Note: HW#2 posted tonight. Due Oct. 25.
Recall: UTXO contains (hash of) ScriptPK
  • simple script: indicates conditions when UTXO can be spent

Limitations:
  • Difficult to maintain state in multi-stage contracts
  • Difficult to enforce global rules on assets

A simple example: rate limiting. My wallet manages 100 UTXOs.
  • Desired policy: can only transfer 2BTC per day out of my wallet
An example: NameCoin

Domain name system on the blockchain: [google.com $\rightarrow$ IP addr]

Need support for three operations:

- **Name.new**(OwnerAddr, DomainName): intent to register
- **Name.update**(DomainName, newVal, newOwner, OwnerSig)
- **Name.lookup**(DomainName)

Note: also need to ensure no front-running on **Name.new()**
A broken implementation

Name.new() and Name.update() create a UTXO with ScriptPK:

```
DUP HASH256 <OwnerAddr> EQVERIFY CHECKSIG VERIFY
<NAMECOIN> <DomainName> <IPaddr> <1>
```

only owner can “spend” this UTXO to update domain data

Contract: (should be enforced by miners)

if domain google.com is registered, no one else can register that domain

Problem: this contract cannot be enforced using Bitcoin script
What to do?

NameCoin: fork of Bitcoin that implements this contract (see also the Handshake, Chia projects)

Can we build a blockchain that natively supports generic contracts like this?

⇒ Ethereum
A world of Ethereum Decentralized apps (DAPPs)

- **New coins**: ERC-20 standard interface
- **DeFi**: exchanges, lending, stablecoins, derivatives, etc.
- **Insurance**
- **DAOs**: decentralized organizations
- **NFTs**: Managing asset ownership (ERC-721 interface)

stateofthedapps.com, dapp.review
Bitcoin as a state transition system

world state

UTXO\_1

UTXO\_2

⋮

updated world state

UTXO\_1

UTXO\_2

⋮

input

Tx: UTXO\_2 \rightarrow UTXO\_3

Bitcoin rules:

\[ F_{\text{bitcoin}} : S \times I \rightarrow S \]

S: set of all possible world states, \( s_0 \in S \) genesis state

I: set of all possible inputs
Ethereum as a state transition system

Much richer state transition functions

⇒ one transition executes an entire program
Running a program on a blockchain (DAPP)

Consensus layer (beacon chain)

Compute layer (execution chain): The EVM

Create a DAPP

State $s_0$, $s_1$, $s_2$...
The Ethereum system

Proof-of-Stake consensus

<table>
<thead>
<tr>
<th>Block</th>
<th>Age</th>
<th>Txn</th>
<th>Fee Recipient</th>
</tr>
</thead>
<tbody>
<tr>
<td>15764027</td>
<td>4 secs ago</td>
<td>91</td>
<td>Fee Recipient: 0x467...263</td>
</tr>
<tr>
<td>15764026</td>
<td>16 secs ago</td>
<td>26</td>
<td>0xedc7ec654e305a38fff...</td>
</tr>
<tr>
<td>15764025</td>
<td>28 secs ago</td>
<td>165</td>
<td>bloXroute: Max Profit Bui...</td>
</tr>
<tr>
<td>15764024</td>
<td>40 secs ago</td>
<td>188</td>
<td>Lido: Execution Layer Re...</td>
</tr>
<tr>
<td>15764023</td>
<td>52 secs ago</td>
<td>18</td>
<td>Fee Recipient: 0xeBe...Acf</td>
</tr>
<tr>
<td>15764022</td>
<td>1 min ago</td>
<td>282</td>
<td>0xd4e96ef8ee8678dbff...</td>
</tr>
<tr>
<td>15764021</td>
<td>1 min ago</td>
<td>295</td>
<td>0xbb3afde35eb9f5feb53...</td>
</tr>
<tr>
<td>15764020</td>
<td>1 min ago</td>
<td>71</td>
<td>Fee Recipient: 0x6d2...766</td>
</tr>
</tbody>
</table>

One block every 12 seconds. About 150 Tx per block.

Block proposer receives Tx fees for block (along with other rewards)
A bit about the beacon chain (Eth2 consensus layer)

To become a validator: stake (lock up) 32 ETH ... or use Lido.

