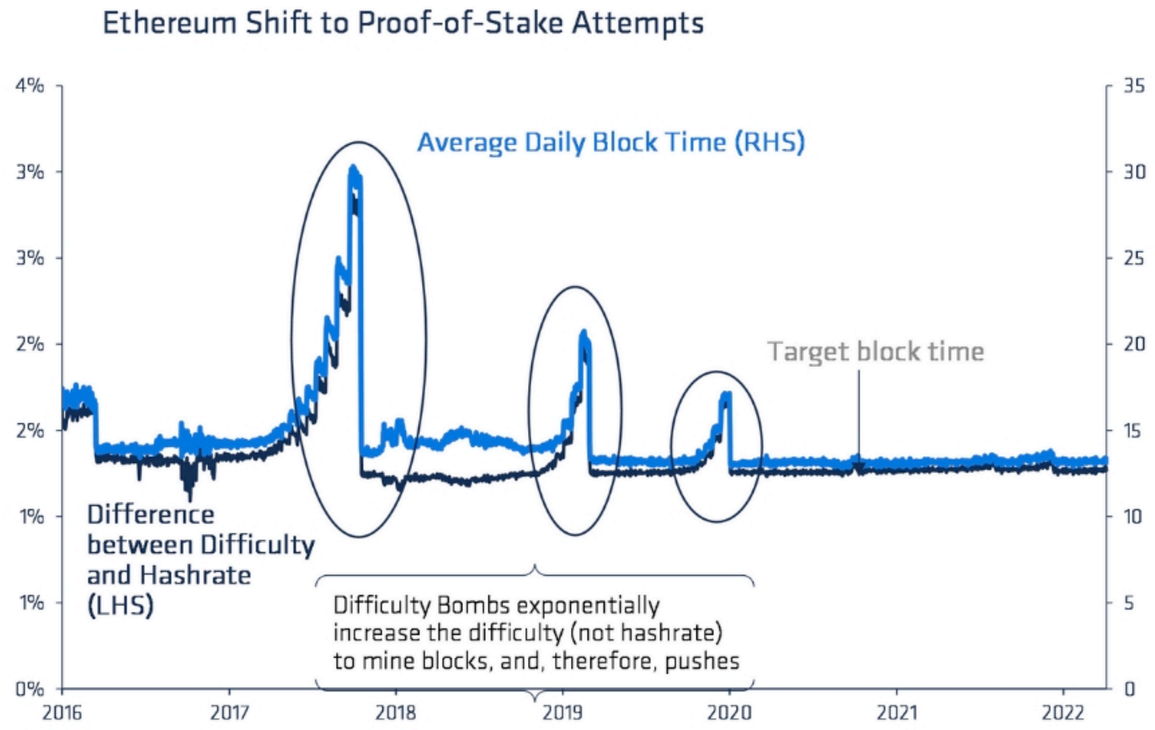


Ethereum consensus (original)

- The original Ethereum consensus algorithm worked very much like Bitcoin
  - Proof-of-work (though with a different hash function)
  - Faster target block time (15 seconds, not 10 minutes)
  - Had a mechanism to reward people who nearly created a block
    - These "not quite a block, but close" were called uncles
  - Miners received coins (like Bitcoin), but the reward schedule was different (not 21 million.)

Difficulty bombs

- To incentivize miners to upgrade, Ethereum included "difficulty bombs"
  - Without a hard-fork of the code, mining difficulty got worse



Source: Bloomberg, CoinShares, data available as of close 25 April 2022

Ethereum consensus (new!)

- **Ethereum has moved to Proof of Stake**
  - This is a multi-phase process:

1. Create a new proof-of-stake "beacon chain"
   this chain generates random values
2. "Dock" the original chain to the beacon chain
   use the random values to select block proposers
   from the set of validators who have put up stake
   (this replaces the probabilistic proof-of-work)
3. Complete the integration
   basically: let stakers withdraw their money

We are here

From concept to practice

- **How does a developer see Ethereum?**
  - So far we have talked about:
    - Init function (at deploy)
    - A single stateUpdate function (triggered by message Tx)
    - Databases and VMs

From concept to practice

- Ethereum programs are in "EVM byte code"
  - This is great for running things in a VM, works across platforms
  - Not made for human comprehension

```
00000d8b  PUSH1   #2 {var_e0_25}
00000d8d  EXP      {var_c0_53}
00000d8e  SUB      {var_c0_53} {var_a0_34}
00000d8f  DUP4     {var_40_4} {var_c0_54}
00000d90  AND      {var_a0_35} {var_a0_34} {var_c0_54}
00000d91  PUSH1   #0 {var_c0_55}
00000d93  SWAP1    {var_a0_35} {var_a0_36} {var_c0_56}
00000d94  DUP2     {var_e0_26}
```

Source: https://blog.ret2.io/2018/05/16/practical-eth-decompilation/

From concept to practice

- **How does a developer see Ethereum?**
  - So far we have talked about:
    - Init function (at deploy)
    - A single stateUpdate function (triggered by message Tx)
    - Databases and VMs

