An Introduction to Smart Contracts

Aydin Abadi

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, Blockchain 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 puts a set of transactions in a block.
- The nodes compete with each other and when one wines it puts its block into a chain. Other (honest) nodes also put the winer's block into their chain. Next winer puts its block on top of the previous block in its chain.
- The block chain provides Bitcoin public ledger.
- So, the ledger is an ordered record of transactions.
- Each full node in the Bitcoin network independently stores the ledger.

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.

Smart Contracts in Bitcoin

- Multisignatures: at least m keys are required to authorise a Bitcoin transaction, where n>=m; applications include:

- 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.

- Business security: at least m stakeholders of a company must sign a transaction in order to transfer some bitcoin.

- Micropayment/payment channels: perform multiple transactions with low cost.

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

Accounts in Ethereum I

- 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.

Accounts in Ethereum II

	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	✓	✓

- A contract can send messages and create another contract(s)!
- Nonce contains the number of transactions made by an externally owned account or the number of contracts made by a contract account.
- A signed transaction is sent only by externally owned account and sending it will cost the sender.

State In Ethereum

- In Ethereum, the **state** of an account includes all accounts' information (e.g. state/storage, Ether balances, contracts code, etc).
- In Ethereum state can be **updated**.
- Miners keep track of accounts Ether balances and update states and balances.

Ethereum & Gas I

- 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.
- The total cost of a transaction in Ether is: Gas Limit * Gas Price.
- Gas limit: the maximum amount of gas a sender wants to spend on a transaction.
- Gas is the execution fee for each operation made on Ethereum. It reflects how much work an action or set of actions takes to perform.

Ethereum & Gas II

- Every operation performed by a transaction or contract, in Ethereum, costs a certain amount of gas, e.g. 2+3 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.

- In Bitcoin, we also have a similar concept: metering, the number of bytes in a transaction.

Ethereum Smart Contract

Some features I

- In Ethereum, a smart contract is an **account** with **code**.
- **A contract cannot invoke itself**.
- **It must be invoked by an** (externally owned or contract) **account**.

Ethereum Smart Contract

Some features II

- To **change/update** a contract's **state,** we have to make a **new transaction** and publish it to the network hosting the contract.

Ethereum Smart Contract

Some features III

- Smart contracts usually do **not** have **private states**!
- Has its own:
  - **pros**: more transparency
  - **cons**: no privacy

Ethereum Smart Contract

Some features IV

- **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.**

Ethereum Smart Contract Programming Language

- **Solidity**.
  - The primary language for writing smart contracts.
  - Object oriented programming Language.
    - Looks like Javascript.
- Serpent.
- LLL.
- Mutant.
- Smart contracts are compiled into byte code before being deployed to the blockchain.

Smart Contract & Solidity

- It is a high level language whose syntax is similar to JavaScript.
- However, it has some **new types** as well.
- There are special variable and functions in Solidity mainly used to get information about the blockchain. For instance:
  - **msg.sender**: returns a message sender's address.
  - **msg.value**: returns the number of Ether/Wei sent with a message.
  - **now**: returns 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.

Ethereum Virtual Machine (EVM)

- EVM is the runtime secure environment for smart contracts in Ethereum. Every nodes in the Ethereum network runs EVM.

- It is a tailor-made compiler to compile smart contracts and resists 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. **We need to have ABI's of a contract, in order to interact with the contract.**

Interacting with a Deployed Contract I

- To interact with a (deployed) contract, callers/users of the contract need to know its specifications, such as:
  - The name of each function of the contract.
  - What arguments/parameters each function takes.
  - The contract's address on the blockchain.
  - The contract's ABI.
- Recall that users need to send **enough Ether** for the contract **execution**.