Validators:
- sign blocks to express correctness (finalized once enough sigs)
- occasionally act as block proposer (chosen at random)
- correct behavior ⇒ issued **new ETH** every epoch (32 blocks)
- incorrect behavior ⇒ slashed

(lots of details)

[Graph # Validators (444K)]

[Graph Staked ETH (15M)]
The economics of staking

Validator locks up 32 ETH. Oct 2022: 15M ETH staked (total)

Expected **annual** validator income:

- Issuance: 1.4 ETH
- Tx fees: 0.9 ETH
- MEV: 0.3 ETH
- Total: 2.6 ETH (8.3% return on 32 ETH staked)

In practice: staking provider (e.g., Lido) takes a cut of the returns
The Ethereum system

The Ethereum system consists of:

- **Consensus layer (beacon chain)**
- **Compute layer (execution chain)**

The **consensus layer** (beacon chain) produces blocks in epochs of 32 blocks. Each block contains transactions that need to be computed. These transactions are processed by the **compute layer** (execution chain) using the function `notify_new_payload(payload)` from the Engine API. This function sends transactions to the compute layer for processing.

After the transactions are processed, the compute layer updates the world state.
The Ethereum Compute Layer:
The EVM
World state: set of accounts identified by 32-byte address.

Two types of accounts:

1. **owned accounts**: controlled by ECDSA signing key pair (pk,sk).
   - sk: signing key known only to account owner

2. **contracts**: controlled by code.
   - code set at account creation time, does not change
# Data associated with an account

<table>
<thead>
<tr>
<th>Account data</th>
<th>Owned</th>
<th>Contracts</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>address</strong> (computed):</td>
<td><strong>H(pk)</strong></td>
<td><strong>H(CreatorAddr, CreatorNonce)</strong></td>
</tr>
<tr>
<td><strong>code:</strong></td>
<td>⊥</td>
<td><strong>CodeHash</strong></td>
</tr>
<tr>
<td><strong>storage root</strong> (state):</td>
<td>⊥</td>
<td><strong>StorageRoot</strong></td>
</tr>
<tr>
<td><strong>balance</strong> (in Wei):</td>
<td>balance</td>
<td>balance</td>
</tr>
<tr>
<td><strong>nonce:</strong></td>
<td>nonce</td>
<td>nonce</td>
</tr>
</tbody>
</table>

(#Tx sent) + (#accounts created): anti-replay mechanism
Account state: persistent storage

Every contract has an associated **storage array** $S[]$:

- $S[0], S[1], \ldots, S[2^{256}-1]$: each cell holds 32 bytes, init to 0.

Account storage root: **Merkle Patricia Tree hash** of $S[]$

- Cannot compute full Merkle tree hash: $2^{256}$ leaves

<table>
<thead>
<tr>
<th>$S[000]$ = a</th>
<th>$S[010]$ = b</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S[011]$ = c</td>
<td>$S[110]$ = d</td>
</tr>
</tbody>
</table>

![Diagram of Merkle Patricia Tree](diagram.png)

- Time to compute root hash: $\leq 2 \times |S|$
  - $|S|$ = # non-zero cells
Transactions: signed data by initiator

- **To**: 32-byte address of target (0 → create new account)
- **From**, [Signature]: initiator address and signature on Tx (if owned)
- **Value**: # Wei being sent with Tx
- **Tx fees (EIP 1559)**: `gasLimit`, `maxFee`, `maxPriorityFee` (later)
- if **To = 0**: create new contract `code = (init, body)`
- if **To ≠ 0**: **data** (what function to call & arguments)
- **nonce**: must match current nonce of sender (prevents Tx replay)
Transaction types:

- owned → owned: transfer ETH between users
- owned → contract: call contract with ETH & data
<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>msg.value</th>
<th>Tx fee (ETH)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xa4ec1125ce9428ae5...</td>
<td>0x2cebe81fe0dcd220e...</td>
<td>0 Ether</td>
<td>0.00404405</td>
</tr>
<tr>
<td>0xba272f30459a119b2...</td>
<td>0xUniswap V2: Router 2</td>
<td>0.14 Ether</td>
<td>0.00644563</td>
</tr>
<tr>
<td>0x4299d864bbda0fe32...</td>
<td>0xUniswap V2: Router 2</td>
<td>89.33910411882671 Ether</td>
<td>0.00716578</td>
</tr>
<tr>
<td>0x4d1317a2a98cfea41...</td>
<td>0xc59f33af5f4a7c8647...</td>
<td>14.501 Ether</td>
<td>0.001239</td>
</tr>
<tr>
<td>0x29ecaa773f052d14e...</td>
<td>0xcryptoKitties: Core</td>
<td>0 Ether</td>
<td>0.00775543</td>
</tr>
<tr>
<td>0x63bb46461696416fa...</td>
<td>0xUniswap V2: Router 2</td>
<td>0.203036474328481 Ether</td>
<td>0.00766728</td>
</tr>
<tr>
<td>0xde70238ae7a35abd...</td>
<td>0xbalancer: ETH/DOUGH...</td>
<td>0 Ether</td>
<td>0.00261582</td>
</tr>
<tr>
<td>0x69aca10fe1394d535f...</td>
<td>0x83d03aa7fc09b8be...</td>
<td>0 Ether</td>
<td>0.00259936</td>
</tr>
<tr>
<td>0xe2f5d180626d29e75...</td>
<td>0xUniswap V2: Router 2</td>
<td>0 Ether</td>
<td>0.00665809</td>
</tr>
</tbody>
</table>
Messages: virtual Tx initiated by a contract

Same as Tx, but no signature  (contract has no signing key)

contract $\rightarrow$ owned: contract sends funds to user

contract $\rightarrow$ contract: one program calls another (and sends funds)

One Tx from user: can lead to many Tx processed.  Composability!

