Smart contracts

Mariano Ceccato

mariano.ceccato@univr.it

Outline

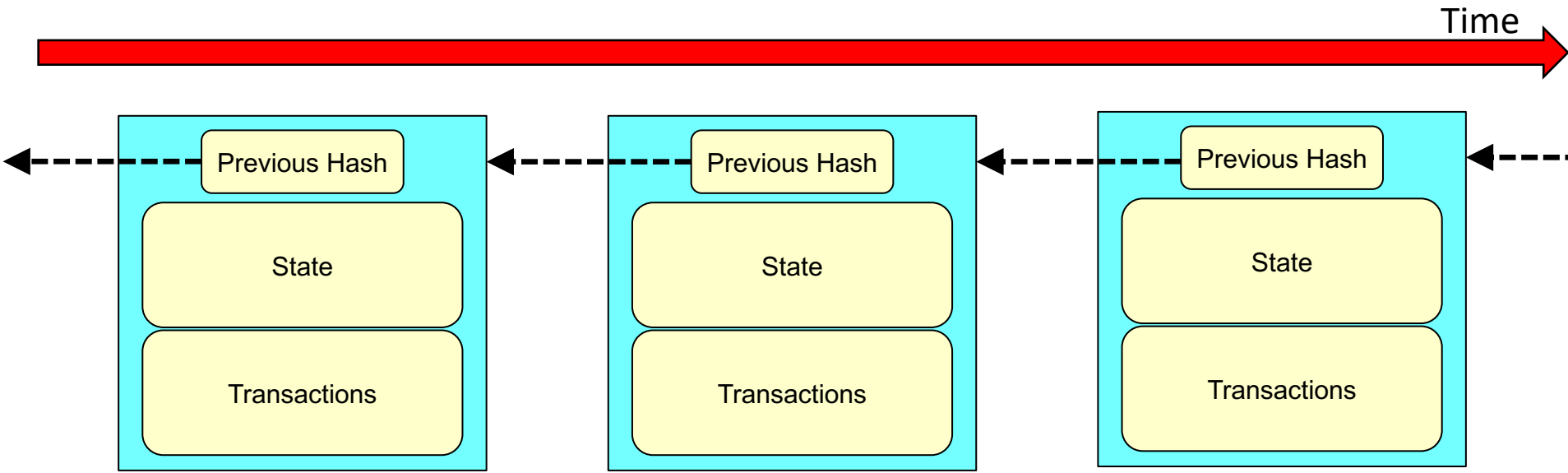
- Introduction to smart contracts
- Security bugs
- Experimental results
- Conclusions

Cryptocurrencies and blockchain

- **Blockchain**:
  - decentralised public ledger used for recording data and transactions
  - Shared
  - Immutable (write once) trusted
  - consensus protocol
  - no need for trusted party

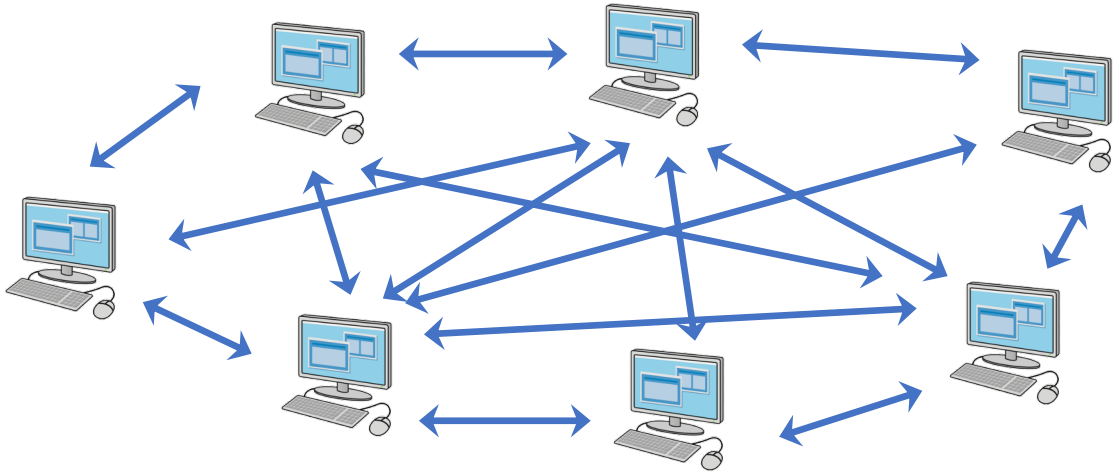
- **Cryptocurrencies**:
  - Use the blockchain to record transactions (payments)
  - e.g., Bitcoin, Ethereum

