Blockchains & Cryptocurrencies

Smart Contracts & Ethereum

Instructors: Matthew Green & Abhishek Jain
Johns Hopkins University - Spring 2019

Many slides adapted from NBFMG

Housekeeping

- We are a week behind…
- **Assignment 2 is coming** (Weds)
- Readings:

News?

News?

Feb 25, 2019, 08:11am

Mastercard, Amazon and Accenture Partner To Establish Transparent Blockchain Supply Chain

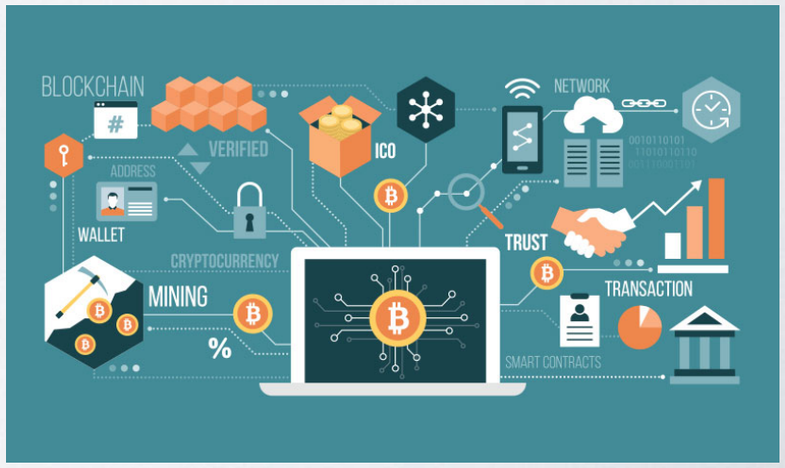
Leslie Ankney Contributor ⓘ

Crypto & Blockchain

I cover blockchain tech, cryptocurrencies and their implications.

Today

- From Bitcoin to smart contracts
- Ethereum
- Applications of smart contracts



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)

