Ethereum: mechanics

Dan Boneh

Note: HW#2 posted tonight. Due Oct. 24.
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: DNS

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
<DNS> <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: a fork of Bitcoin that implements this contract
(see also the Ethereum Name Service -- ENS)

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)
Bitcoin as a state transition system

world state

UTXO_1
UTXO_2
⋮

input

Tx: UTXO_2 \rightarrow UTXO_3

updated world state

UTXO_1
UTXO_3
⋮

Bitcoin rules:

F_{bitcoin} : S \times I \rightarrow S

S: set of all possible world states, \quad s_0 \in S \text{ 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**

Program code

Create a DAPP

State transition:
- $state_0$
- $Tx_1$
- $state_1$
- $Tx_2$
- $state_2$
- ...
The Ethereum system

Proof-of-Stake consensus

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

<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</td>
<td>91</td>
<td>Fee Recipient: 0x467...263</td>
</tr>
<tr>
<td>15764026</td>
<td>16 secs</td>
<td>26</td>
<td>0xedc7ec654e305a38fff...</td>
</tr>
<tr>
<td>15764025</td>
<td>28 secs</td>
<td>165</td>
<td>bloXroute: Max Profit Bui...</td>
</tr>
<tr>
<td>15764024</td>
<td>40 secs</td>
<td>188</td>
<td>Lido: Execution Layer Re...</td>
</tr>
<tr>
<td>15764023</td>
<td>52 secs</td>
<td>18</td>
<td>Fee Recipient: 0xeBe...Acf</td>
</tr>
<tr>
<td>15764022</td>
<td>1 min</td>
<td>282</td>
<td>0xd4e96e8ee8678dbff...</td>
</tr>
<tr>
<td>15764021</td>
<td>1 min</td>
<td>295</td>
<td>0xbb3afde35eb9f5feb53...</td>
</tr>
<tr>
<td>15764020</td>
<td>1 min</td>
<td>71</td>
<td>Fee Recipient: 0x6d2...766</td>
</tr>
</tbody>
</table>

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)

# Validators
(843K)

Staked ETH
(27M)
The economics of staking

Validator locks up 32 ETH. Oct 2023: 27M ETH staked (total)

Annual validator income (an example):

- Issuance: 1.0 ETH
- Tx fees: 0.4 ETH
- MEV: 0.4 ETH
- Total: 1.8 ETH (5.6% return on 32 ETH staked)

