Blockchains & Cryptocurrencies

Smart Contracts II / Ethereum
feat. random pictures of my dogs

Instructor: Matthew Green & Abhishek Jain
Spring 2023

News?

News?

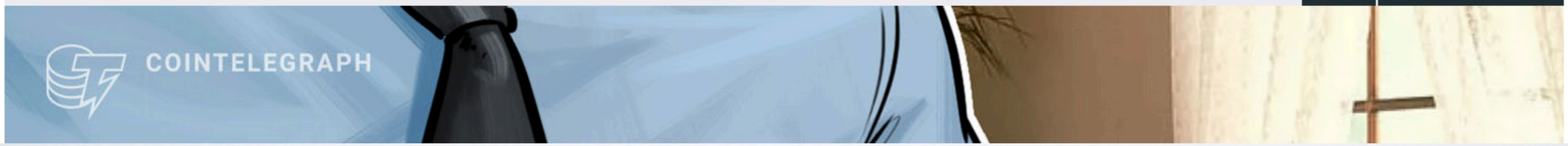
BRAYDEN LINDREA

FEB 20, 2023

‘Unworkable’ bill to ban blockchain immutability is introduced in Illinois

Florida-based lawyer Drew Hinkes described the bill as “the most unworkable state law” related to blockchain and cryptocurrency that he has ever seen.

8306 Total views **238** Total shares Listen to article 5:23



News?

SYNOPSIS AS INTRODUCED:

New Act

Creates the Digital Property Protection and Law Enforcement Act. Provides that upon a valid request from the Attorney General or a State's Attorney, made pursuant to the substantive or procedural laws of the State, a court may order any appropriate blockchain transaction for digital property or for the execution of a smart contract. Provides that a blockchain network that processes a blockchain transaction originating in the State at any time after the effective date of the Act shall process a court-ordered blockchain transaction without the need for the private key associated with the digital property or smart contract. Provides that upon a petition by the Attorney General or a State's Attorney, the court shall assess a civil penalty of between $5,000 and $10,000 for each day that the blockchain network fails to comply with the order. Sets forth

News?

PRESS RELEASES

U.S. Treasury Sanctions Notorious Virtual Currency Mixer Tornado Cash

August 8, 2022

WASHINGTON – Today, the U.S. Department of the Treasury's Office of Foreign Assets Control (OFAC) sanctioned virtual currency mixer Tornado Cash, which has been used to launder more than $7 billion worth of virtual currency since its creation in 2019. This includes over $455 million stolen by the Lazarus Group, a Democratic People's Republic of Korea (DPRK) state-sponsored hacking group that was sanctioned by the U.S. in 2019, in the largest known virtual currency heist to date. Tornado Cash was subsequently used to launder more than $96 million of malicious cyber actors' funds

News?

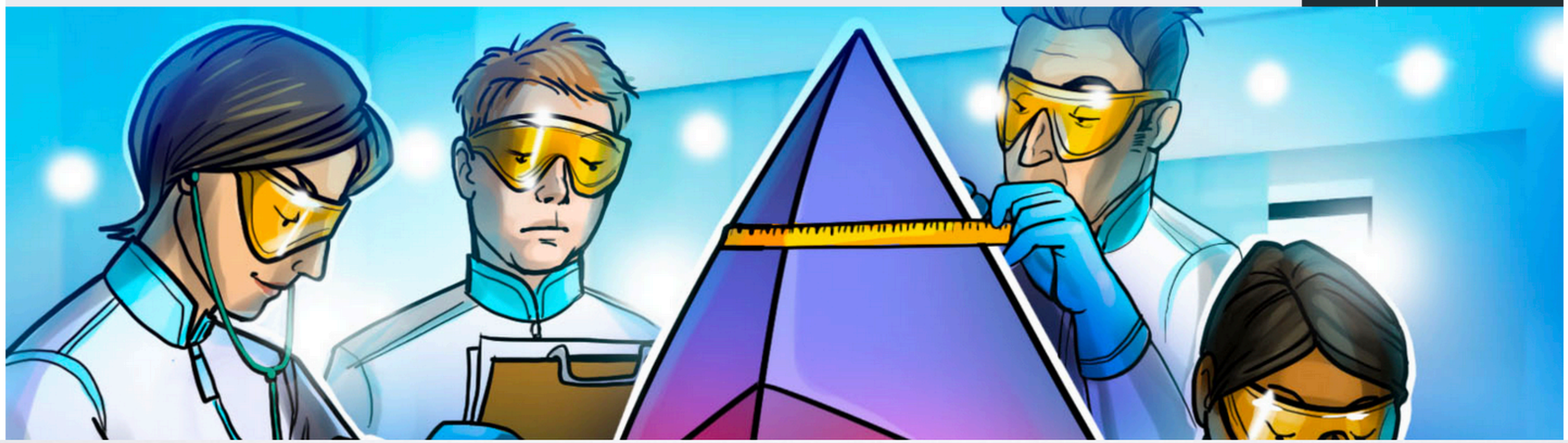
BRAYDEN LINDREA

FEB 14, 2023

OFAC-compliant blocks on Ethereum hit three-month low of 47%

Back in November, the percentage of Ethereum blocks complying with orders from the Office of Foreign Asset Control peaked at 79%.

3686 Total views **15** Total shares Listen to article 2:56



Today

- Finish up our basic intuition for how a smart contract blockchain might work
- Then: Ethereum!

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: what could we do with more programmability?

Example: custom tokens

- Bitcoin supports a single currency (BTC)
  - You can spin up a new network (e.g., Litecoin, Dogecoin, etc.)
  - Can we support a second currency on Bitcoin?
    - Applications: coupons, stablecoins, NFTs
    - Major calls: **Mint** (e.g., centralized party), **Burn**, **Pay**

Example 2: prediction market

- Simple program that executes the following:
  - Accepts "bets" on one of two possible outcomes (e.g., Presidential election outcomes)
  - Maintains odds based on all previous bets (simple calculation)
  - Receives a signed message from a pre-chosen judge (e.g., "Candidate A wins!")
  - Pays funds to all winners

