

smart contracts

spending the bitcoins of a utxo: the easy story

- this is similar to a challenge response protocol
- txin of a transaction tx provides...
 - public key whose hash should match the address in txout
 - **signature** of a string X
- X is a string derived from...
 - tx where signatures are omitted
 - the destination address contained in referred txout

the reality: the conditions for unlocking funds can vary

- **only one subject can spend**
- anyone can spend
- nobody can spend (logging)
- M-of-N subjects should agree to spend
- one subject can spend after a certain amount of funds are accumulated (e.g., for crowdfunding)
- one (or many) can spend after a certain time
- etc...
- a combination of the above

scripts

- **locking** script (a.k.a. scriptPubKey)
 - associated with txout
 - states conditions to spend the output (a “question”)
 - usually it specifies at least the (hash of) a public key
- **unlocking** script (a.k.a. scriptSig)
 - associated with txin
 - should «match» the conditions of the corresponding txout (the “answer”)
 - usually it contains a signature
- the output of the unlocking script (answer) is used as input for the locking script (question)
 - essentially: (1) exec the unlocking script (2) keep the stack and exec the locking script (3) success if top of the stack is not zero and no operation failed
- executed as part of consensus checks

the bitcoin scripting language

- proprietary
- simple
- stack-based
- **no state**
- same execution on all nodes
- no iteration instructions
 - Turing incomplete

the bitcoin scripting language

- read and executed from left to right
- **constants** push themselves onto the stack
- arithmetic: ADD, SUB, ...
- stack: DUP, DROP, ROT, 2DUP, ...
- flow: IF, ELSE, ENDIF, VERIFY, RETURN, ...
- crypto: **HASH160**, SHA1, **CHECKSIG**,
CHECKMULTISIG ...
- time: CHECKLOCKTIME,

<https://en.bitcoin.it/wiki/Script>

examples

- anyone-can-spend
 - unlock: (empty)
 - lock: TRUE
- provably-unspendable, just to store data
 - lock: RETURN <data max 80 bytes> (never considered an UTXO for efficiency)
- **pay-to-public-key-hash (P2PKH, the “standard” one)**
 - unlock: <sig> <pubKey>
 - lock: DUP HASH160 <pubKeyHash> EQUALVERIFY CHECKSIG
- A or B can spend
 - unlock for A: <sig> <ApubKey> <1>
 - unlock for B: <sig> <BpubKey> <0>
 - lock: IF DUP HASH160 <ApubKeyHash>
ELSE DUP HASH160 <BpubKeyHash> ENDIF
EQUALVERIFY CHECKSIG
- freezing funds until a time in the future
 - unlock: <sig> <pubKey>
 - lock: <expiry time> CHECKLOCKTIMEVERIFY DROP
DUP HASH160 <pubKeyHash> EQUALVERIFY CHECKSIG

smart contracts

- each one of these scripts is called *smart contract*
 - it is not a “legal contract”, it is just a script!
 - they may realize/support legal contracts
 - it might be recognized as a contract, if parties agree that “code is law”, since the execution is checked by consensus
 - but in Italy they do are recognized as legal contracts by Legge 11 feb. 2019 n.12
- enable to use the bitcoin blockchain for other purposes:
 - colored coins
 - *tokens*, that are distinct from bitcoin, whose transactions are recorded in the bitcoin blockchain
 - obsoleted by the rising of the transaction fees
 - record of transaction for generic assets
 - settlement of off-chain transactions
 - so called “payment channels”, see the Lightning Network

bitcoin for smart contracts: limits

- high fees
- limited expressiveness
 - Turing incomplete
- slow
- the blockchain records any “state change”
 - unhandy for software execution

Ethereum

- Ethereum is a DLT targeted to smart contracts

| | Bitcoin | Ethereum |
|-------------------------------|---|---|
| Turing completeness | NO | YES |
| persistent values for scripts | not supported, complex, just UTXO, usually need external code | contracts accounts can store variables , easy to retrieve |
| blockchain contains | just transactions | current status |
| language | simple stack based | high level language compiled to a bytecode for the <i>Ethereum Virtual Machine</i> |
| block time | 10 minutes | 20 seconds |
| consensus | PoW | PoW->PoS? |
| block size limit | 1MB | adjusted dynamically, no limit |

accounts

- in bitcoin the lock script states what should be provided to unlock funds
 - it is a feature of every UTXO
 - some standard scripts (P2PKH, 2-of-3, etc.)
 - potentially infinite kinds of UTXO
 - depending on the lock script
- in Ethereum we have just two kinds of accounts
 - Externally Owned Accounts (EOA)
 - contract accounts

contracts (accounts)

- each contract account is associated an software object
 - very much like a software object of OOP
- it has a **state**
 - persisted in the blockchain
- it has **operations**

operations

- an operation is executed within a transaction
- it can...
 - change the state of the object
 - take parameters
 - return values
- essentially they are the methods of the object/contract

accounts

| | EOA | contract accounts |
|-----------------------------------|--|---|
| associated private keys | yes | no |
| balance | yes | yes |
| other persistent values/variables | no | yes it also stores EVM bytecode |
| as a transaction sender... | <ul style="list-style-type: none">• can send ETH• can call operations on a contract | <ul style="list-style-type: none">• can send ETH• can call operations on another contract in the same calling transaction |
| as a transaction recipient... | <ul style="list-style-type: none">• can receive ETH | <ul style="list-style-type: none">• can receive ETH• always executes an operation (possibly the fallback one) |

transactions fields

- (sender address)
- recipient address
- value (exchanged ETH)
- data
- nonce (increasing, to avoid replay attack)
- gas price
- gas limit

