

Weaknesses of blockchain applications

How to shoot yourself in the foot playing with the blockchain



Simone Bronzini
CTO @ Chainside



Summary

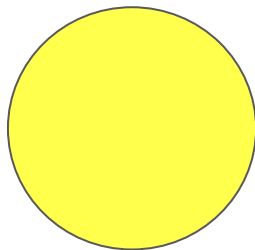
1. Blockchain recap:
 - a. Immutability
 - b. Consensus
 - c. Smart contracts
 - i. Turing completeness
 - ii. Turing completeness consequences
2. A bunch of horror stories

Blockchain Recap

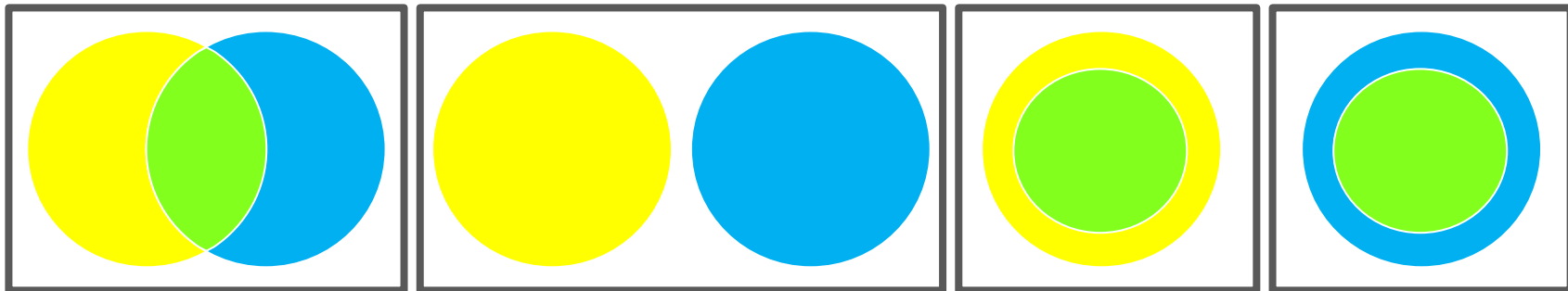
- The blockchain was created with the aim of deploying an electronic cash system without trusted third parties
- Using the Proof of Work, consensus on the state of the database is achieved, modifying past history is impractical
- Due to the large number of network participants, the possibility of modifying the rules is extremely limited and in some cases it could cause a consensus failure and a split of the network

Consensus rules

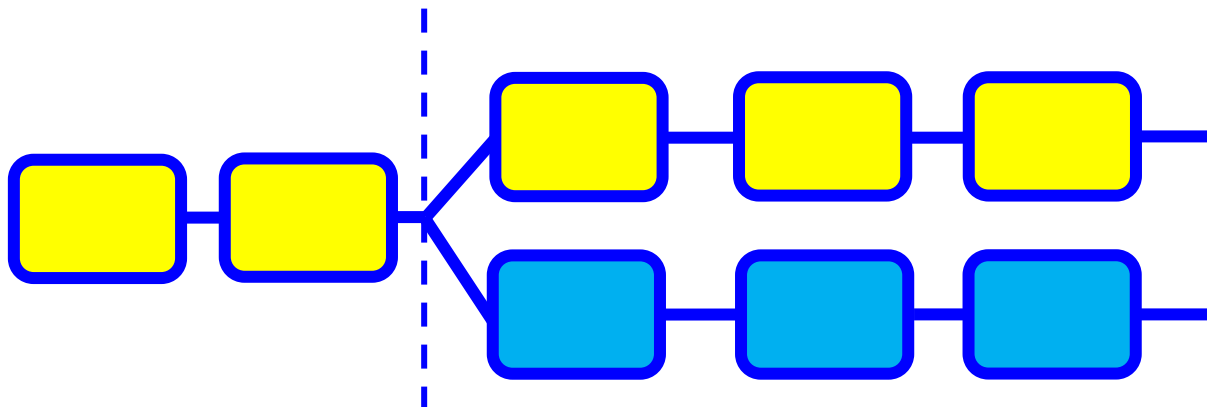
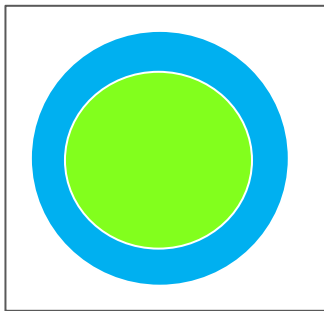
In any consensus-based system, nodes follow the same set of rules



What happens if nodes follow different rules?