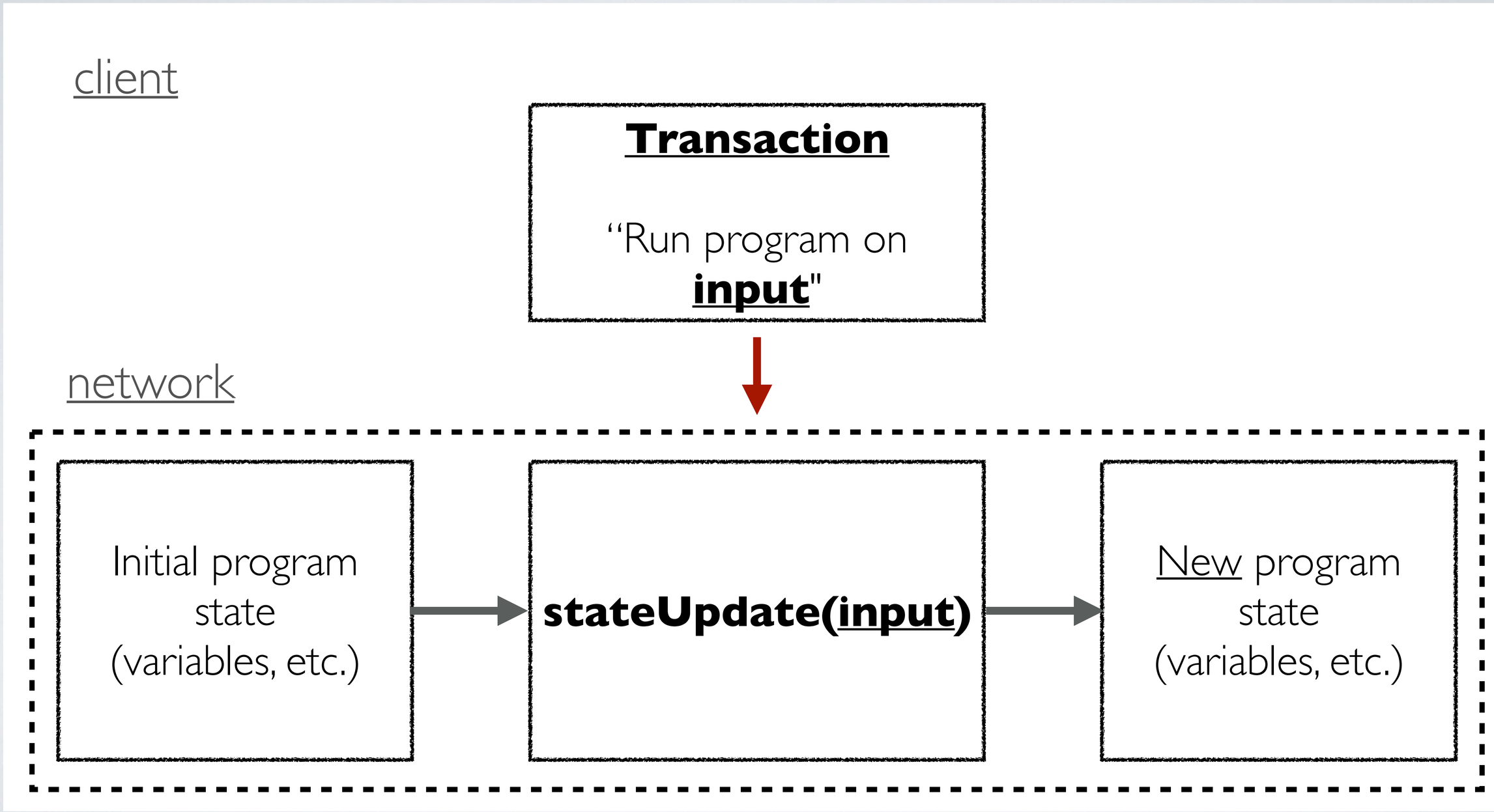
Example 3: decentralized exchange

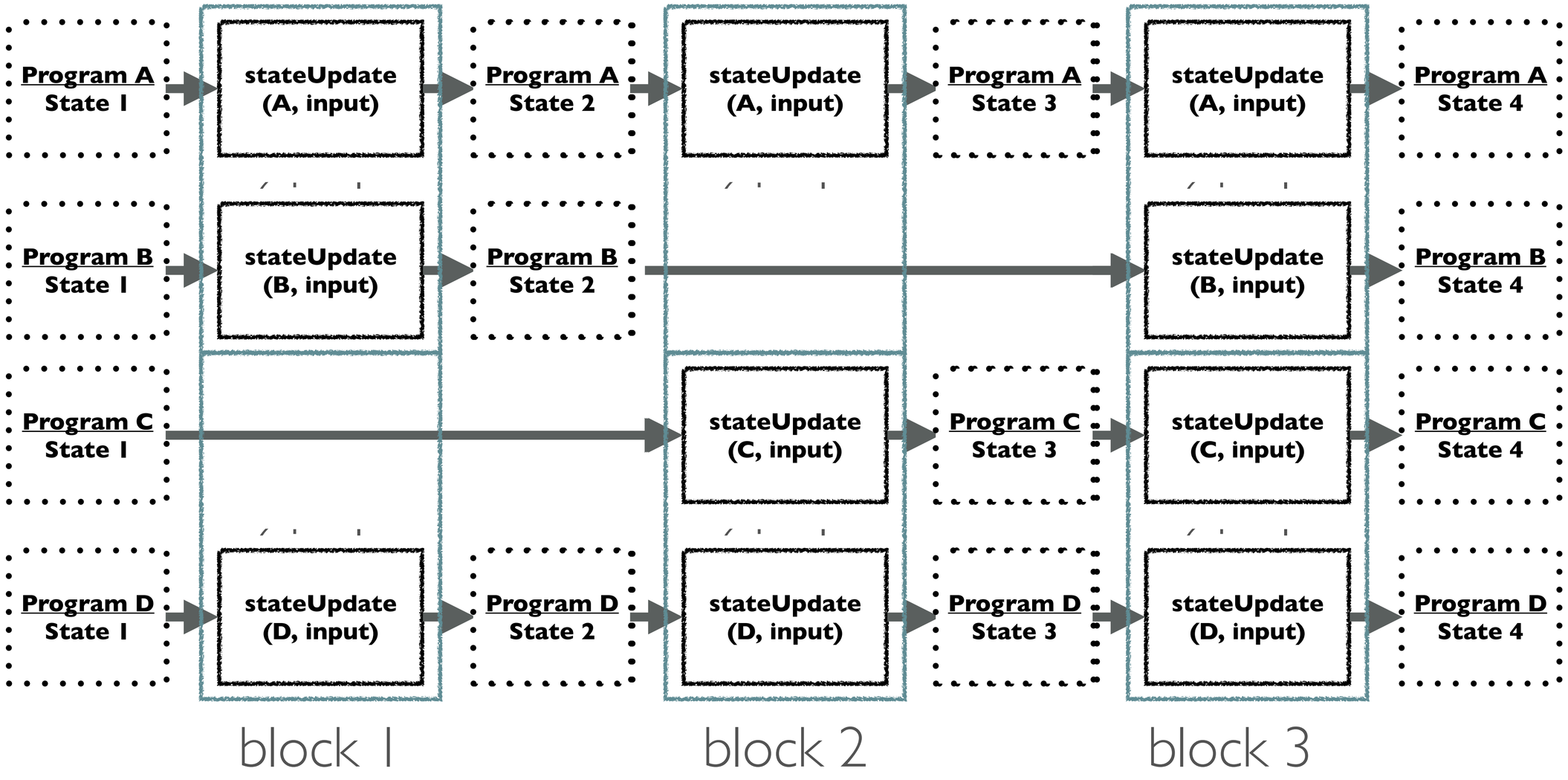
- Assume we have at least two different "tokens"
  - Accept deposits of Token A/Token B
    - Keep a separate for both tokens for each user
  - Build a program that can accept "bids" and "asks" to exchange Token A for Token B
  - Match bids to complete trades
  - Allow withdrawals of the exchanged tokens

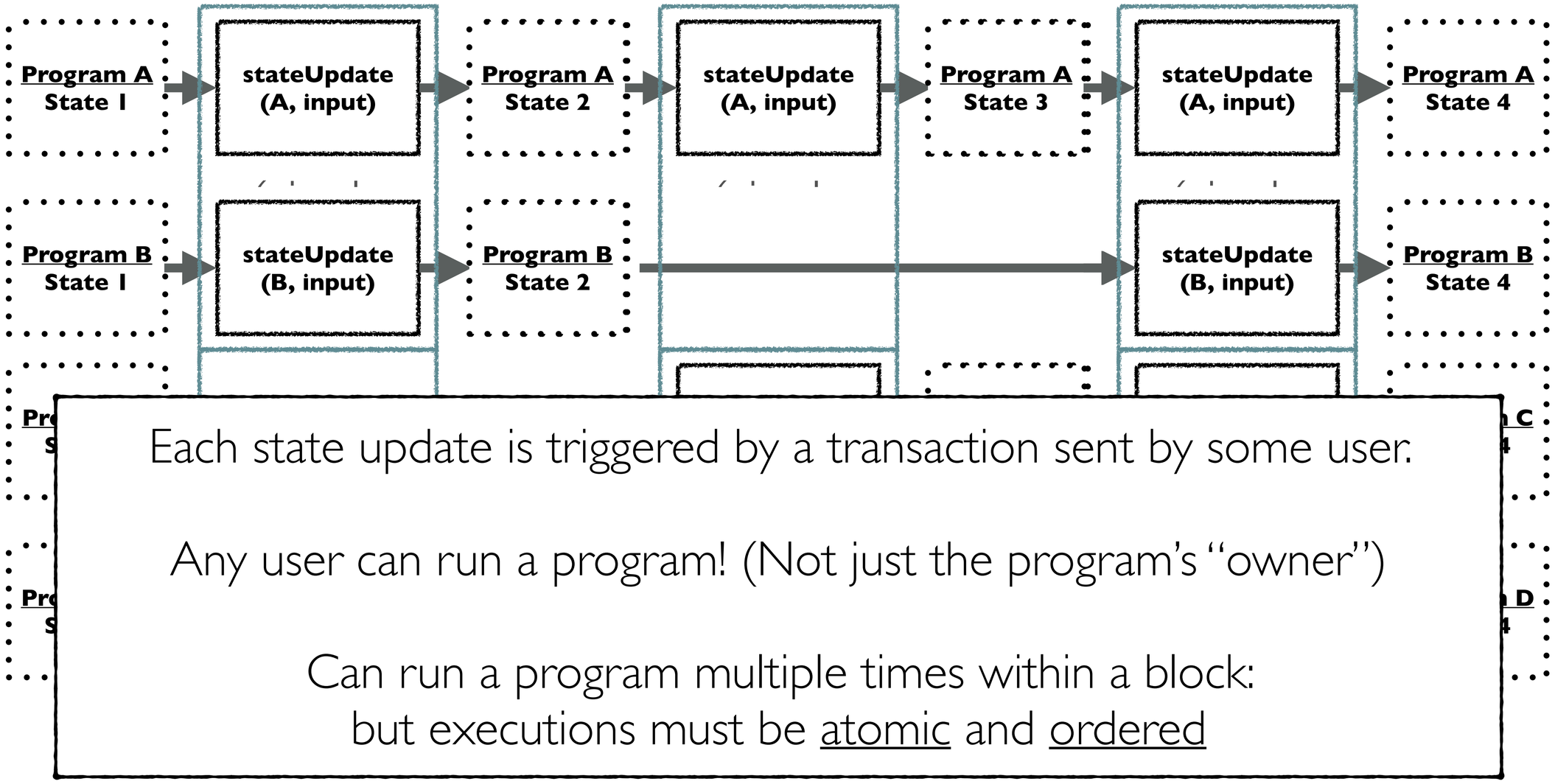
Example 4: "algorithmic stablecoin"

- Cryptocurrencies like BTC and ETH have "unstable" value against fiat currencies (e.g., BTC-USD)
  - Is there some way we could make a virtual currency…
  - Backed by a pile of floating currencies, e.g., ETH/BTC etc.
  - Such that the value of the currency stays "close" to $1?

Review: what might a programmable architecture look like?


client
Transaction
"Run program on input"
network
Initial program state (variables, etc.)
stateUpdate(input)
New program state (variables, etc.)


Program A
State 1
stateUpdate
(A, input)
Program A
State 2
stateUpdate
(A, input)
Program A
State 3
stateUpdate
(A, input)
Program A
State 4
Program B
State 1
stateUpdate
(B, input)
Program B
State 2
stateUpdate
(B, input)
Program B
State 4
Program C
State 1
stateUpdate
(C, input)
Program C
State 3
stateUpdate
(C, input)
Program C
State 4
Program D
State 1
stateUpdate
(D, input)
Program D
State 2
stateUpdate
(D, input)
Program D
State 3
stateUpdate
(D, input)
Program D
State 4
block 1
block 2
block 3



Each state update is triggered by a transaction sent by some user.

Any user can run a program! (Not just the program's "owner")

Can run a program multiple times within a block:
but executions must be atomic and ordered

Transaction

"Run program **A** on **input**"

stateUpdate (A, input)

calls program B as a "subroutine"

stateUpdate (B, args)

Transaction

"Run program **A** on **input**"

stateUpdate (A, input)

calls program B as a "subroutine"

args

stateUpdate (B, args)

1. An external user calls program A
2. Program A calls program B as a subroutine

Transaction

"Run program **A** on **input**"

stateUpdate (A, input)

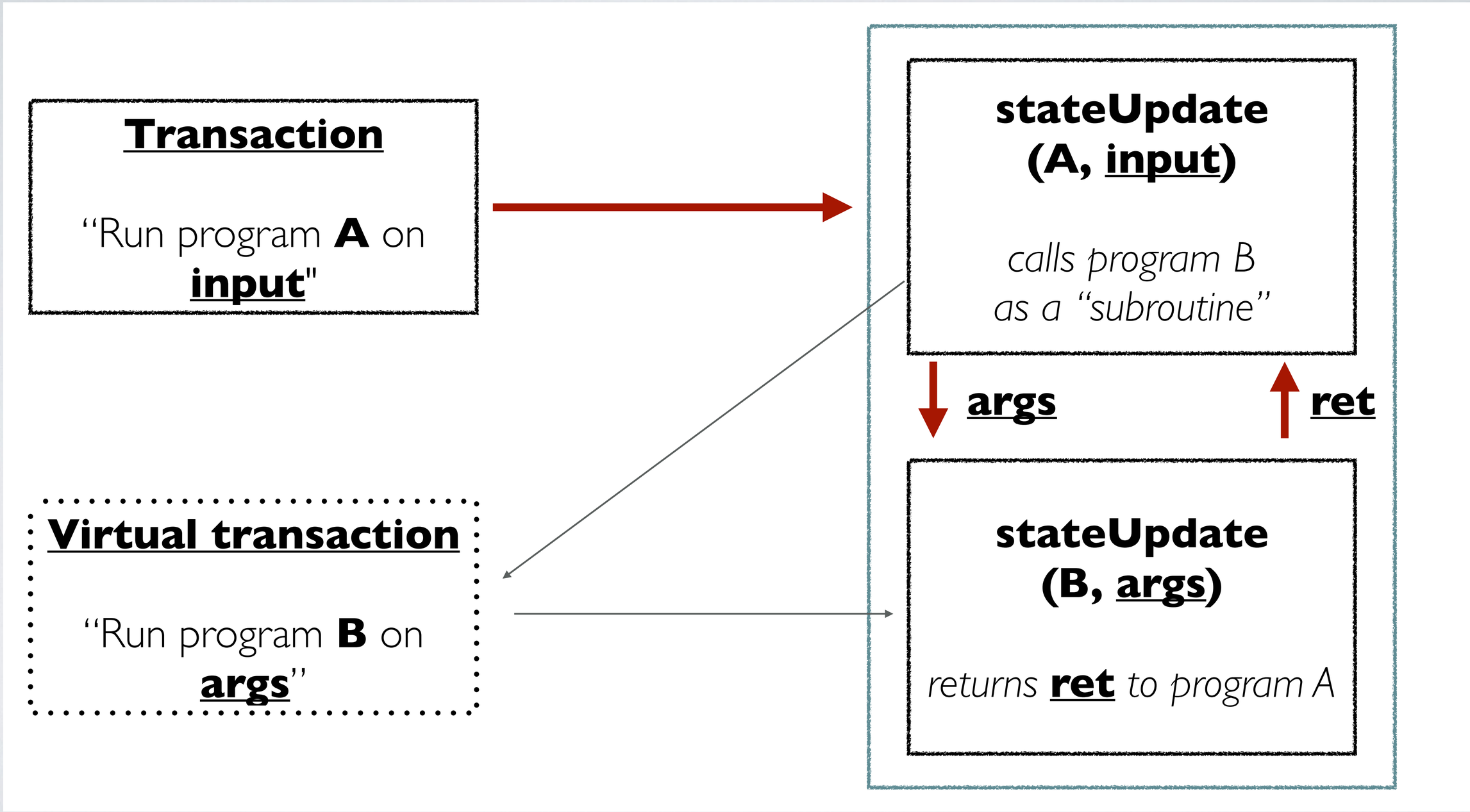
calls program B as a "subroutine"

args ↓ ↑ **ret**

stateUpdate (B, args)

returns **ret** *to program A*

1. An external user calls program A
2. Program A calls program B as a subroutine
3. Program B returns **ret** to A (and updates its own state)


