

Blockchains & Cryptocurrencies

Ethereum



Instructor: Abhishek Jain
Johns Hopkins University - Spring 2021

Today

- From Bitcoin to smart contracts
- Ethereum
- Applications



Review: Bitcoin

- **Each block is a list of transactions**
 - Each transaction consumes one or more inputs;
 - Each transaction includes a set of outputs (amount + destination)
 - Input consumption has conditions (e.g., valid script, typically enforcing valid signature)

Review: Bitcoin

- **Each block is a list of transactions**
 - Each transaction consumes one or more inputs;
 - Each transaction includes a set of outputs (amount + destination)
 - Input consumption has conditions (e.g., valid signature)

In practice, each input/output has script:

ScriptPubKey (outputs)

ScriptSig (inputs)

Review: Bitcoin

- OP_DUP OP_HASH160 Bytes to push
89 AB CD EF AB BA 88 AC
Data to push OP EQUALVERIFY OP CHECKSIG

- Each transaction includes a set of outputs

```
scriptPubKey: OP_2DUP OP_EQUAL OP_NOT OP_VERIFY OP_SHA1 OP_SWAP OP_SHA1 OP_EQUAL  
scriptSig: <preimage1> <preimage2>
```

- Input consumption has conditions (e.g., valid signature)

```
scriptPubKey: <expiry time> OP_CHECKLOCKTIMEVERIFY OP_DROP OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG  
scriptSig: <sig> <pubKey>
```

ScriptPubKey (outputs)
ScriptSig (inputs)

Review: Bitcoin script

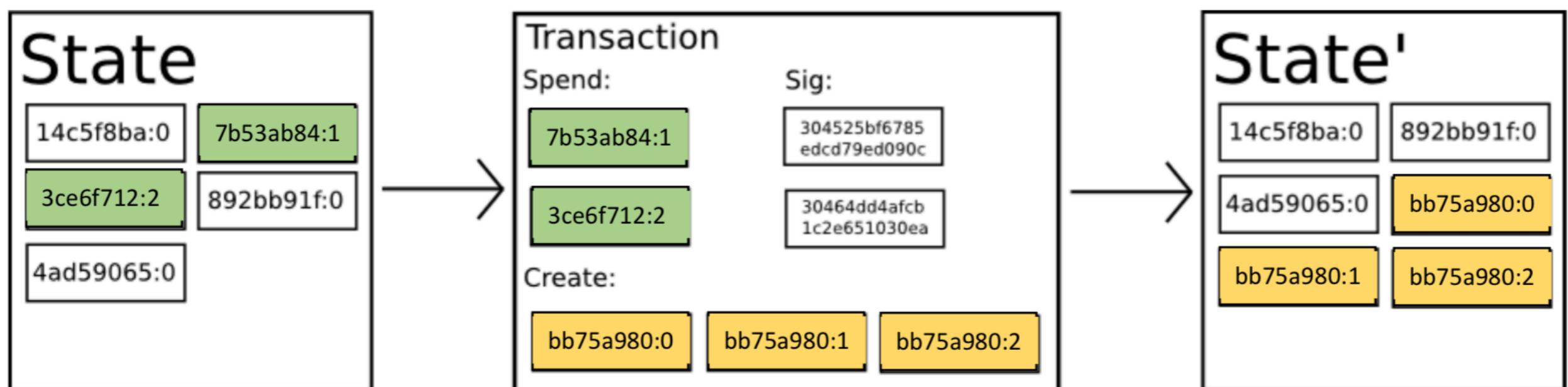
- **Bitcoin script allows us to attach conditions to payments**
 - However script is deliberately limited
 - Stack-based FORTH-type language, many original opcodes disabled
 - Highly limited access to global data

Generalizing Bitcoin

- **Let's consider each Bitcoin transaction as a state transition function**
 - What is the input state?
 - What is the output state?
 - What does a Transaction do to the state?

Generalizing Bitcoin

- Let's consider each Bitcoin transaction as a state transition function
 - Input state: list of coins available for spending (UTXO set)
 - Transaction: set of instructions for updating the UTXO set
 - Output state: New blockchain hash, Updated UTXO set



“Smart contracts”

- Idea proposed by Nick Szabo (1994)
 - Bitcoin script is a ‘contract’ in the sense that it provides programmable conditions for redeeming a coin
 - However, conditions are highly limited
 - Example: pay out a coin iff a user has a signing key
 - Example: implement a second currency/asset
 - Example: pay out a coin iff a candidate wins the US election
 - Example: pay out a coin iff a majority of users votes to invest in a service

“Smart contracts”

- Idea proposed by Nick Szabo (1994)
 - Bitcoin script is a ‘contract’ in the sense that it provides programmable conditions for redeeming a coin
 - However, conditions are highly limited

• Example: pay out a coin iff a user has a signing key Bitcoin 

• Example: implement a second currency/asset Bitcoin  

• Example: pay out a coin iff a candidate wins the US election Bitcoin 

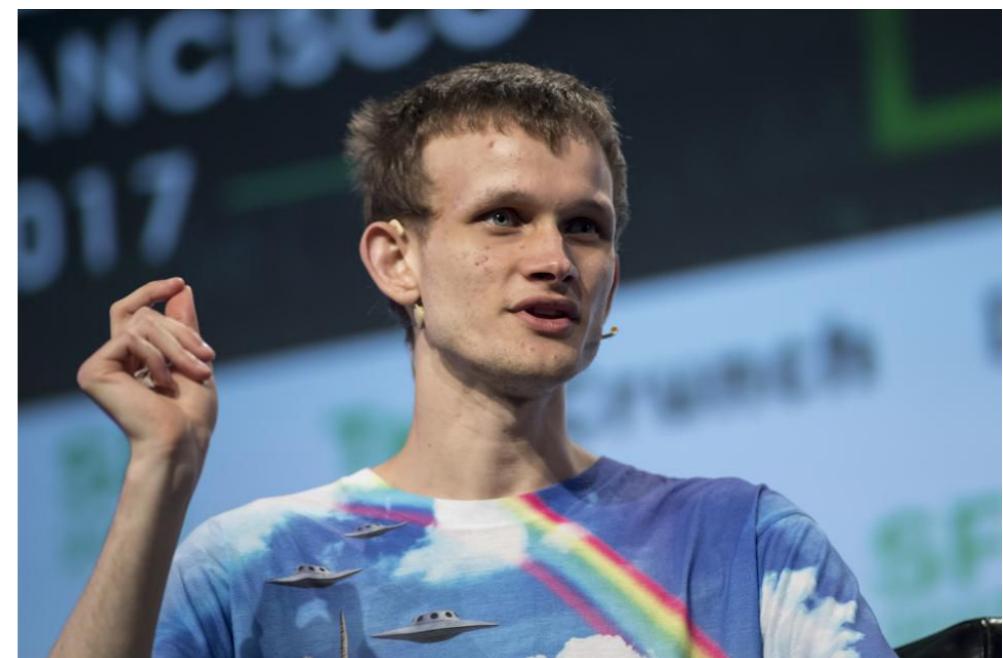
• Example: pay out a coin iff a majority of users vote to invest in a service Bitcoin 

Why not Bitcoin?

