Blockhains and Distributed Ledgers Lecture 03

Aggelos Kiayias

credit to Aydin Abadi
for slide preparation

THE UNIVERSITY of EDINBURGH

Lecture 03

- The ledger as a platform.
- Smart Contracts
- Ethereum

The Idea of Smart Contracts

- **Contract**: formalizes a relationship and contains a set of promises made between principals.
- **Smart Contract** (put forth in 1994 by Nick Szabo): provides new ways to formalize and secure digital relationships which can be more functional compared to paper-based.
    - can reduce costs, for either principals, third parties, or their tools.
    - uses cryptographic and other security mechanisms, in order to secure algorithmically specifiable relationships from being breached and ensure the agreed upon terms are satisfied.

Smart Contracts

- The decentralised nature of cryptocurrencies, such as Bitcoin provides an infrastructure for hosting smart contracts.
  - The code is run by the consensus peers and the correctness of execution is guaranteed by the consensus protocol of the blockchain.
  - We can think of smart contracts as being executed by a trusted global machine that will honestly perform every instruction.
- In this context, a smart contract is a piece of computer program stored and executed on the blockchain.
- The program code captures the logic of contractual clauses between parties.
- Thus, smart contracts minimise trust (and maybe operation costs).

Bitcoin
Transaction, Block chain and Ledger

- A transaction is a transfer of Bitcoin value broadcast to the Bitcoin network.

- When someone wants to spend/transfer its bitcoin, it generates a transaction, signs it and sends the transaction to the network of nodes.

- Each node collects the transaction it receives, checks its correctness, and orders the set of transactions in a block structure.

- The nodes compete with each other and when one solves the proof-of-work puzzle emits the set of transactions.

Bitcoin Transaction

- Each transaction consists of the following main fields:
  - a list of inputs: an input points to an output of a previous transaction from which it wants to spend some Bitcoin. Each input itself has the following main fields: (a) previous transaction address, (b) index, (c) and ScriptSig.
  - a list of outputs: An output comprises instructions for sending bitcoins. Each output has the following main fields: (a) value (that the transaction want to send) and (b) ScriptPubKey.

Bitcoin Transaction's Input Main Fields

- Previous transaction address (txid): Contains a hash value of previous transaction.

- Index: recall that the previous transaction may have a list of outputs, the index refers to a position of desirable output in that list.

- ScriptSig (signature script): Allows the spender of bitcoin to provide evidence that (some of) the bitcoin in the previous transaction belong to it. This field contains the transaction's creator (or spender) public key and a signed message (or a signature). Specifically, message it signs includes:

  - The txid and output index of the previous transaction, the previous output's ScriptPubKey.

Bitcoin Transaction's Output Main Fields

- Value: the number of Bitcoin that this output will be worth when claimed by the next spender.
- ScriptPubKey: Various Bitcoin script commands including address of recipient.

Validating a Transaction

- To validate a transaction each node:
  - concatenates ScriptSig of the current transaction with ScriptPubKey of the referenced transaction.
  - checks if the value is successfully compiled with no errors (and the result value is true). If yes, the transaction is valid.

Bitcoin Script

- Is simple, stack-based and processed from left to right. The script words are also called opcodes, commands or functions.
- Any data in the script is enclosed in <>, e.g. <sig>, <pubKey>
- The opcodes can be categorised as follows:
  - Arithmetic, e.g. OP_ABS, OP_ADD
  - Stack, e.g. OP_DROP, OP_SWAP
  - Flow control, e.g. OP_IF, OP_ELSE
  - Bitwise logic, e.g. OP_EQUAL, OP_EQUALVERIFY
  - Crypto for
    - Hashing, e.g. OP_SHA1, OP_SHA256
    - (Multiple) Signature Verification, e.g. OP_CHECKSIG, OP_CHECKMULTISIG
  - Locktime, e.g. OP_CHECKLOCKTIMEVERIFY, OP_CHECKSEQUENCEVERIFY

Bitcoin Script Execution Example

Block n

Input: // something
Output:

OP_DUP
OP_HASH160
<pubKeyHash2>
OP_EQUALVERIFY
OP_CHECKSIG
OP_DUP
OP_HASH160
<pubKeyHash1>
OP_EQUALVERIFY
OP_CHECKSIG

ScriptPubKey

. . .