Chain of blocks in the distributed ledger



Consensus protocol

- In every epoch, each miner can propose a block of new transactions to update the blockchain
- A leader is elected probabilistically
- The leader broadcasts the proposed block to all miners
- All miners update the blockchain and include the new block



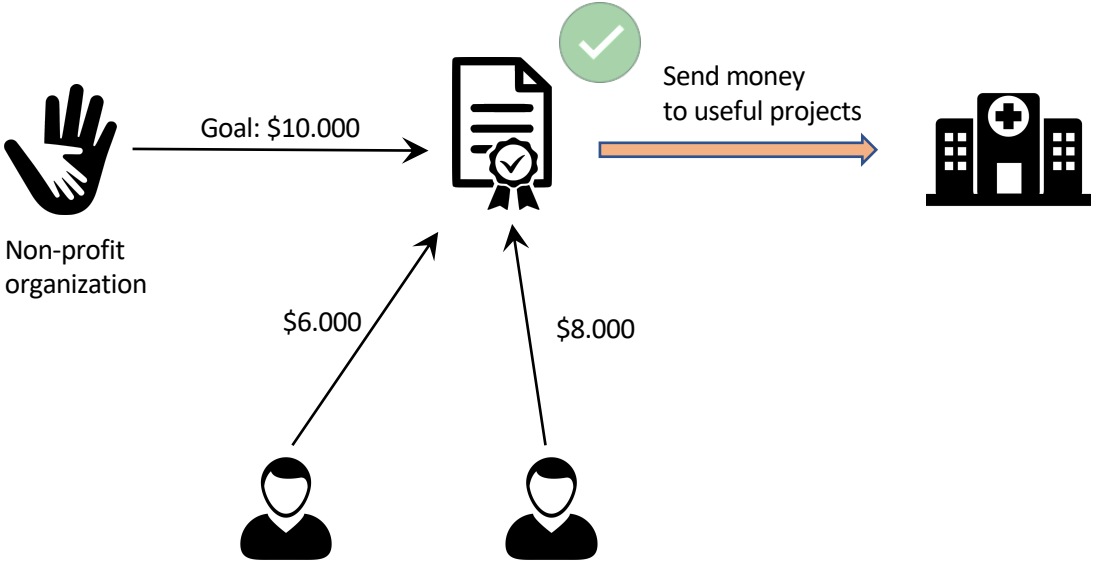
From cryptocurrencies to smart contracts

- Full-fledged program that
  - Runs in the blockchain
  - Immutable code
  - Immutable transactions
    - E.g., saving wallets, investments, insurances, games, etc.

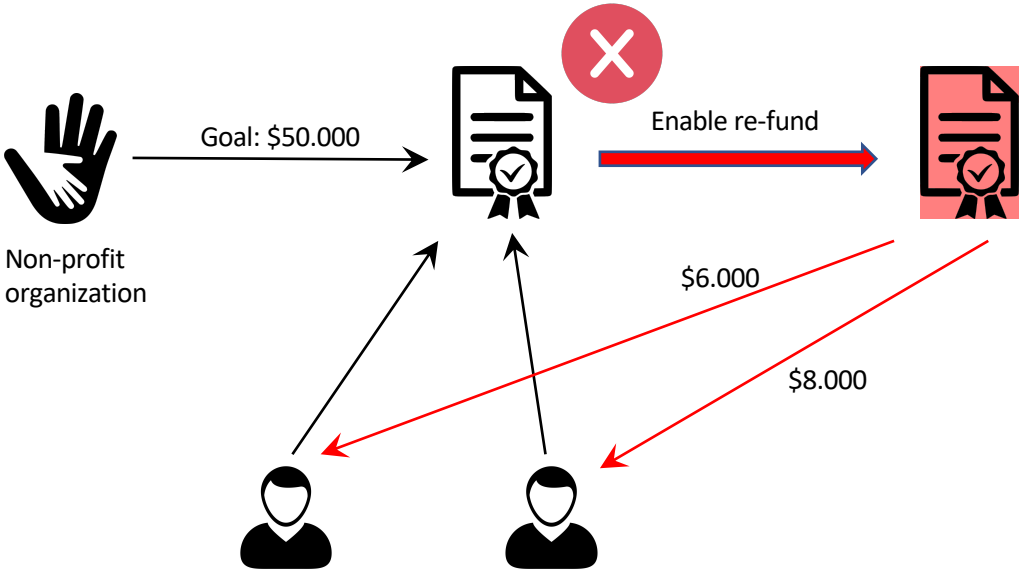
Example: Smart contract for charity

- A new un-known non-profit organization want to run a crowdfunding campaign to start a new charity project.
- The organization fixes a goal, and wants the contributors to be able to ask for a re-fund if the goal is not reached.

