Ethereum Mechanics

Dan Boneh
Stanford University
First, some background: signatures

Physical signatures: bind transaction to author

Bob agrees to pay Alice 1$

Bob agrees to pay Alice 100$

Problem in the digital world:
anyone can copy Bob’s signature from one doc to another
Digital signatures

Solution: make signature depend on document

Signer

Verifier

Bob agrees to pay Alice 1$

'accept' or 'reject'

secret signing key (sk)

public verification key (pk)

signature

signing algorithm

verify
Digital signatures: syntax

Def: a signature scheme is a triple of algorithms:

- **Gen()**: outputs a key pair \((pk, sk)\)
- **Sign(sk, msg)** outputs sig. \(\sigma\)
- **Verify(pk, msg, \sigma)** outputs ‘accept’ or ‘reject’

Secure signatures: (informal)

Adversary who sees signatures **on many messages** of his choice, cannot forge a signature on a new message.
Families of signature schemes

1. RSA signatures (old ... not used in blockchains):
   • long sigs and public keys (≥256 bytes), fast to verify

2. Discrete-log signatures: Schnorr and ECDSA (Bitcoin, Ethereum)
   • short sigs (48 or 64 bytes) and public key (32 bytes)

3. BLS signatures: 48 bytes, aggregatable, easy threshold
   (Ethereum 2.0, Chia, Dfinity)

4. Post-quantum signatures: long (≥600 bytes)

   details in CS255
Signatures on the blockchain

Signatures are used everywhere:
• ensure Tx are valid,
• DAO governance votes,
• consensus protocol votes.

\[ \text{sk_1, sk_2} \]

\[ \text{data, signatures} \]
The layers of a blockchain

- **Data Availability / Consensus Layer**
- **Execution engine** (blockchain computer)
- **applications** (DAPPs, smart contracts)
- **user facing tools** (cloud servers)

most relevant to this course

The network
A bit about the lower layers: how blocks are added

I am the block proposer

validate block

signed

validate block

sk_A

sk_B

sk_C
A bit about the lower layers: how blocks are added

blockchain

\[ \text{sk}_A \]

\[ \text{sk}_B \]

\[ \text{sk}_C \]

I am the block proposer
The consensus layer (informal)

A **public** append-only data structure:

- **Persistence**: once added, data can never be removed*
- **Safety**: all honest participants have the same data**
- **Liveness**: honest participants can add new transactions
- **Open(?)**: anyone can add blocks (no authentication)

*achieved by replication
Ethereum Mechanics

The Ethereum execution engine
Why the EVM?

Full-Time Developers Since Launch | 50+ Avg Developers

Years Since First Commit

source: electric capital
Ethereum: enables a world of applications

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
**Ethereum as a state transition system**

A rich state transition function

⇒ one transition executes an entire program

---

**Diagram:**

- **Ethereum world state**: $S_i$
- **Updated Ethereum world state**: $S_{i+1}$
- **Input**: $Tx$

... → $S_i$ → **Input**: $Tx$ → $S_{i+1}$ → ...
Running a program on a blockchain (DAPP)

- **Consensus layer (beacon chain)**
- **Compute layer (execution chain):** The EVM

Create a DAPP

- Program code
- States: $\text{state}_0$, $\text{state}_1$, $\text{state}_2$, ...
- Transactions: $\text{Tx}_1$, $\text{Tx}_2$
The Ethereum system

Proof-of-Stake consensus

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

Block proposer receives 
Tx fees for block  
(along with other rewards)

<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 Bu...</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>0xd4e96e8ee8678dbff...</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>
The Ethereum system (post merge)

Execution layer

- `notify_new_payload(payload)` [Engine API]
  - Sends transactions to compute layer

Consensus layer (beacon chain)

Update world state

- 32 blocks in an epoch
The Ethereum Compute Layer: The EVM
World state: set of accounts identified by 32-byte address.

Two types of accounts:

1. **owned accounts (EOA):** controlled by a signing key pair \((pk,sk)\).
   - \(sk\): owned by account owner

2. **contracts:** controlled by code (set by creator)
### Data associated with an account

<table>
<thead>
<tr>
<th>Account data</th>
<th>Owned</th>
<th>Contracts</th>
</tr>
</thead>
<tbody>
<tr>
<td>address (computed):</td>
<td>H(pk)</td>
<td>H(CreatorAddr, CreatorNonce)</td>
</tr>
<tr>
<td>balance (in Wei):</td>
<td>balance</td>
<td>balance (10^{18} Wei = 1 ETH)</td>
</tr>
<tr>
<td>code:</td>
<td>⊥</td>
<td>CodeHash</td>
</tr>
<tr>
<td>storage root (state):</td>
<td>⊥</td>
<td>StorageRoot</td>
</tr>
<tr>
<td>nonce:</td>
<td>nonce</td>
<td>nonce</td>
</tr>
</tbody>
</table>