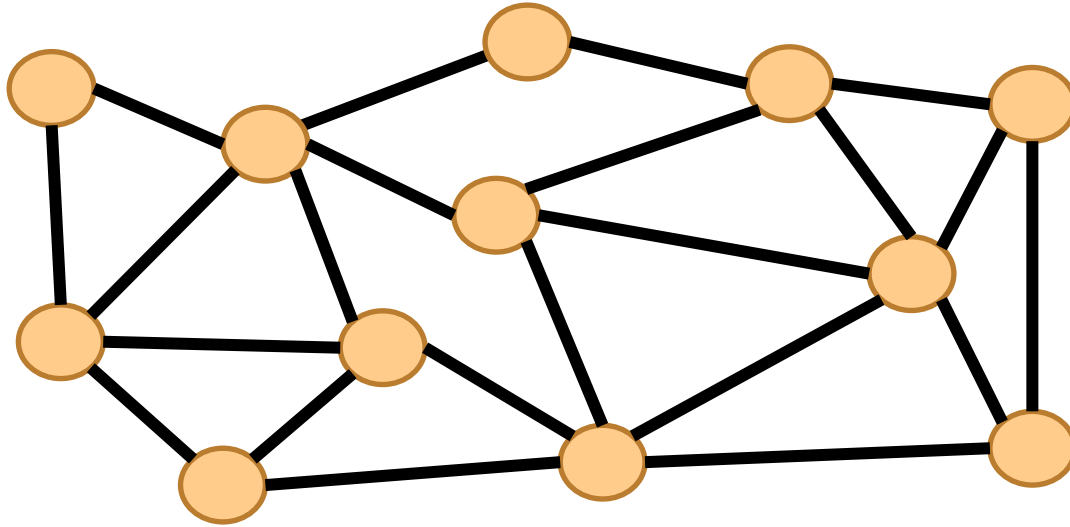


Hard fork

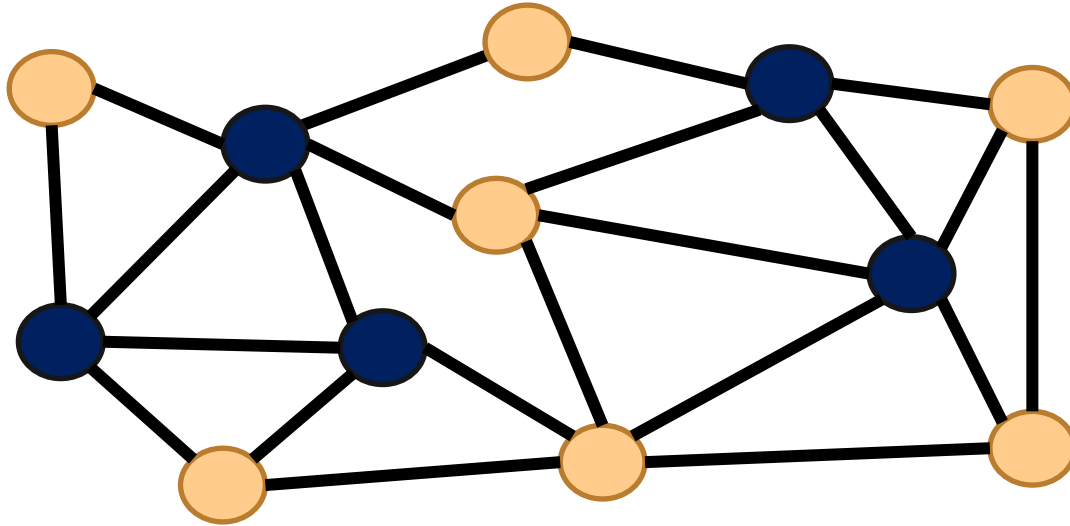


As soon as a blue block appears, there is no way to recover the split

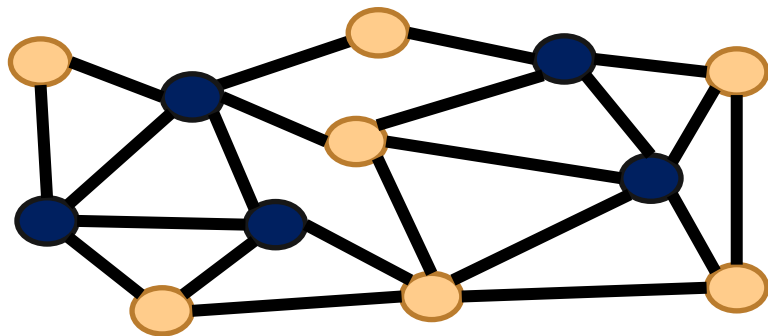
Consensus rules - transactions




Consensus rules - transactions



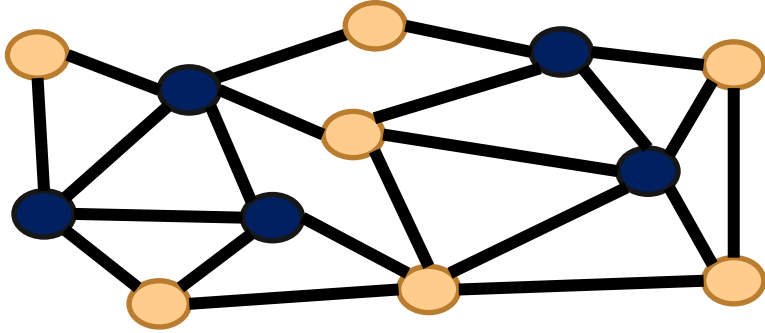
Consensus rules - transactions





Smart contracts accepted by yellow nodes

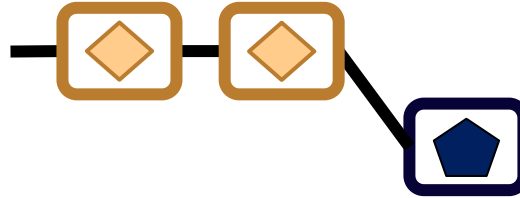
 Smart contracts accepted by blue nodes

Consensus rules - transactions

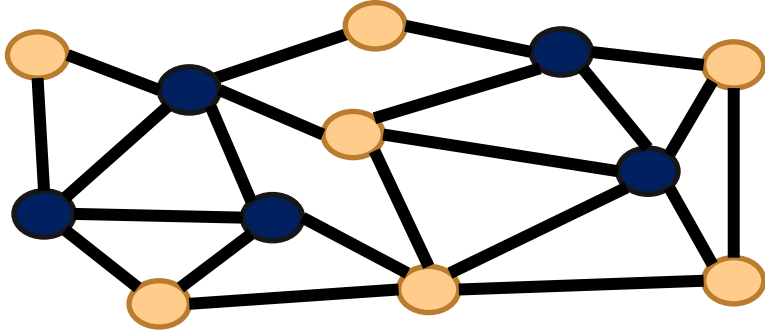



 Smart contracts accepted by yellow nodes


 Smart contracts accepted by blue nodes

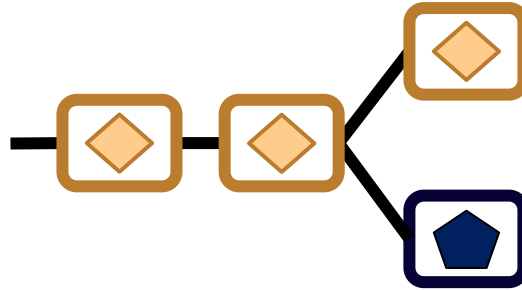


Consensus rules - transactions

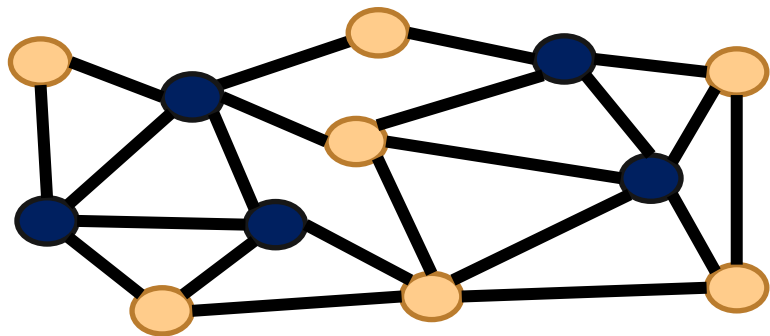



 Smart contracts accepted by yellow nodes


 Smart contracts accepted by blue nodes

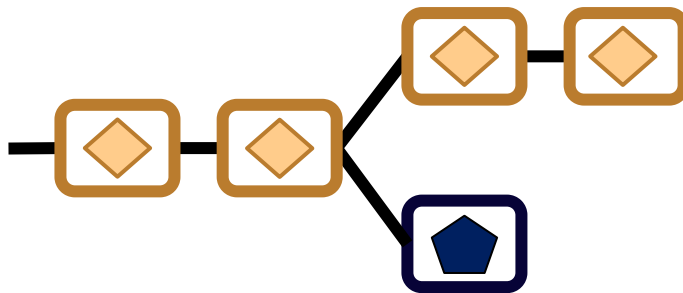


Consensus rules - transactions

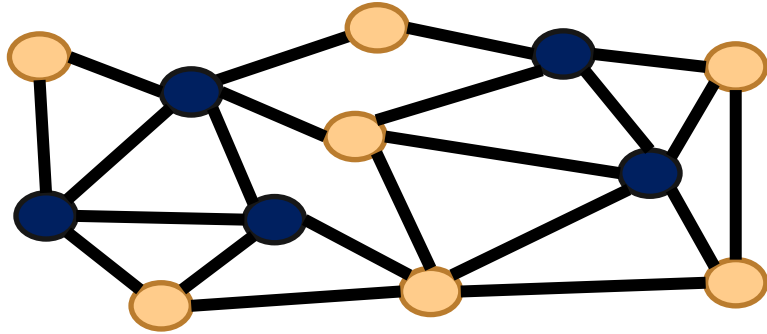



 Smart contracts accepted by yellow nodes


 Smart contracts accepted by blue nodes

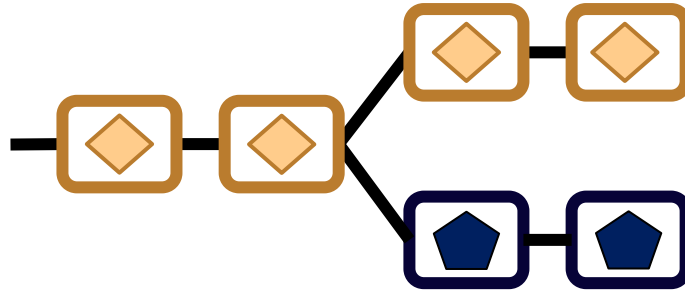


Consensus rules - transactions

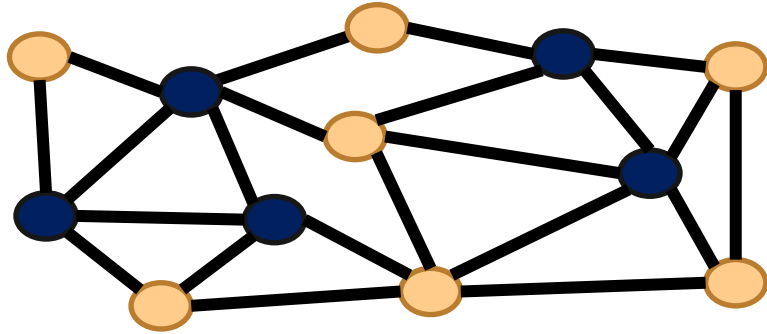



 Smart contracts accepted by yellow nodes


 Smart contracts accepted by blue nodes

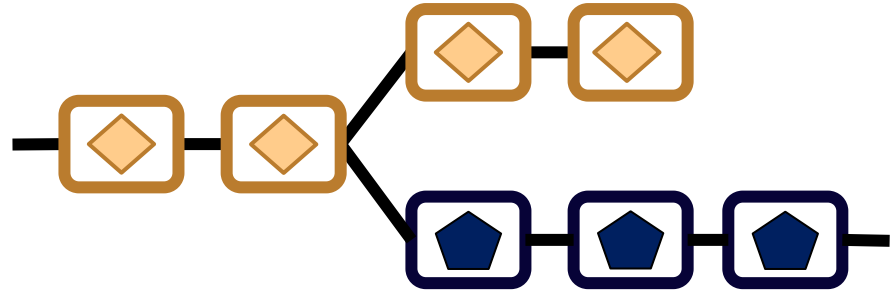


Consensus rules - transactions

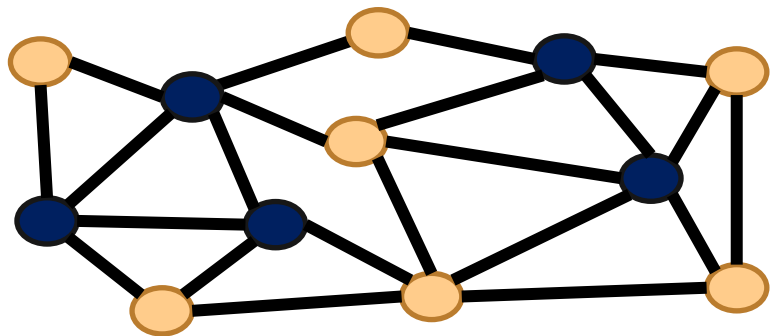



 Smart contracts accepted by yellow nodes


 Smart contracts accepted by blue nodes

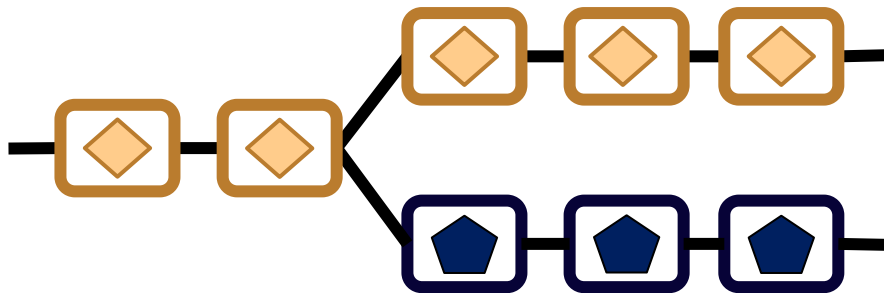


Consensus rules - transactions



 Smart contracts accepted by yellow nodes

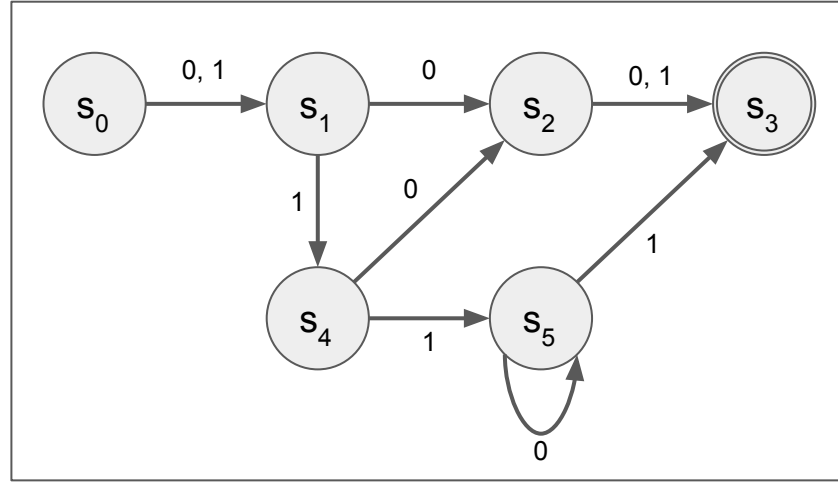
 Smart contracts accepted by blue nodes



Smart contracts