- Bitcoin script is not Turing complete
 - So script size (hence TX size) grows with complexity
 - Bitcoin blocks are capped at 1MB, which means TX space is at a premium
 - Limited script opcodes (security reasons)
 - No access to “global state”

Ethereum

- Proposed in 2013 by Vitalik Buterin
 - Basic idea: extend Bitcoin by adding Turing-complete scripting, with full access to chain state (“smart contracts”)
 - Scripts run inside of an Ethereum virtual machine (EVM), can call other scripts & each other (recursively)
 - Includes a native token (ETH) to pay for transactions, but users can create additional tokens using contracts



What challenges does this raise?

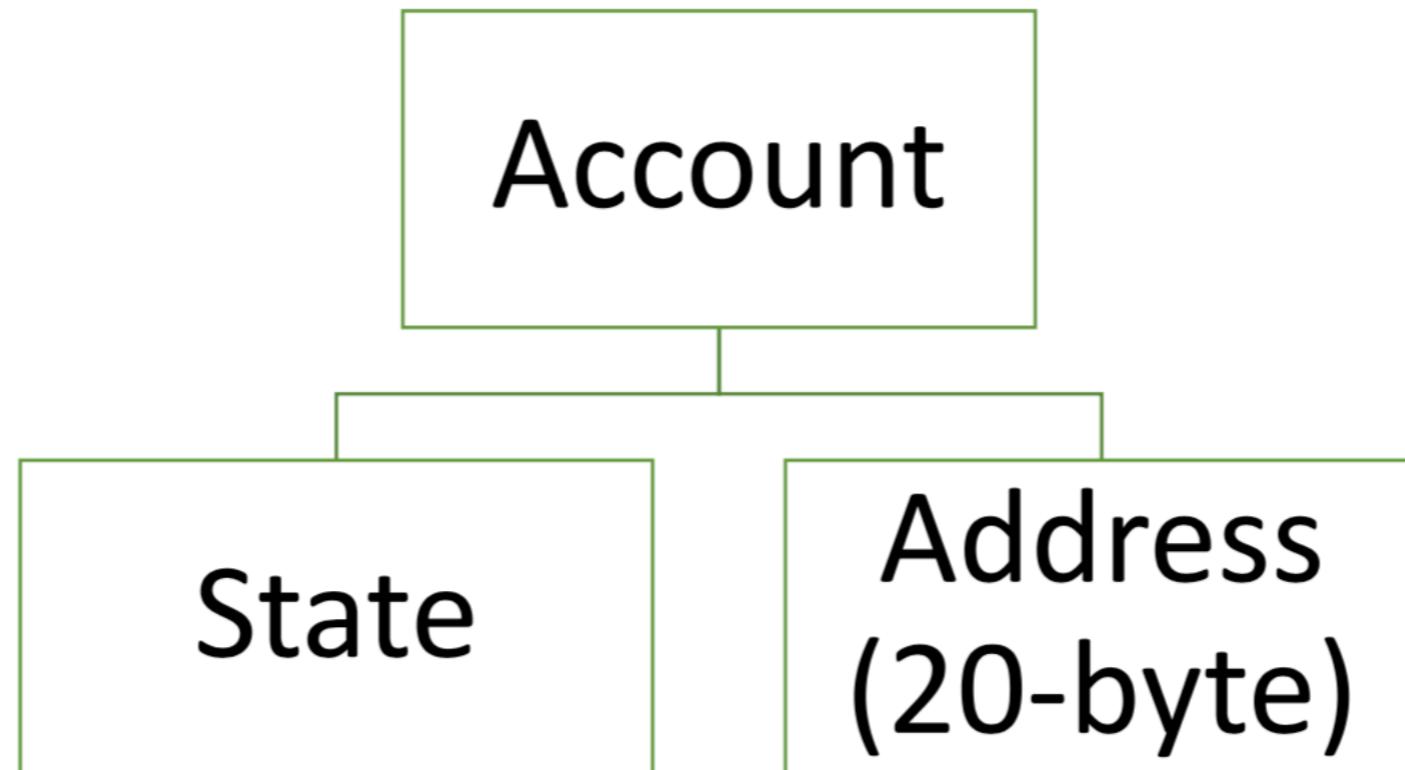
- What if the code *crashes*?
 - Code run in EVM
- What if code requires *randomness* (e.g., lottery)?
 - Use blockchain metadata (e.g., blockhash) as randomness that can be used by all miners. (Requires care to prevent *grinding attacks*.)
- What if the code doesn't *terminate* (in reasonable time)?
 - Require transaction senders to pay “fee” (for computation and storage)

What challenges does this raise? (contd.)

- **What if we want multi-query, *stateful* smart-contracts?**
 - Maintain contract state (must be same across miners)
- **What if we want to use the same “code” for *different* contracts?**
 - Many reasons for doing so, e.g., think of “libraries”, or say a “currency”
 - Different state for each instance.
- **How to achieve modularity?**
 - Contracts can “call” other contracts (using specified “interface”)

Ethereum: Account-Based Cryptocurrency

- Global state consists of many small “objects”, aka **accounts**
- Accounts can interact with each other via “message passing”



Ethereum: Accounts

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



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

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

Ethereum: Accounts

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



Can initiate messages to other externally owned accounts or contract accounts

Can only send messages in response to other received messages.