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

```
  76        A9              14
OP_DUP OP_HASH160     Bytes to push

89 AB CD EF AB BA AB BA AB BA AB BA AB BA AB BA 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

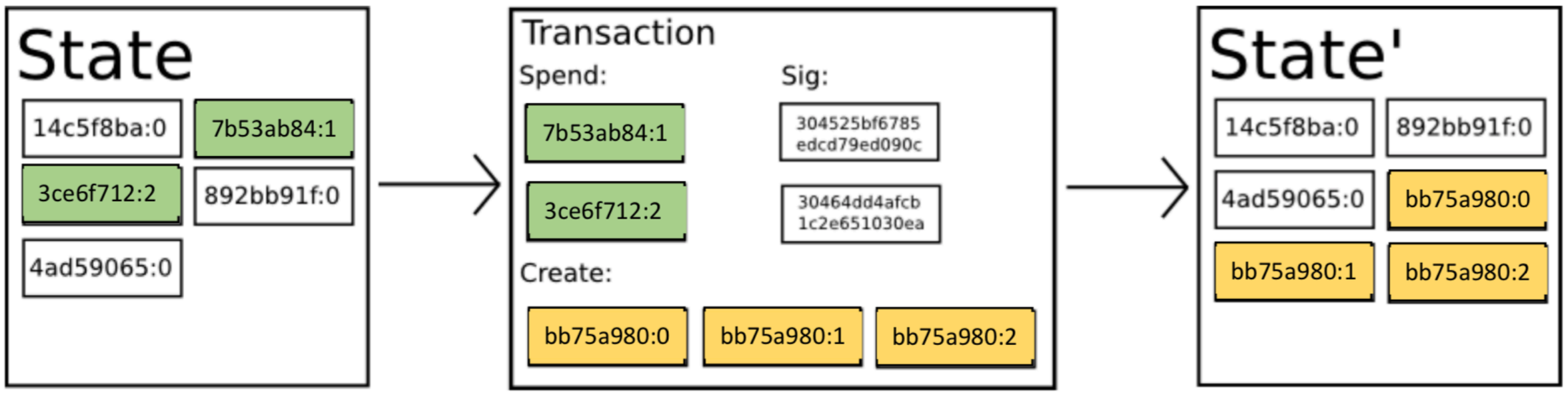
- **Bitcoin script allows us to attach conditions to payments**
  - However script is deliberately *limited*
    - Stack-based FORTH-type language, many original opcodes disabled
    - Highly limited access to global data

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: 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
    - Example: implement a second currency/asset
    - Example: pay out a coin iff a candidate wins the US
    - Example: pay out a coin iff a majority of users votes to invest in a service

Bitcoin ✅

Bitcoin ✅👎

Bitcoin 👎

Bitcoin 👎