Interacting with a Deployed Contract II

- 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 it ourselves.
- 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 contract (steps 1 and 2) and then call the contract method (step 3):

```
1.var myContract= web3.eth.contract (contract_abi);

2.var contractInstance= myContract.at ("Deployed_contract_address");

3.myContractInstance.contractMethod.call ();  or
  myContractInstance.contractMethod.sendTransaction ();
```

Ethereum
Creating Accounts & Addresses

- We can create an (externally owned) account and address in Ethereum by using one of the following methods:
  - Wallet/software, e.g. MetaMask, MyEtherWallet. The wallet also produces a private key. A password is used to encrypt the private key. MetaMask generates password for us, but we have to provide our own password to MyEtherWallet.
  - 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 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 & 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 & Solidity

- The language supports big integer, so you do not need to use any library for big integer. For example, we can have:

```
uint x = 10000000000000000000000000000009;
```

- 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;
```

Smart Contract & Mapping Data Type

- One of the useful types in Solidity is mapping:

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

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

- It is like a hash table:
- Store a value using a key:

```
reputation [0x111] = 5;
```

- Later, given the key, you can retrieve the value:

5 will be assigned to it.

```
var val = reputation [0x111];
```

- It is very useful structure for defining and maintaining **token balance**.
- If Solidity did provide mapping, then we had to use a library (another contract) or implement it.

Smart Contract Structure

```
pragma solidity ^0.4.4;

contract ContractName {

  uint public a;
  address public b;

  function constructor();
  function f(uint b);

}
```

Variables

Functions

Smart Contract: Example 1

- Let's see 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 writes it to the state.

Smart Contract Code. 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) external {
        variable = value * 2;
    }
}
```

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 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 changes the contract state, i.e. the value of "variable" is changed by function: double_it ().

Smart Contract Code.
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 Code

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

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 cost).

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().

Smart Contract Code 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} );
```

- We did not define these parameters, in our original function, but EVM allows us to include them in any function.

Smart Contract
Example 3: Weather Betting

- A form of conditional payments.
- Let's see a contract that:
  - Allows Alice and Bob deposit 2 Ether.
  - They bet on tomorrow weather: **if** the temperature was above 15°, **then** all the money goes to Alice. **Otherwise**, it goes to Bob.
  - We require a (trusted) 3rd party to provide weather information, i.e. 3rd **party verifier**.

Smart Contract Code
Example 3: Weather Betting

```
pragma solidity ^0.4.4;

contract Weather_bet {

    uint public amount;
    address public owner;
    address public Alice;
    address public Bob;
    address verifier;
    mapping (address => uint) public deposite_balance;

    constructor() public {
        amount = 2 ether;
        owner = msg.sender;
        Alice = 0x1890DD250108CFb2AFdA9b2583A0281a5172ee68;
        Bob = 0x03fcc7B438483414aEF817d0D761dd80E2036AB7;
        verifier = 0xAE355B827eaF295b5E5A629E36CeFacf8d8fadE8;
    }
    function deposit() payable external{
       require(msg.sender == Alice || msg.sender == Bob);
       require(msg.value >= amount);
       deposite_balance[msg.sender] = msg.value;
    }
    function verify(uint val) external {
      require(msg.sender == verifier);
      require(deposite_balance[Bob] * deposite_balance[Alice] >= 2*amount);
      if (val > 15){
         Alice.transfer(amount*2);
      }
      else{
          Bob.transfer(amount*2);
      }
      deposite_balance[Alice] = deposite_balance[Bob] = 0;
    }
}
```

This is another way of defining a constructor

Allows each party to send some Ether to the contract

If the temperature is greater than 15 **then** pay Alice

Otherwise pay Bob

We could define **time tags** for verify() and deposit() functions.

Feeding Real World info. To Smart Contracts I

- In Example 3, an external verifier has to verify occurrence of an event in the real world! e.g. the temperature was beyond/ below 15°.
- What if the verifier is **not available**, i.e. single point of failure?
  - Then, the money would be **locked** forever in the contract!
- Does the verifier always **tell the truth**, i.e. trust issue?
  - If not, even though the contract is correct, then the result would be **unfair**.

Feeding Real World info. To Smart Contracts II

- To address the availability issue:
  - **Use a threshold scheme**: define a set of verifiers (e.g. n verifiers) where a subset of them is required to verify the event, (e.g. t verifiers, where t < n).
  - Define a **withdraw function**: a function, time tagged, that allows the parties to get their shares back if the verifier does not show up.

Feeding Real World info. To Smart Contracts III

- To address the trust issue:
  - Use a well-known **trusted hardware**. Now we have to trust the hardware producers.
  - Use the **threshold scheme**, again. Now we have to assume the majority of verifiers are trusted.

Example 4: Simple Programable Transparent Donation

- Let's design a contract that:
  - Sets token rate (value of each token in Ether).
  - Sets threshold.
  - Allows people/donors to buy tokens.
  - Keeps track of the number of token it sells via an unforgeable public token balance.
  - Donors can transfer tokens among themselves.
  - 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/ beneficiary (e.g. "0x7dff4ad270b09c5847072796ee7e7c6c6ae209df").

Smart Contract Code

Example 4:
Simple Programable Transparent Donation