Tx from owned addr $\rightarrow$ contract $\rightarrow$ another contract

another contract $\rightarrow$ different owned
Example Tx

**State**
- 14c5f8ba: 1024 eth
- bb75a980: 5202 eth
  - If !contract.storage[tx.data[0]]:
    - contract.storage[tx.data[0]] = tx.data[1]
- 892bf92f: 0 eth
  - send(tx.value / 3, contract.storage[0])
  - send(tx.value / 3, contract.storage[1])
  - send(tx.value / 3, contract.storage[2])
- 4096ad65: 77 eth

**Transaction**
- From: 14c5f8ba
- To: bb75a980
- Value: 10 eth
- Data: 2, CHARLIE
- Sig: 30452fdeeb3d f7959f2ceb8a1

**State'**
- 14c5f8ba: 1014 eth
- bb75a980: 5212 eth
- 892bf92f: 0 eth
- 4096ad65: 77 eth

world state (four accounts)  updated world state
An Ethereum Block

Validators collect Txs from users ⇒ proposer creates a block of nTx

• To produce a block do:
  • for i=1,...,n: execute state change of Tx_i sequentially
    (can change state of >n accounts)
  • record updated world state in block

Other validators re-execute all Tx to verify block ⇒ sign block if valid ⇒ enough sigs, epoch is finalized.
(1) consensus data: proposer ID, parent hash, votes, etc.

(2) address of gas beneficiary: where Tx fees will go

(3) **world state root**: updated world state

  Merkle Patricia Tree hash of all accounts in the system

(4) **Tx root**: Merkle hash of all Tx processed in block

(5) **Tx receipt root**: Merkle hash of log messages generated in block

(5) Gas used: used to adjust gas price (target 15M gas per block)
The Ethereum blockchain: abstractly

prev hash

accts.
updated world state

Tx
log messages

prev hash

accts.
updated world state

Tx
log messages
Amount of memory to run a node

ETH total blockchain size (archival): 12 TB (Oct. 2022)
An example contract: NameCoin

```solidity
contract nameCoin { // Solidity code (next lecture)

    struct nameEntry {
        address owner; // address of domain owner
        bytes32 value; // IP address
    }

    // array of all registered domains
    mapping (bytes32 => nameEntry) data;
}
```
An example contract: NameCoin

function nameNew(bytes32 name) {

    // registration costs is 100 Wei

    if (data[name] == 0 && msg.value >= 100) {
        data[name].owner = msg.sender // record domain owner
        emit Register(msg.sender, name) // log event
    }
}

Code ensures that no one can take over a registered name

Serious bug in this code! Front running. Solved using commitments.
An example contract:  NameCoin

function **nameUpdate** (bytes32 name, bytes32 newValue, address newOwner) {

    // check if message is from domain owner,
    // and update cost of 10 Wei is paid

    if (data[name].owner == msg.sender && msg.value >= 10) {

        data[name].value = newValue;  // record new value
        data[name].owner = newOwner;  // record new owner

    }
An example contract: NameCoin

function `nameLookup` (bytes32 name) {
    return data[name];
}

} // end of contract

Used by other contracts
Humans do not need this (use etherscan.io)
EVM mechanics: execution environment

Write code in Solidity (or another front-end language)

⇒ compile to EVM bytecode
  (some projects use WASM or BPF bytecode)

⇒ validators use the EVM to execute contract bytecode in response to a Tx
The EVM

Stack machine (like Bitcoin) but with JUMP
• max stack depth = 1024
• program aborts if stack size exceeded; block proposer keeps gas
• contract can create or call another contract

In addition: two types of zero initialized memory
• Persistent storage (on blockchain): SLOAD, SSTORE (expensive)
• Volatile memory (for single Tx): MLOAD, MSTORE (cheap)
• LOG0(data): write data to log

see https://www.evm.codes
Every instruction costs gas, examples:

**SSTORE**  \( \text{addr} \) (32 bytes),  \( \text{value} \) (32 bytes)

- zero \( \rightarrow \) non-zero:  20,000 gas
- non-zero \( \rightarrow \) non-zero:  5,000 gas  (for a cold slot)
- non-zero \( \rightarrow \) zero:  15,000 gas refund  (example)

Refund is given for reducing size of blockchain state

**CREATE** :  \( 32,000 + 200 \times \text{(code size)} \)  gas;

**CALL**  \( \text{gas, addr, value, args} \)

**SELFDESTRUCT**  \( \text{addr} \) :  kill current contract  (5000 gas)
Gas calculation

Why charge gas?