Output coin condition:

This coin can be paid out to user **A** on 1/20/2020 if the transaction contains a signature (under some "notary" public key) of the following candidate name (**X**)

Or

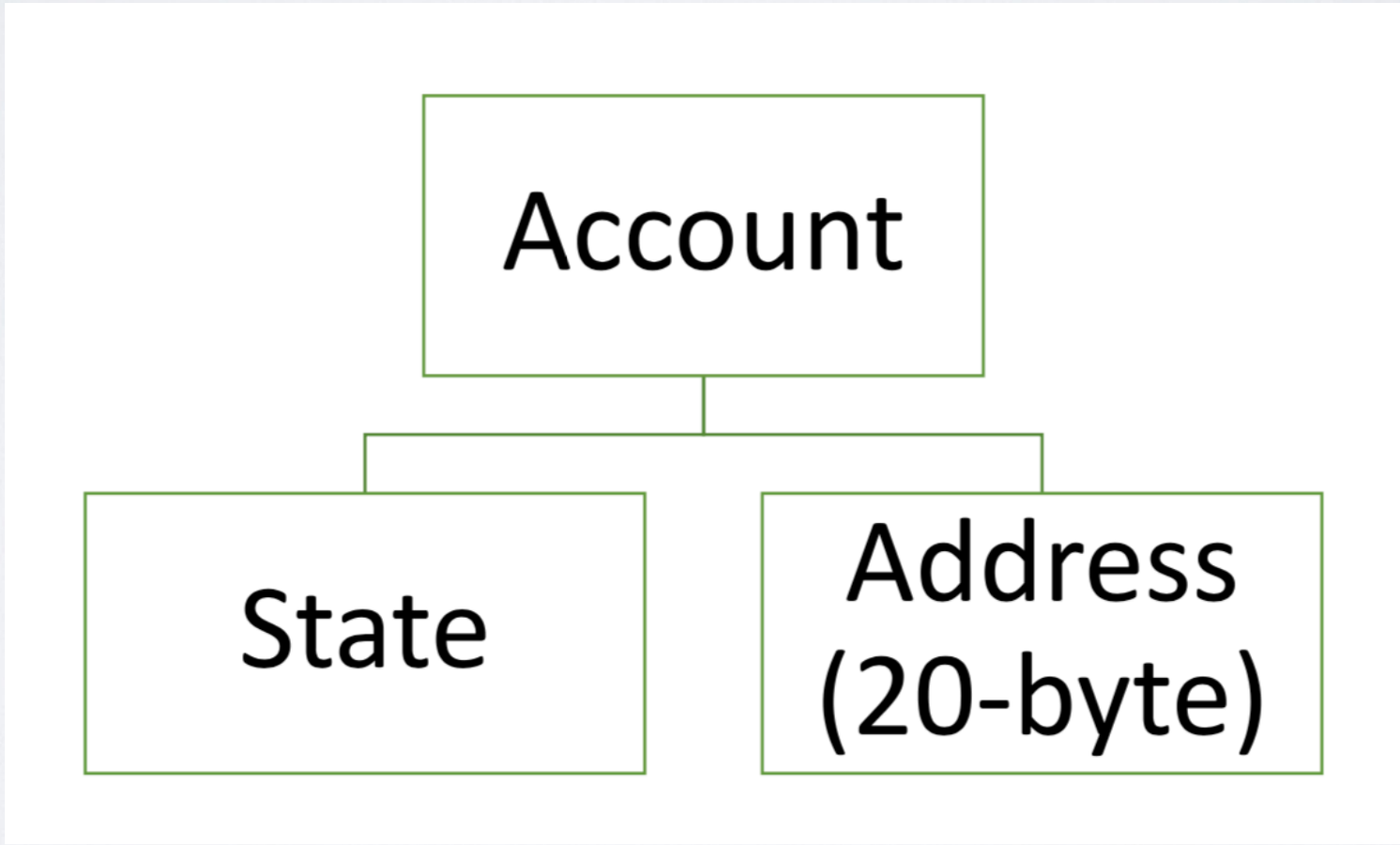
This coin can be paid out to user **B** on 1/20/2020 if the transaction contains a signature (under some "notary" public key) of the following candidate name (**Y**)

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" (only sort of a problem)

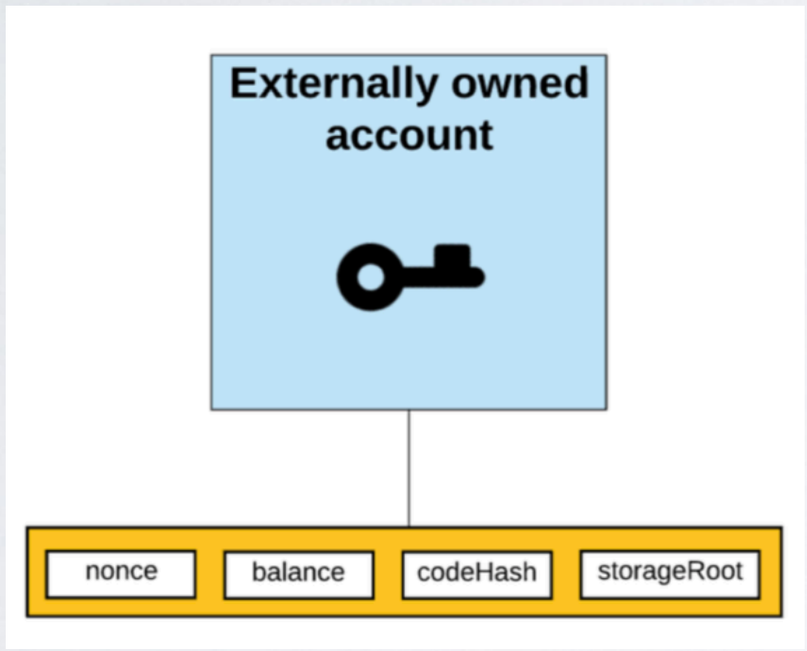
Ethereum

- Account-based currency
  - Many small "accounts"

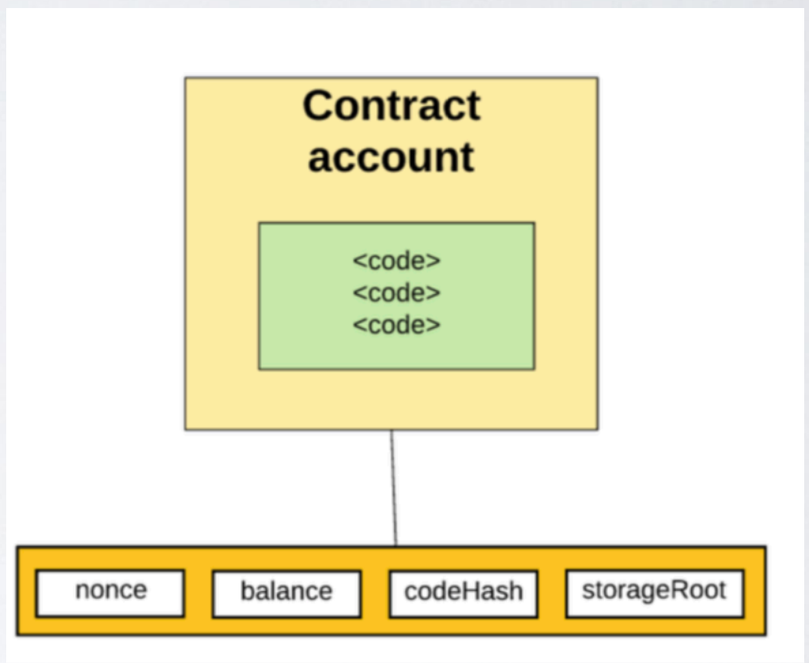


Ethereum: Accounts

- 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.

Ethereum: Accounts

nonce: # transactions sent/ # contracts created

- balance: # Wei owned (1 ether=$10^{\#\$}$Wei)