Block n+1

Input:
…..

<sig1>
<pubKey1>
<sig2>
<pubKey2>

Output: // something
…
…

ScriptSig

<sig1> <pubKey1> <sig2> <pubKey2> OP_DUP OP_HASH160 <pubKeyHash2> OP_EQUALVERIFY OP_CHECKSIG OP_DUP OP_HASH160 <pubKeyHash1> OP_EQUALVERIFY OP_CHECKSIG

Given:
a. Transaction senders address: <pubKeyHash1> and <pubKeyHash2>
b. Their public keys: <pubKey1> and <pubKey2>
c. Their signatures: <sig1> and <sig2>

we want to authenticate the transaction senders, by checking if their address (i.e. <pubKeyHash>) belongs to them. To this end, we check:

1. the transaction senders's address equals the hash of the public key they provide.
2. given the public we can successfully verify the signature (or signed message).

Stack top

Bitcoin Script Execution Example

Stack	Script	Description
Empty	<sig1> <pubKey1> <sig2> <pubKey2> OP_DUP OP_HASH160 <pubKeyHash2> OP_EQUALVERIFY OP_CHECKSIG OP_DUP OP_HASH160 <pubKeyHash1> OP_EQUALVERIFY OP_CHECKSIG	
<sig1> <pubKey1> <sig2> <pubKey2>	OP_DUP OP_HASH160 <pubKeyHash2> OP_EQUALVERIFY OP_CHECKSIG OP_DUP OP_HASH160 <pubKeyHash1> OP_EQUALVERIFY OP_CHECKSIG	We add constant values from left to right to the stack until we reach an opcode.
<sig1> <pubKey1> <sig2> <pubKey2> <pub2Key>	OP_HASH160 <pubKeyHash2> OP_EQUALVERIFY OP_CHECKSIG OP_DUP OP_HASH160 <pubKeyHash1> OP_EQUALVERIFY OP_CHECKSIG	We duplicate the item at the top of the stake.
<sig1> <pubKey1> <sig2> <pubKey2> <pub2Hash>	<pubKeyHash2> OP_EQUALVERIFY OP_CHECKSIG OP_DUP OP_HASH160 <pubKeyHash1> OP_EQUALVERIFY OP_CHECKSIG	We hash the item at the top of the stack and replace it with the hash value.
<sig1> <pubKey1> <sig2> <pubKey2> <pub2Hash> <pubKeyHash2>	OP_EQUALVERIFY OP_CHECKSIG OP_DUP OP_HASH160 <pubKeyHash1> OP_EQUALVERIFY OP_CHECKSIG	We add the constant value to the stack
<sig1> <pubKey1> <sig2> <pubKey2>	OP_CHECKSIG OP_DUP OP_HASH160 <pubKeyHash1> OP_EQUALVERIFY OP_CHECKSIG	We check if top two items are equal.
<sig1> <pubKey1>	OP_DUP OP_HASH160 <pubKeyHash1> OP_EQUALVERIFY OP_CHECKSIG	We check if the signature matches the public key.
<sig1> <pubKey1> <pubKey1>	OP_HASH160 <pubKeyHash1> OP_EQUALVERIFY OP_CHECKSIG	We duplicate top stack item
<sig1> <pubKey1> <pub1Hash>	<pubKeyHash1> OP_EQUALVERIFY OP_CHECKSIG	We hash at the top of the stack
<sig1><pubKey1> <pub1Hash><pubKeyHash1>	OP_EQUALVERIFY OP_CHECKSIG	We push the value to the stack
<sig1> <pubKey1>	OP_CHECKSIG	we check if top two items are equal
Empty	TRUE	we verify the signature.

Smart Contracts in Bitcoin

some examples

- Multisignatures: at least m keys are required to authorise a Bitcoin transaction, where n>=m; applications include:
    1. Increased security: keys are distributed among n devices (e.g. smart phone, laptop, etc). Even if less than m devices are corrupted and the keys are stollen, you'd still posses your bitcoin.
    2. Business account management: at least m stakeholders of a company must sign a transaction in order to transfer some bitcoin.
- CoinJoin: To make tracing transactions harder. For example, a group of bitcoin holders get together and mix their coins. At the end of the procedure, nobody, other than the participants, can determine for sure which bitcoin belongs to which one of them.