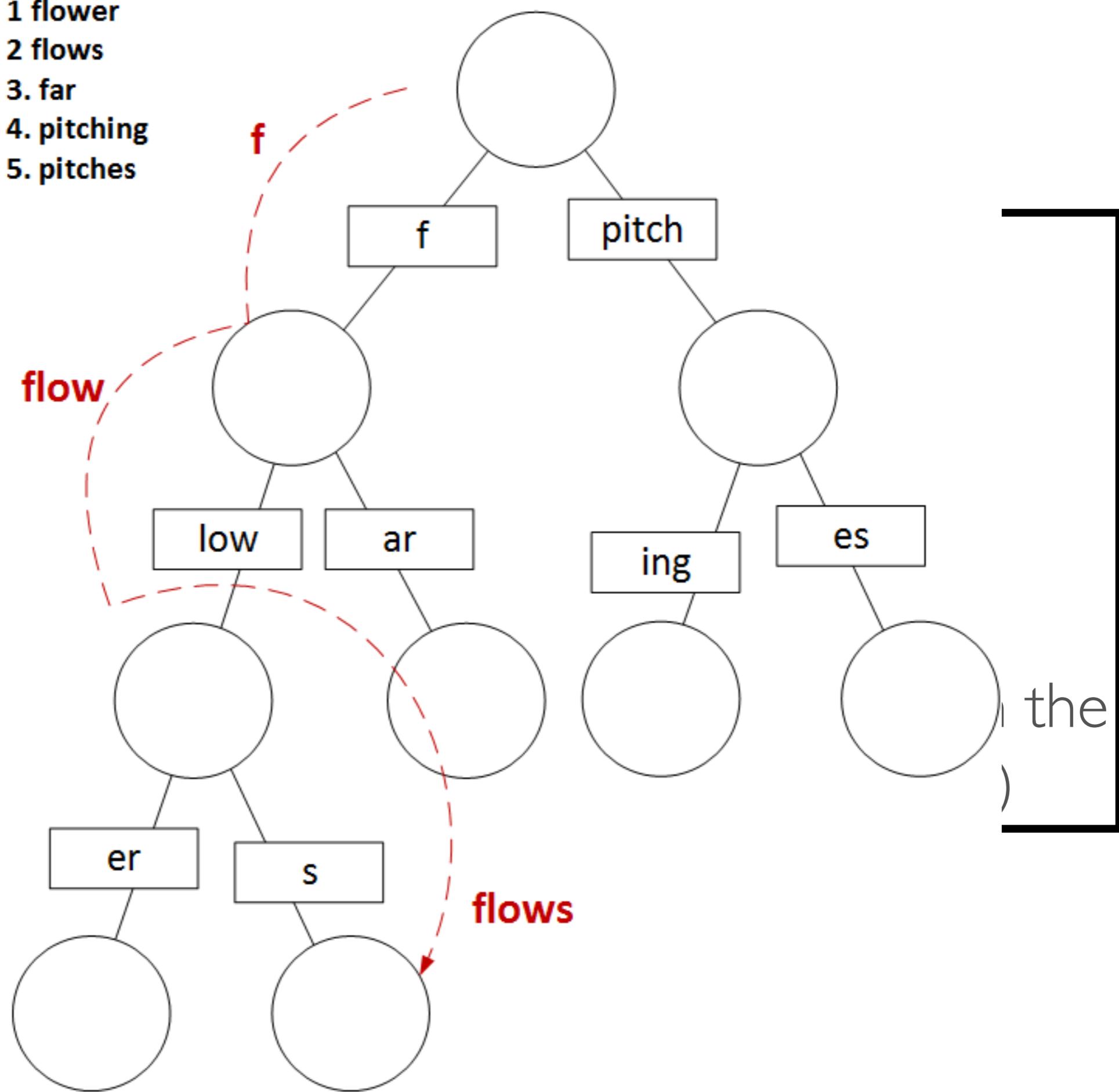
Ethereum: Accounts

- **nonce:** # transactions sent/ # contracts created
- **balance:** # Wei owned (1 ether = 10^{18} Wei)
- **storageRoot:** Hash of the root node of a Merkle Patricia tree of storage contents of the account. The tree is empty by default.
- **codeHash:** Hash of empty string / Hash of the EVM (Ethereum Virtual Machine) code of this account

Ethereum: Accounts

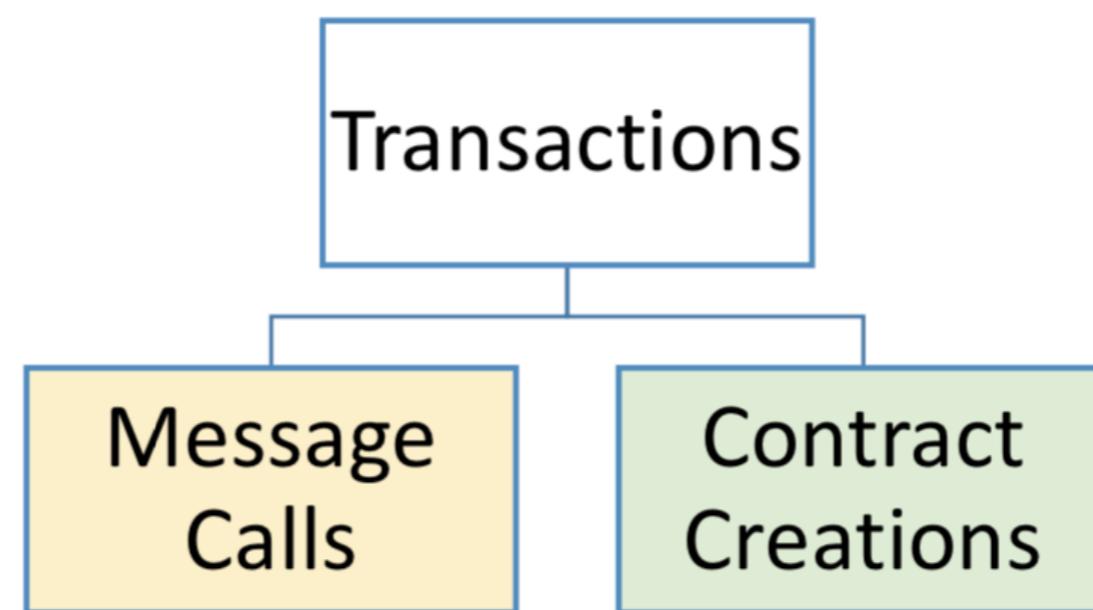
- no
 - ba
 - sto
 - tre
 - co
 - (Et
- Merkle Patricia tree:
- A data structure for storing key/value pairs in a cryptographically authenticated manner.
- (I.e., the tree root is a hash of all key/values in the structure, and updates/deletions are fast.)