- Tx fees (gas) prevents submitting Tx that runs for many steps.
- During high load: block proposer chooses Tx from mempool that maximize its income.

Old EVM: (prior to EIP1559, live on 8/2021)

- Every Tx contains a gasPrice ``bid” (gas → Wei conversion price)
- Producer chooses Tx with highest gasPrice ($\max \sum (\text{gasPrice} \times \text{gasLimit})$)

$\Rightarrow$ not an efficient auction mechanism (first price auction)
Gas prices spike during congestion

GasPrice in Gwei:

\[ 86 \text{ Gwei} = 86 \times 10^{-9} \text{ ETH} \]

Average Tx fee in USD
Gas calculation: EIP1559

Every block has a “baseFee”:

   the **minimum** gasPrice for all Tx in the block

baseFee is computed from **total gas** in earlier blocks:

- earlier blocks at gas limit (30M gas) \(\Rightarrow\) base fee goes up 12.5%
- earlier blocks empty \(\Rightarrow\) base fee decreases by 12.5%

If earlier blocks at “target size” (15M gas) \(\Rightarrow\) base fee does not change
EIP1559 Tx specifies three parameters:

- **gasLimit**: max total gas allowed for Tx
- **maxFee**: maximum allowed gas price (max gas → Wei conversion)
- **maxPriorityFee**: additional “tip” to be paid to block proposer

Computed **gasPrice** bid:

\[
\text{gasPrice} \leftarrow \min(\text{maxFee}, \text{baseFee} + \text{maxPriorityFee})
\]

Max Tx fee: **gasLimit** × **gasPrice**
(1) if \texttt{gasPrice} < \texttt{baseFee}: abort
(2) If \texttt{gasLimit} \times \texttt{gasPrice} < \texttt{msg.sender.balance}: abort
(3) deduct \texttt{gasLimit} \times \texttt{gasPrice} from \texttt{msg.sender.balance}

(4) set \texttt{Gas} \leftarrow \texttt{gasLimit}
(5) execute Tx: deduct gas from \texttt{Gas} for each instruction
   if at end (\texttt{Gas} < 0): abort, Tx is invalid (proposer keeps \texttt{gasLimit} \times \texttt{gasPrice})
(6) Refund \texttt{Gas} \times \texttt{gasPrice} to \texttt{msg.sender.balance}

(7) \texttt{gasUsed} \leftarrow \texttt{gasLimit} \times \texttt{Gas}
   (7a) BURN \texttt{gasUsed} \times \texttt{baseFee}
   (7b) Send \texttt{gasUsed} \times (\texttt{gasPrice} \times \texttt{Gas}) \times \texttt{baseFee} to block producer
Example baseFee and effect of burn

<table>
<thead>
<tr>
<th>block #</th>
<th>gasUsed</th>
<th>baseFee (Gwei)</th>
<th>ETH burned</th>
</tr>
</thead>
<tbody>
<tr>
<td>15763570</td>
<td>21,486,058</td>
<td>16.92</td>
<td>0.363</td>
</tr>
<tr>
<td>15763569</td>
<td><strong>14,609,185</strong> (&lt;15M)</td>
<td>16.97</td>
<td>0.248</td>
</tr>
<tr>
<td>15763568</td>
<td>25,239,720</td>
<td>15.64</td>
<td>0.394</td>
</tr>
<tr>
<td>15763567</td>
<td>29,976,215</td>
<td>13.90</td>
<td>0.416</td>
</tr>
<tr>
<td>15763566</td>
<td><strong>14,926,172</strong> (&lt;15M)</td>
<td>13.91</td>
<td>0.207</td>
</tr>
<tr>
<td>15763565</td>
<td><strong>1,985,580</strong> (&lt;15M)</td>
<td>15.60</td>
<td>0.031</td>
</tr>
</tbody>
</table>

≈ gasUsed × baseFee

baseFee < 16Gwei  ⇒  new issuance > burn  ⇒  ETH inflates
baseFee > 16Gwei  ⇒  new issuance < burn  ⇒  ETH deflates
EIP1559 goals (informal):

• users incentivized to bid their true utility for posting Tx,
• block proposer incentivized to not create fake Tx, and
• disincentivized off chain agreements.

Suppose no burn (i.e., baseFee given to block producer):

⇒ in periods of low Tx volume proposer would try to increase volume by offering to refund the baseFee *off chain* to users.
Note: transactions are becoming more complex

Gas usage is increasing $\Rightarrow$ each Tx takes more instructions to execute
Next lecture: writing Solidity contracts