Bitcoin Limitations

- The scripting language as implemented in Bitcoin has limitations:

1. **Lack of Turing-completeness** (e.g., no loops): although Bitcoin scripting language supports a large set of computation, it does not support everything.
    - Bitcoin purposefully picked a non Turing-complete language.
2. Lack of arbitrary state variables: contracts on Bitcoins cannot hold arbitrary state variables; i.e. a variable that remembers preceding events and user interactions and it can be changed by a program/contract.
    - So it is hard to build stateful multi-stage contracts on bitcoin.
    - Unspent transaction outputs (UTXO) is an implicit state in Bitcoin, but Bitcoin scripts do not modify it as they either succeed or fail.

Extending Bitcoin Functionalities

- There are some alternative blockchains that try to add more capabilities and features to Bitcoin:
  - Distributed domain name service, e.g. Namecoin.
  - Distributed system to enable financial functions, e.g. Omni.
  - Decentralised network for peer to peer ecommerce (or a decentralised version of eBay or Amazon), e.g. OpenBazzar.

How to bridge the gap?

- Can we have a generic platform on which users can create their own (decentralised) projects/applications easily without having to extend/modify Bitcoin?

Ethereum

- Ethereum is an open blockchain platform on which user defined decentralised applications can be build and run.

- Ethereum, similar to Bitcoin, is based on a distributed public blockchain, peer to peer network and consensus.

- … a programmable blockchain.

- In order for users to build an application on Ethereum, they design smart contract(s).

- Each smart contract is stored in the Ethereum blockchain and run by the network nodes when it is invoked.

Ethereum

- “Ether” is the internal currency of Ethereum.
- Ether is used to pay transaction and computation fees.
- Ethereum has a list of denominations each of which has its own name. The smallest denomination is called Wei.

Unit	Wei
Wei	1
Kwei	1,000
Mwei	1,000,000
Gwei	1,000,000,000
Szabo	1,000,000,000,000
Finney	1000,000,000,000,000
Ether	1000,000,000,000,000,000

Ethereum and Accounts

- Two types of account in Ethereum:
    - Externally owned account.
    - (Smart) contract account.
- Each Account has a ether balance and stores the number of Ether it possesses in Wei.

Ethereum and Accounts

	Externally owned account	Contract account
Identified by an address	✓	✓
Holds the account's Ether balance	✓	✓
Hold contract code	✗	✓
Holds the account's storage	✓	✓
Associated private-public key	✓	✗
Can send signed transactions	✓	✗
Can create contracts	✓	✓
Can send unsigned transactions	✗	✓
Holds a nonce	✓	✓

- Note that contract can send messages and create another contract(s)!

- A signed transaction is sent only by externally owned account and sending it will incur costs.

State Machine Replication

- Is a general method for implementing a fault-tolerant distributed service by replicating servers and coordinating client interactions with server replicas.
- Formally, a (deterministic) state machine is defined as: (I, S, s_0, σ, F)

  where:

  - I : a set of inputs.
  - S : a set of states.
  - s_0 : an initial state, and $s_0 \in S$
  - σ : a state-transition function defined as $\sigma : S \times I \to S$
  - F : a set of final states, and $F \subset S$
- At any given time, a state machine can be in exactly one state.
- In a distributed setting, all machine replicas must be in an identical state, given the same input.

State Machine Replication Bitcoin

- Bitcoin is a specialised version of a state machine.
- The ledger of Bitcoin can be thought of as a state defined as the ownership status of all existing bitcoin.
- The initial state is the genesis block (i.e. 1st block in the blockchain).
- State-transition function takes the state and a transaction and outputs a new state.

State Machine Replication Ethereum

- In Ethereum, the state includes all accounts' information (e.g. state/storage, Ether balances, contracts code, etc).

- The state-transition function takes the state and a transaction and outputs a new state.

- Miners keep track of accounts Ether balances and update them.

Ethereum & Gas

- As Ethereum supports a Turing-complete language, any code can be written in it, including code that can make the miners work forever something that results in a Denial of Service (DOS) Attack. In order to force all executions to terminate, each operation is tagged with an explicit cost, called gas.
- In Ethereum, each transaction incurs a cost to the sender.
- Gas is the unit of all computation tasks in Ethereum. It reflects how much work an action or set of actions takes to perform.