- no
 - ba
 - sto
 - tre
 - CO
 - (Et
 - (l.e., t
 - st
- p

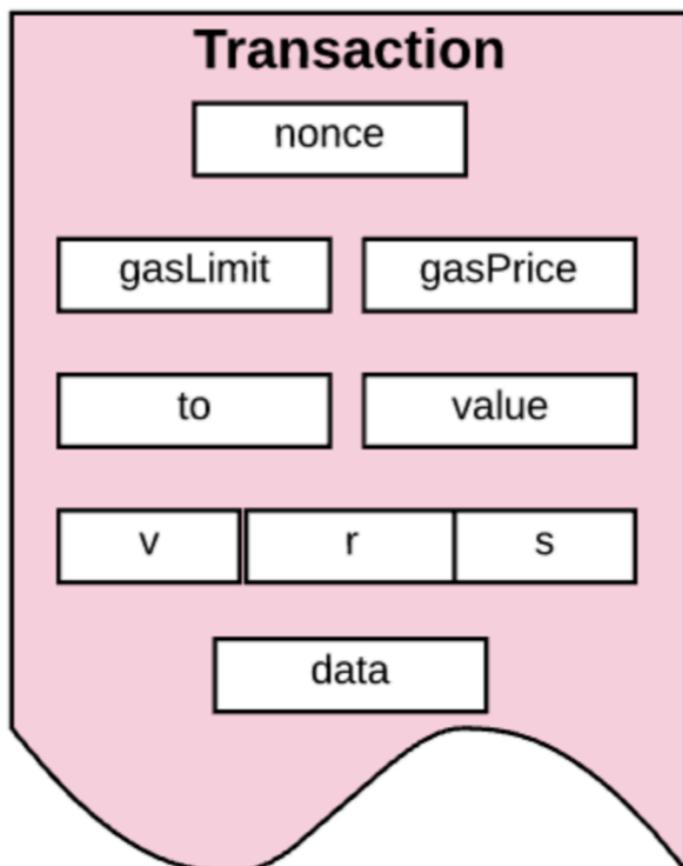


Ethereum Transactions

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



Ethereum Transactions

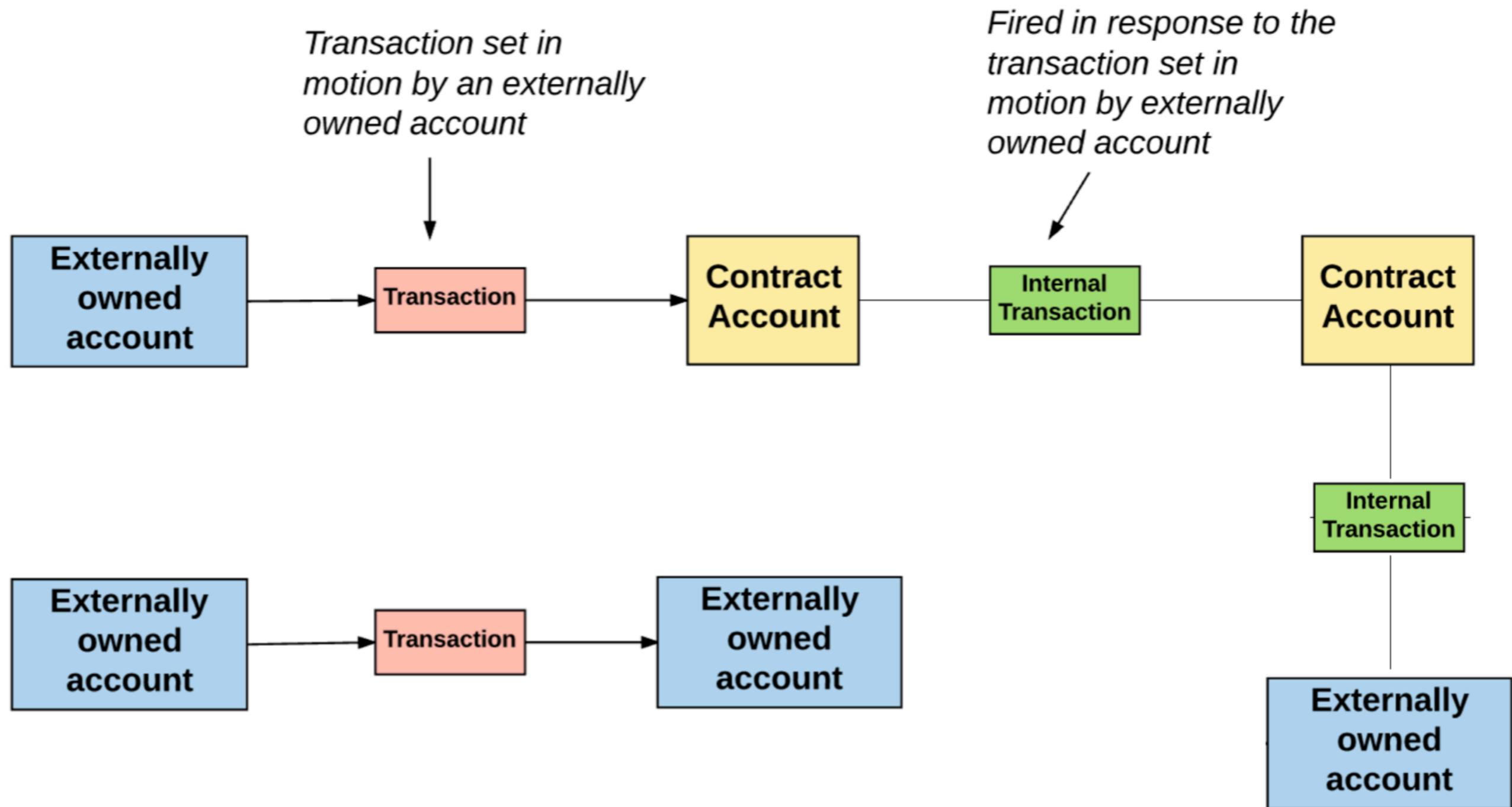


- **nonce**: A count of the number of transactions sent by the sender.
- **gasPrice**
- **gasLimit**
- **to**: Recipient's address
- **value**: Amount of Wei Transferred from sender to recipient.
- **v,r,s**: Used to generate the signature that identifies the sender of the transaction.
- **init**: EVM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

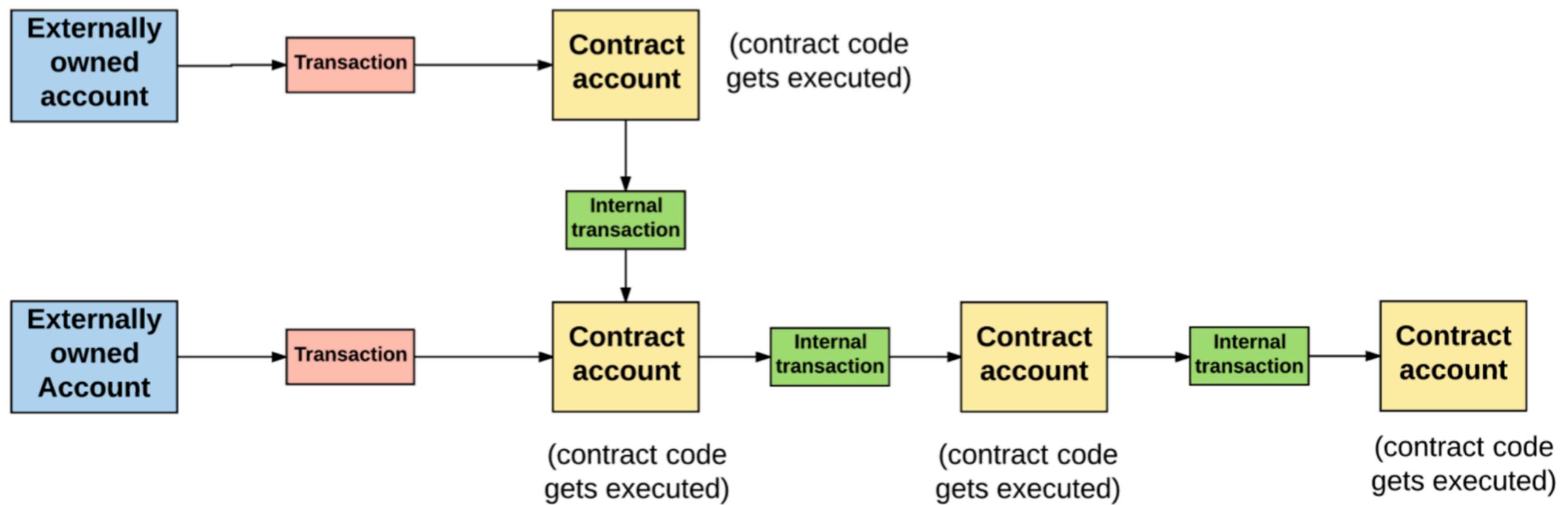
Transaction Types

- 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 not put on the blockchain!