- Code which defines the conditions to spend a transaction
- Are executed by all nodes in the network when validating transactions
- Can express pretty complex redeem conditions
- Examples:
 - Only the owner of private key matching public key K_1 can redeem
 - Only the owners of private keys matching public keys K_1 and K_2 can redeem
 - Either the owners of private keys matching public keys K_1 and K_2 can redeem right away, or the owner of the private key matching public key K_3 can redeem after 2 months from now
 - Providing the solution to the equation $3 \cdot x + 2 = 38$ anyone can redeem

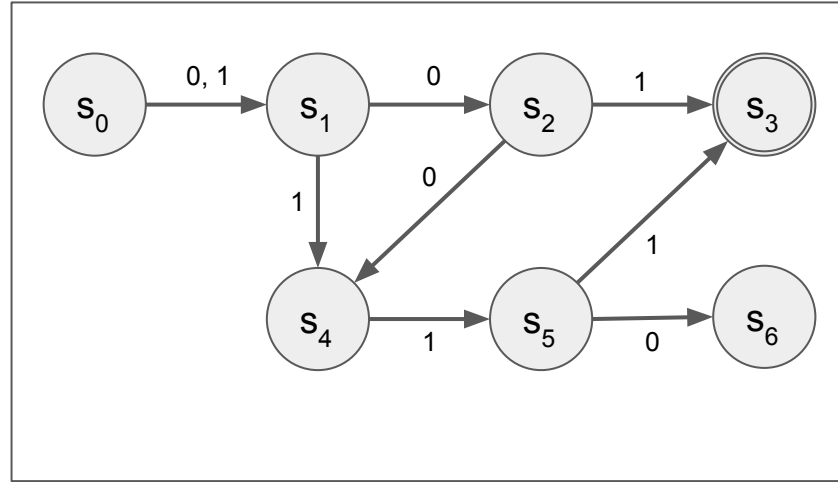
Turing completeness - General concept



Turing completeness - In real life

```
1
2 a = int(input())
3
4 if a > 10:
5
6     while a < 100:
7         a = a + 1
8         print(a)
9
10 else:
11     print(a + 10)
12
```

Turing completeness - Bitcoin formalisation



Turing completeness - Bitcoin in real life

```
1 IF
2     HASH160 <hashed_data> EQUAL
3
4     <pubk1> CHECKSIG
5
6 ELSE
7
8     <timestamp> CHECKLOCKTIMEVERIFY
9
10    2 <pubk2> <pubk3> <pubk4> 3 CHECKMULTISIG
11
12 ENDIF
```

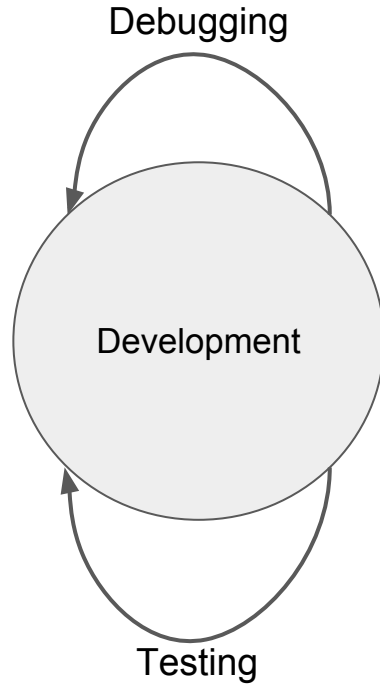
Turing completeness - Problems

find the bug!

```
sorted_currencies = sorted(currencies)
for gateway in [g.replace('/', '-') for g in Account.known_users]:
    for field in ('from_users', 'from_gateways'):
        with open('trust_by_gateway/%s/%s' % (field, gateway), 'w') as outfile:
            outfile.write('time ')
            for curr in sorted_currencies:
                outfile.write('{0}_amount {0}_volume '.format(curr))
            outfile.write('overall_amount overall_volume\n')
for ledg in ledger_gen(START, LAST+1, STEP, True):