Ethereum & Gas

- Every operation performed by a transaction or contract, in Ethereum, costs a certain amount of gas, e.g. 2+2 costs less than Hash(2).
- If the value of “Gas Limit“ in a transaction is higher than the computation requires, the difference will be refunded to the transaction sender.

Ethereum Smart Contract

- An ethereum smart contract is an account with code.
- It cannot invoke itself.
- It must be invoked by an (externally owned or contract) account.

Ethereum Smart Contract

- All miners:
    - process and verify all incoming transactions.
    - run the called contracts.
    - perform consensus on the new state using the blockchain.
- That is why the execution of smart contract and its state is said to be secure.

Smart Contract Programming Language

- Solidity.
  - The primary language for writing smart contracts.
  - Object oriented programming Language.
  - Looks like Javascript.
- Serpent.
- LLL.
- Mutant.

* They are compiled into byte code before being deployed to the blockchain.

Solidity

- It is a high level language whose syntax is similar to JavaScript.
- It is an object oriented language and supports inheritance.
- There are special variable and functions in Solidity mainly used to get information about the blockchain. For instance:
    - msg.sender: it provides the message sender's address.
    - msg.value: it provides the number of Ether/Wei sent with the message.
    - now: current/head block timestamp.
- An easy to use Solidity compiler for beginners: remix. It is an online compiler to write, compile and debug smart contracts. It can be found here: remix.ethereum.org. Remember to set Environment option to JavaScript VM.
- Note that the caller/user of a smart contract needs to know the contract specifications (e.g. what functions each contract has, the functions name, and what arguments/parameters they take).

Smart Contract- Solidity

- Types in Solidity:
  - bool: true and false.
  - Integers: int (int8,…,int256) and uint (uint8,…,uint256).
  - address: has two members:
    - balance
    - send
  - byte arrays: bytes1,…, bytes32, bytes.
  - dynamic arrays.
  - string.
  - enum.
  - struct.
  - mapping (KeyType => ValueType).

Smart Contract Maps

- A useful type in Solidity is the mapping:

```
mapping (keyType => ValueType) name;

e.g. mapping (address => uint) reputation;
```

- It is like a hash table:

1. Store a value using a key: `reputation [0x111] = 5;`

   5 will be assigned to it.

2. Later, given the key, you can retrieve the value `var val = reputation [0x111];`

Smart Contract
Solidity

- The language supports big integers (256-bit). For example, we can have:

uint x = 1000000000000000000000000000009;

- Type deduction is supported by Solidity compiler, so instead of writing

string x=“Hi everyone”;

you can write

var x=“Hi everyone”;

- What is missing? Floating data type.
- Solidity does not support floating data type. So, in Solidity we cannot write

float x = 2.3899;

Ethereum Virtual Machine (EVM)

- EVM is the runtime environment for smart contracts in Ethereum. Every node in the Ethereum network runs EVM.
- It is a tailor-made compiler to compile smart contracts and resist DOS attacks (that can exploit the features of Turing-complete language).
- Contracts written in a high-level language (e.g. Solidity) are compiled into byte-code using the EVM compiler and uploaded on the blockchain.
- Application Binary Interface (ABI): It is an interface between modules of high-level contract program (i.e. Solidity) and low-level EVM byte-code. It allows us to tell to EVM which function in the contract we want to invoke.

Interacting with a Deployed Contract

- To interact with a contract deployed on the blockchain, we can either use ready-to-use user interfaces (UI), e.g. www.myetherwallet.com, or design our own UI.
- If we want to develop a web-based UI, we can use web3.js: a library that provides a collection of modules and specific functionalities for the Ethereum ecosystem.
- Web3 contains web3.eth, that includes methods to interact with the Ethereum network and blockchain.
- For instance, we can make an instance of a contract (steps 1 and 2) and then call the contract method (step 3):
    1. var myContractInstance= web3.eth.contract (contract_abi);
    2. myContractInstance.options.address = 0x…; (20 byte address)
    3. myContractInstance.*contractMethod*.call (); or myContractInstance.*contractMethod*.sendTransaction ();

Contract Instance Options

- `address` - `String`: The address where the contract is deployed.
- `jsonInterface` - `Array`: The json interface of the contract.
- `data` - `String`: The byte code of the contract.
- `from` - `String`: The address transactions should be made from.
- `gasPrice` - `String`: The gas price in wei to use for transactions.
- `gas` - `Number`: The maximum gas provided for a transaction (gas limit).