- $\text{max fee} = \text{gas price} * \text{gas limit}$
 - actual fee depends on the executed code
 - if a tx runs “out of gas”, state changes are reverted, but fee is taken anyway

contract lifecycle

- written in a high-level language
- compiled to EVM bytecode
- deployed
 - transaction sent to special address 0x0 and bytecode as data
- operations are called on the contract
 - as part of transactions, which may update its state
- cannot be deleted, but the contract can destruct itself

a solidity example

- anyone can withdraw funds from this contract

```
1 // Our first contract is a faucet!
2 contract Faucet {
3
4     // Give out ether to anyone who asks
5     function withdraw(uint withdraw_amount) public {
6
7         // Limit withdrawal amount
8         require(withdraw_amount <= 10000000000000000000);
9
10        // Send the amount to the address that requested it
11        msg.sender.transfer(withdraw_amount);
12    }
13
14    // Accept any incoming amount
15    function () public payable {}
16
17 }
```

evolution

```
4 contract owned {
5   address owner;
6   // Contract constructor: set owner
7   constructor() {
8     owner = msg.sender;
9   }
10  // Access control modifier
11  modifier onlyOwner {
12    require(msg.sender == owner,
13           "Only the contract owner can call this function");
14  };
15 }
16 }
17
18 contract mortal is owned {
19   // Contract destructor
20   function destroy() public onlyOwner {
21     selfdestruct(owner);
22   }
23 }
24
25 contract Faucet is mortal {
26   event Withdrawal(address indexed to, uint amount);
27   event Deposit(address indexed from, uint amount);
28
29   // Give out ether to anyone who asks
30   function withdraw(uint withdraw_amount) public {
31     // Limit withdrawal amount
32     require(withdraw_amount <= 0.1 ether);
33     require(this.balance >= withdraw_amount,
34            "Insufficient balance in faucet for withdrawal request");
35     // Send the amount to the address that requested it
36     msg.sender.transfer(withdraw_amount);
37     emit Withdrawal(msg.sender, withdraw_amount);
38   }
39   // Accept any incoming amount
40   function () public payable {
41     emit Deposit(msg.sender, msg.value);
42   }
43 }
```

- state variables
- constructors
- inheritance
- custom modifiers
- assertions
- events

simple things might be complex

- for example, requiring a multisignature to unlock funds

libraries

- libraries can be imported in a project as included code...
- ...or from the blockchain!
 - ...if you trust it!

remix

- a basic web based editor, emulator, debugger
- <https://remix.ethereum.org>

The screenshot displays the Remix IDE interface. On the left, a code editor shows Solidity code for a 'Hello' contract. The code includes a pragma statement for Solidity version 0.4.8, a constructor, a 'setGreeting' function, and a 'greet' function. On the right, the compiler interface shows the Solidity version (0.4.8+commit.60cc1668), compilation options (Auto Compile checked), and a transaction interface for the 'Hello' contract. The transaction interface includes a 'Create' button, a 'Bytecode' field with a long hexadecimal string, an 'Interface' field with a JSON schema, and a 'Web3 deploy' field with a JavaScript snippet. The metadata location is also visible at the bottom.

```
1 pragma solidity ^0.4.8;
2
3 contract Hello {
4     // A string variable
5     string public greeting;
6
7     // Events that gets logged on the blockchain
8     event GreetingChanged(string _greeting);
9
10
11     // The function with the same name as the class is a constructor
12     function Hello(string _greeting) {
13         greeting = _greeting;
14     }
15
16     // Change the greeting message
17     function setGreeting(string _greeting) {
18         greeting = _greeting;
19
20         // Log an event that the greeting message has been updated
21         GreetingChanged(_greeting);
22     }
23
24     // Get the greeting message
25     function greet() constant returns (string _greeting) {
26         _greeting = greeting;
27     }
28 }
29
```

Solidity version: 0.4.8+commit.60cc1668.Emscripten.clang
Change to: 0.4.10-nightly.2017.3.3+commit.6bfd894f

Text Wrap Enable Optimization Auto Compile

Attach Transact Transact (Payable) Call

Hello 1403 bytes

At Address string _greeting

Bytecode: 6060604052346100005760405161057b38038061057b833981016040528

Interface: [{"constant":false,"inputs":[{"name":"_greeting","type":"string"}],"name":"set

Web3 deploy: var _greeting = /* var of type string here */ ;
var helloContract = web3.eth.contract([{"constant":fal
var hello = helloContract.new(
_greeting,
{
 from: web3.eth.accounts[0],
 data: '0x6060604052346100005760405161057b38038061
 gas: '4700000'
}, function (e, contract){
 console.log(e, contract);
 if (typeof contract.address !== 'undefined') {
 console.log('Contract mined! address: ' + con
 }
})

Metadata location: bzzr://a63d0b3449ebe3923dda93af66f138c1aef28f4a1d3a51f6c4f1c6326C

[Toggle Details](#)

contracts security

- contracts are usually not very long
- writing contracts is easy
- **writing secure contracts is difficult**
 - solidity/EVM semantic may be subtle
 - mistakes may cost a lot of money!

Atzei N. et al. **A survey of attacks on ethereum smart contracts.**
International Conference on Principles of Security and Trust 2017

references

- A. M. Antonopoulos – Mastering Bitcoin
- A. M. Antonopoulos, G. Wood - Mastering Ethereum