    time = ledg.close_time(True)
    output = {user: {'from_users': {c: 0.0 for c in currencies+['overall']}, 'from_gateways': {c: 0.0 for c in currencies+['overall']}}}
for user in Account.known_users:
    received = {user: {'from_users': {c: set() for c in currencies+['overall']}, 'from_gateways': {c: set() for c in currencies+['overall']}}}
    for user in Account.known_users:
        for edge in ledg.trustlines(lambda x: x.dest.known and converter.convertible_in_time(x.amount.currency)):
            name = edge.dest.get_name().replace('/', '-')
            currency = edge.amount.currency
            if edge.orig.known:
                field = 'from_gateways'
            else:
                field = 'from_users'
            converted_value = converter.convert_in_time(edge.amount.value, currency, conversion, time)
            if currency in currencies:
                output[name][field][currency] += converted_value
                received[name][field][currency].add(edge.orig.get_name())
            output[name][field]['overall'] += converted_value
            received[name][field]['overall'].add(edge.orig.get_name())
for gateway in output:
    for field in ('from_gateways', 'from_users'):
        with open('trust_by_gateway/%s/%s' % (field, gateway), 'a') as outfile:
            outfile.write("%s " % (time.strftime("%Y%m%d%H%M%S"),))
            outfile.write(' '.join(["%s %d" % (str(output[gateway][field][sorted_currencies[i]]), len(received[gateway][field][sorted_currencies[i]])) for i in xrange(len(sorted_currencies))]))
            outfile.write(" %s %d\n" % (str(output[gateway][field]['overall']), len(received[gateway][field]['overall'])))
```