Can be adjusted (BASE_REWARD_FACTOR)
A function of congestion

In practice: staking provider (e.g., Lido) takes a cut of the returns
The Ethereum system consists of two layers: the consensus layer (beacon chain) and the compute layer (execution chain). The consensus layer updates the world state by notifying the compute layer with new payload data using the `notify_new_payload(payload)` function. This payload is sent by the Engine API and is used to update the world state. Each epoch contains 32 blocks that are processed by the compute layer.
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>$H(pk)$</td>
<td>$H(CreatorAddr, CreatorNonce)$</td>
</tr>
<tr>
<td><strong>code</strong>:</td>
<td>$\bot$</td>
<td>CodeHash</td>
</tr>
<tr>
<td><strong>storage root</strong> (state):</td>
<td>$\bot$</td>
<td>StorageRoot</td>
</tr>
<tr>
<td><strong>balance</strong> (in Wei):</td>
<td>balance</td>
<td>balance (1 Wei = $10^{-18}$ ETH)</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>root</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>10, d</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>
```

<table>
<thead>
<tr>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0, a</td>
</tr>
<tr>
<td>⊥, b</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>⊥, c</td>
</tr>
</tbody>
</table>

\[
time \text{ to compute root hash: } \leq 2 \times |S|\]

$|S|$ = # 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  \((1 \text{ Wei} = 10^{-18} \text{ ETH})\)
- **Tx fees (EIP 1559)**: `gasLimit, maxFee, maxPriorityFee` (later)
- if **To = 0**: create new contract  \(\text{code} = (\text{init}, \text{body})\)
- if **To ≠ 0**: **data** (what function to call & arguments)
- **nonce**: must match current nonce of sender (prevents Tx replay)
- **chain_id**: ensures Tx can only be submitted to the intended chain
Transaction types:

- owned $\rightarrow$ owned: transfer ETH between users
- owned $\rightarrow$ 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>Uniswap V2: Router 2</td>
<td>0.14 Ether</td>
<td>0.00644563</td>
</tr>
<tr>
<td>0x4299d864bbda0fe32...</td>
<td>Uniswap V2: Router 2</td>
<td>89.83910411882671 Ether</td>
<td>0.00716578</td>
</tr>
<tr>
<td>0x4d1317a2a98cfe4a41...</td>
<td>0xc59f33af5f4a7c8647...</td>
<td>14.501 Ether</td>
<td>0.001239</td>
</tr>
<tr>
<td>0x29ecaa773f052d14e...</td>
<td>CryptoKitties: Core</td>
<td>0 Ether</td>
<td>0.00775543</td>
</tr>
<tr>
<td>0x63bb46461696416fa...</td>
<td>Uniswap V2: Router 2</td>
<td>0.203036474328481 Ether</td>
<td>0.00766728</td>
</tr>
<tr>
<td>0xde70238aef7a35abd...</td>
<td>Balancer: ETH/DOUGH...</td>
<td>0 Ether</td>
<td>0.00261582</td>
</tr>
<tr>
<td>0x69aca10fe1394d535f...</td>
<td>0x837d03aa7fc09b8be...</td>
<td>0 Ether</td>
<td>0.00259936</td>
</tr>
<tr>
<td>0xe2f5d180626d29e75...</td>
<td>Uniswap 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 → owned: contract sends funds to user
- contract → contract: one program calls another (and sends funds)

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

- Tx from owned addr → contract → another contract
- another contract → different owned
Example Tx

world state (four accounts)

Transaction
From: 14c5f8ba
To: bb75a980
Value: 10 eth
Data: 2, CHARLIE
Sig: 30452fde0d3d8f7959f2ce88a1

updated world state
An Ethereum Block

Block proposer creates a block of n Tx:  (from Txs submitted by users)

• 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
Amount of memory to run a node

ETH total blockchain size (archival): 16 TB (Oct. 2023)
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;
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
    }
}
```
function \texttt{nameLookup}(\texttt{bytes32} \texttt{name}) \{ \\
    \text{return data[\texttt{name}];} \\
\}\end{contract} // end of contract

Used by other contracts
Humans do not need this (use etherscan.io)
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**  \texttt{addr} (32 bytes), \texttt{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

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

\texttt{CALL gas, addr, value, args}

\texttt{SELFDESTRUCT 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 Gwei = 86 × 10^-9 ETH

Average Tx fee in USD
congestion
Gas calculation: EIP1559
(since 8/2021)

EIP1559 goals (informal):

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

[ Transaction Fee Mechanism Design, by T. Roughgarden, 2021 ]
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 $\rightarrow$ 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: \(\text{gasLimit} \times \text{gasPrice}\)
Gas calculation (informal)

gasUsed $\leftarrow$ gas used by Tx

Send $\text{gasUsed} \times (\text{gasPrice} - \text{baseFee})$ to block proposer

BURN $\text{gasUsed} \times \text{baseFee}$

$\Rightarrow$ total supply of ETH can decrease
Gas calculation

(1) if gasPrice < baseFee: abort
(2) If gasLimit × gasPrice < msg.sender.balance: abort
(3) deduct gasLimit × gasPrice from msg.sender.balance

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

(7) gasUsed ← gasLimit – Gas
   (7a) BURN gasUsed × baseFee
   (7b) Send gasUsed × (gasPrice – 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>

\[ \approx \text{gasUsed} \times \text{baseFee} \]

- \( \text{baseFee} < 16\text{Gwei} \) \implies \text{new issuance} > \text{burn} \implies \text{ETH inflates}
- \( \text{baseFee} > 16\text{Gwei} \) \implies \text{new issuance} < \text{burn} \implies \text{ETH deflates}
Recall: EIP1559 goals (informal)
• users incentivized to bid their true utility for posting Tx,
• block proposer incentivized to not create fake Tx, and
• disincentivize 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

Total Gas Usage

Evolution of the total gas used by the Ethereum network per day

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