Transaction
"Run program A on input"
stateUpdate (A, input)
calls program B as a "subroutine"
args
ret
Virtual transaction
"Run program B on args"
stateUpdate (B, args)
returns ret to program A

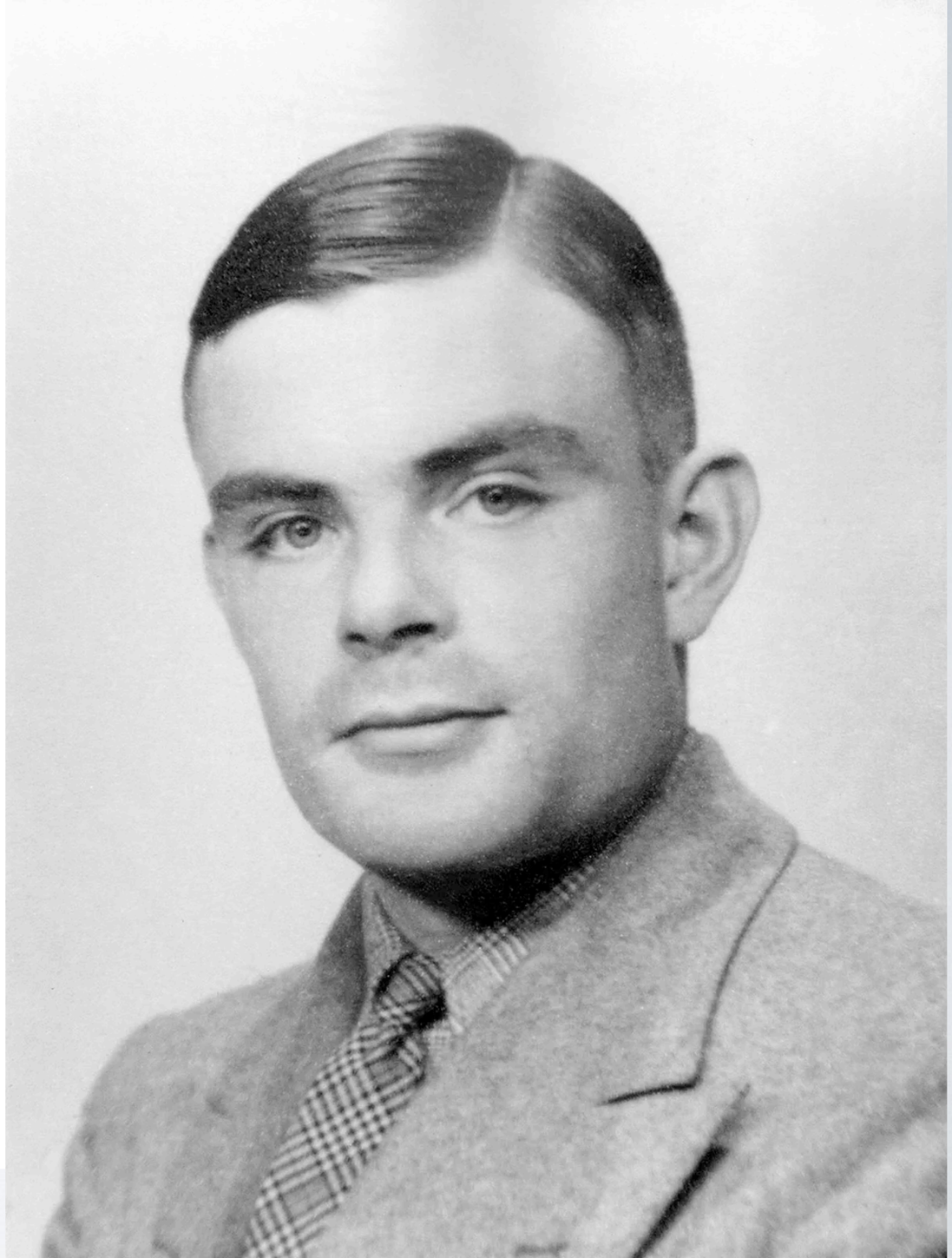
You should have many questions

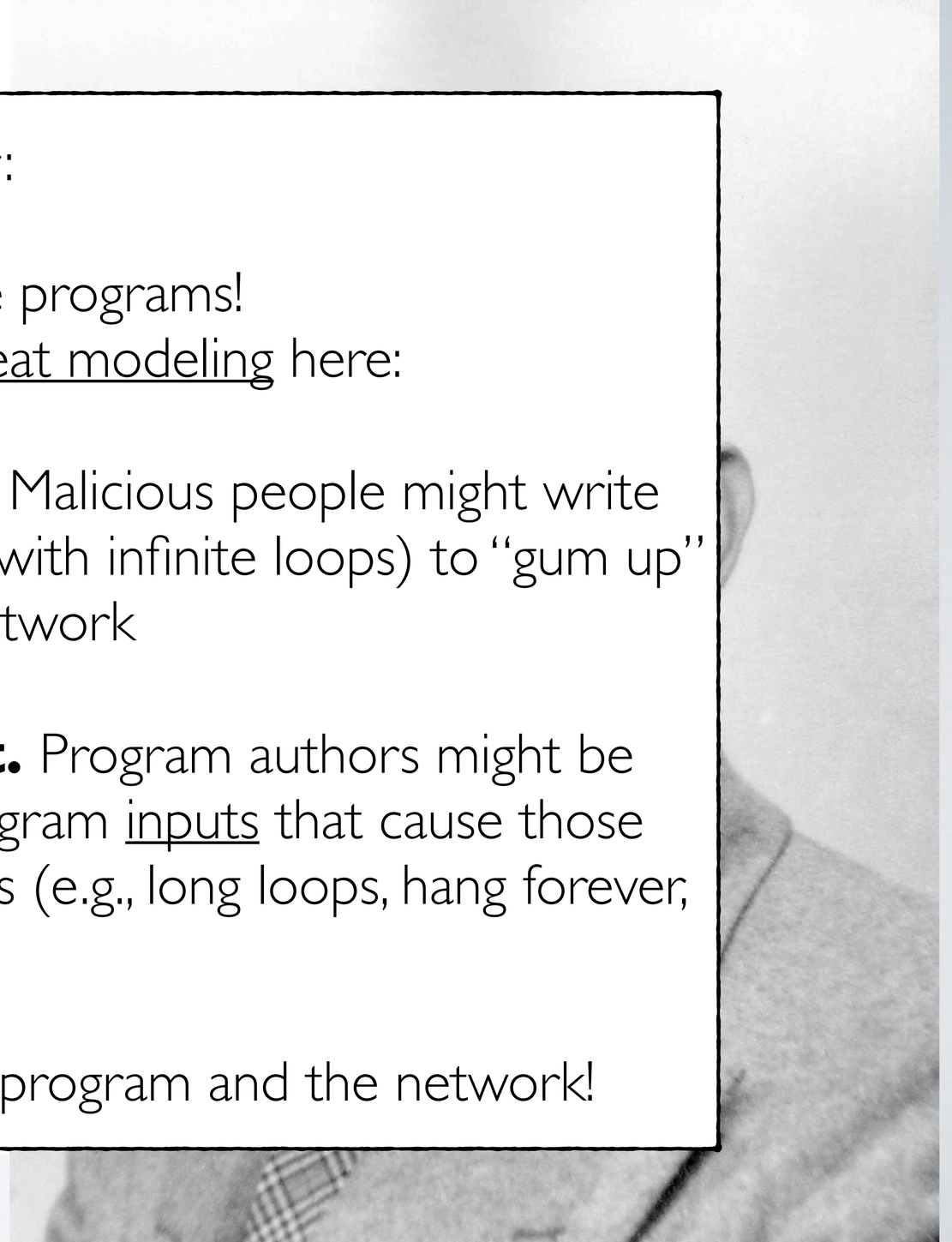
- Where do these programs come from?
  - E.g., how do I submit a new program to the system
- What if running a program takes too long? Uses too much state?
- What language are these programs written in?
- Where does all the program state get stored?
- If this is a Bitcoin-like system, how do many computers stay in consensus?

What if running a program takes too long?

What if a program "fills up" the database with junk?

- We cannot allow each stateUpdate to run for an unlimited number of cycles
  - Otherwise people will gum up the network and cause it to “halt”!
  - We could solve the halting problem…
- Ditto with state variables:
  - Programs can’t be allowed to just fill up the database with junk





Remember:

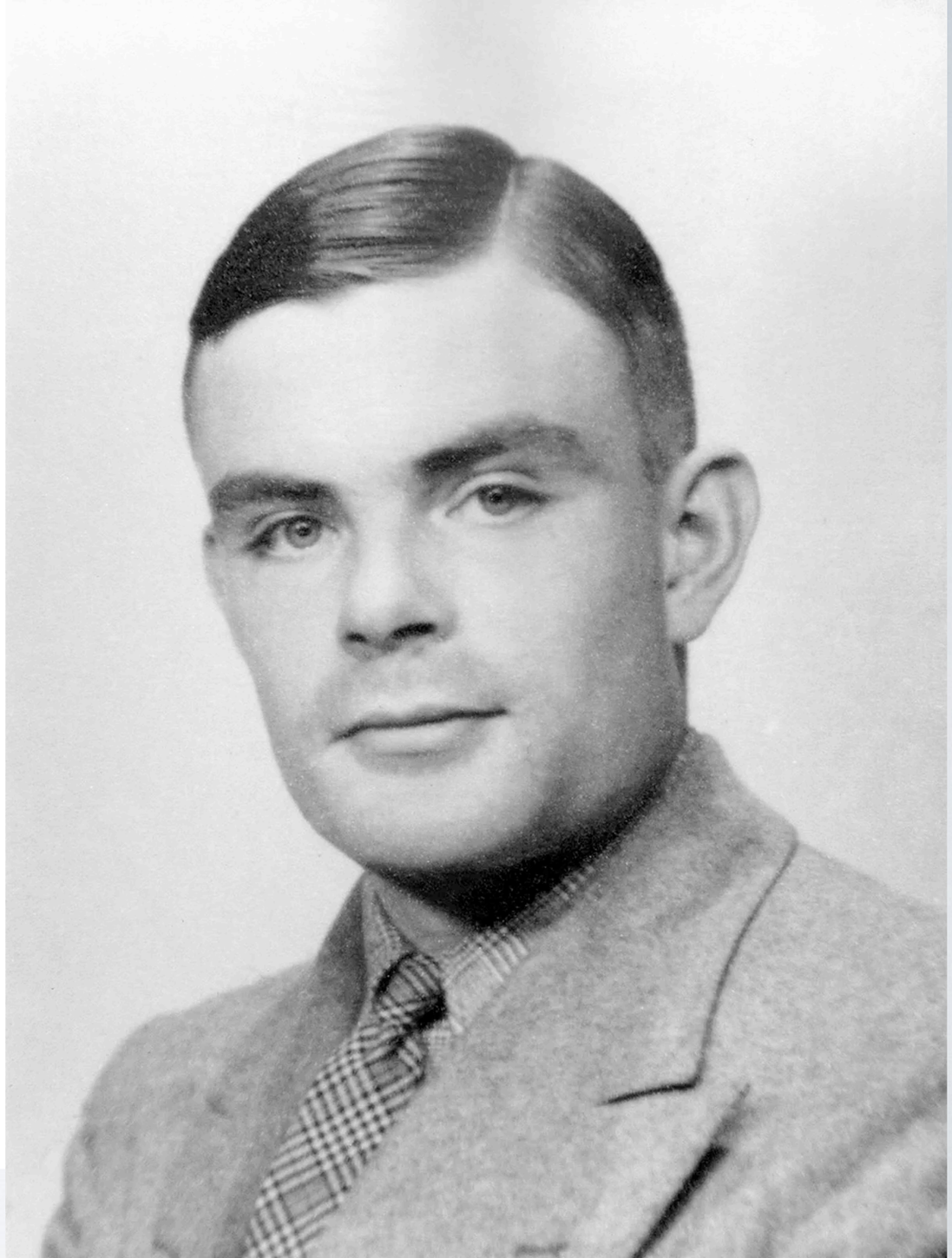
Anyone can call these programs!
We need to think about threat modeling here:

1. **If the network is the target.** Malicious people might write deliberately expensive programs (e.g., with infinite loops) to "gum up" the whole network

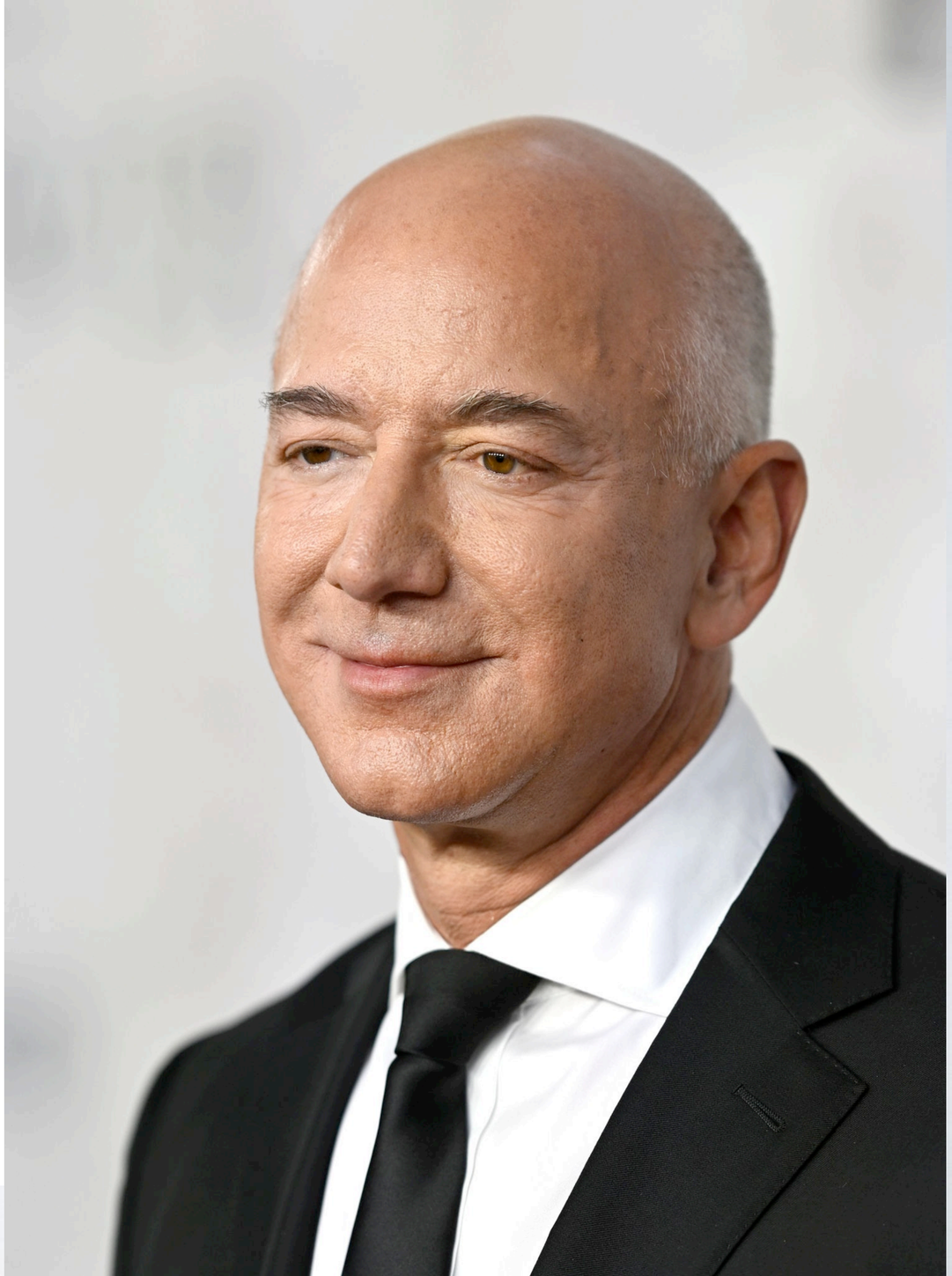
2. **If the program is the target.** Program authors might be honest, but attackers might find program inputs that cause those programs to do expensive calculations (e.g., long loops, hang forever, etc.)

Hence: we want to protect both the program and the network!

- **Answer #1: cap resource consumption**
  - Put a "max cycles allowed" limit on each stateUpdate function
  - When a program runs for too many steps, abort it
    - What do we do with any partial state changes it's made?
  - We can place similar caps on database (storage) consumption



- **Answer #2: <u>charge for resource consumption</u>**
  - If we have cryptocurrency in this network, we can charge people for compute and storage resources!
    - We'll need a universal system for measuring compute cycles (don't want measurement to be wildly varying for different processors)
    - We may also need a "fee market" that adjusts the price of resources when the network gets congested



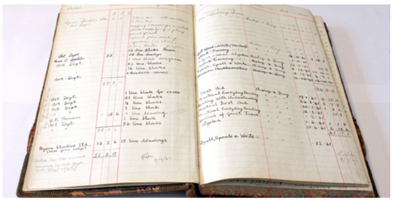
- **Why not both?**
  - Set systemwide caps
  - Also let users specify their own per-transaction resource cap (e.g., "I want this to run for at most X cycles")
  - Let users bid on a fee (calculated as currency-per-cycle)
  - Q: when someone sends a transaction, do they know how many cycles it will need to run?

- We can do both!
  - Set systemwide caps
  - Also let users specify their own cap (transactions run for max X cycles)
  - Let users specify how much they're willing to pay for each cycle
- **Q:** when someone sends a transaction, do they know how many cycles it will need to run?
- **A:** They might not! (Runtime can change based on other callers' previous inputs.)

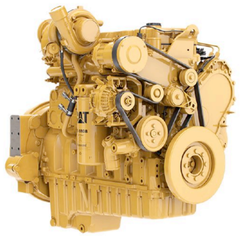
How does this network stay in consensus?

- Network with single node (N=1)
  - One computer, has a database with program code and program state, ordering logic, VM
    - Receives transactions
    - Orders them into blocks (like Bitcoin)
    - Executes each transaction
    - Updates the database

Database
(program code/state, blocks, transactions)



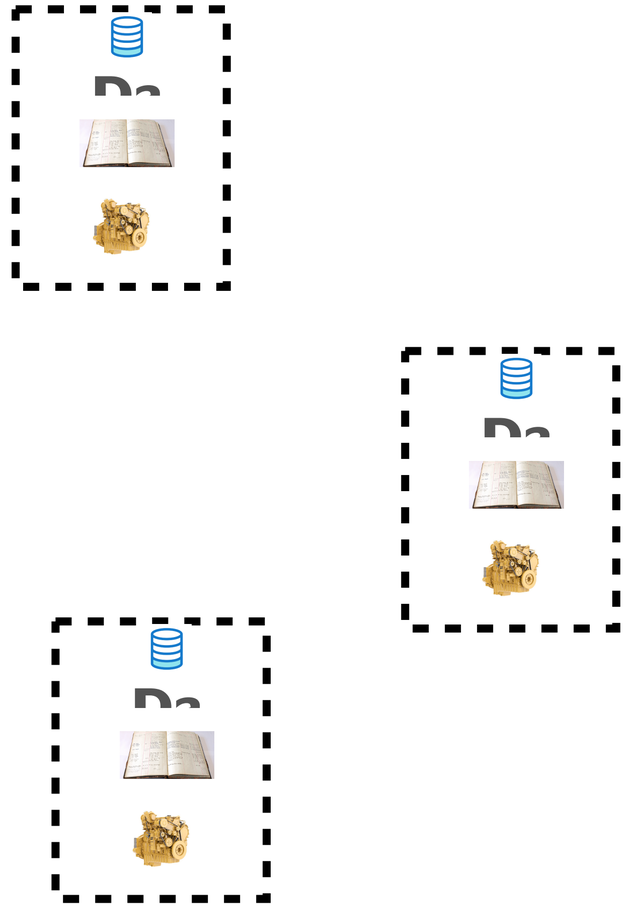
Transaction ordering



Execution engine (VM)

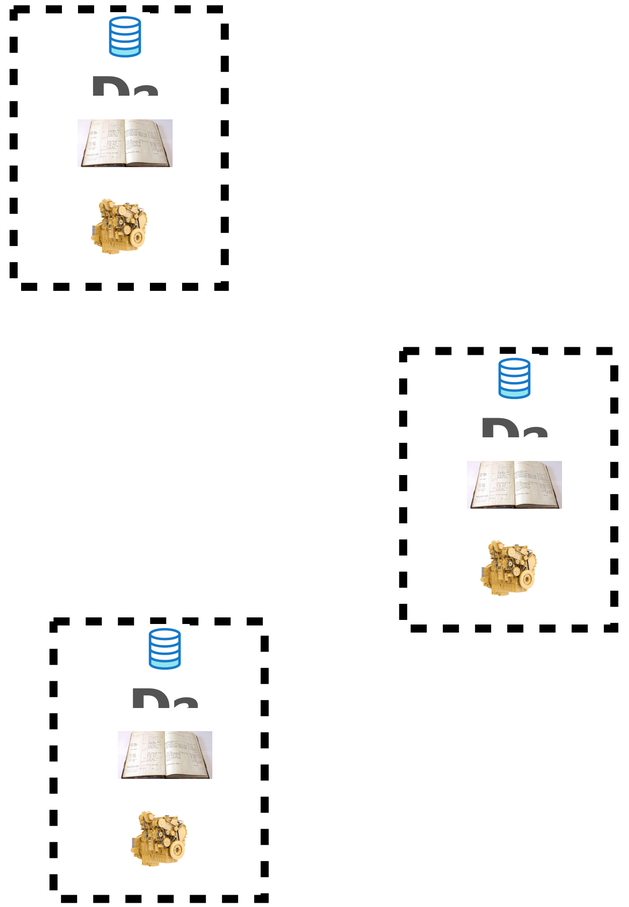
Example node

How do many nodes stay in sync?



How do many nodes stay in sync?

- Idea: just agree on transaction ordering?
  - If all nodes start from the same state (empty database)
  - And all nodes agree on a unique transaction ledger, execute transactions in that order (e.g., Bitcoin approach)
  - And all transaction execution is deterministic and completely repeatable
  - **Then… in theory, every node should always end up agreeing on the same overall state!**



How do many nodes stay in sync?