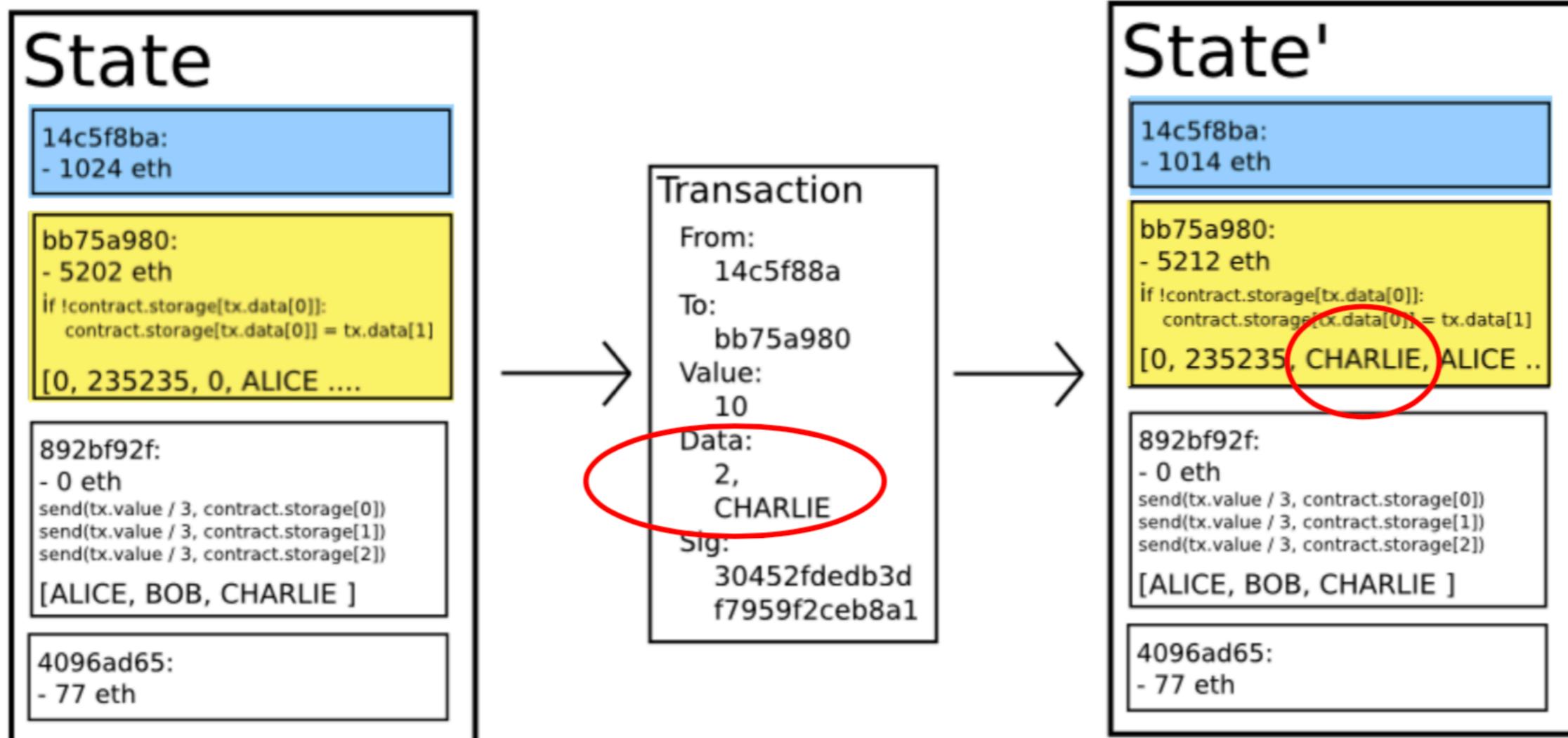
Transaction Types



Transaction Flow



Transaction Execution



Contracts

- Smart contracts are often written in a high-level object-oriented language (e.g., Solidity)
 - The “contract” object has “methods”, which can be public or private (internal)
 - Public methods can be called by anyone, private methods can only be called from other methods within that contract
 - External transactions contain the contract address + the data (arguments) for a method call

- Smart contracts are object-oriented
- The “contract” part is public or private
- Public methods can only be called from outside
- External transactions (arguments)

```
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 {
```

methods
+ the data

Contract Creation

- The same piece of code 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

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?

How do we stop DoS?

- EVM code is “Turing complete”, e.g., has loops and may not halt
 - Users pay fees to submit TXes to the network
 - (Pays for both computation and storage of data)
 - Contract computation/storage costs are denominated in “gas” (e.g., one hash call requires 30 “gas”)
 - Each transaction contains a “gas limit”, and a “gas price” price the sender will pay per unit of gas, denominated in ETH (currency)
 - Along with sufficient ETH to pay the fee

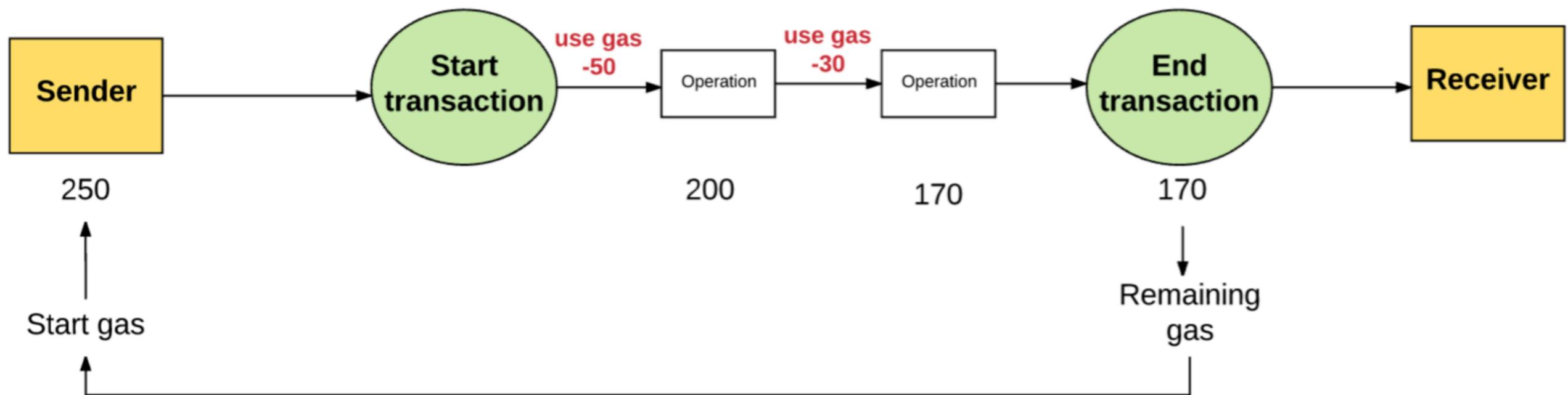
Gas limits/price

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

$$\begin{array}{ccc} \boxed{\text{Gas Limit}} & \times & \boxed{\text{Gas Price}} \\ \boxed{50,000} & & \boxed{20 \text{ gwei}} \\ & & = \\ & & \boxed{\text{Max transaction fee}} \\ & & \boxed{0.001 \text{ Ether}} \end{array}$$