Ethereum
Creating an Account and Address

- We can create an (externally owned) account and address in Ethereum by using one of the following methods:
    - Wallet, e.g. MetaMask, MyEtherWallet. The wallet produces the private key of the account. A password is used to encrypt the private key in local storage (either selected by the wallet or by the user).
    - Command line interface (e.g. geth) in Ethereum client node, e.g. geth new account. It also generates a private key and allows us to provide a password.
- On the other hand, when a transaction sends a contract to the chain, an account and address for the contract are generated by the network and provided to the sender.

Smart Contract Example 1

- How does a smart contract look like?
- Let’s design a contract that:
  - Holds its owner/creator address.
  - Assigns an arbitrary value to its state.
  - After it is deployed to the blockchain, each time a function of it is called and a value is passed to it, it doubles the value and returns the result.

Smart Contract Example 1

```
pragma solidity ^0.4.4;

contract Example1 {

    uint public variable;
    address public owner;
    function Example1 () {
        variable = 30;
        owner = msg.sender;
    }
    function double_it (uint value) returns (uint){
     var temp = value * 2;
     return temp;
    }
}
```

This function is called constructor. It is run only once when the contract is sent to the blockchain. It is not considered as a contract function and cannot be called, after the contract is created.

We can call double_it() from outside as: instance.double_it.call (5, {from:"0x11"}); "instance" is just an instance of the contract and is defined (using web3.eth) when we want to interact with.

This function doubles the value it receives and returns the result.

msg.sender here is the address of the account that sends the contract to the blockchain.

In this example, the contract function does not change the contract state, i.e. the value of "variable" is not changed by function: double_it ().

Smart Contract Example 2: Token System

- Token systems have a variety of applications (e.g. sub-currencies, secure unforgeable coupons, or reputation systems).
- Let's design a contract that:
    - Sets token rate (value of each token in Ether).
    - Allows people to buy tokens by paying Ether within a predefined period of time.
    - Maintains an unforgeable public token balance.

Smart Contract
Example 2: Token System

```
pragma solidity ^0.4.4;

contract Example2 {

mapping (address=> uint) public token_balances;
uint exchange_Rate;
uint public end;
uint validity_period;
function Example2 () {
     exchange_Rate = 2;
     validity_period = 10000;
     end = now + validity_period;
}
 modifier notExpired {
        require (now <= end);
        _;
     }
function buyToken () payable notExpired external {
     require (msg.value >= exchange_Rate);
     uint amount = (msg.value) / exchange_Rate;
     token_balances[msg.sender] += amount;
     }
}
```

Head block timestamp (sec)

require(): is a function for error handling. When the condition is not met, it will undo all changes made to the sate in the current call (but it will keep the gas).

the tag: payable allows the function to receive Ether; otherwise, it cannot receive Ether.

the tag: notExpired allows the function to accept calls only within a period of time. We can defined it by using a modifier.

msg.sender here is the address of the account that calls buyToken().

In this example, the contract function does change the contract state, i.e. the value of "token_balances" is updated by function: buyToken ().

Smart Contract Example 2: Token System

```
function buyToken () payable external {
    require (msg.value >= exchange_Rate);
    uint amount = (msg.value) / exchange_Rate;
    token_balances[msg.sender] += amount;
    }
}
```

Where and how are *msg.sender* and *msg.value* defined ?

They are defined in the transaction that invokes the function. Function *buyToken()* in a transaction is invoked as follows:

contractInstance.buyToken.sendTransaction ({from: “0x111”, value:10, gas: 4200000});

Note: we did not define these parameters, in our original function, but EVM allows us to include them in any function.

Smart Contract
Invoking Contract Functions

- A contract function can be invoked via:
    - **Call**: Originated from an externally owned account. It results in a local invocation of a contract function. Does not publish anything on the blockchain. It does not change the contract state. Does not consume any Ether. Recall example 1 to see how we invoke “double_it()” using an API: call.
    - **Transaction.**
        - **External Transaction:** Originated from an externally owned account. It is signed and broadcasted to the network. Miners process it and, if valid, publish it on the blockchain. It can change the contract state. It consumes Ether. Recall example 2 to see how we can invoke “buyToken()”, using an API: sendTransaction.
        - **Internal Transaction:** Originated from a contract account. It is not signed and not broadcasted to the network. It is sent to another contract (or externally owned account). This kind of transaction is invoked by an external transaction and can change the state of called contract.