- Idea: just agree on transaction

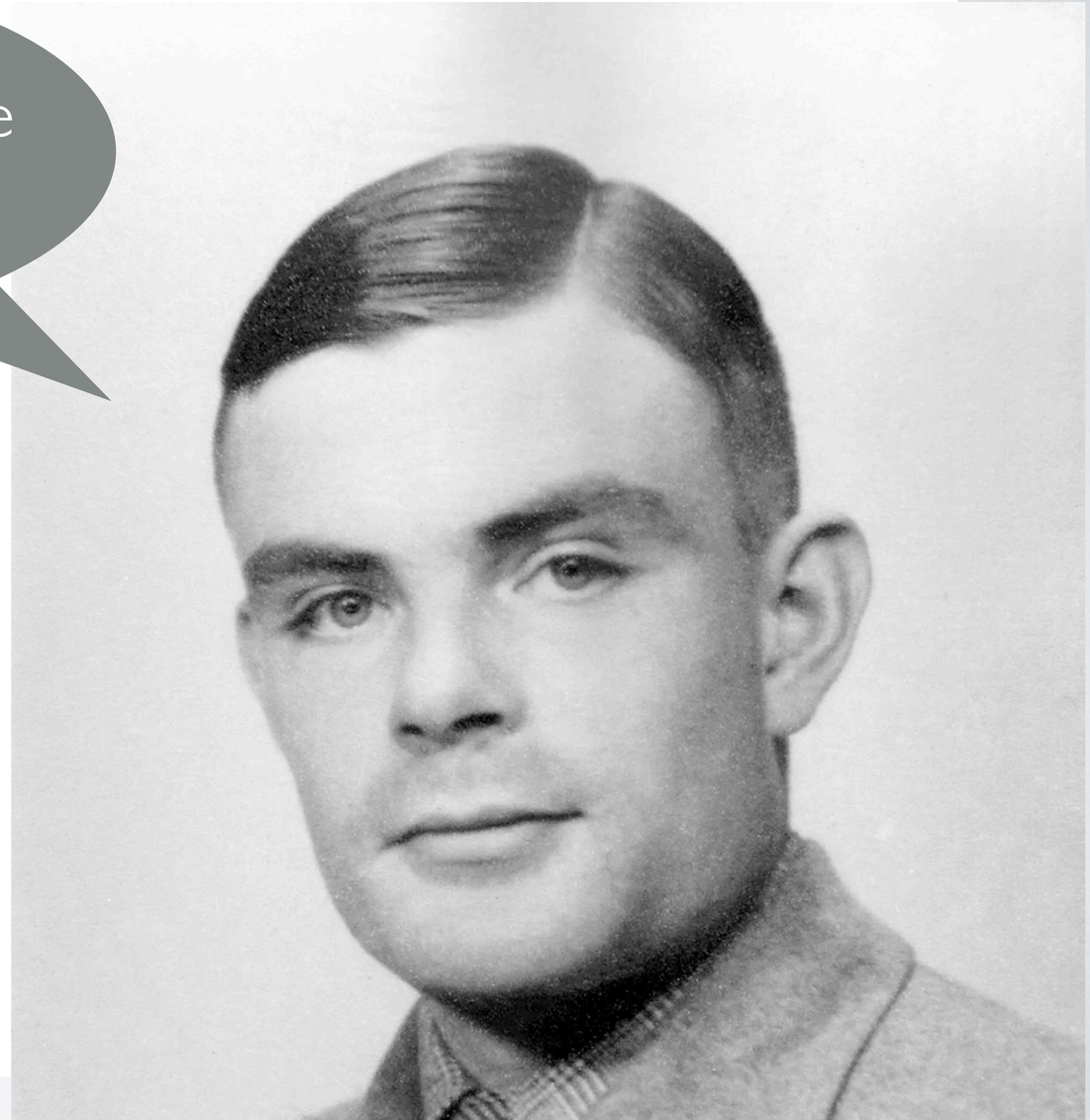
  - If all nodes start fr
    (empty database)

  - And all nodes agree on a unique transaction ledger, execute transactions in that order (e.g., Bitcoin approach)

  - And all transaction execution is <u>deterministic</u> and completely repeatable

  - Then… in theory, every node should always end up agreeing on the database!

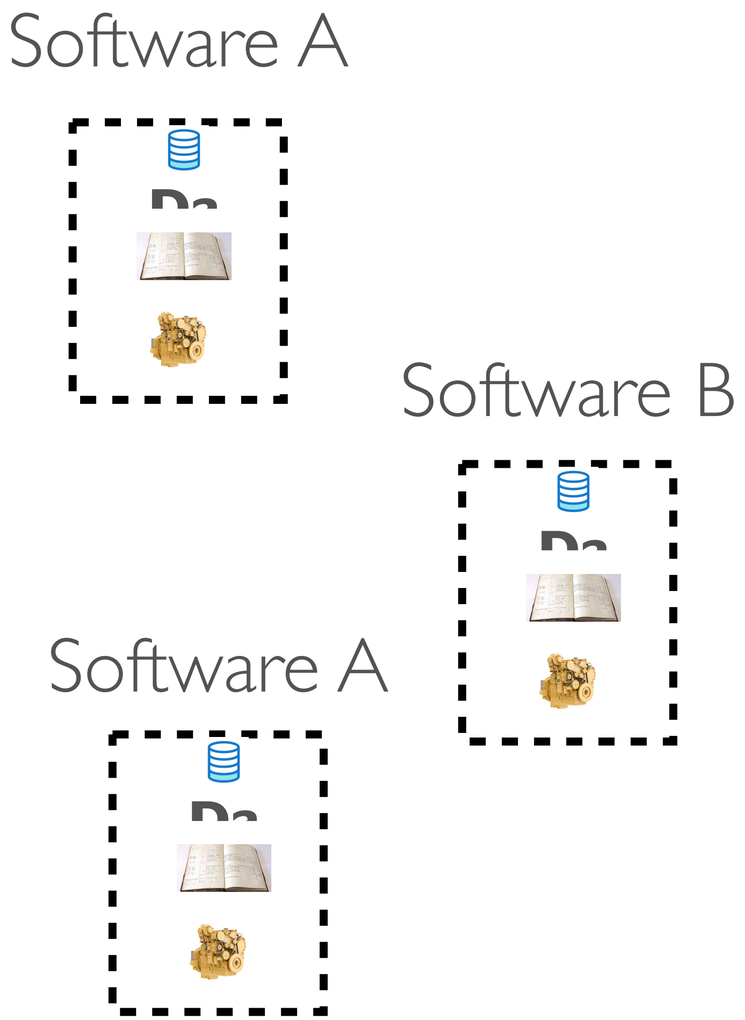
Does this give you the willies?



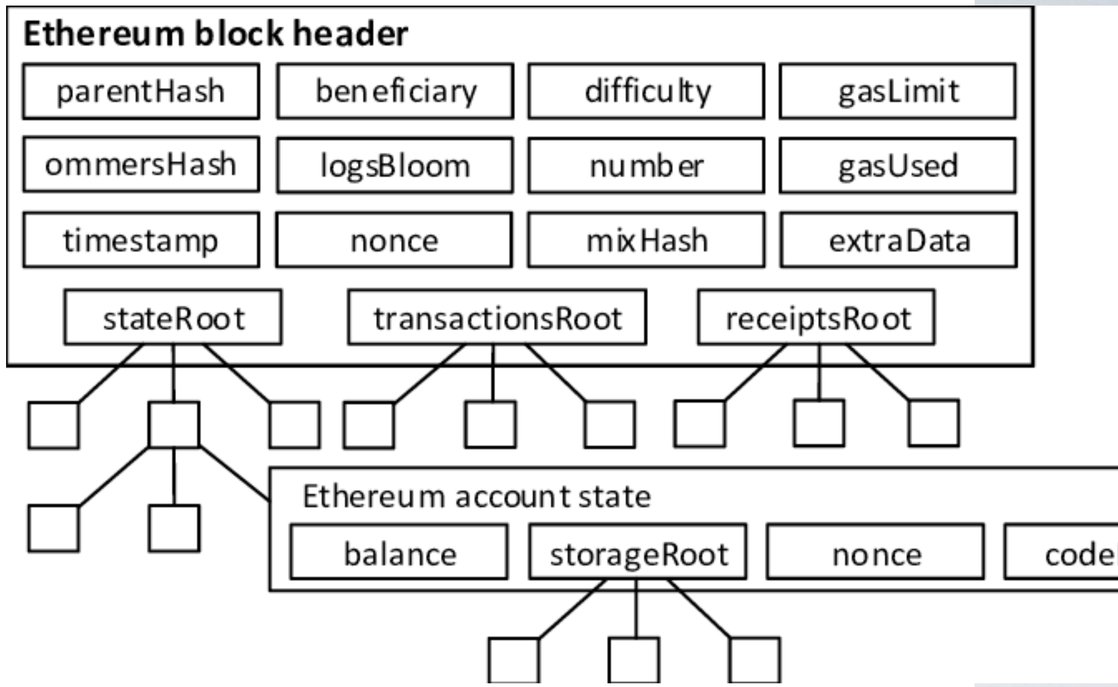
- What if something goes wrong?
  - All nodes can achieve consensus on the contents of the transaction ledger (aka Blockchain)

    Nakamoto consensus: nodes compare the most recent hash of the blockchain!

  - But what if A's execution gets a different result than B's execution engine?
    - Scary! The transactions might be in consensus, but the database contents might not be. How do we detect this?

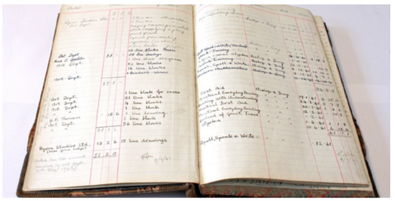


- Answer: hash the database too!
  - We build a hash tree over all the database records
  - Each time we update the database (write things out), we update the hash tree
  - When a transaction runs, we store these hash trees into the block
  - (In theory, these trees can enable light clients to verify execution.)

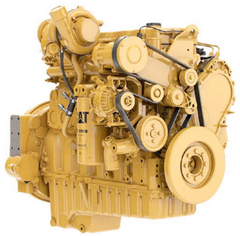


- Now we have an architecture
  - Nakamoto consensus for transaction ordering
  - Transactions for deploying, executing programs (smart contracts)
  - Add "state root" (hash of database) to each block, to ensure database stays in sync
  - Built-in currency to pay for resource usage (Tx execution, database writes)

Database
(program code/state, blocks, transactions)



Transaction ordering

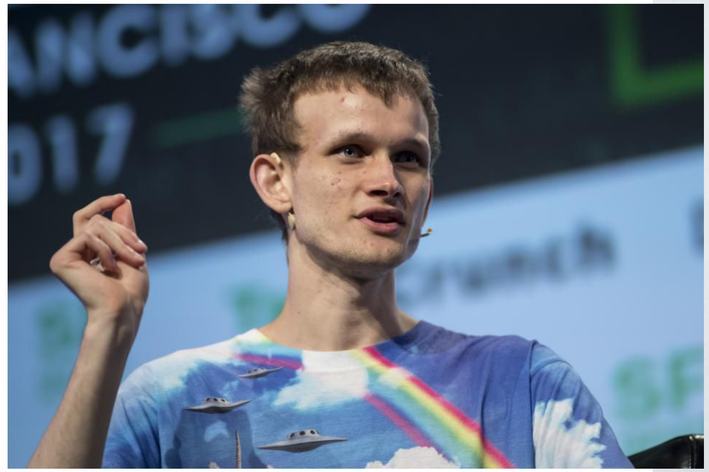


Execution engine (VM)

Example node

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

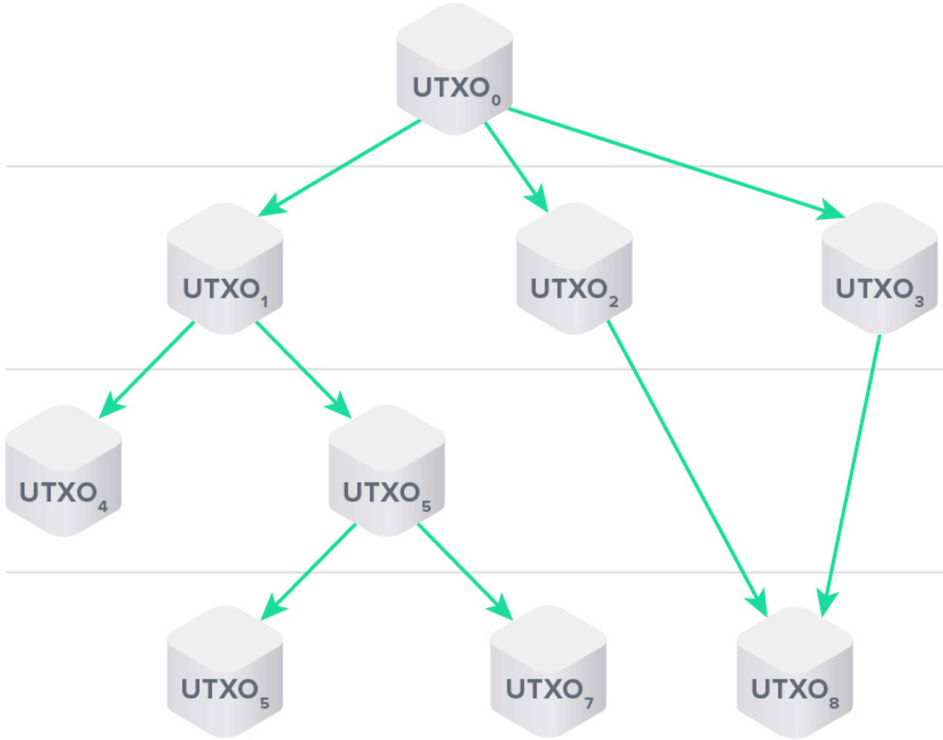


Ethereum vs. Bitcoin (payments)

- Bitcoin uses a UTXO model for its payments
  - Ethereum uses an account-based model
  - Every address has an "account" in the system database
    - Contains things like "account balance" and other state hashes
    - If an address has never transacted on chain, it "exists"
      (but only virtually: there is no actual record in the database.)