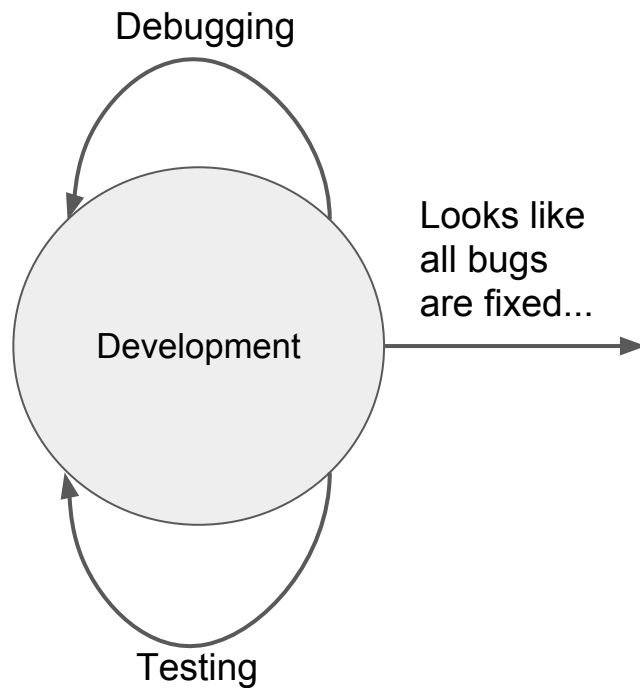
Turing completeness - Problems

- Classical software development process



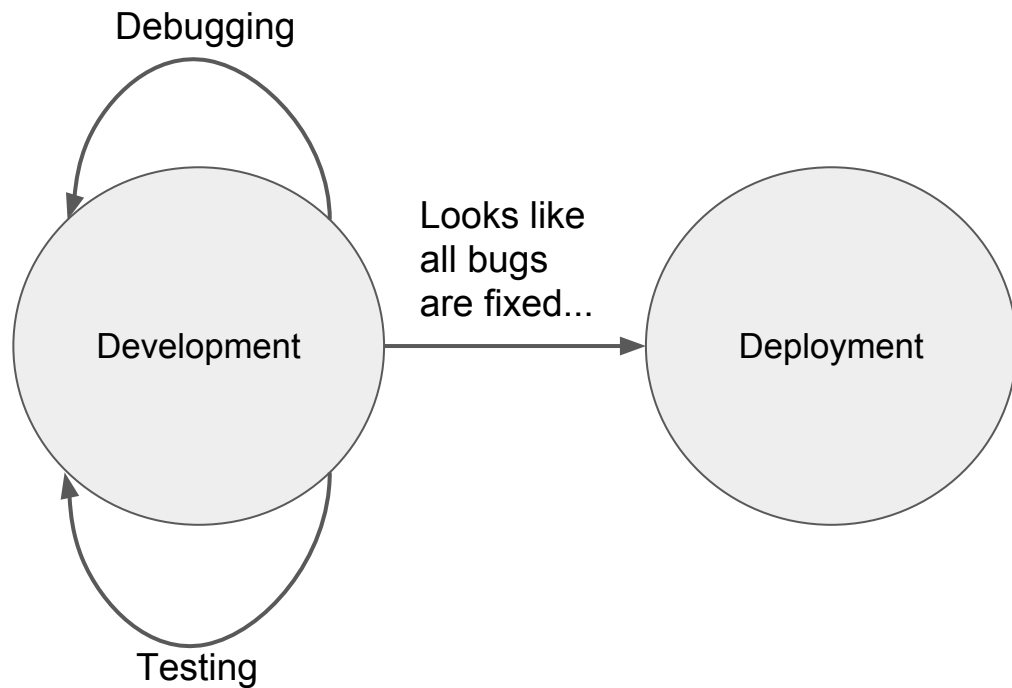
Turing completeness - Problems

- Classical software development process



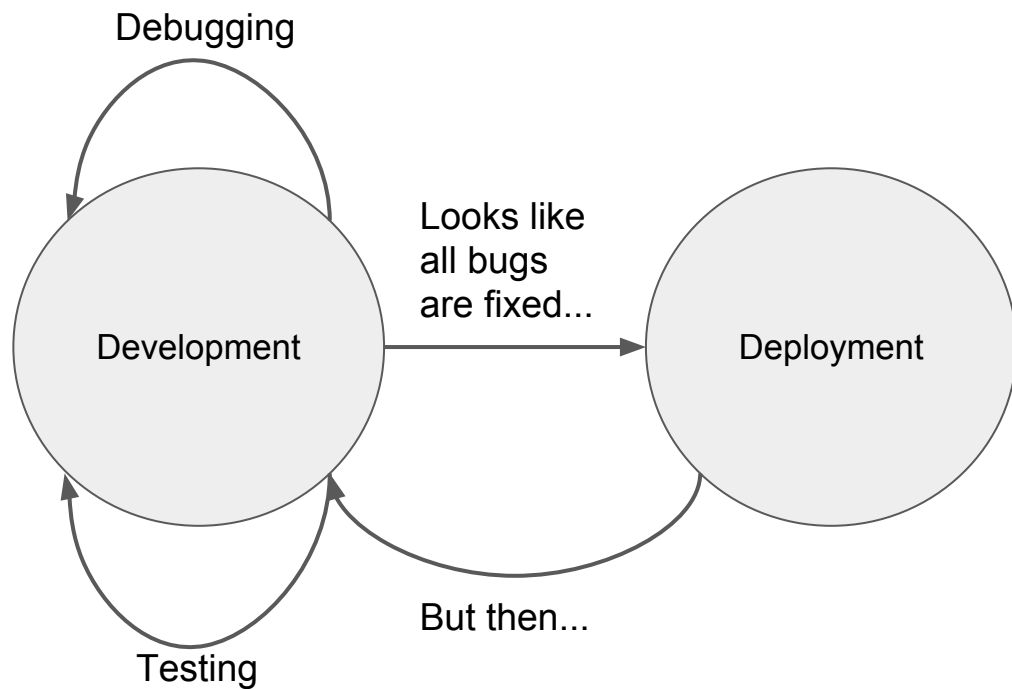
Turing completeness - Problems

- Classical software development process



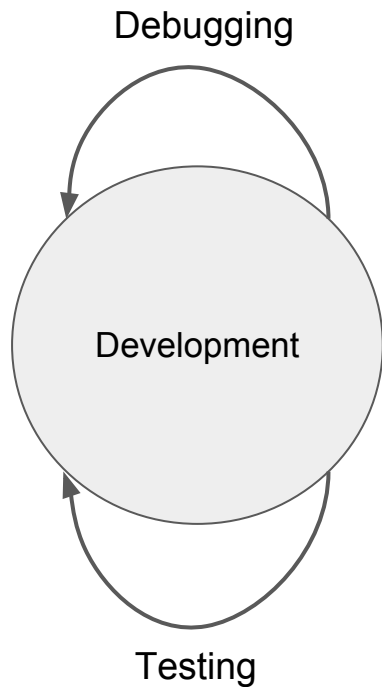
