ECE 443/518 – Computer Cyber Security
Lecture 17 Smart Contract, Oblivious Transfer

Professor Jia Wang
Department of Electrical and Computer Engineering
Illinois Institute of Technology

October 26, 2022

Outline

Smart Contract

Oblivious Transfer (OT)

Reading Assignment

- This lecture: Smart Contract, Oblivious Transfer
- Next lecture: Secure Multi-Party Computation

Outline

Smart Contract

Oblivious Transfer (OT)

From Ledger to State Machine

- The ledger as stored in the block chain can be treated as a very simple state machine.
  - Initial state: initial account balances
  - Currect state: current account balances
  - State transitions: each blockchain transaction updates account balances by addition and subtraction.
- The blockchain can support more complex state machines.
  - Allow accounts to define state variables in addition to balance.
  - Allow blockchain transactions to perform more operations on state variables than simple addition and subtraction.
- This is similar to how we build computer hardware and software to support general purpose computing need.
  - E.g. Ethereum Virtual Machine (EVM) defined by the Ethereum blockchain uses 8-bit opcode and a stack to organize its 256-bit registers, and supports high-level programming languages like Solidity.

Smart Contract

- What are the benefits of running state machines and thus programs in a blockchain?
  - Not for efficiency since each computation needs to be executed as many times as anyone would need to validate the blockchain, using the same inputs and generating the same output.
  - Nonrepudiation: the account initiates a computation must sign the request.
  - Integrity: the outcome is permanentely recorded in the blockchain and cannot be reverted.
  - As long as there is no branch.
- That is what is necessary to execute a contract.
  - Smart contract: a program running inside a blockchain.

A Smart Contract Example

```
pragma solidity 0.8.7;

contract VendingMachine {
  // Declare state variables of the contract
  address public owner;
  mapping (address => uint) public cupcakeBalances;

  // When 'VendingMachine' contract is deployed:
  // 1. set the deploying address as the owner of the contract
  // 2. set the deployed smart contract's cupcake balance to 100
  constructor() {
    owner = msg.sender;
    cupcakeBalances[address(this)] = 100;
  }
  ...
```

- A smart contract that you can buy cupcakes on Ethereum.
- No you don't receive an actual cupcake.
  - What you received could be treated as a ticket or token to redeem a physical cupcake somewhere.

Smart Contract Account

```
...
constructor() {
  owner = msg.sender;
  cupcakeBalances[address(this)] = 100;
}
...
```

- Once created, a smart contract will has its own address, as indicated by `address(this)`
- Other accounts interact with the smart contract by sending (signed) messages to the smart contract account.
- The smart contract will handle these messages in member functions.
  - `constructor` is a special one called for the first message which deploys the smart contract.

The Message Sender

```
contract VendingMachine {
  // Declare state variables of the contract
  address public owner;
  mapping (address => uint) public cupcakeBalances;

  // When 'VendingMachine' contract is deployed:
  // 1. set the deploying address as the owner of the contract
  // 2. set the deployed smart contract's cupcake balance to 100
  constructor() {
    owner = msg.sender;
    cupcakeBalances[address(this)] = 100;
  }
  ...
```

- `msg.sender` indicates who initiates the computation.
  - The payer of cryptocurrency.
- The sender should in addition specify what transactions (member function) is to be performed (called).
  - E.g. one of `constructor`, `refill`, and `purchase`
  - Plus other necessary parameters.

Transactions

```
contract VendingMachine {
  ...
  // Allow the owner to increase the smart contract's cupcake balance
  function refill(uint amount) public {
    require(msg.sender == owner, "Only the owner can refill.");
    cupcakeBalances[address(this)] += amount;
  }

  // Allow anyone to purchase cupcakes
  function purchase(uint amount) public payable {
    require(msg.value >= amount * 1 ether, "1 ETH per cupcake");
    require(cupcakeBalances[address(this)] >= amount, "Not enough in stock");
    cupcakeBalances[address(this)] -= amount;
    cupcakeBalances[msg.sender] += amount;
  }
}
```

- `msg.value` indicates money the sender pays the the contract.
  - The money is transfered from the sender address to the contract address automatically if the computation completes successfully.
- How could one withdraw money from the contract?