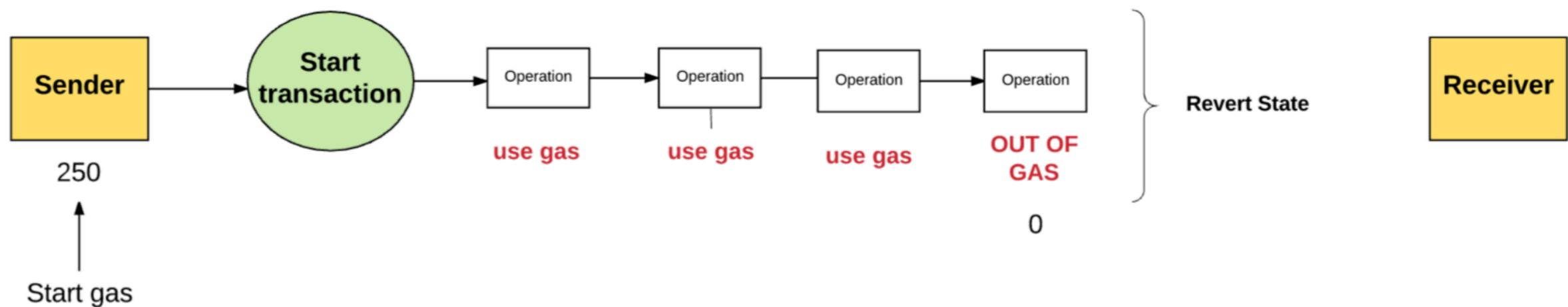
Gas limits/price

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



Gas limits/price

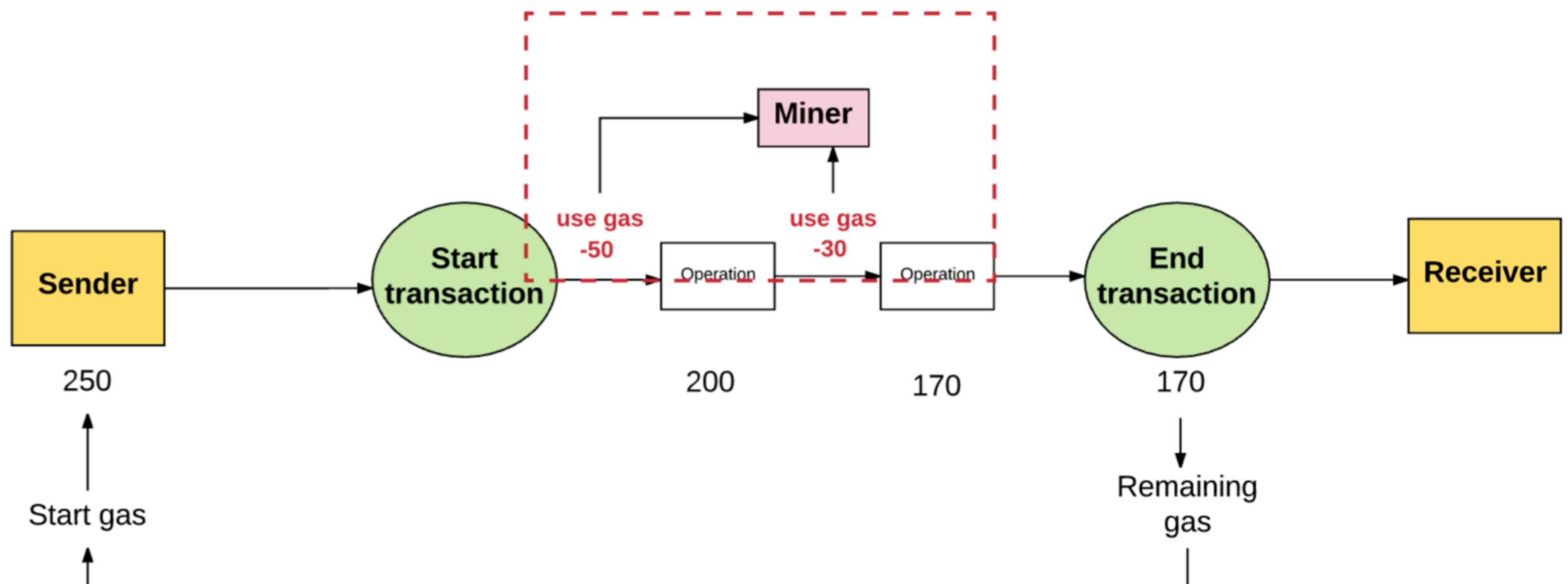
If sender does not provide the necessary gas to execute the transaction, the transaction runs “out of gas” and is considered invalid.



- 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.