Turing completeness - Problems

- Classical software development process



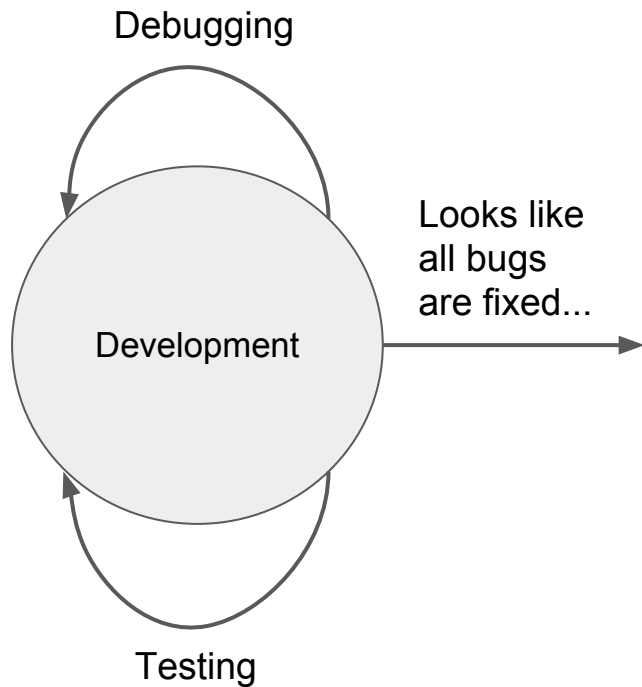
Turing completeness & immutability

- Software development process on smart contracts



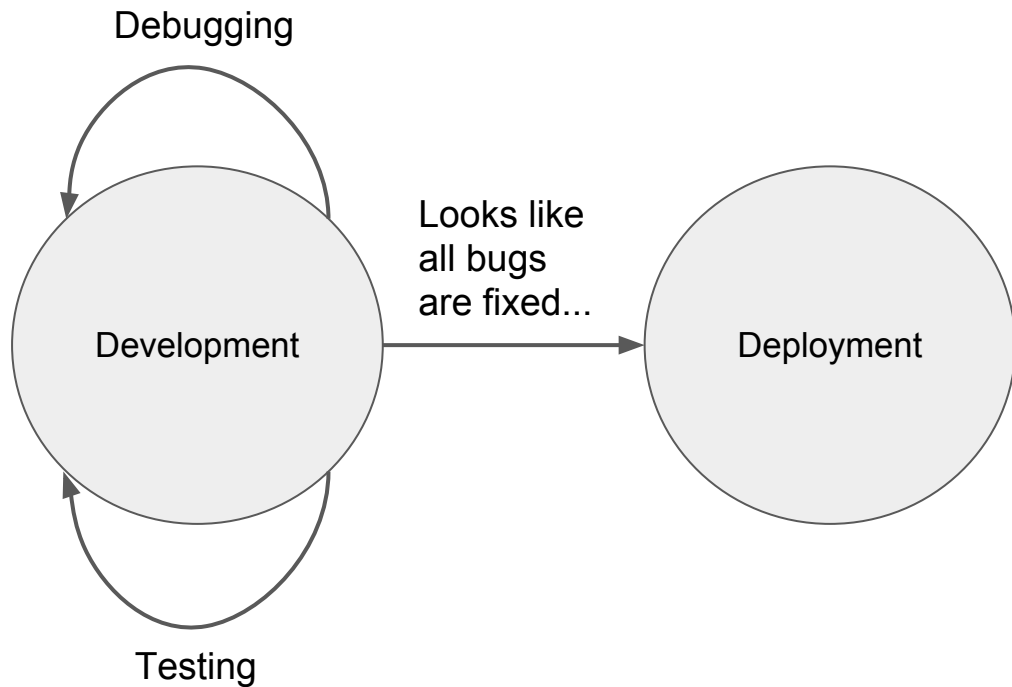
Turing completeness & immutability

- Software development process on smart contracts



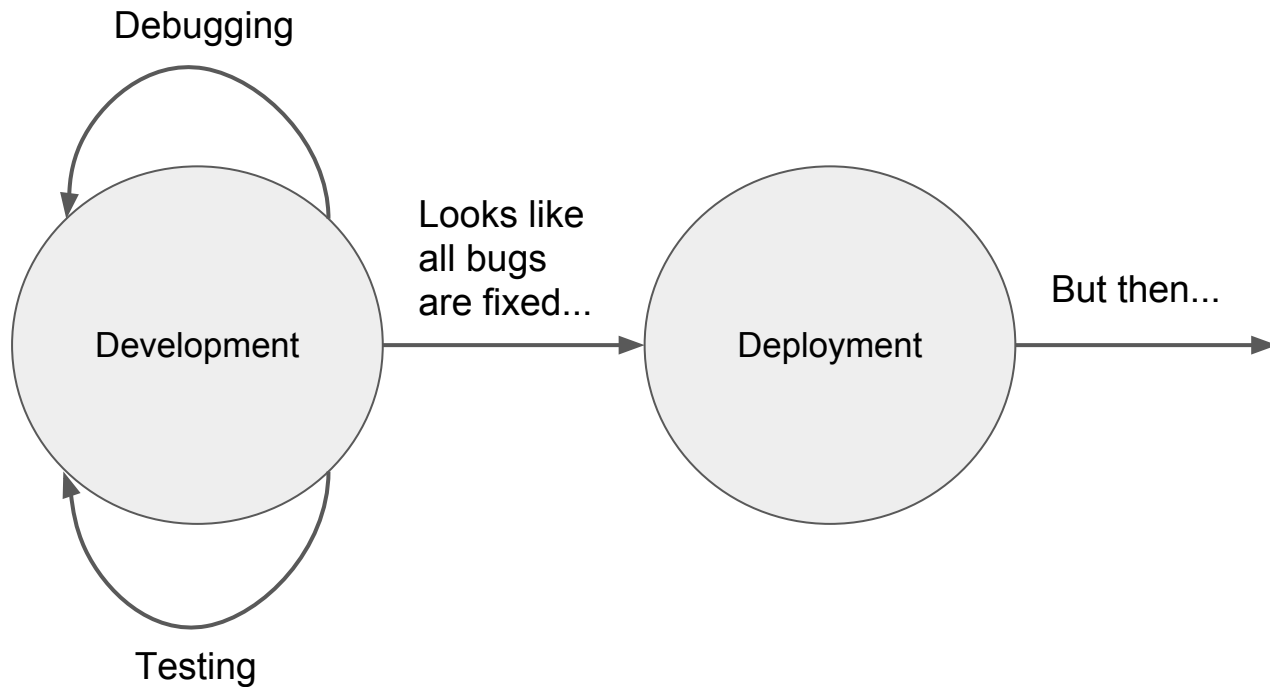