- storageRoot: Hash of the root node of a Merkle Patricia tree. The tree is empty by default.
- codeHash: Hash of empty string / Hash of the EVM (Ethereum Virtual Machine) code of this account

Ethereum: Accounts

nonce

• bala

• stor

tree i

• cod

Virtu

. The

eum

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.)

nonce

• bala

• stor

tree i

• cod

Virtu

(I.e., t

st

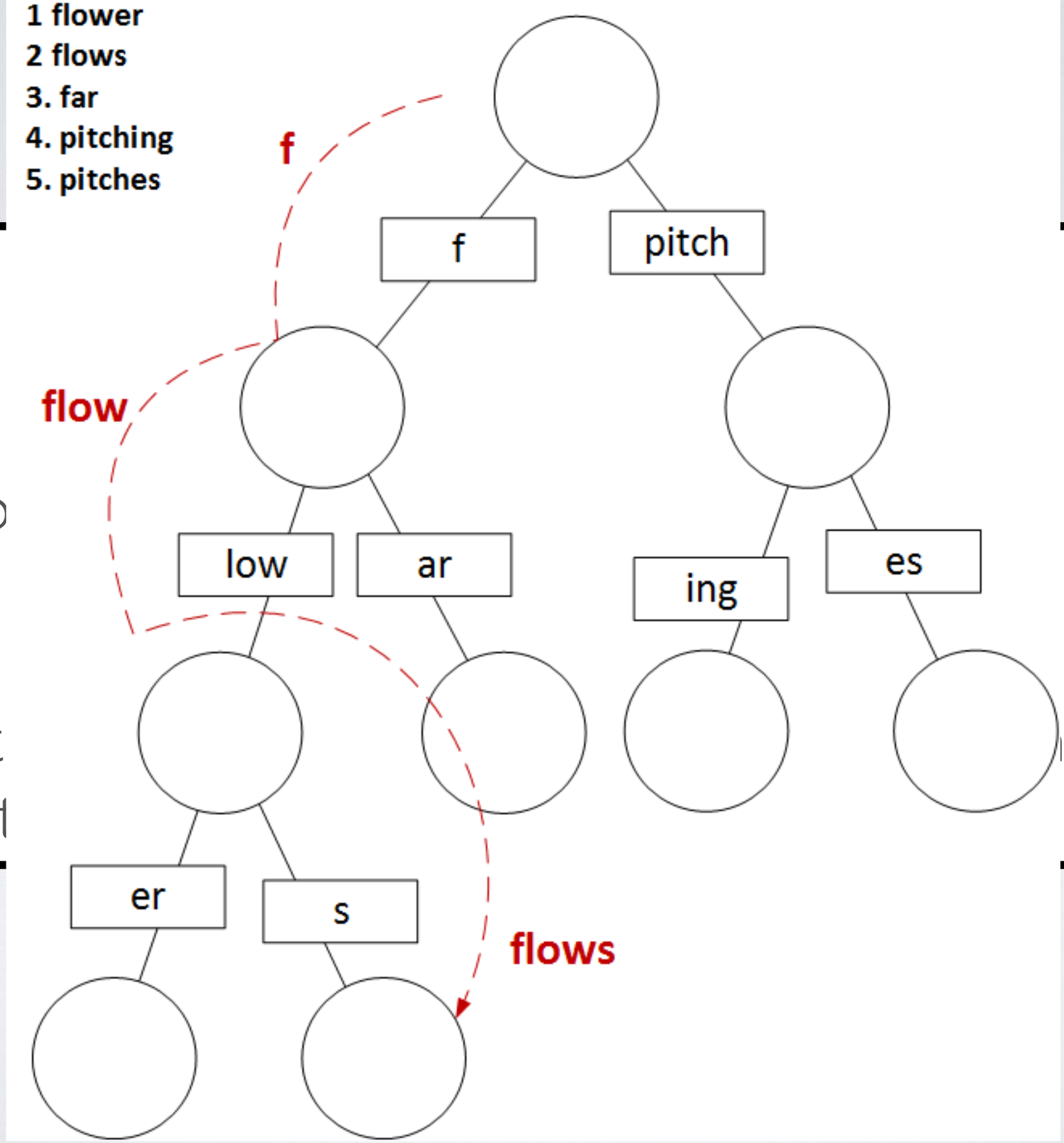
p

e. The

eum

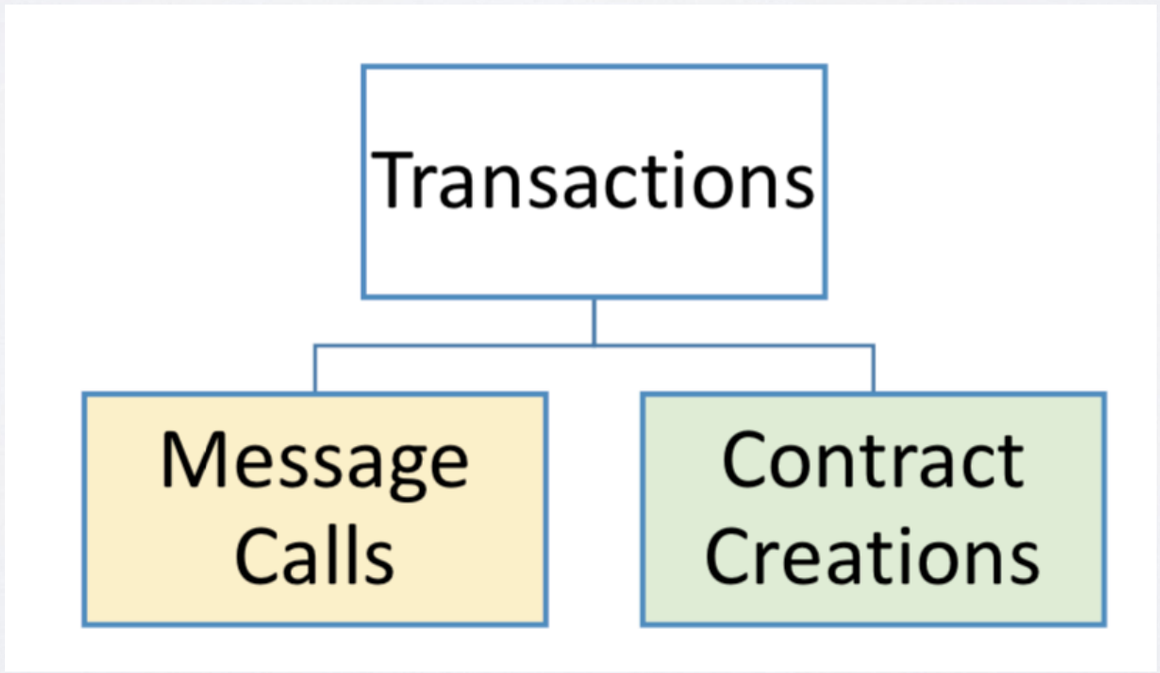
the

1 flower
2 flows
3. far
4. pitching
5. pitches

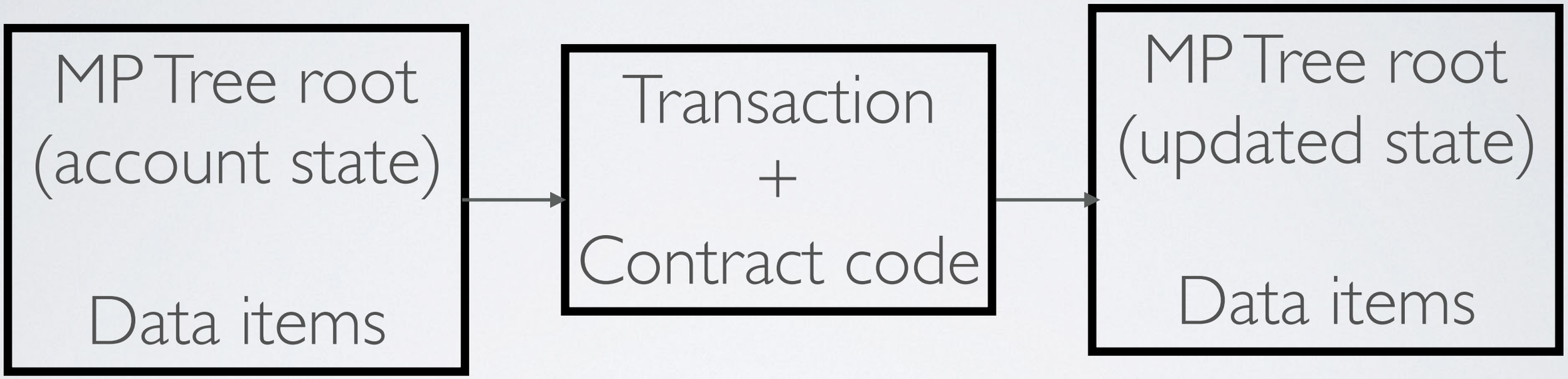


Ethereum Transactions

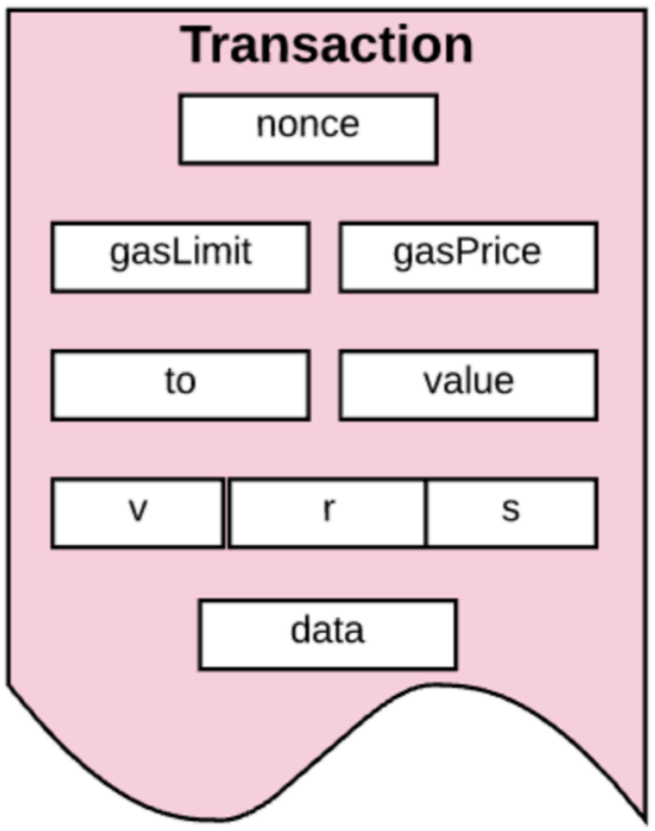
- Two types:
  - Message calls: update state in a given contract, by executing code (or simply transferring money)
  - Contract creations: make a new contract account, with new state



