Financial applications of blockchains and distributed ledgers

Master's program in Financial Engineering

Jiahua (Java) Xu

Session 3

Housekeeping

Time and venue

Three sessions: 15:15 – 16:10, 16:25 – 17:20, 17:35 – 18:30

Tuesdays, on Zoom, https://epfl.zoom.us/j/4897861984

To-do's

1. From a group (12 students have done so).
2. Vote for the submission deadline.
3. (optional but appreciated) Contribute to the class discussion, on Moodle or live on Zoom.

From the previous lecture

Hard forks

1. Ethereum vs. Ethereum classic
   - One processes the same amount of ETH and ETC at the time of the fork (Pauls 2016)
2. EOS token migration
   - EOS Token transfer on Ethereum blockchain freezes on 1 June 2018 22:59:59 UTC/GMT
   - Register Ethereum mainnet address prior to the deadline: associate the public key of one's Ethereum wallet with the public key of his (native) EOS wallet (Masnavi 2018)

Game-theoretic modeling on blockchain

1. Bitcoin mining pools: A cooperative game theoretic analysis (Lewenberg et al. 2015)
2. Bitcoin Mining as a Contest (Dimitri 2017)
3. Game-theoretic analysis of DDoS attacks against bitcoin mining pools (Johnson et al. 2014)
4. On Power Splitting Games in Distributed Computation: The Case of Bitcoin Pooled Mining (Luu et al. 2015)

Recap

1. Bitcoin
    - Miners' inventive
    - Merkle tree
    - Adjustable difficulty level
2. Proof-of-Stake and other consensus mechanisms
3. Blockchain vs. Ethereum
    - Record-keeping model
    - Language
4. Decentralized autonomous organization

Cross-platform communication

Oracle

Data feed services that provide smart contracts with external information / off-chain information. (Klages-Mundt et al. 2020)

1. Centralized oracle
   - can be proven that the data feed is an authentic representation of a particular source
   - but it is still inherently manipulable by the source
2. Decentralized oracle: remain an open research question

Atomic swap

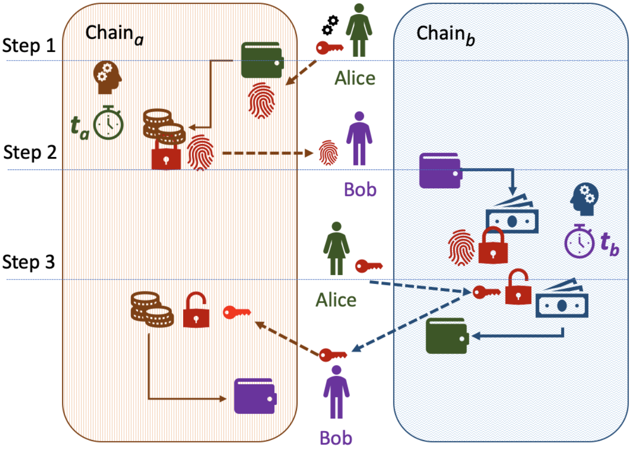

Figure 1: Hash Timelock Contracts

1. Alice initiates the transaction by generating a secret (a key) that will be used to unlock the asset transfers later on. She then deploys a smart contract on Chain_a, that will lock her assets until time t_a. * This contract will transfer to Bob the assets only if the secret generated by Alice is revealed and entered in the smart contract.
   - After time t_a, should the secret have not been revealed, the smart contract expires and Alice's assets will be unlocked and returned to her wallet
2. Bob can verify the contract deployed by Alice on Chain_a (assets, delivery address, etc.) and use the hash submitted by Alice in order to deploy a similar contract on Chain_b.
3. Alice can verify the contract deployed on Chain_b, unlock the assets, and initiate their transfer to her wallet by revealing the secret on Chain_b.
4. As early as when the secret is revealed in the mempool of Chain_b (even before the Alice's transfer is confirmed), Bob can use the secret to unlock the assets on Chain_a and complete the cross-ledger transaction.

Decentralized finance

Decentralized exchange (DEX)

- DEXs on Ethereum
  - Automated market makers (AMM): Uniswap, Bancor ...
- DEXs on XRPL
  - Ledger gateway

Trading platform

- fidentiaX: secondary life insurance trading on blockchain

Lending platform

- Compound

Stablecoin (Klages-Mundt et al. 2020)

- Custodial
  - Reserve Fund
  - Fractional Reserve Fund
  - Central Bank Digital Currency (CBDC)
- Non-custodial

Coding smart contracts (Perez 2019)

Flow of a Bitcoin redeem script

1. Write script
2. Hash script to create address
3. Receive Bitcoins
4. Publish script and required data (usually signature) using a transaction

From Bitcoin Scripting to Smart Contracts

We need more features to write general programs

- Persistent state
  $\rightarrow$ account-based
  $\rightarrow$ storage primitives
- Turing-completeness (loops)
  $\rightarrow$ jump primitive
- More transparency?
  $\rightarrow$ code deployed before usage

Smart Contract implementing a simple coin

```
contract Coin {
  address public minter;
  mapping (address => uint) public balances;

  constructor() public { minter = msg.sender; }

  function mint(address receiver, uint amount) public {
    require(msg.sender == minter);
    require(amount < 1e60);
    balances[receiver] += amount;
  }

  function send(address receiver, uint amount) public {
    require(amount <= balances[msg.sender]);
    balances[msg.sender] -= amount;
    balances[receiver] += amount;
  }
}
```

What can/can't Smart Contracts do?