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

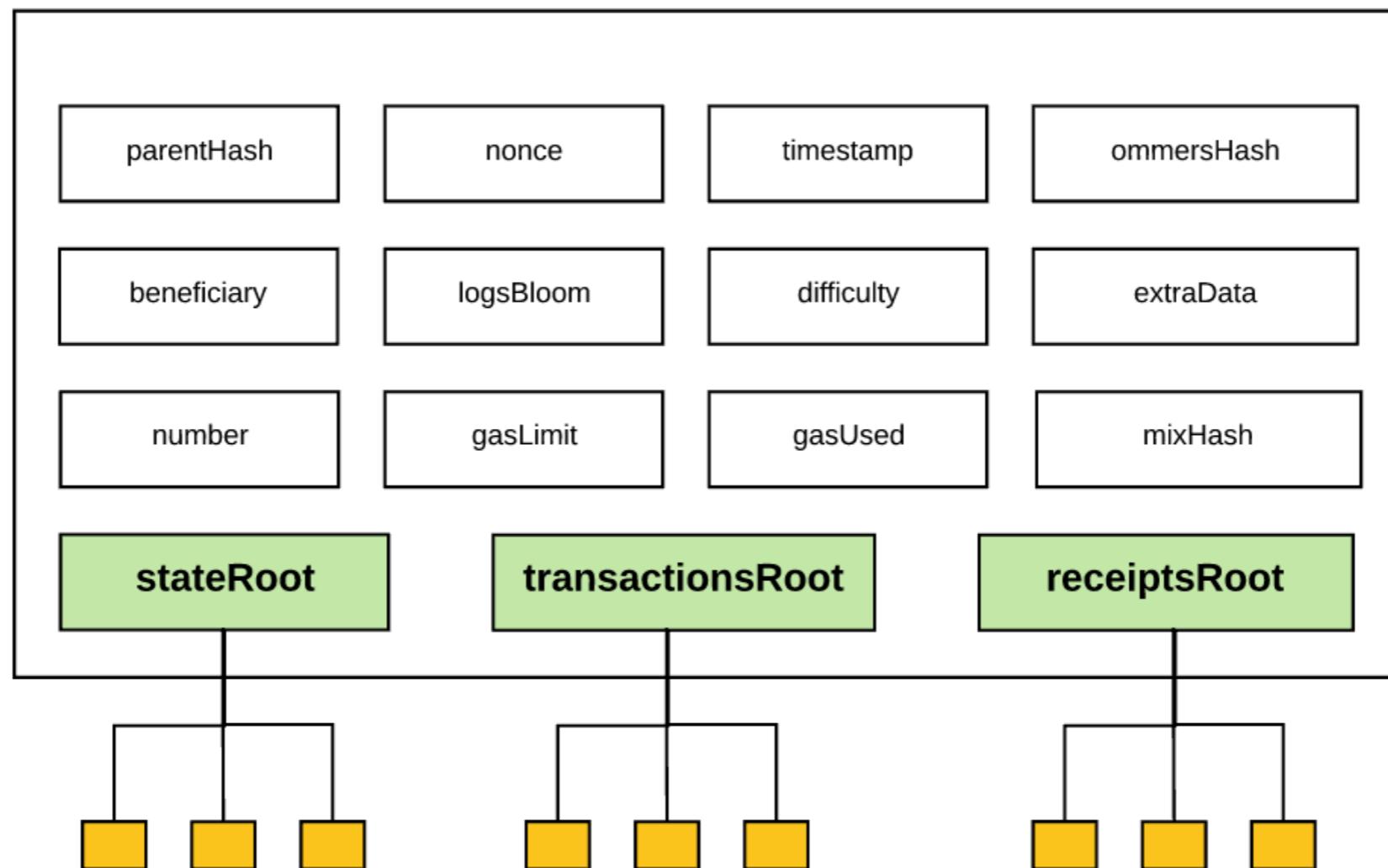
Ethereum Blocks

In Ethereum, a block consists of:

- The **block header**
- Information about the **set of transactions** included in that block.
- A set of other **block headers** for the current block's **uncles**.

Block Header

Block header



Block Header

- **parentHash**: a hash of the parent block's header
- **ommersHash**: a hash of the current block's list of ommers
- **beneficiary**: the account address that receives the fees for mining this block
- **stateRoot**: the hash of the root node of the state tree
- **transactionsRoot**: the hash of the root node of the tree that contains all transactions listed in this block
- **receiptsRoot**: the hash of the root node of the tree that contains the receipts of all transactions listed in this block
- **logsBloom**: a Bloom Filter (data structure) that consists of log information
- **difficulty**: the difficulty level of this block
- **number**: the count of current block
- **gasLimit**: the current gas limit per block
- **gasUsed**: the sum of the total gas used by transactions in this block
- **timestamp**: the timestamp of this block's inception
- **extraData**: extra data related to this block
- **mixHash**: a hash that, when combined with the nonce, proves that this block has carried out enough computation
- **nonce**: a hash that, when combined with the mixHash, proves that this block has carried out enough computation

Why Trees?

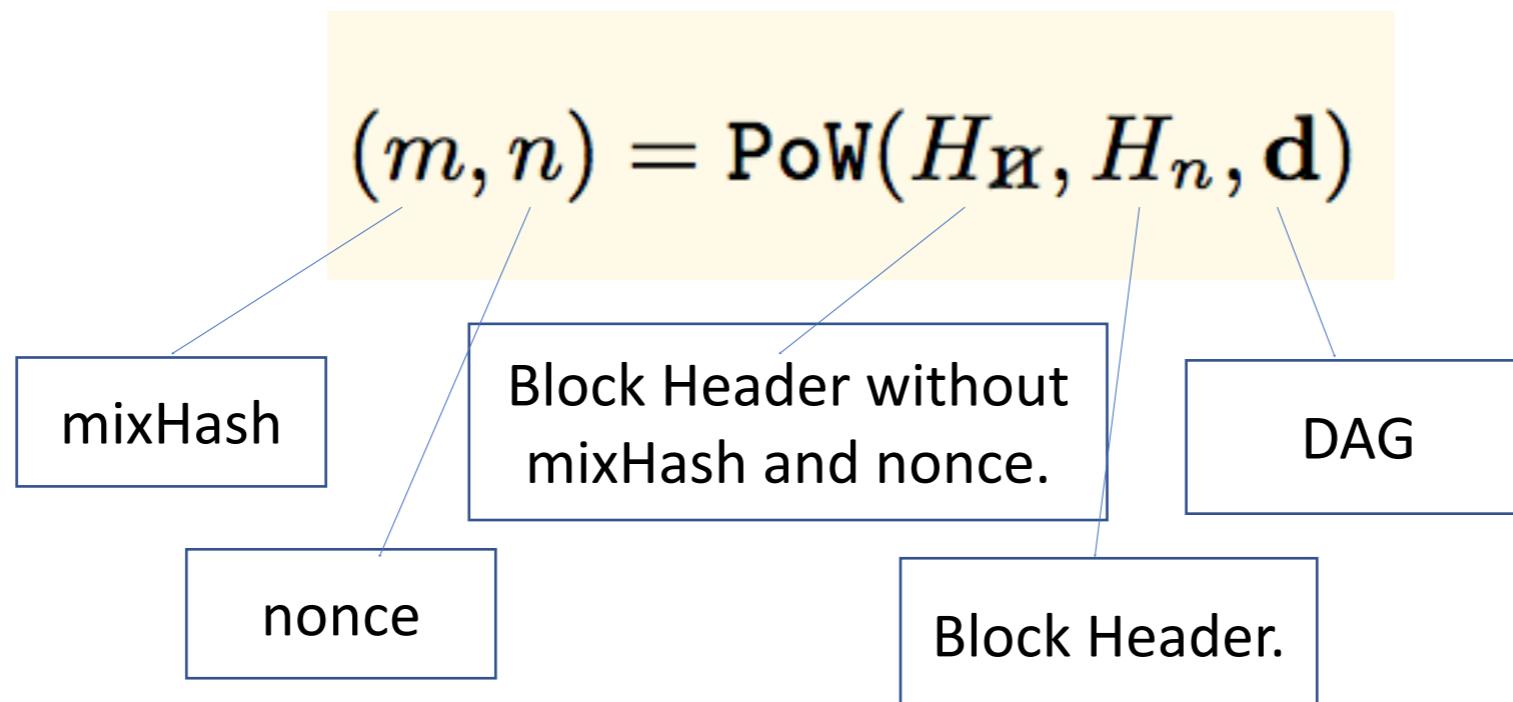
- A “light” client does not need to have the entire state of the world
 - A server can construct a Merkle proof that data exists in the state tree
 - Provided the state is encoded into the Merkle (Patricia) trees and we are given inclusion proofs, we can verify any block given just the previous block’s 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