Successful campaign

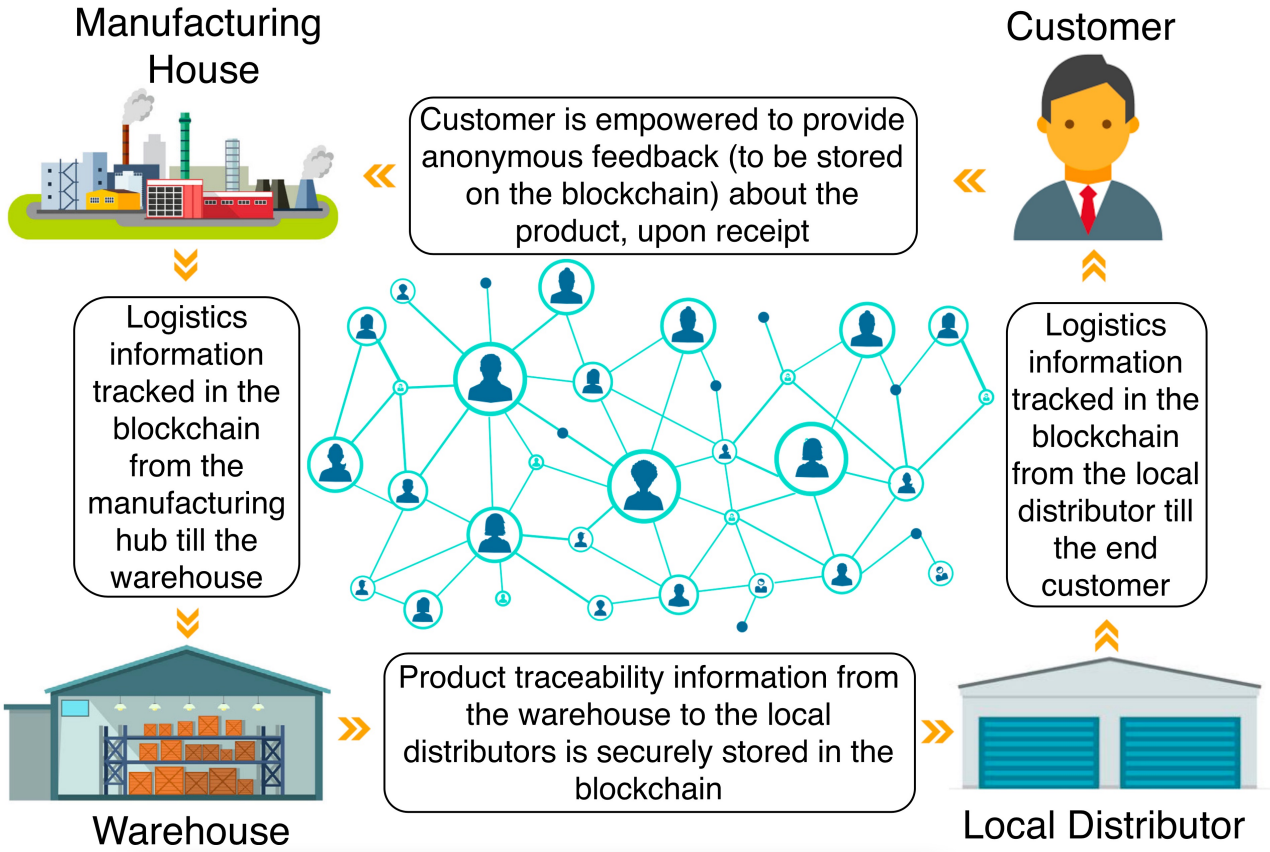


Unsuccessful campaign

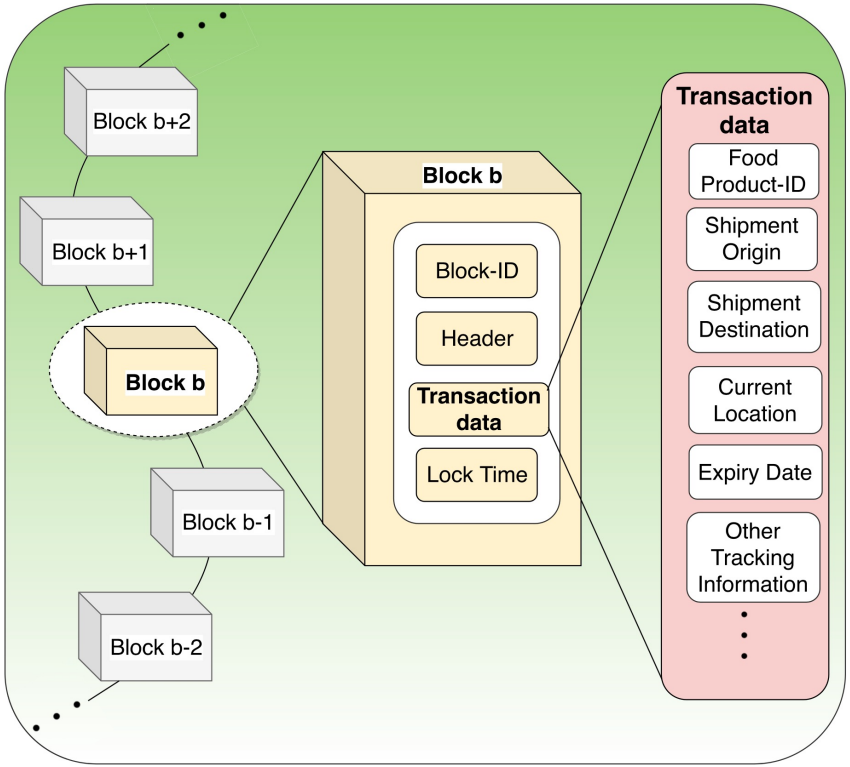