Ethereum: Accounts



Ethereum Transactions

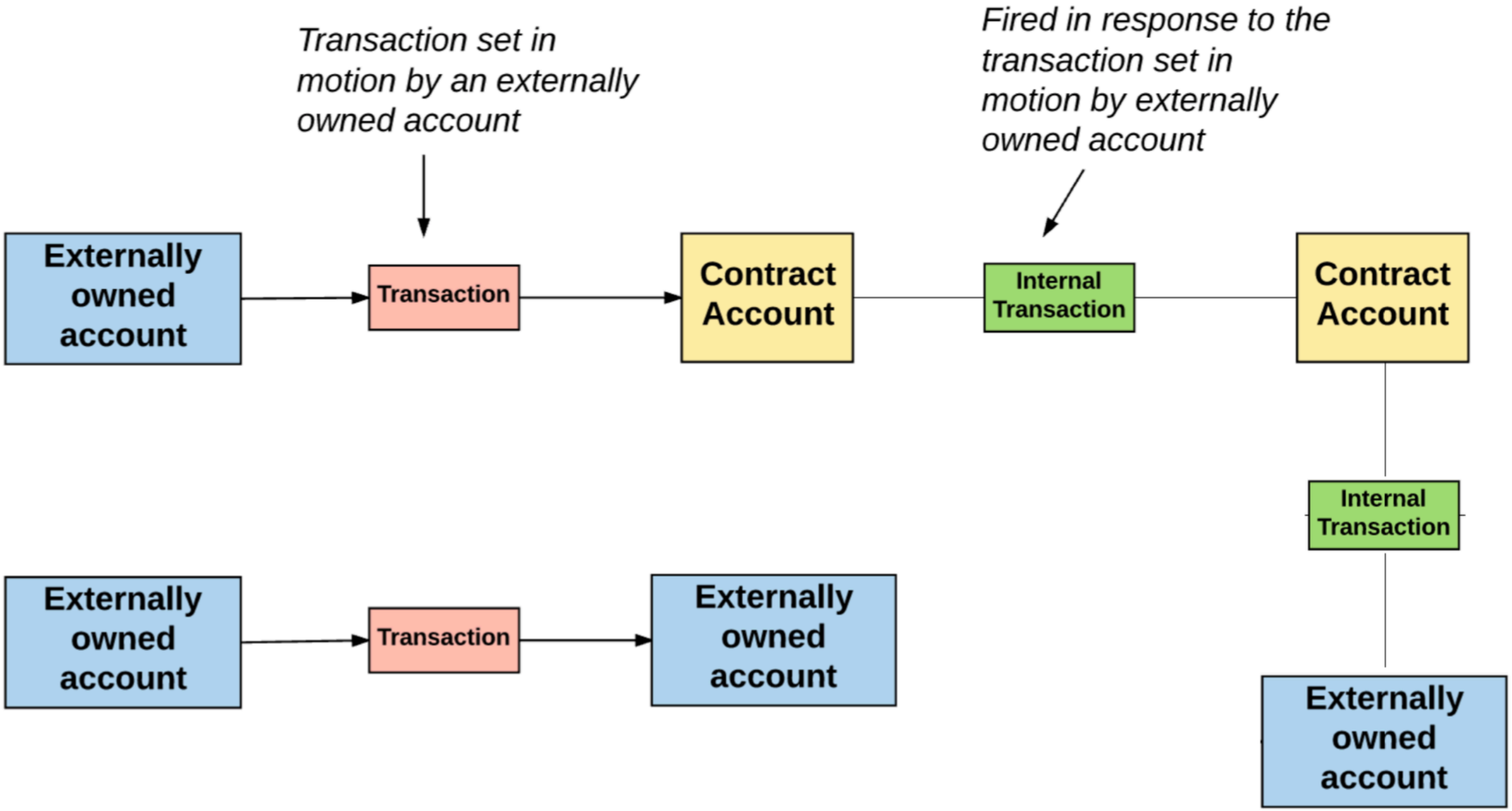


- nonce: A count of the number of transactions sent by the sender.
- gasPrice
- gasLimit
- to: Recepient's address
- value: Amount of Wei Transferred from sender to recipient.
- v,r,s: Used to generate the signature that identifies the sender of the transaction.
- init: EVM code used to initialize the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

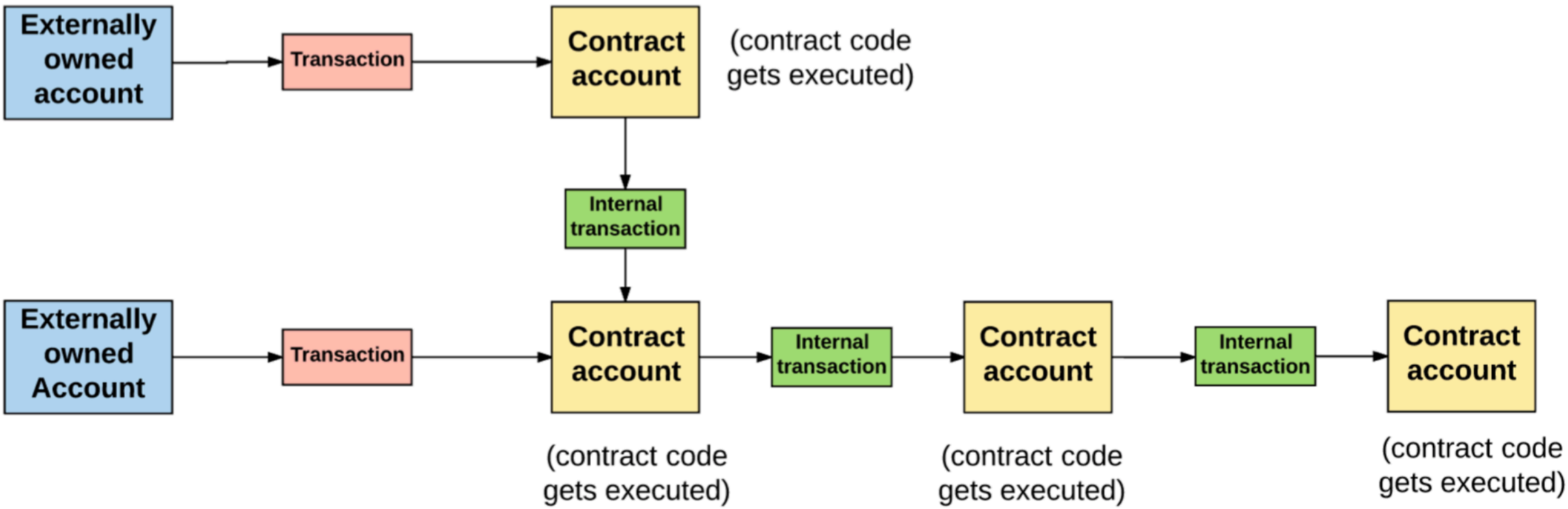
Transaction Types

- 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!

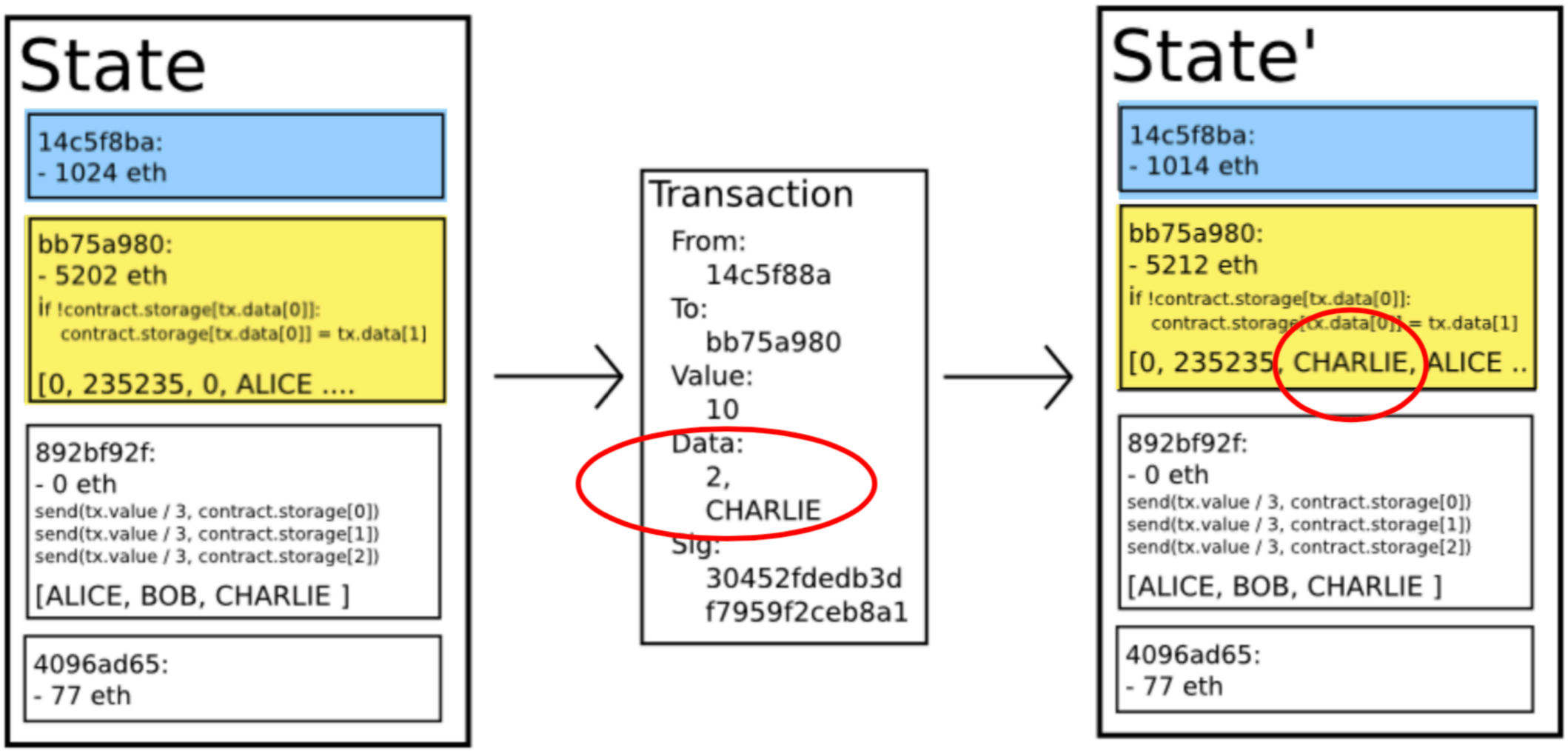
Transaction Types



Transaction Flow



Transaction Execution (Pay)



Contracts

- 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 contr
  object-orien
  - The "cont
    public or
- Public me
  can only b
- External t
  (argumen