(#Tx sent) + (#accounts created): anti-replay mechanism
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 $S[]$ (simplified)

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

S[000] = a
S[010] = b
S[011] = c
S[110] = d

Time to compute root hash: $\leq 2 \times |S|$ |S| = # non-zero cells
State transitions:  Tx and messages

Transaction types:

- **owned** \(\rightarrow\) **owned**: transfer ETH between users
- **owned** \(\rightarrow\) **contract**: call contract with ETH & data

After a contract is called:

- **contract** \(\rightarrow\) **contract**: one program calls another (composability)
- **contract** \(\rightarrow\) **owned**: contract sends funds to user

Calling a contract can start a chain of transactions: A \(\rightarrow\) B \(\rightarrow\) C \(\rightarrow\) D
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:** data (what function to call & arguments)
- **if To = 0:** create new contract `code = (init, body)`
- **nonce:** must match current nonce of sender (prevents Tx replay)
- **chain_id:** ensures Tx can only be submitted to the intended chain
## Example (block #10993504)

<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>0x2cebe81fe0dcd220e...</td>
<td>0.14 Ether</td>
<td>0.00644563</td>
</tr>
<tr>
<td>0x4299d864bbda0fe32...</td>
<td>0x2cebe81fe0dcd220e...</td>
<td>89.839104111882671 Ether</td>
<td>0.00716578</td>
</tr>
<tr>
<td>0x4d1317a2a98cfeae41...</td>
<td>0xc59f33af5f4a7c8647...</td>
<td>14.501 Ether</td>
<td>0.001239</td>
</tr>
<tr>
<td>0x29ecaa773f052d14e...</td>
<td>0x2cebe81fe0dcd220e...</td>
<td>0 Ether</td>
<td>0.00775543</td>
</tr>
<tr>
<td>0x63bb46461696416fa...</td>
<td>0x2cebe81fe0dcd220e...</td>
<td>0 Ether</td>
<td>0.00766728</td>
</tr>
<tr>
<td>0xde70238aef7a35abd...</td>
<td>0x2cebe81fe0dcd220e...</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>0x2cebe81fe0dcd220e...</td>
<td>0 Ether</td>
<td>0.00665809</td>
</tr>
</tbody>
</table>
The Ethereum blockchain: abstractly

prev hash

updated world state

log messages

block #X

accts.

block #X+1

accts.

updated world state

log messages

prev hash

Tx

Tx

....

....
Amount of memory to run a node

ETH total blockchain size (archival): 13 TB (Feb. 2023)
An example contract: NameSystem

A name system on Ethereum: [uniswap → addr]

(a simplified ENS)

Need to support three operations:

- **Name.new**(OwnerAddr, Name): intent to register
- **Name.update**(Name, newVal, newOwner)
- **Name.lookup**(Name)
An example contract: NameSystem

```solidity
contract nameSys { // Solidity code

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

    // array of all registered domains
    mapping (bytes32 => nameEntry) data;
}
```
function **nameNew**(bytes32 name) {
    // registration fee is 100 Wei
    if (data[name] == 0 && msg.value >= 100) {
        data[name].owner = msg.sender // record 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 commit-reveal.
function nameUpdate(
    bytes32 name, bytes32 newValue, address newOwner) {

    // check if message is from owner, and fee 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 `nameLookup` (bytes32 name) {
    return data[name];
}
} // end of contract

EVM contracts cannot keep secrets
(we need practical iO)

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
• code can CREATE or CALL another contract

In addition: several types of memory
• Persistent storage (on blockchain): SLOAD, SSTORE (expensive)
• Volatile memory (for single Tx): MLOAD, MSTORE (cheap)
• LOG0(data): write data to log
• CallData: arguments in Tx (persistent, but only readable by current Tx)
Every instruction costs gas, examples:

**MLOAD, MSTORE**: 3 gas (cheap)

**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)

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

**CALL** \( \text{gas} \), \( \text{addr} \), \( \text{value} \), \( \text{args} \)
Why charge gas?
• Tx fees (gas) prevents submitting Tx that runs for many steps.
• During high load: block proposer chooses set of Tx from mempool that maximize its income.

Old EVM: (prior to EIP1559, live on 8/2021)
• Every Tx contains a gasPrice ``bid'' (gas $\rightarrow$ Wei conversion price)
• Producer chooses Tx with highest gasPrice $\left( \max \sum (\text{gasPrice} \times \text{gasUsed}) \right)$
  $\Rightarrow$ not an efficient auction mechanism (first price auction)
Gas prices spike during congestion

Average Tx fee in USD
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 (since 8/2021)

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: \( \text{gasLimit} \times \text{gasPrice} \)
Gas calculation

\[ \text{gasUsed} \leftarrow \text{gas used by Tx} \]

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

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

\[ \Rightarrow \] total supply of ETH can decrease
END OF LECTURE