```
pragma solidity ^0.4.4;

contract Prog_Donation{

    uint exchange_rate;
    uint threshold;
    uint public tokens_sold;
    address recipient;
    mapping (address => uint) public token_balance;
    constructor () public {
        exchange_rate = 4;
        threshold = 8;
        recipient = 0x03fcc7B438483414aEF817d0D761dd80E2036AB7;
    }
    function buyToken() payable external{
        uint amount = msg.value/exchange_rate;
        token_balance[msg.sender] += amount;
        tokens_sold += amount;
        if (tokens_sold * exchange_rate > threshold){
            recipient.transfer((tokens_sold * exchange_rate) /2);
            tokens_sold = 0; }
    }
    function transfer_token(address rec, uint amount) external{
        require(token_balance[msg.sender] >= amount);
        token_balance[msg.sender] -= amount;
        token_balance[rec] += amount;
    }
}
```

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

Smart Contract Code

Compiler Version

- The given examples and codes are compatible with Geth version: 1.6.7 or remix compiler version: v0.24.

- Make sure you always **remember** the **version** of the **compiler** you are working with for your smart contracts!

Smart Contract Applications 1

- **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 (e.g. via voting) to take a certain action, then the contract automatically does it and delivers the result.
- **Decentralised Crowd Funding**.
  - Central authority who receives the funding is substituted with a smart contract.
  - Donors pay smart contracts and when the funding reaches to certain value, the funding automatically delivered to the funding recipient.

Smart Contract Applications 2

- **Modern transparent and accountable charity that supports cooperative governance** (e.g. OxChain project).
  - Donors can **pay** the contract and **buy** stakes.
  - The contract supports **cooperative governance** in a sense that:
  - The charity can create different **project proposals** for funding.
  - The stakeholders **vote** and **decide** whether a project should receive the funding.
  - The funding would be **automatically sent** to the recipient if a project gets enough positive votes.

Smart Contract Applications 3

- **Modern transparent and accountable charity that supports cause-based donations** (e.g. [1]).
  - Donations are transferred to predetermined beneficiaries if *something* happens (e.g. Earthquake, flood).

Smart Contract Applications 4

- **Robust and Fair Multi-party Computation** (e.g. [2]).
  - Allows all parties in a multi-party computation to get the output of computation; otherwise, they will be monetary compensated via **deposit paradigm.**
- **Efficient Verifiable Computation** (e.g. [3])
  - To **disincentivized** malicious behavior and detect misbehaving parties with low cost.

Smart Contract Applications 5

Proofs of Storage:

- Problem:
  - Alice wants to store its big data in dropbox or Google drive server, and leave it there for years.
  - But, she wants to pay the server every month only if the server proves its data is intact.
- Solution:
  - A smart contract can hold Alice's money. The server every month proves to the contract that Alice's data is intact.
  - The contract checks the proofs and if it accepts them, it pays the server.
- Advantages:

Due to smart contracts availability

1- Alice does not need to be online to check the proofs every month.

Due to smart contracts integrity

2- Alice is guaranteed that she pays only if the server provides a correct result.

3- The server is guaranteed that it will get paid if it provides correct result.

References

[1] **OxChain Project**

[2] **Fair and Robust Multi-Party Computation using a Global Transaction Ledger**, Aggelos Kiayias and Hong-Sheng Zhou and Vassilis Zikas, 2015

[3] **Betrayal, Distrust, and Rationality: Smart Counter-Collusion Contracts for Verifiable Cloud Computing**, Changyu Dong et al., 2017.

[4] **SmartPool: Practical decentralised pooled mining**, Lio Luu et al., 2017.

[5] **Application of smart contracts in proof of storage:** https://github.com/AydinAbadi/CR-LP/blob/master/main.pdf

The End.