```sol
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 {
```

ethods

+ the 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

To prevent cheating, the network works like Bitcoin:

Every single node in the network must also run the EVM machine 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?

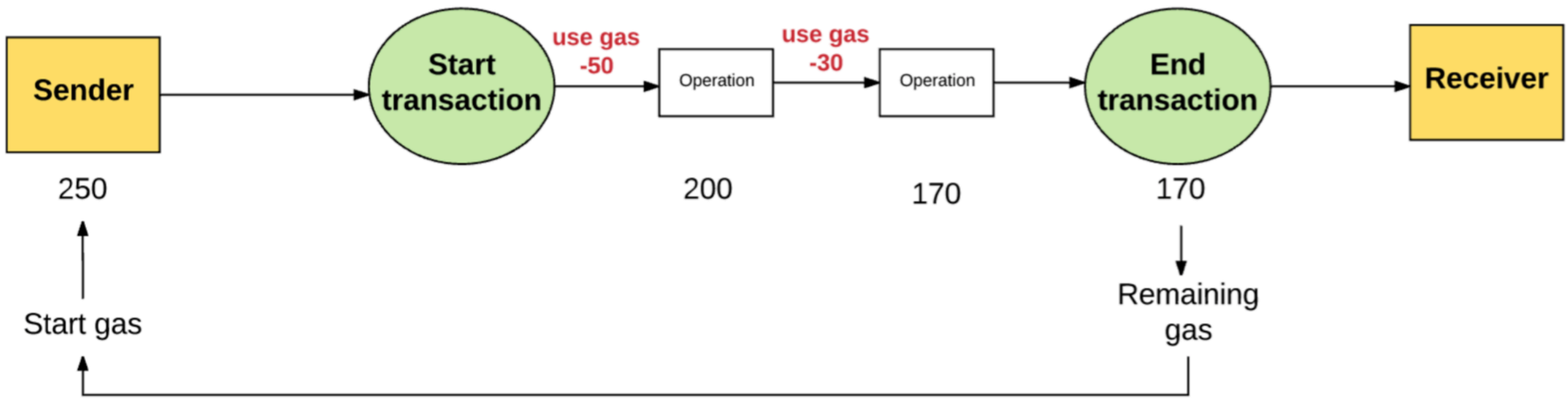
- EVM code is "Turing complete", e.g., has loops and may not halt
  - Users pay fees to submit TXes to the network
    - (Pays for both computation and storage of data)
      - Contract computation/storage costs are denominated in "gas" (e.g., one hash call requires 30 "gas")
      - Each transaction 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.

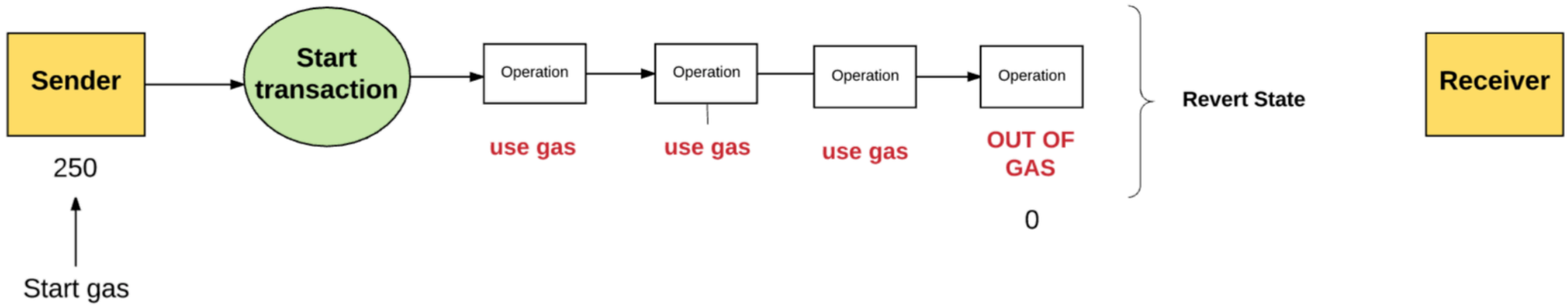
Gas limits/price

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



Gas limits/price

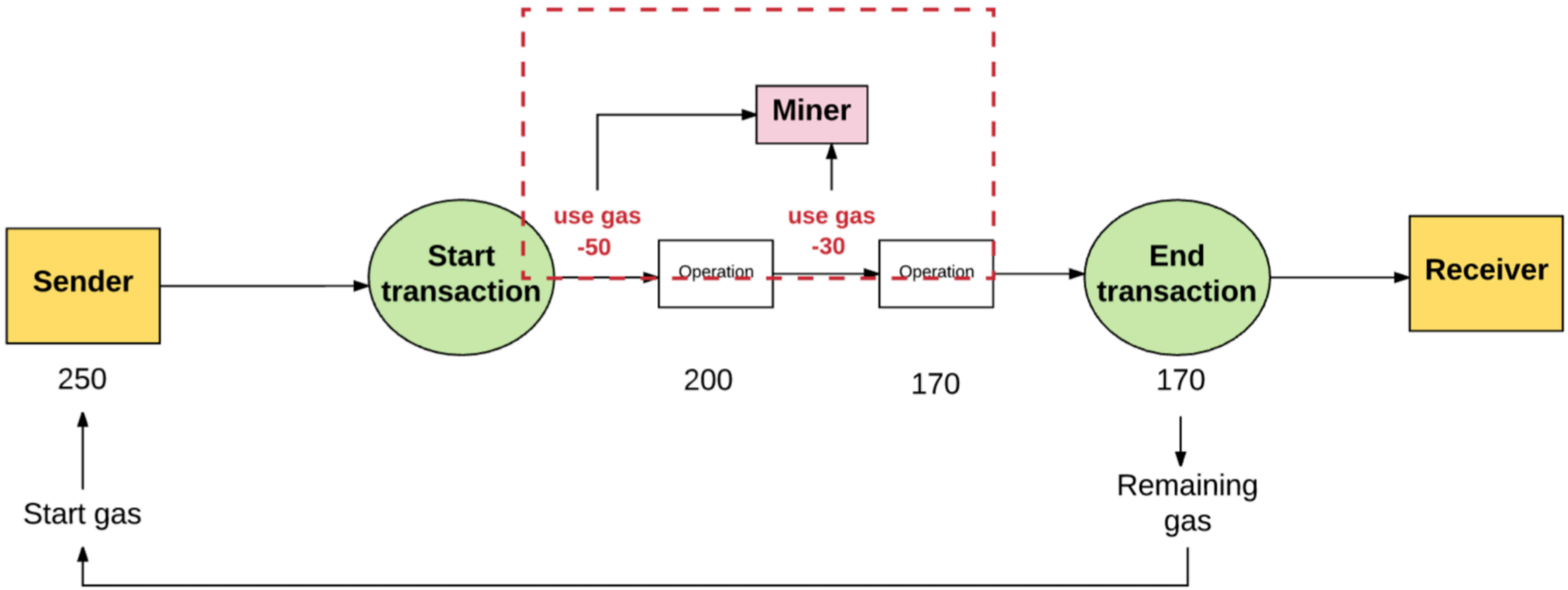
If sender does not provide the necessary gas to execute the transaction, the transaction runs “out of gas” and is considered invalid.



