Financial applications of blockchains and distributed ledgers

Master's program in Financial Engineering

Jiahua (Java) Xu

Session 3

EPFL

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.
2. Vote for the submission deadline.
3. (optional but appreciated) Contribute to the class discussion, on Moodle or live on Zoom.
4. (optional) The missing recording from last time ...

From the previous lecture

Hyperinflation in Zimbabwe and Venezuela

1. Deficit government spending 1.f

Special Drawing Right (SDR)

1. "IMF members can also use SDRs in operations and **transactions involving the IMF**, such as the **payment of interest on and repayment of loans, or payment for future quota increases**."
2. Facebook's Libra:

Recap

1. Double spending

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

Cross-platform communication

Oracle

Data feed services that provide smart contracts with external information / off-chain information.

Atomic swap

Hash Timelock Contracts

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

Ethereum Virtual Machine (EVM) Bytecode ...

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 (loop count
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 × 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)`

Smart Contract Security

Smart Contracts: What could go wrong?

TheDAO hack (2016)

- TheDAO raised ~*$150M in ICO*
- Soon after, it got hacked ~*$50M*
- Price of Ether halved
- Ethereum community decided to hard-fork
- Attacker used a *re-entrancy* vulnerability

Parity Wallet bug (2017)

- Parity wallet library was used to manage multisig wallet contracts
- Parity *wallet* has been *removed* due to a "bug"
- *Dependent contracts* became unable to send funds
- Around *$280M frozen*

Common vulnerabilities / bugs

- Re-entrancy
  - Can allow an attacker to drain funds
- Unhandled exceptions
  - Can result in lost funds
- Dependency on destructed contract
  - Can result in locked funds
- Transaction order dependency
  - Can allow an attacker to manipulate prices
- Integer overflow
  - Can result in locked fund

Re-entrancy

- Vulnerable contract sends money before updating state
- Attacker contract's fallback function is called
- Attacker contract makes re-entrant call to attacker

Vulnerable contract

```
function payMe(address account) public {
  uint amount = getAmount(account);
  // XXX: vulnerable
  if (!account.send(amount))
    throw;
  balance[account] -= amount;
}
```

Attacker contract

```
function () {
  victim.payMe(owner);
}
```

Unhandled exception

- In Solidity, not all failed "calls" raise an exception
- If the failed call returns a boolean, it must be checked correctly
- Failure to do this could result in inconsistent state or even locked funds

Problematic contract

```
// allows user to withdraw funds
function withdraw(address account) public {
  uint amount = getAmount(account);
  balance[account] -= amount;
  account.send(amount); // could silently fail
}
```

Dependency on destructed contract

- Contracts can use other contracts as library
- If the library contract gets destructed, the call becomes a no-op
- If the only way for a contract to send money is to use the library, Ether can be locked

Library contract

```
function sendEther(address recipient,
                   uint value) public onlyOwner {
  address.send(value)
}
```

Contract using library

```
address public library = 0xdcc703c0E500B653Ca82273B7BFAd8045D8
function sendEther(address recipient, uint value) public {
  bytes4 sig =  bytes4(sha3("sendEther(address, uint)"));
  library.delegatecall(sig, recipient, value);
}
```

Transaction Order Dependency

- Result can change depending on the order of the transactions
- Miners are free to choose the miner order of the transactions in a block
- There can be financial incentives to perform such manipulations

Contract vulnerable to transaction order dependency

```
function solve() {
  if (msg.sender == owner) {// update reward
    owner.send(reward);
    reward = msg.value;
  } else if (puzzleSolved(msg.data)) {
    msg.sender.send(reward); //send reward
    solution = msg.data;
  }
}
```

Integer Overflow

- Solidity has many different numeric types
  - `int8` to `int256` and `uint8` to `uint256`
- Types are encoded in EVM using bit manipulations
  - If `a` is `uint8`, `a AND a 0xff` would be generated
- Variables may therefore overflow or underflow during execution

Contract vulnerable to integer overflow

```
function overflow(uint fee) {
  uint amount = 100;

  // underflows if fee > 100
  amount -= fee;

  // tries to send a large value
  // and fails on underflow
  msg.sender.send(amount);
}
```

Smart Contract analysis tools

- Usually static analysis and/or symbolic execution
- Work either on Solidity or on the EVM bytecode
- Check for known vulnerabilities/patterns

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 {.shrink, .plain}

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

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

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

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

// Approves `_spender` to spend `_value` on behalf of the s
```

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: .text.minor[migrations/2_my_token.js]

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

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

Implement and test the project

.small-margin.text.minor[

Download the specs for the project

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

Run the tests

```
truffle test
```

Get the contract skeleton (optional)
.text.minor[If you are not confident, you can get the skeleton to get started]

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

]

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