RECORDING THE STATE OF THE SYSTEM

UTXO Model



Directed graph of assets (UTXOs) moving between users

Account Model

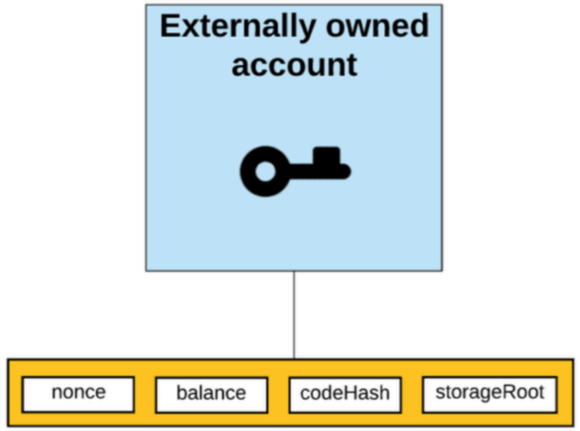
State n	Account A	Balance t_0
State (n+1)	Account A	Balance t_1
	Account B	Balance t_1
State (n+2)	Account A	Balance t_2
	Account B	Balance t_2
State (n+3)	Account A	Balance t_3
	Account B	Balance t_3
	Account C	Balance t_3

Database of network states

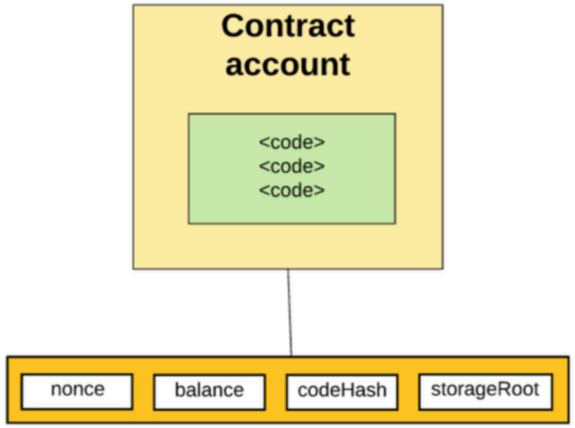
Diagram respectfully excerpted from: https://www.horizen.io/blockchain-academy/technology/expert/utxo-vs-account-model/

Ethereum: Accounts

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

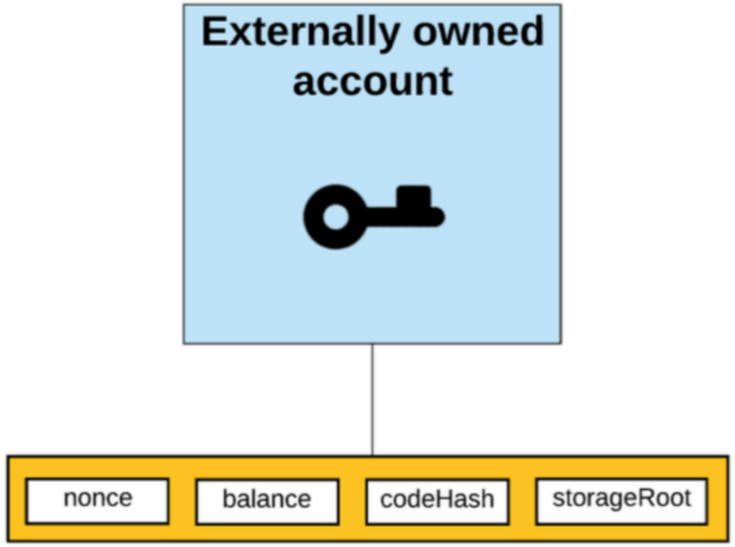


Like Bitcoin, updates require an signature by an external private key



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

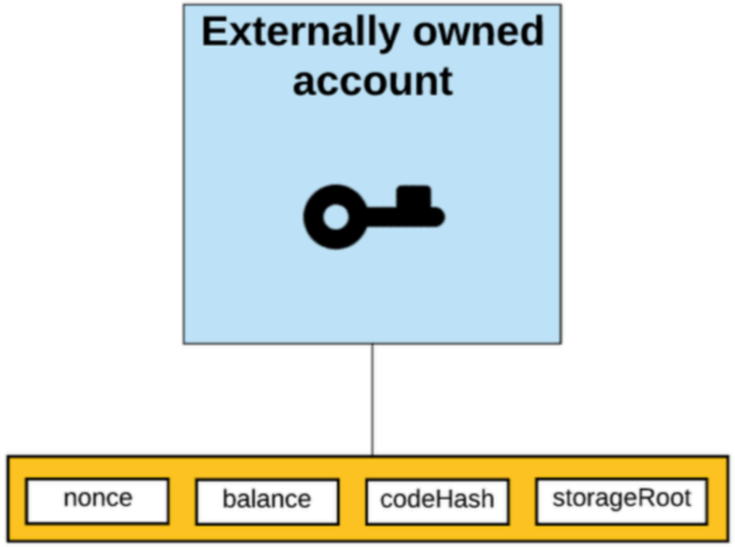
External accounts



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.

External accounts



The nonce field counts number of transactions sent by that account.

(an account that has never transacted has "implicit" nonce = 0)

Each transaction must embed the "next" nonce for that account.

External accounts

Address 0xC1b634853Cb333D3aD8663715b08f41A3Aec47cc

Overview

ETH BALANCE

5.002448759761403114 ETH

ETH VALUE

$8,315.97 (@ $1,662.38/ETH)

More Info

PRIVATE NAME TAGS

+ Add

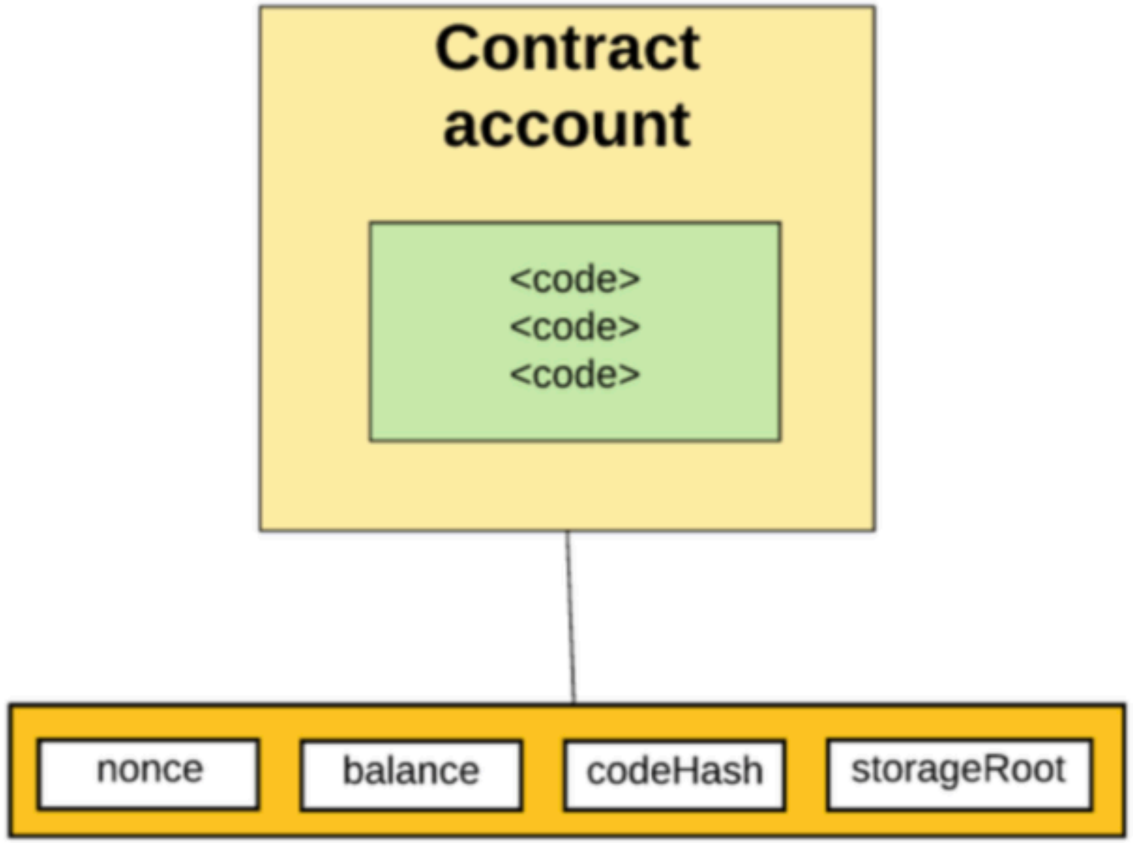
LAST TXN SENT

0x4286e9c6c9fcf... from 10 secs ago

FIRST TXN SENT

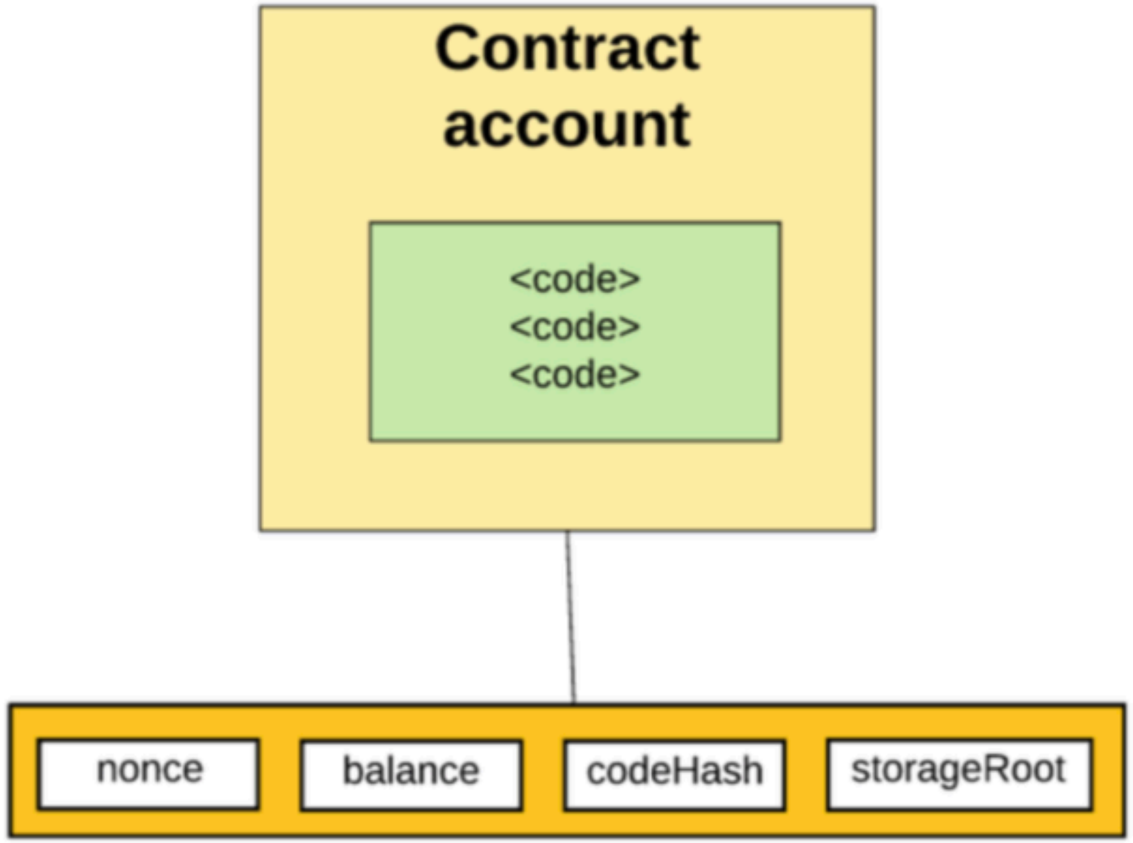
0x0cbbe9eaa29fd... from 47 days 20 hrs ago