Turing completeness & immutability

- Software development process on smart contracts



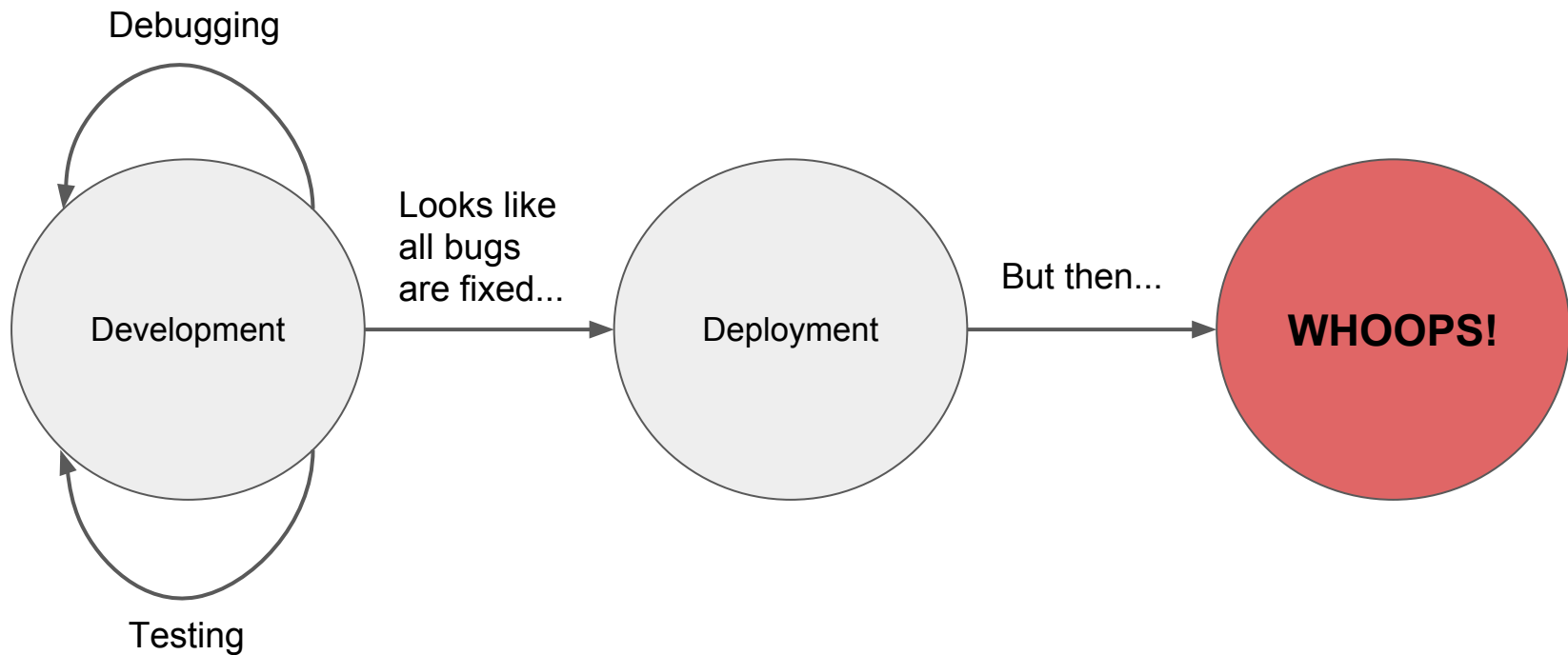
Turing completeness & immutability

- Software development process on smart contracts

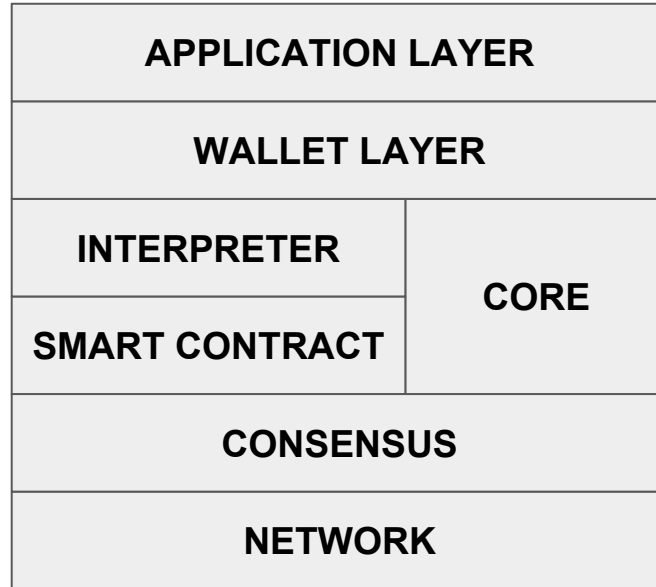


Turing completeness & immutability

- Software development process on smart contracts



Weakness surfaces



Now, a collection of horror stories...