- 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.



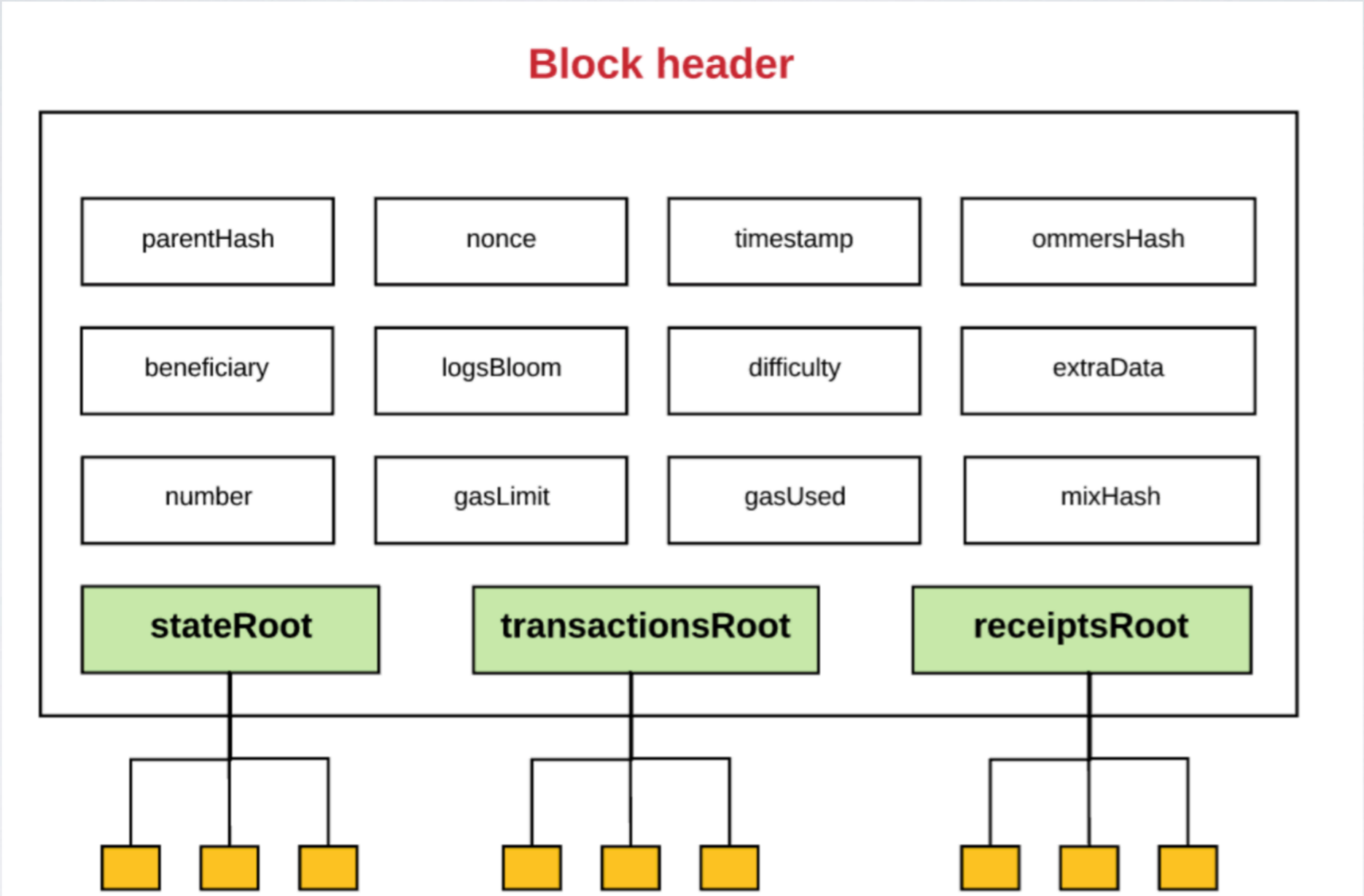
Incentive problems?

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

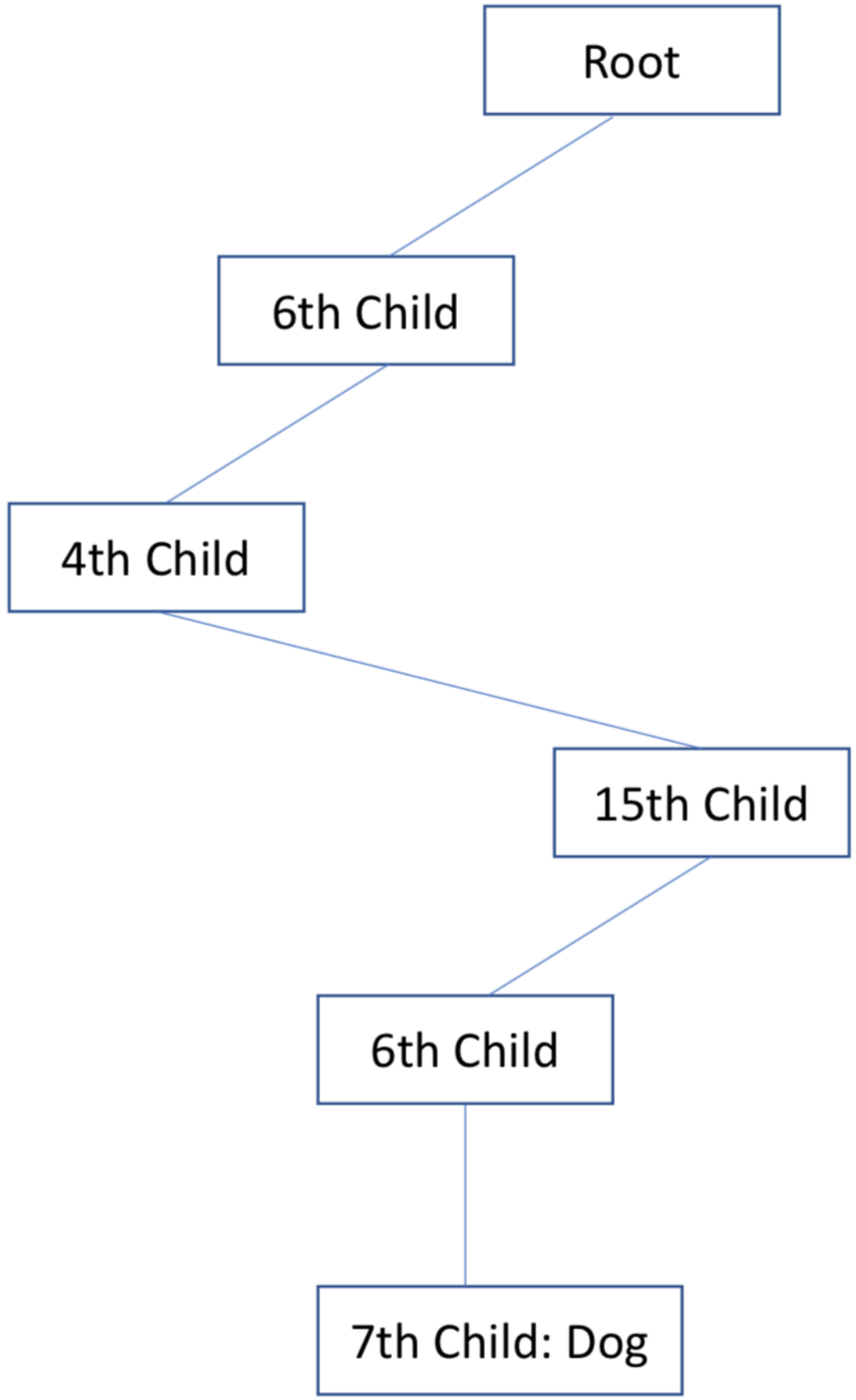
  - A server can construct a Merkle proof that data exists in the state tree

- 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

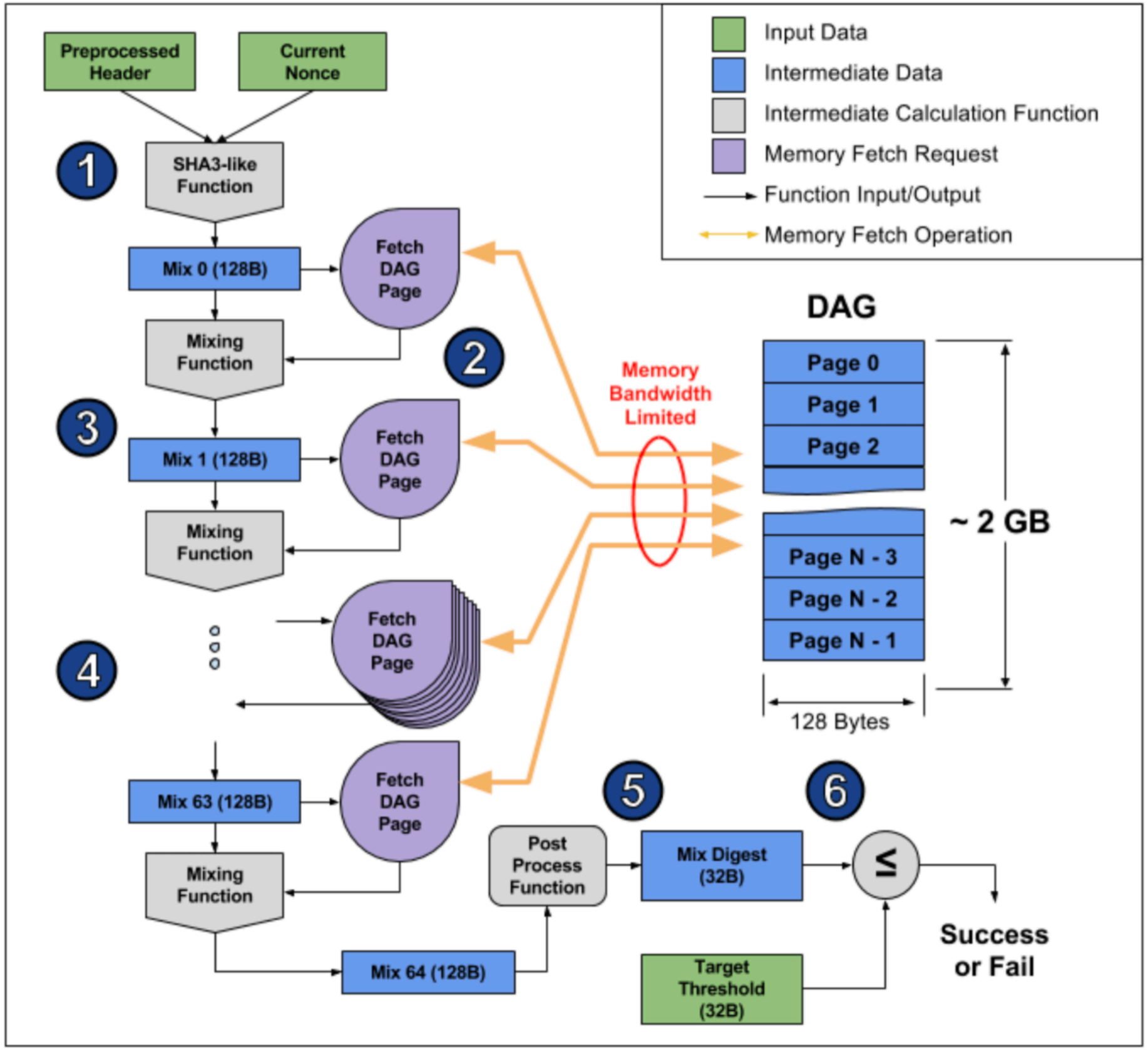
- Full nodes do need to keep current state data in order to work, but can "prune" (throw away) old states that are no longer valid

Merkle Patricia Trees

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



Ethash Hashing Algorithm



Constructing the Chain

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

Constructing the Chain

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.

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.

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

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 }
```

```
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;
    }
}
```