Transactions | Internal Transactions | Token Transfers (ERC-20) | Analytics | Comments

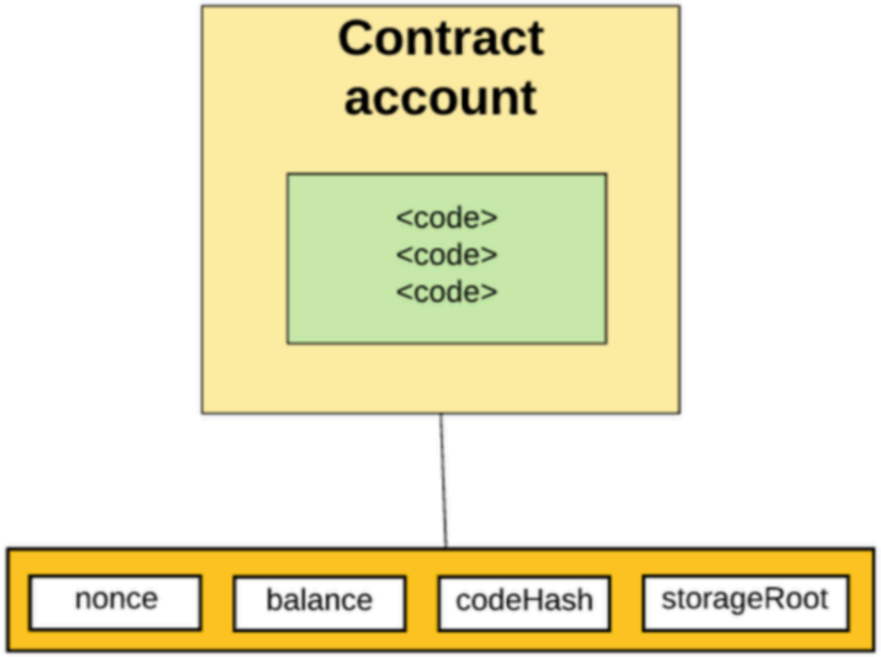


Contract addresses represent programs on the network.

There is no public or secret key associated with these programs.



Q: How do you compute contract addresses, then?



Address =
Hash(Contract code || nonce of the creator's account)

Note: does not include constructor code!

Contract 0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48

Centre: USD Coin | Source Code | # Centre | # Token Contract

Overview

ETH BALANCE

0 ETH

ETH VALUE

$0.00

TOKEN HOLDINGS

>$63,203.13 (>102 Tokens)

More Info

PRIVATE NAME TAGS

+ Add

CONTRACT CREATOR

Circle: Deployer at txn 0xe7e0fe390354509cd08...

TOKEN TRACKER

USD Coin (USDC) (@$1.00)

Creators' rights?

- Congratulations, you just created a contract!
  - What powers do you have over it?

Creators' rights?

- Congratulations, you just created a contract!
  - What powers do you have over it?
  - **A:** None, specifically granted by Ethereum.

    Your code can, however, choose to give you rights.

    Example: setup code can record an "owner" address (you) inside the contract's state. When you call it later the code can check that it's you.

    But we can also write code that does not give its creator any special privileges.

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 defau
- codeHash: Hash of empty string / Hash of the EVM (Ethereum Virtual Machine) code

this account

Ethereum: Accounts

nonce

• bala

• stora

• code

this ac

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.)

. The tree is empty

um Virtual Machin

nonce

• bala

• stora

• code

this ac

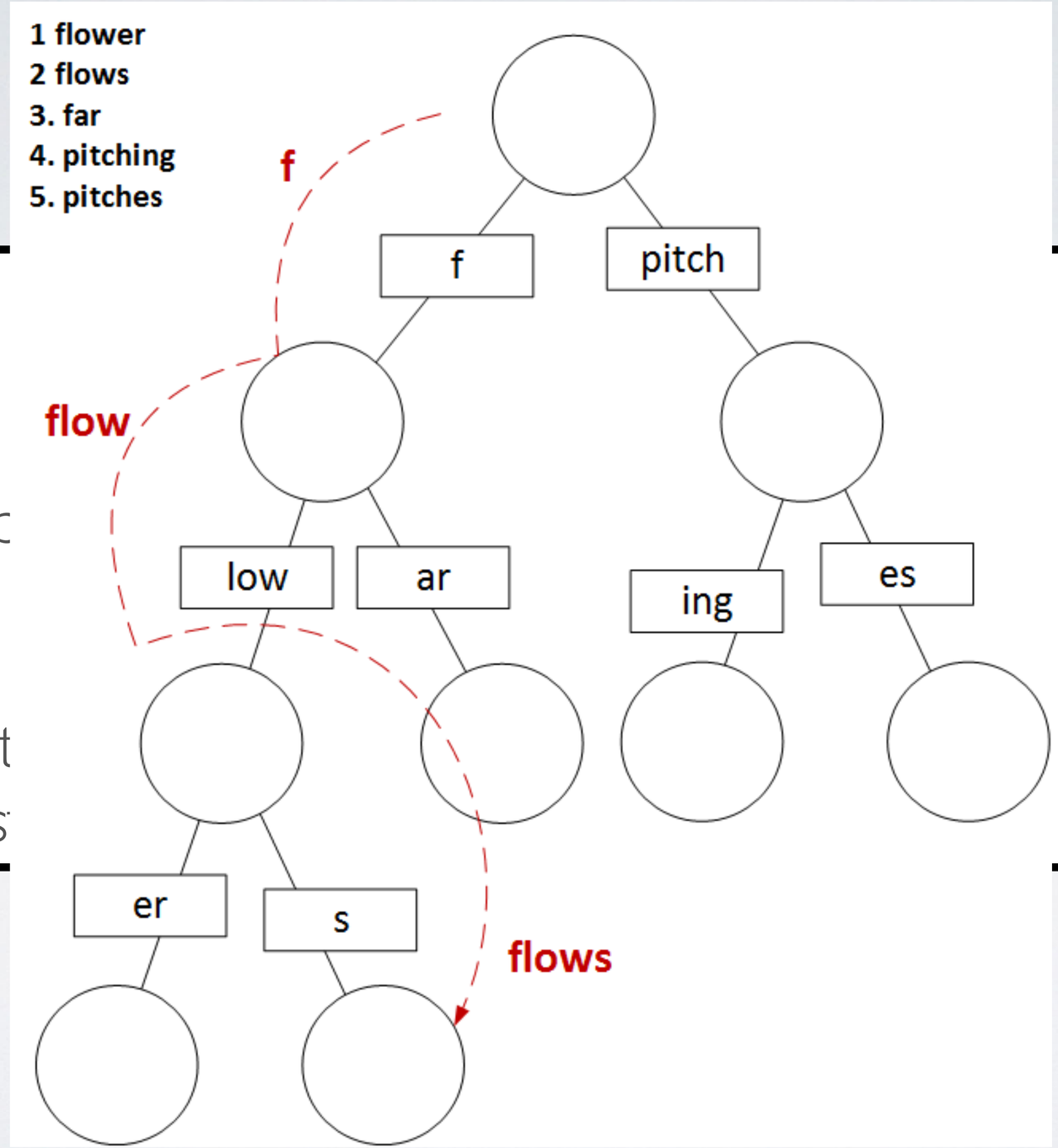
(I.e., t

s

. The tree is empty

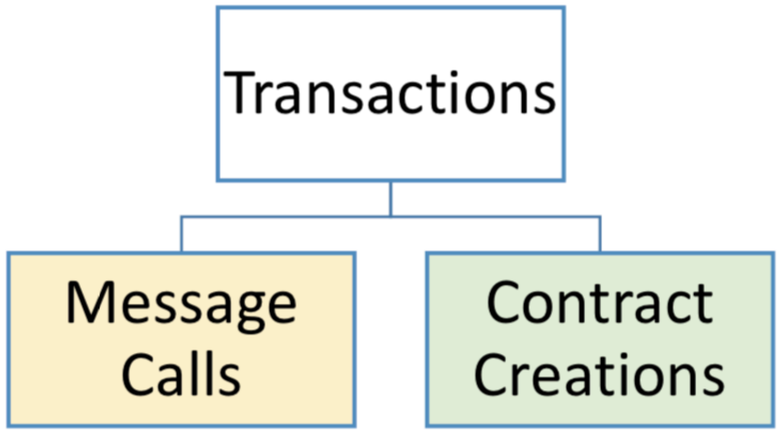
um Virtual Machir

the

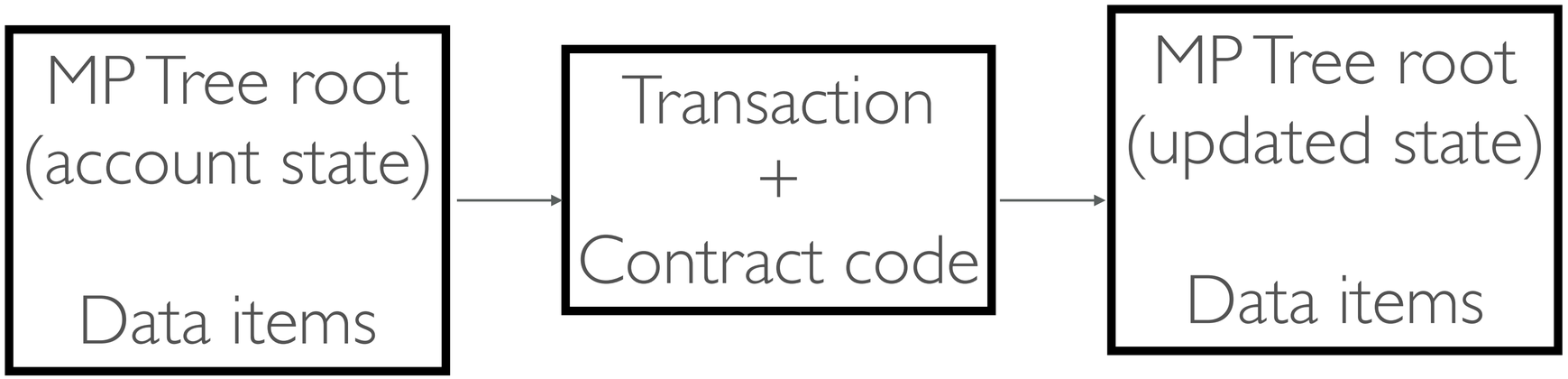


Ethereum Transactions

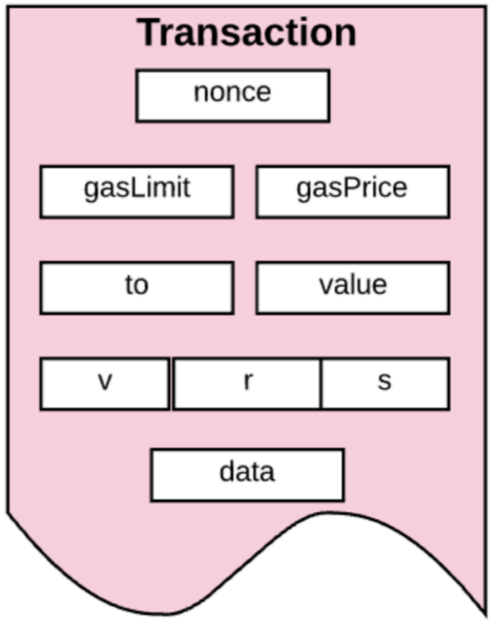
- Two types:
  - Message calls: update state in a given contract, by executing code (or simply transferring money)
  - Contract creations: make a new contract account, with new state