Mt Gox

- In Bitcoin transactions are identified by their TXID, but unfortunately it was possible for third parties to change the TXID of a transaction without making it invalid
- MtGox used the TXID to track deposits and withdrawals of its users
- Attacker could request a withdraw, malleate the withdrawing transaction and contact MtGox support claiming that the transaction didn't occur
- MtGox would then send a second withdraw transaction to the user

APPLICATION LAYER	
WALLET LAYER	
INTERPRETER	CORE
SMART CONTRACT	
CONSENSUS	
NETWORK	

Bitcoin - Value overflow incident

- On August 15 2010, it was discovered that block 74638 contained a transaction that created 184,467,440,737.09551616 bitcoins
- This happened because the code used for checking transactions before including them in a block didn't account for the case of outputs so large that they overflowed when summed
- A new version of the client was published, containing a soft forking change to the consensus rules that rejected output value overflow transactions
- The blockchain forked until the “good” chain eventually became the longest

APPLICATION LAYER	
WALLET LAYER	
INTERPRETER	CORE
SMART CONTRACT	
CONSENSUS	
NETWORK	

Bitcoin - Multisig evaluation

- In its first version, bitcoin operation `OP_CHECKMULTISIG` expected $N+1$ values as an input to validate N signatures:

```
2 <pubkeyA> <pubkeyB> <pubkeyC> 3 CHECKMULTISIG
```

Should be spent with:

```
<sigA> <sigB>
```

- This meant: providing the correct number of signatures had the interpreter crash
- The solution was actually easy (just provide one more dummy parameter):

```
<null> <sigA> <sigB>
```

- To this day we still add a dummy parameter to multisig scripts

APPLICATION LAYER	
WALLET LAYER	
INTERPRETER	CORE
SMART CONTRACT	
CONSENSUS	
NETWORK	

Bitcoin - 24 blocks long fork

- When version 0.8 was released, it used a new database engine: Level DB
- The previously used database was unable to process blocks that required more than 10.000 locks on DB rows
- As soon as such block was released, clients still using version < 0.8 saw that block as invalid, new clients saw it as valid
- A 24 blocks long fork resulted which was resolved when all nodes **downgraded** their software

APPLICATION LAYER	
WALLET LAYER	
INTERPRETER	CORE
SMART CONTRACT	
CONSENSUS	
NETWORK	

Ethereum - Parity 1

- Parity provided a multisig wallet smart contract which was probably overcomplicated (hundreds of lines of code vs. `2 <pubkey1> <pubkey2>`
`2 CHECKMULTISIG`)
- In the contract there was a bug that allowed third parties to change the ownership of the contract, stealing the funds inside
- \$31M dollars were stolen, and other \$100M were taken by white hat hackers and later given back to the owners

APPLICATION LAYER	
WALLET LAYER	
INTERPRETER	CORE
SMART CONTRACT	
CONSENSUS	
NETWORK	

Ethereum - Parity 1

```
contract WalletLibrary {
    address owner;

    // called by constructor
    function initWallet(address _owner) {
        owner = _owner;
        // ... more setup ...
    }

    function changeOwner(address _new_owner) external {
        if (msg.sender == owner) {
            owner = _new_owner;
        }
    }

    function () payable {
        // ... receive money, log events, ...
    }

    function withdraw(uint amount) external returns (bool success) {
        if (msg.sender == owner) {
            return owner.send(amount);
        } else {
            return false;
        }
    }
}
```

```
contract Wallet {
    address _walletLibrary;
    address owner;

    function Wallet(address _owner) {
        // replace the following line with "_walletLibrary = new WalletLibrary();"
        // if you want to try to exploit this contract in Remix.
        _walletLibrary = <address of pre-deployed WalletLibrary>;
        _walletLibrary.delegatecall(bytes4(sha3("initWallet(address)")), _owner);
    }

    function withdraw(uint amount) returns (bool success) {
        return _walletLibrary.delegatecall(bytes4(sha3("withdraw(uint)")), amount);
    }

    // fallback function gets called if no other function matches call
    function () payable {
        _walletLibrary.delegatecall(msg.data);
    }
}
```

APPLICATION LAYER	
WALLET LAYER	
INTERPRETER	CORE
SMART CONTRACT	
CONSENSUS	
NETWORK	

Ethereum - Parity 2

- Parity multisig wallet contract refers to a library contract in the wallet logic
- The library contract however was actually an unutilized wallet contract, and it could be initialized by anyone
- Somebody did it, and as the new owner of the contract was able to kill it, freezing the funds of all the other wallets depending on it

APPLICATION LAYER	
WALLET LAYER	
INTERPRETER	CORE
SMART CONTRACT	
CONSENSUS	
NETWORK	



devops199 @devops199

07:49

will i get arrested for this? 🤔

0x642483b7936b505dbe2e735cc140f29ddfdbb3f3e39efa549707d98e0298de6e4dea99

a30da4872869e36322c9dfcdd06d9aa389e746dc6b92fdc0414e0b18421b

0xae7168deb525862f4fee37d987a971b385b96952



Tienus @Tienus

07:51

@devops199 you are the one that called the kill tx?



devops199 @devops199

07:51

yes

i'm eth newbie..just learning



qx133 @qx133

07:52

you are famous now haha



devops199 @devops199

07:52

sending kill() destroy() to random contracts

you can see my history

😞((((((((((((((((((((((((((((((((



Xavier @n3xco

07:52

can't make an omelet without breaking some eggs

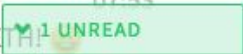
i guess



Tienus @Tienus

07:53

Let me know next time you decide to kill some contracts so I can sell my E



Ethereum - DAO 1

- The DAO was a complex Ethereum-based smart contract that was supposed to act as a decentralised investment fund and collected over \$150M
- Due to a bug in its code, it was possible for an attacker to withdraw more money than deposited, draining funds out of the DAO
- The attacker stole \$50M

APPLICATION LAYER	
WALLET LAYER	
INTERPRETER	CORE
SMART CONTRACT	
CONSENSUS	
NETWORK	

Ethereum - DAO 2

- To fix the issue, miners agreed to operate a hard fork and give the money back to the DAO investors
- As a result of the hard fork, the Ethereum network split and still today there are two incompatible versions of the chain

APPLICATION LAYER	
WALLET LAYER	
INTERPRETER	CORE
SMART CONTRACT	
CONSENSUS	
NETWORK	

Bitcoin - Single transaction double spending

- September 2018, a bug was found that was introduced in PR #9049 (merged in November 2016)
- The bug allowed two attacks:
 - A miner could DoS a large part of the network (nodes would crash) (November 2016)
 - A miner could generate new BTCs out of thin air by spending the same output multiple times inside the same transaction (0.15.0, September 2017)
- It was fixed before anyone exploited it, apart from testnet

APPLICATION LAYER	
WALLET LAYER	
INTERPRETER	CORE
SMART CONTRACT	
CONSENSUS	
NETWORK	

Conclusions

- Blockchains are a very powerful tool, but:
 - Using them is very complex
 - Mistakes can cost millions
 - There is a huge need for experts
 - There is a huge lack of experts

If you feel like experimenting

We released an open source Python3 library to create complex Bitcoin scripts:

<https://github.com/chainside/btcpy>

OR

```
pip3 install chainside-btcpy
```

QUESTIONS?