Complications

- What if there is an infinite loop into a smart contract?
  - Can be exploited by adversaries to jam the blockchain.
  - In theory, we cannot detect if there is an infinite loop in a program.
  - On blockchain, we can solve the issue by limiting the number of instructions a smart contract may execute by the transaction fee the sender would like to pay.
- Since the program of a smart contract need to be deployed to the blockchain, everyone can see and analyze it.
  - Bugs in the program could be found and exploited by adversaries.

Outline

Smart Contract

Oblivious Transfer (OT)

Oblivious Transfer (OT)

- Alice runs a pay-per-view service that provides access to n messages $m_1, m_2, \ldots, m_n$.
- Bob would like to access a particular message m_k.
- Bob don't want to let Alice know what is k.
  - For privacy reasons.
- Bob don't want to pay Alice a lot of money to obtain all the messages in order to hide k.
- Let's consider the simple case for two messages ($n = 2$).
  - Alice's secret: m_1, m_2.
  - Bob's secret: $k \in \{1, 2\}$.
  - At the end, Bob learns m_k but not the other among the two messages, and Alice learns nothing about k.
- How could this even be possible?
  - Assume Alice and Bob are honest but curious.

- Alice's RSA key pair: $k_{pr} = (n = pq, d)$, $k_{pub} = (n, e)$.

1. Alice sends Bob two random messages x_1 and x_2.
2. Bob generates a random message y and sends Alice v.
   - $v = (y^e + x_k) \bmod n$.
3. Alice sends Bob m_1' and m_2'.
   - $m_1' = m_1 + ((v - x_1)^d \bmod n)$.
   - $m_2' = m_2 + ((v - x_2)^d \bmod n)$.
4. Bob computes $m_k' - y$ to recover m_k.
   - For $k = 1$, RSA guarantees that $m_1' = m_1 + ((v - x_1)^d \bmod n) = m_1 + (y^{ed} \bmod n) = m_1 + y$.
   - Same applies when $k = 2$.
   - So Bob indeed learns m_k.

Analysis for Alice

- The only piece of information Alice directly learns from Bob is the message v.
  - $v = (y^e + x_k) \bmod n$.
  - Note that Alice has no kwowledge about y and k.
- With x_1 and x_2, Alice may derive y_1 and y_2.
  - $y_1 = (v - x_1)^d \bmod n$.
  - $y_2 = (v - x_2)^d \bmod n$.
- $v \equiv y_1^e + x_1 \equiv y_2^e + x_2 \pmod{n}$.
  - Alice cannot decide which of y_1 and y_2 is y.
- Alice learns nothing about Bob's secret k.
  - No matter how powerful Alice is.

Analysis for Bob

- Assume $k = 1$ for Bob.
  - Bob will learn m_1.
  - Does Bob learn anything about m_2?
- Bob learns x_1, x_2, m'_1, m'_2 directly from Alice.
  - x_1 and x_2 are simply random messages, providing no information on m_2.
  - $m'_1 = m_1 + y$, having nothing to do with m_2.
- $m'_2 \equiv m_2 + (v - x_2)^d \equiv m_2 + (y^e + x_1 - x_2)^d \pmod{n}$.
  - Bob may learn m_2 if and only if he can decrypt the ciphertext $y^e + x_1 - x_2$ encrypted with Alice's public key.
  - Since Alice chooses x_1 and x_2, to decrypt $y^e + x_1 - x_2$ implies Bob could decrypt any message encrypted with Alice's public key – this breaks RSA.
- Bob, if computationally bounded, learns nothing about m_2.

Summary

- Smart contracts are programs running inside a blockchain, reacting to blockchain events.
- Oblivious transfer (OT) as a building block for more complicated protocols.