Ethereum: Accounts



Ethereum Transactions

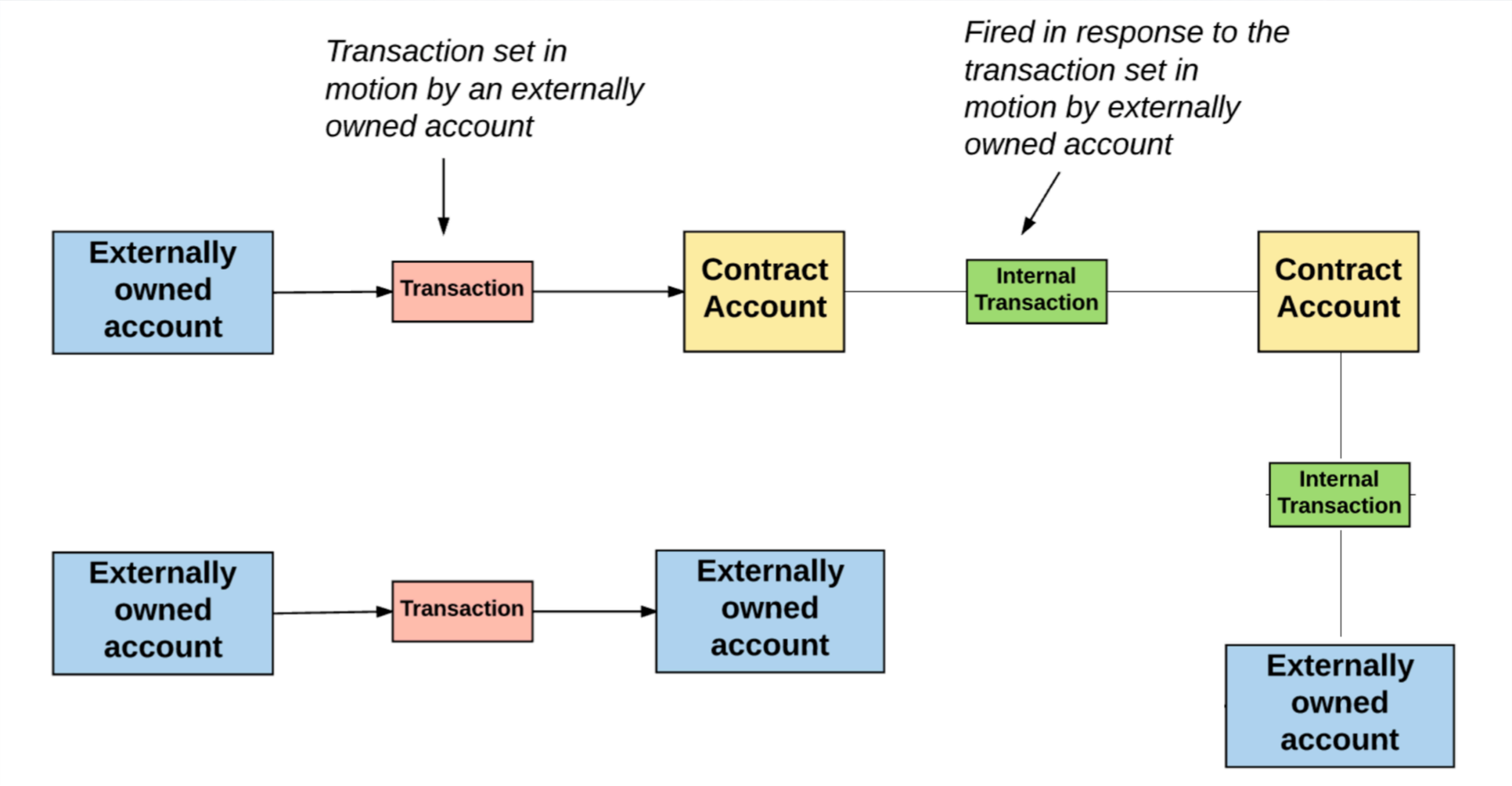


- nonce: A count of the number of transactions sent by the sender.
- gasPrice
- gasLimit
- to: Recepient's address
- value: Amount of Wei Transferred from sender to recipient.
- v,r,s: Used to generate the signature that identifies the sender of the transaction.
- init: EVM code used to initialize the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

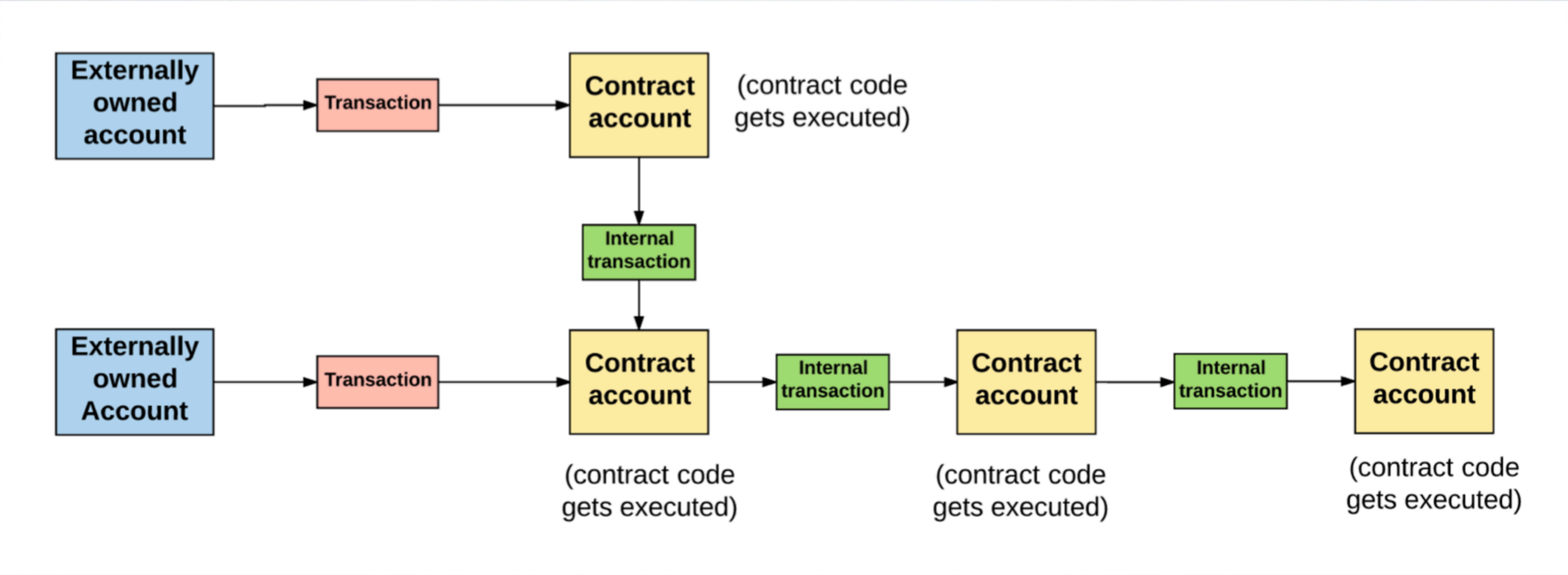
Transaction Types

- 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!

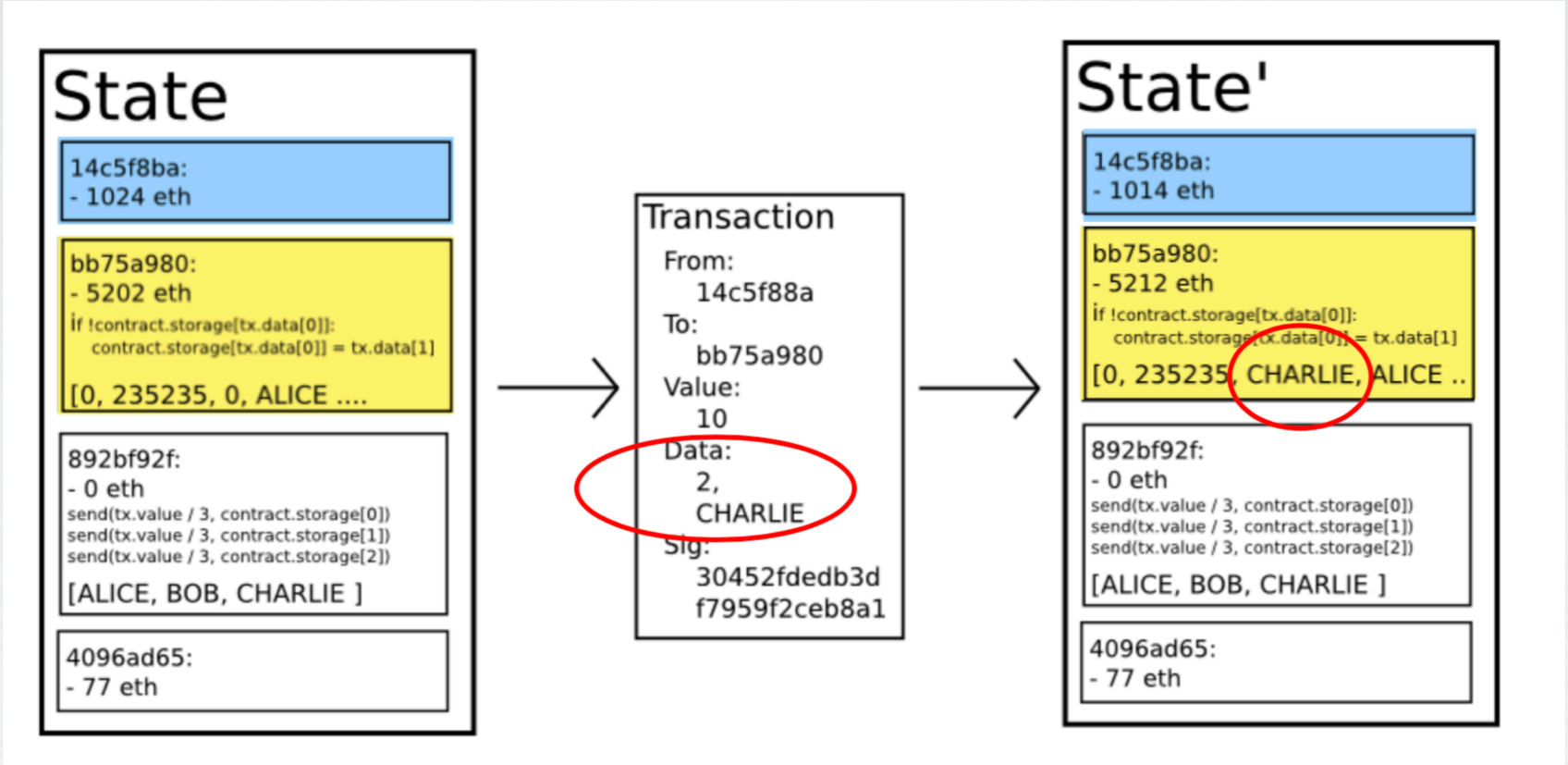
Transaction Types



Transaction Flow



Transaction Execution (Pay)



Contracts

- **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?