Smart Contract
Example 3: Events and Logs

```
pragma solidity ^0.4.4;

contract Example3 {

    uint public variable;

    function Test (uint val) returns (uint){
     var result = val * val;
      variable = result + 5;
      return result;
    }
}
```

This function both changes the state and returns a value.

How can we invoke it from the outside?

* If we do: “contractInstance.Test.call({from:”0x111”});” we would get the returned value, but the state will not be updated.
* If we do: contractInstance.Test.sendTransaction({from: “0x111”, value:10, gas: 4200000}); the state would be updated but we will not get the returned value.

In example 3, to get “result”, we can store it in the contract’s log and read it from there.

Smart Contract
Example 3: Events and Logs

When "event" is called, its arguments are stored in a log associated with the contract address and the log is stored in the blockchain.

```
pragma solidity ^0.4.4;

contract Example3{

    uint public variable;
    event CheckVal (uint val);

    function Test (uint val) returns (uint){
      var result = val * val;
      variable = result + 5;
      CheckVal (result);
    }
}
```

Storing values in a log is cheaper than storing it in a contract state variable.

However, a contract cannot read its own log and the log can only be read from the contract outside.

event is defined here.

We can read the contract's log, from the outside, as follows:

var eventx = myContractInstance.CheckVal ({from:"0x111"});

var res;

eventx.watch (function(error, result){ res = result.args.val.toNumber();});

Smart Contract Example 4

- Let’s design a contract that:
  - Sets token rate (value of each token in Ether).
  - Sets threshold in Ether.
  - Keeps track of the number of token it sold.
  - Allows people to buy and transfer tokens.
  - Maintains an unforgeable public token balance.
  - When the value of token sold exceeds the threshold, it automatically sends half of the value of sold tokens (in Ether) to an account address (e.g. “0x7dff4ad270b09c5847072796ee7e7c6c6ae209df”).

Smart Contract Example 4

```
pragma solidity ^0.4.4;

contract Example4 {

    mapping (address => uint) public token_balance;
    uint exchange_Rate;
    uint public threshold;
    uint public total_token_sold;
    address Ether_recipient;
    uint converter;
    function Example4 (){
        exchange_Rate = 2;
        threshold = 8;
        converter = 1000000000000000000;
        total_token_sold = 0;
        Ether_recipient = 0x7dff4ad270b09c5847072796ee7e7c6c6ae209df;
    }
     function buyToken () payable external {
        uint amount = (msg.value) / exchange_Rate;
        token_balance[msg.sender] += amount / converter;
        total_token_sold += amount / converter;
        if (total_token_sold * exchange_Rate > threshold) {
           Ether_recipient.transfer (converter * total_token_sold);
           total_token_sold = 0;}
     }
    function transfer_token (address recipient, uint amount){
        require (token_balance[msg.sender] >= amount);
        token_balance[msg.sender] -= amount;
        token_balance[recipient] += amount;
    }
}
```

This function, provided by Solidity, allows contract to transfer Ether to another (externally owned or contract) account.

we also can use:
converter = 1 ether;

Smart Contract Types

1. Token.
2. Authorization.
3. Time constraint.
4. Termination.
5. Oracle.

Smart Contract Applications

- Distributed Autonomous Organisations (DAO).
  - Smart contract acts as a virtual organisation with a predefined set of rules and actions/functions.
  - If the majority of it's members/stakeholders decide (via voting) to take certain action, the contract automatically does it and delivers the result.
- Decentralised Crowd Funding.
  - Central authority who receives the funding is substituted by a smart contract.
  - Donors pay smart contract and when the funding reaches a certain value, the funding is automatically delivered to the funding recipient.
- Robust and Fair Multi-party Computation.
  - Allows all parties to engage in a multi-party computation to get the output of computation; in case of an abort, they will be monetarily compensated.
- Efficient Verifiable Computation.
  - To incentivise certain computations of high cost and monetarily compensate them.

End of Lecture 03

- Next lecture
  - Incentives in Distributed Ledgers.