From concept to practice

- **How does a developer see Ethereum?**
  - So far we have talked about:
    - Init function (at deploy)
    - A single stateUpdate function (triggered by message Tx)
    - Databases and VMs
  - But this sucks for software developers
    - Let's instead think of contracts as object-oriented programs

From concept to practice

- Developers typically write programs in a high-level language
  - Technically any language can compile to EVM bytecode
  - And there are a few: Agoric (Javascript), Vyper
    - Some other chains (e.g., Solana) use rust
  - However, most Ethereum smart contracts are written in **Solidity**

Source: https://blog.ret2.io/2018/05/16/practical-eth-decompilation/

```
contract TokenContractFragment {

    // Balances for each account
    mapping(address => uint256) balances;

    // Owner of account approves the transfer of an amount to another account
    mapping(address => mapping (address => uint256)) allowed;

    // Get the token balance for account `tokenOwner`
    function balanceOf(address tokenOwner) public constant returns (uint balance) {
        return balances[tokenOwner];
    }

    // Transfer the balance from owner's account to another account
    function transfer(address to, uint tokens) public returns (bool success) {
        balances[msg.sender] = balances[msg.sender].sub(tokens);
        balances[to] = balances[to].add(tokens);
        Transfer(msg.sender, to, tokens);
        return true;
    }

    // Send `tokens` amount of tokens from address `from` to address `to`
    // The transferFrom method is used for a withdraw workflow, allowing contracts to send
    // tokens on your behalf, for example to "deposit" to a contract address and/or to charge
    // fees in sub-currencies; the command should fail unless the _from account has
    // deliberately authorized the sender of the message via some mechanism; we propose
    // these standardized APIs for approval:
    function transferFrom(address from, address to, uint tokens) public returns (bool success) {
        balances[from] = balances[from].sub(tokens);
        allowed[from][msg.sender] = allowed[from][msg.sender].sub(tokens);
        balances[to] = balances[to].add(tokens);
        Transfer(from, to, tokens);
        return true;
    }

    // Allow `spender` to withdraw from your account, multiple times, up to the `tokens` amount.
    // If this function is called again it overwrites the current allowance with _value.
    function approve(address spender, uint tokens) public returns (bool success) {
        allowed[msg.sender][spender] = tokens;
        Approval(msg.sender, spender, tokens);
        return true;
    }
}
```

```
contract TokenContractFragment {

    // Balances for each account
    mapping(address => uint256) balances;

    // Owner of account approves the transfer of an amount to another account
    mapping(address => mapping (address => uint256)) allowed;

    // Get the token balance for account `tokenOwner`
    function balanceOf(address tokenOwner) public constant returns (uint balance) {
        return balances[tokenOwner];
    }

    // Transfer the balance from owner's account to another account
    function transfer(address to, uint tokens) public returns (bool success) {
        balances[msg.sender] = balances[msg.sender].sub(tokens);
        balances[to] = balances[to].add(tokens);
        Transfer(msg.sender, to, tokens);
        return true;
    }

    // Send `tokens` amount of tokens from address `from` to address `to`
    // The transferFrom method is used for a withdraw workflow, allowing contracts to send
    // tokens on your behalf, for example to "deposit" to a contract address and/or to charge
    // fees in sub-currencies; the command should fail unless the _from account has
    // deliberately authorized the sender of the message via some mechanism; we propose
    // these standardized APIs for approval:
    function transferFrom(address from, address to, uint tokens) public returns (bool success) {
        balances[from] = balances[from].sub(tokens);
```

```
14      // Transfer the balance from owner's account to another account
15      function transfer(address to, uint tokens) public returns (bool success) {
16          balances[msg.sender] = balances[msg.sender].sub(tokens);
17          balances[to] = balances[to].add(tokens);
18          Transfer(msg.sender, to, tokens);
19          return true;
20      }
21
22      // Send `tokens` amount of tokens from address `from` to address `to`
23      // The transferFrom method is used for a withdraw workflow, allowing contracts to send
24      // tokens on your behalf, for example to "deposit" to a contract address and/or to charge
25      // fees in sub-currencies; the command should fail unless the _from account has
26      // deliberately authorized the sender of the message via some mechanism; we propose
27      // these standardized APIs for approval:
28      function transferFrom(address from, address to, uint tokens) public returns (bool success) {
29          balances[from] = balances[from].sub(tokens);
30          allowed[from][msg.sender] = allowed[from][msg.sender].sub(tokens);
31          balances[to] = balances[to].add(tokens);
32          Transfer(from, to, tokens);
33          return true;
34      }
35
36      // Allow `spender` to withdraw from your account, multiple times, up to the `tokens` amount.
37      // If this function is called again it overwrites the current allowance with _value.
38      function approve(address spender, uint tokens) public returns (bool success) {
39          allowed[msg.sender][spender] = tokens;
40          Approval(msg.sender, spender, tokens);
41          return true;
42      }
43  }
```