Ethereum: mechanics

Dan Boneh

Note: HW#2 is posted on the course web site. Due Oct. 12.
Limitations of Bitcoin

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 project)

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

⇒ Ethereum
Ethereum: enables many applications

About 3000 Ethereum Decentralized Apps (DAPPs)

• New coins: ERC-20 interface to DAPP
• DeFi: exchanges, lending, stablecoins, derivatives, etc.
• Insurance
• Games: assets managed on chain (e.g. CryptoKitties)
• Managing distinguished assets (ERC-821)

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

![Diagram of Bitcoin state transition system]

**World state**

- \( UTXO_1 \)
- \( UTXO_2 \)
- \( \vdots \)

**Input**

- \( Tx: UTXO_2 \rightarrow UTXO_3 \)

**Updated world state**

- \( UTXO_1 \)
- \( UTXO_3 \)
- \( \vdots \)

**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)

Layer 1: consensus layer

Layer 2: compute layer (executed by miners)

create a DAPP

program code

state₀ → Tx₁ → state₁ → Tx₂ → state₂ → ...

... blockchain ...
The Ethereum system

Layer 1: PoW consensus. Block reward = 2 ETH + Tx fees (gas)

avg. block rate = 15 seconds.
(variant of Nakamoto)

about 150 Tx per block.
Ethereum Layer 1.5: compute layer

World state: set of accounts identified by 160-bit address.

Two types of accounts:

(1) **owned accounts**: controlled by ECDSA signing key pair (PK,SK).
   SK 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></td>
<td>$H(PK)$</td>
<td>$H(CreatorAddr, CreatorNonce)$</td>
</tr>
<tr>
<td><strong>code:</strong></td>
<td>⊥</td>
<td>CodeHash</td>
</tr>
<tr>
<td><strong>storage root</strong></td>
<td>⊥</td>
<td>StorageRoot</td>
</tr>
<tr>
<td><strong>balance</strong></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
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

### Example

- \( S[000] = a \)
- \( S[010] = b \)
- \( S[011] = c \)
- \( S[110] = d \)

---

Time to compute root hash:

\[ \leq 2 \times |S| \]

| \( S[] \) | = # non-zero cells
Transactions: signed data by initiator

- **To:** 32-byte address of target (0 ➔ create new account)
- **From, Signature:** initiator address and signature on Tx
- **Value:** # Wei being sent with Tx
- **gasPrice, gasLimit:** Tx fees (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)
State transitions: Tx and messages

Transaction types:

owned $\rightarrow$ owned: transfer ETH between users
owned $\rightarrow$ contract: call contract with ETH & data
### 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>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><strong>89.33910411882671 Ether</strong></td>
<td>0.00716578</td>
</tr>
<tr>
<td>0x4d1317a2a98cfe4a41...</td>
<td>0xc59f33af5f4a7c8647...</td>
<td><strong>14.501 Ether</strong></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><strong>0.203036474328481 Ether</strong></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 ➔ different owned addr
Example Tx

world state (four accounts)
An Ethereum Block

Miners collect Txs from users ⇒ leader creates a block of n Tx

- Miner does:
  - for i=1,...,n: execute state change of Tx_i
    (can change state of >n accounts)
  - record updated world state in block

Other miners re-execute all Tx to verify block

- Miners should only build on a valid block
- Miners are not paid for verifying block (note: verifier’s dilemma)
Block header data (simplified)

(1) consensus data: parent hash, difficulty, PoW solution, 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: tells verifier how much work to verify block
The Ethereum blockchain: abstractly

The diagram illustrates the Ethereum blockchain, showing the relationship between the previous hash, updated world state, transactions (Tx), logs (log messages), and accounts (accts.).
Amount of memory to run a node (in GB)

ETH total blockchain size: 5.2 TB  (Oct. 2020)
An example contract: NameCoin

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

} // end of contract
Write code in Solidity (or another front-end language)

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

⇒ miners 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; miner 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
## Gas prices: examples

**SSTORE** `addr` (32 bytes), `value` (32 bytes)

- zero → non-zero: 20,000 gas
- non-zero → non-zero: 5,000 gas
- non-zero → zero: 15,000 gas refund

**SUICIDE**: kill current contract. 24,000 gas refund

Refund is given for reducing size of blockchain state
Gas calculation

Tx fees (gas) prevents submitting Tx that runs for many steps

Every EVM instruction costs gas:

- Tx specifies `gasPrice`: conversion: gas → Wei
  `gasLimit`: max gas for Tx
Gas calculation

Tx specifies

gasPrice: conversion gas $\rightarrow$ Wei

gasLimit: max gas for Tx

(1) if \( \text{gasLimit} \times \text{gasPrice} > \text{msg.sender.balance} \): abort

(2) deduct \( \text{gasLimit} \times \text{gasPrice} \) from msg.sender.balance

(3) set Gas = gasLimit

(4) execute Rx: deduct gas from Gas for each instruction

    if (Gas < 0): abort, miner keeps \( \text{gasLimit} \times \text{gasPrice} \)

(5) Refund \( \text{Gas} \times \text{gasPrice} \) to msg.sender.balance
Transactions are becoming more complex

GasLimit is increasing over time ⇒ each Tx takes more instructions to execute
Gas prices: spike during congestion

GasPrice in Gwei:

83B Gwei = 83 \times 10^{-9} \text{ ETH}
END OF LECTURE

Next lecture: writing Solidity contracts