"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….

DAO

- Decentralized Autonomous Organization
  - "Like a VC fund" but decentralized
  - Implementation: a contract that controls money, and directs its disbursement according to "shareholder votes"
- Shareholders buy in, pool their ETH (sending to contract)
- Then vote on investments, which are made together
- Users can "split" a DAO

“The DAO”

```
function splitDAO(
  uint _proposalID,
  address _newCurator
) noEther onlyTokenholders returns (bool _success) {

  ...
  // XXXXX Move ether and assign new Tokens.  Notice how this is done first!
  uint fundsToBeMoved =
      (balances[msg.sender] * p.splitData[0].splitBalance) /
      p.splitData[0].totalSupply;
  if (p.splitData[0].newDAO.createTokenProxy.value(fundsToBeMoved)(msg.sender)
== false) // XXXXX This is the line the attacker wants to run more than once
      throw;

  ...
  // Burn DAO Tokens
  Transfer(msg.sender, 0, balances[msg.sender]);
  withdrawRewardFor(msg.sender); // be nice, and get his rewards
  // XXXXX Notice the preceding line is critically before the next few
  totalSupply -= balances[msg.sender]; // XXXXX AND THIS IS DONE LAST
  balances[msg.sender] = 0; // XXXXX AND THIS IS DONE LAST TOO
  paidOut[msg.sender] = 0;
  return true;
}
```

How to upgrade a contract?

- If there is no useful ongoing state, don't: just replace it
- If there is ongoing state (e.g., account balances) then:
  - Don't allow upgrades — and pray you got the code right
  - Call upgradeable/replaceable library code
  - Create a complex mechanism to transfer state from an old contract instance to a new contract instance