Can

- Perform pretty much any computation
- Persist data (e.g. balance of users)
- Transfer money to other addresses or contracts

Can't

- Interact with anything outside of the blockchain
- Be scheduled to do something periodically

Flow to use a Smart Contract

1. Write high-level code for the contract
2. Test the contract
3. Compile the contract into bytecode
4. Send a transaction to deploy the contract
5. Interact with the contract by sending transactions to the generated address

How Smart Contracts are executed

We want to execute smart contract at address A

- User sends a transaction to address A
- Transaction is broadcasted in the same way as other transactions
- Miner executes the smart contract at address A
- If execution succeeds, new state is computed
- When receiving the block containing the transaction, other nodes re-execute smart contract at address A

A few issues

How do we make sure that

- Execution terminates
- Users do not use too much storage
- Execution on different machines always yields the same result

Simple loop from 0 to 10 using EVM instructions

```
for (uint i = 0; i < 10; i++) {}
```

. . . will look something like

```
PUSH1 0x00
PUSH1 0x00
MSTORE          ; store 0 at position 0 in memory
JUMPDEST        ; set a place to jump (PC = 6)
PUSH1 0x0a      ; push 10 on the stack
PUSH1 0x00
MLOAD           ; load loop counter
PUSH1 0x01
ADD             ; increment loop counter
DUP1
PUSH1 0x00
MSTORE          ; store updated loop counter
LT              ; check if loop counter is less than 10
PUSH1 0x06
JUMPI           ; jump to position 6 if true
```

Metering

Ethereum uses the concept of *gas*

- Transactions have a base gas cost
- Each instruction costs a given amount of gas to execute
- Transactions have a gas budget to execute
- Blocks have a total gas budget

Gas has two main purposes

- Protect against DoS attacks
- Incentivize miners

Gas computation

Back to the previous example

```
PUSH1 0x00      ; 3 gas
PUSH1 0x00      ; 3 gas
MSTORE          ; 3 gas
JUMPDEST        ; 1 gas
PUSH1 0x0a      ; 3 gas
PUSH1 0x00      ; 3 gas
MLOAD           ; 3 gas
PUSH1 0x01      ; 3 gas
ADD             ; 3 gas
DUP1            ; 3 gas
PUSH1 0x00      ; 3 gas
MSTORE          ; 3 gas
LT              ; 3 gas
PUSH1 0x06      ; 3 gas
JUMPI           ; 10 gas
```

Total 410 gas: 10 for first 4 instructions, then 40 x 10

Gas computation: special cases

Some instructions, have special rules. For example, SSTORE rules are:

- If allocate storage: 20,000
- If modify allocated storage: 5,000
- If free storage: -15,000

```
PUSH 0x01
PUSH 0x00
SSTORE     ; allocate: 20,000 gas
PUSH 0x02
PUSH 0x00
SSTORE     ; modify: 5,000 gas
PUSH 0x00
PUSH 0x00
SSTORE     ; free: -15,000 gas
```

Gas and incentives

Miners are rewarded proportionally to the amount of gas each transaction consumes.

- Transaction senders set a *gas price*
  - Amount of money/gas that the sender is ready to pay
  - Miners are incentivized to include transactions with higher gas price
- Miners receive gas used $\times$ gas price for each transaction in the mined block
  - If gas budget is not fully used, gas left is returned to sender
  - If execution fails, the gas used is not returned

Ethereum Smart Contract Programming

Solidity

- High-level language targeting the EVM
- Looks vaguely like JavaScript
- Strongly typed, with a fairly simple type-system
- Contains smart contract related primitives
- Supports multiple inheritance

Compiling Smart Contracts: functions

- EVM bytecode has no concept of functions, only conditional jumps
- Solidity creates a conditional jump for each function
- Solidity uses function signatures to choose which function to call
- Transaction sent to the contract must contain the necessary data to trigger the function

Sample signature

`claimFunds(address receiver)`

Conditional jumps

```
CALLDATASIZE       ; load data size
ISZERO
PUSH2 0x00c4       ; default function location
JUMPI
CALLDATALOAD       ; load data
DUP1
PUSH4 0x24600fc3   ; function signature hash
EQ
PUSH2 0x00db       ; function location
JUMPI
DUP1
PUSH4 0x30b67baa
EQ
PUSH2 0x00e6
JUMPI
```

Compiling Smart Contracts: types

- EVM only has 256 bit words
- Solidity has a simple type system including
  - integer types
  - data structures (lists, maps)
- Integer types are encoded using bitwise operations
  e.g. `uint8`: `uint256 & 0xff`
- Data structures are encoded using hash
  e.g. `key(list[5]) = keccak256(index(list) . 5)`