Ommers/Uncles

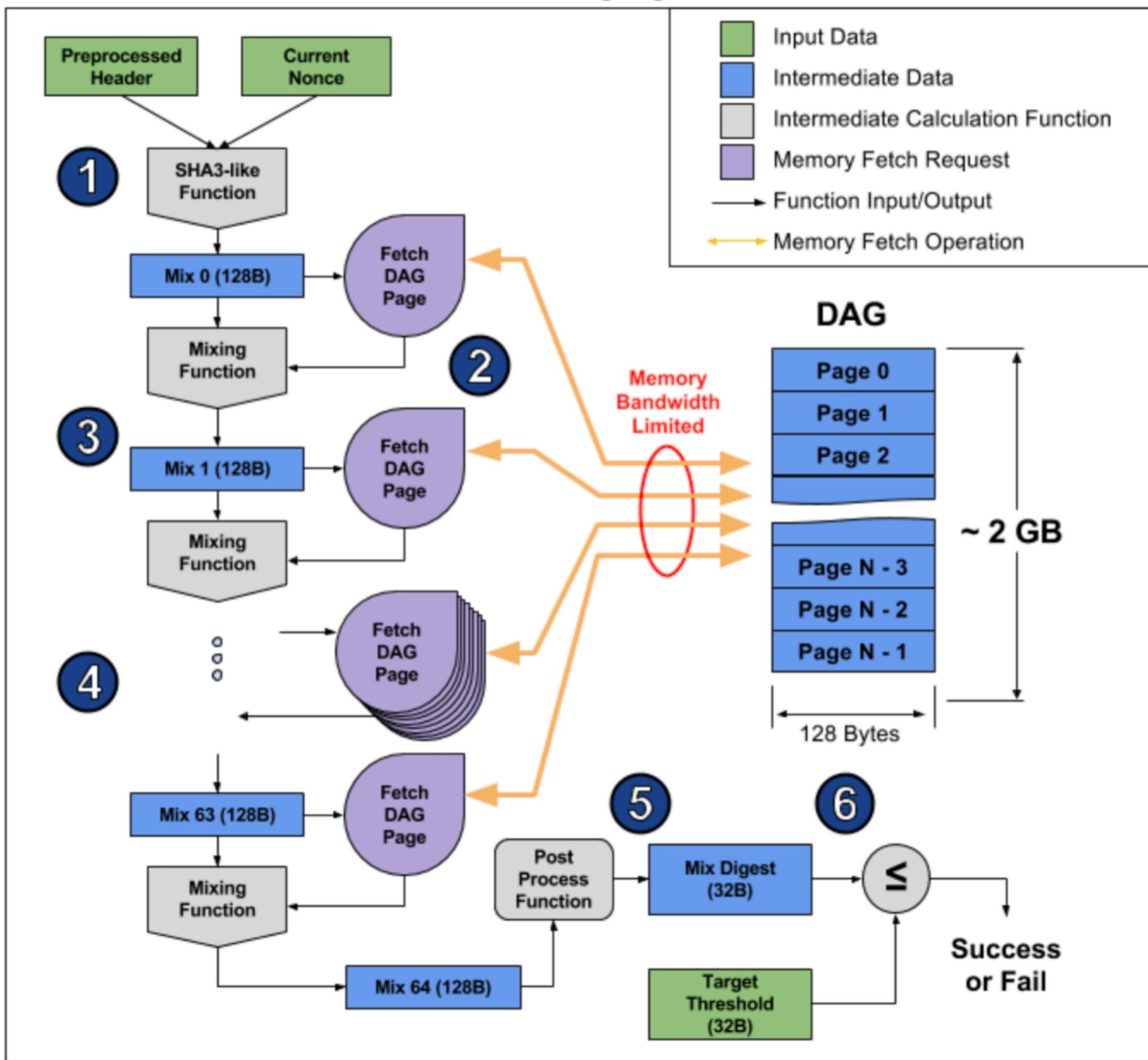
- An ommer is a block whose parent is equal to the current block's parent's parent.
- Block times in Ethereum are around 15 sec. This is much lower than that in Bitcoin (10 min).
- This enables faster transaction. But there are more competing blocks, hence a higher number of **orphaned** blocks
- The purpose of ommers is to help reward miners for including these orphaned blocks.
- The ommers that miners include must be within the sixth generation or smaller of the present block.

Mining: Proof of Work

- Ethereum's proof-of-work algorithm is called Ethash.
- Ethash is memory hard (or memory bound).
- The algorithm is formally defined as:



Ethash Hashing Algorithm



There are plans to replace Ethereum PoW with “proof of stake”, first through a “finality gadget” and later through a full PoS protocol.

This keeps getting delayed.

Contract examples

- Simple “custom token” contract (ERC20)

```
1 // -----
2 // ERC Token Standard #20 Interface
3 // https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md
4 //
5 contract ERC20Interface {
6     function totalSupply() public view returns (uint);
7     function balanceOf(address tokenOwner) public view returns (uint balance);
8     function allowance(address tokenOwner, address spender) public view returns (uint remaining);
9     function transfer(address to, uint tokens) public returns (bool success);
10    function approve(address spender, uint tokens) public returns (bool success);
11    function transferFrom(address from, address to, uint tokens) public returns (bool success);
12
13    event Transfer(address indexed from, address indexed to, uint tokens);
14    event Approval(address indexed tokenOwner, address indexed spender, uint tokens);
15 }
```

```
1 contract TokenContractFragment {
2
3     // Balances for each account
4     mapping(address => uint256) balances;
5
6     // Owner of account approves the transfer of an amount to another account
7     mapping(address => mapping (address => uint256)) allowed;
8
9     // Get the token balance for account `tokenOwner`
10    function balanceOf(address tokenOwner) public constant returns (uint balance) {
11        return balances[tokenOwner];
12    }
13
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 }
```

‘NameCoin’ in Ethereum

```
contract Namespace {  
  
    struct NameEntry {  
        address owner;  
        bytes32 value;  
    }  
  
    uint32 constant REGISTRATION_COST = 100;  
    uint32 constant UPDATE_COST = 10;  
    mapping(bytes32 => NameEntry) data;  
  
    function nameNew(bytes32 hash){  
        if (msg.value >= REGISTRATION_COST){  
            data[hash].owner = msg.sender;  
        }  
    }  
  
    function nameUpdate(bytes32 name, bytes32 newValue, address newOwner){  
        bytes32 hash = sha3(name);  
        if (data[hash].owner == msg.sender && msg.value >= UPDATE_COST){  
            data[hash].value = newValue;  
            if (newOwner != 0){  
                data[hash].owner = newOwner;  
            }  
        }  
    }  
  
    function nameLookup(bytes32 name){  
        return data[sha3(name)];  
    }  
}
```

Credit: Andrew Miller and Joe Bonneau for this code

Multisig and filters

- Can create “filter” contracts that execute another contract and/or pay out money if complex conditions are satisfied
 - E.g., If k-out-of-N signers sign (in Bitcoin this is called “multisig”)
 - Verify that a certain number of blocks have elapsed
 - Check that another contract executed

Prediction Markets

- Remember that contracts can “control” a balance (in ETH)
 - (They can control balances in e.g., ERC20 tokens as well)
 - Can make payouts of ETH conditional on certain events — e.g., signed by a notary
 - Requires: method to “place bet”
 - Method to “claim bet” (verify sig/conditions), pay to an address
 - Relies on a centralized notary! (See Augur)
 - Look at Chainlink

Future of Ethereum

- Proof of stake
- Rollups
- Sharding