Programming hands-on

Ecosystem Overview

- Solc: Solidity compiler
- Truffle: Framework to help build/test
- Ganache: Easy setup of local private chain
- Mythril, Securify, etc: Static analysis tools

Installing software

NodeJS (if not already installed)

Follow instructions at: https://nodejs.org/en/download/

Truffle

```
npm install -g truffle
```

What we will build

A simple token compliant with the ERC-20 standard

This is how most “coins” or “tokens” are implemented on Ethereum. It defines a common interface to

- Transfer tokens
- Allow other parties to transfer tokens
- Check balance for tokens
- Emit events for token transfers

ERC-20 interface

```
// Returns the total supply of tokens
function totalSupply() public view returns (uint256)

// Returns the balance of `_owner`
function balanceOf(address _owner) public view returns (uint256 balance)

// Transfers `_value` from sender to `_to`
function transfer(address _to, uint256 _value) public returns (bool success)

// Transfers `_value` from `_from` to `_to` if `_from` authorized the send
function transferFrom(
address _from, address _to, uint256 _value
) public returns (
bool success
)

// Approves `_spender` to spend `_value` on behalf of the sender
function approve(
address _spender, uint256 _value
) public returns (
bool success
)

// Returns how much `_spender` is allowed to spend on behalf of `_owner`
function allowance(
address _owner, address _spender
) public view returns (
uint256 remaining
)

// Is emitted when `_from` transfers `_value` to `_to`
event Transfer(address indexed _from, address indexed _to, uint256 _value)
// Is emitted when `_owner` allows `_spender` to spend `_value` on his behalf
event Approval(address indexed _owner, address indexed _spender, uint256 _value)
```

Token specifics

We will build a very simple token:

- Fixed total supply (1,000,000 for the sake of example)
    - No tokens can be created or burned after creation
- All tokens belong to owner at contract creation time
- No other particular limitation

Starting to develop

Start a new project

```
mkdir my-token
cd my-token
truffle init
truffle create contract MyToken
```

Create migration file: `migrations/2_my_token.js`

```
const MyToken = artifacts.require("MyToken");

module.exports = function(deployer) {
  deployer.deploy(MyToken);
};
```

Download the specs for the project

```
wget https://git.io/smart-contract-intro-spec -O test/my-token-test.js
```

Run the tests

```
truffle test
```

Get the contract skeleton (optional)

If you are not confident, you can get the skeleton to get started

```
wget https://git.io/smart-contract-intro-skel -O contracts/MyToken.sol
```

Now, implement the contract and run the tests regularly.
Check the ERC-20 standard for more details about each function.

Thank you!

Contact

Jiahua (Java) Xu

Ecole Polytechnique Fédérale de Lausanne (EPFL)

EXTRA 249 (Extranef UNIL)

Quartier UNIL-Dorigny

jiahua.xu@epfl.ch

References I

Dimitri, Nicola. 2017. "Bitcoin Mining as a Contest." *Ledger* 2: 31–37. https://doi.org/10.5195/ledger.2017.96.

Johnson, Benjamin, Aron Laszka, Jens Grossklags, Marie Vasek, and Tyler Moore. 2014. "Game-theoretic analysis of DDoS attacks against bitcoin mining pools." In *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 8438:72–86. Springer Verlag. https://doi.org/10.1007/978-3-662-44774-1_6.

Klages-Mundt, Ariah, Dominik Harz, Lewis Gudgeon, Jun-You Liu, and Andreea Minca. 2020. "Stablecoins 2.0: Economic Foundations and Risk-based Models." In *2nd Acm Conference on Advances in Financial Technologies (Aft '20)*. New York. https://doi.org/10.1145/3419614.3423261.

References II

Lewenberg, Yoad, Yoram Bachrach, Yonatan Sompolinsky, Aviv Zohar, and Jeffrey S. Rosenschein. 2015. "Bitcoin mining pools: A cooperative game theoretic analysis." In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems, Aamas*, 2:919–27. www.ifaamas.org.

Luu, Loi, Ratul Saha, Inian Parameshwaran, Prateek Saxena, and Aquinas Hobor. 2015. "On Power Splitting Games in Distributed Computation: The Case of Bitcoin Pooled Mining." In *2015 Ieee 28th Computer Security Foundations Symposium*, 2015-Septe:397–411. IEEE. https://doi.org/10.1109/CSF.2015.34.

Masnavi, Siamak. 2018. "EOS Blockchain Upgrade (Mainnet Launch) and Token Migration: Everything You Need to Know." https://www.cryptoglobe.com/latest/2018/05/eos-blockchain-upgrade-mainnet-launch-and-token-migration-everything-you-need-to-know/.

References III

Pauls. 2016. "Ethereum Fork: Step-by-Step Guide to Safely Splitting your ETH/ETC." https://steemit.com/ethereum/@pauls/ethereum-fork-step-by-step-guide-to-safely-splitting-your-eth-etc.

Perez, Daniel. 2019. "Introduction to Smart Contracts." https://daniel.perez.sh/talks/2019/smart-contract-intro/%7B/#%7D1.