Blockchain and Industrial IoT

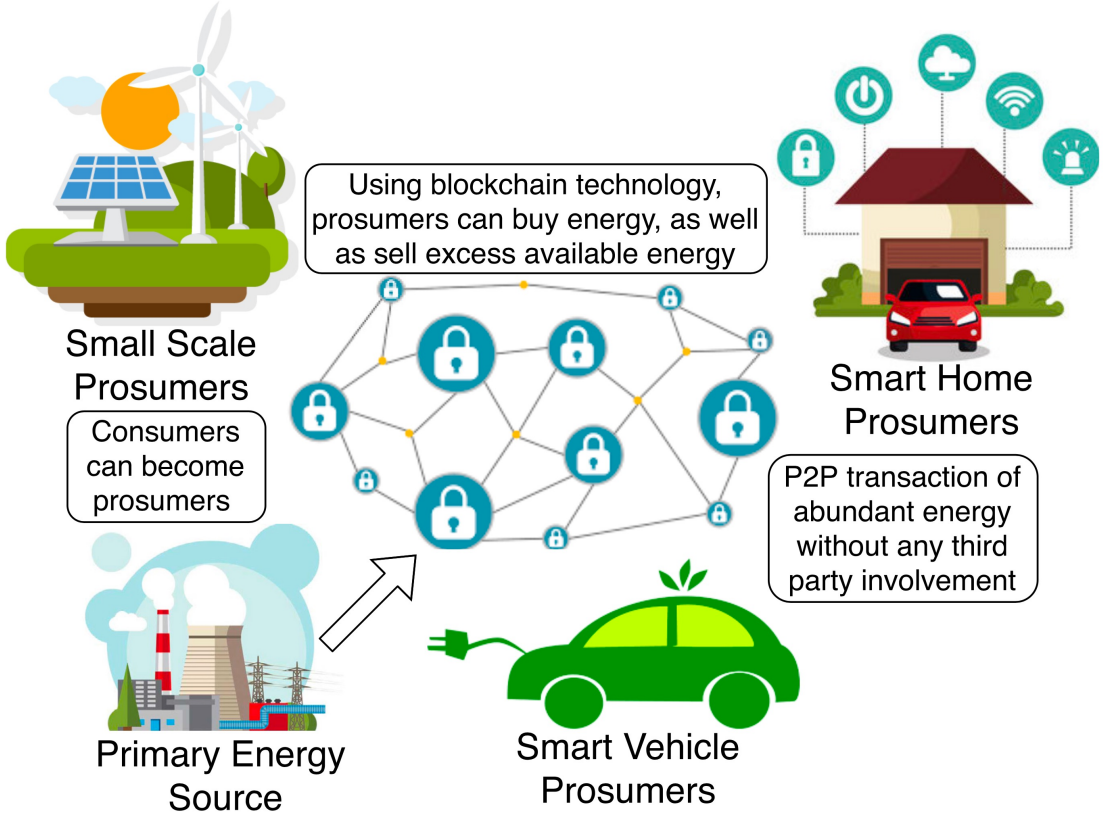
Supply chain logistics



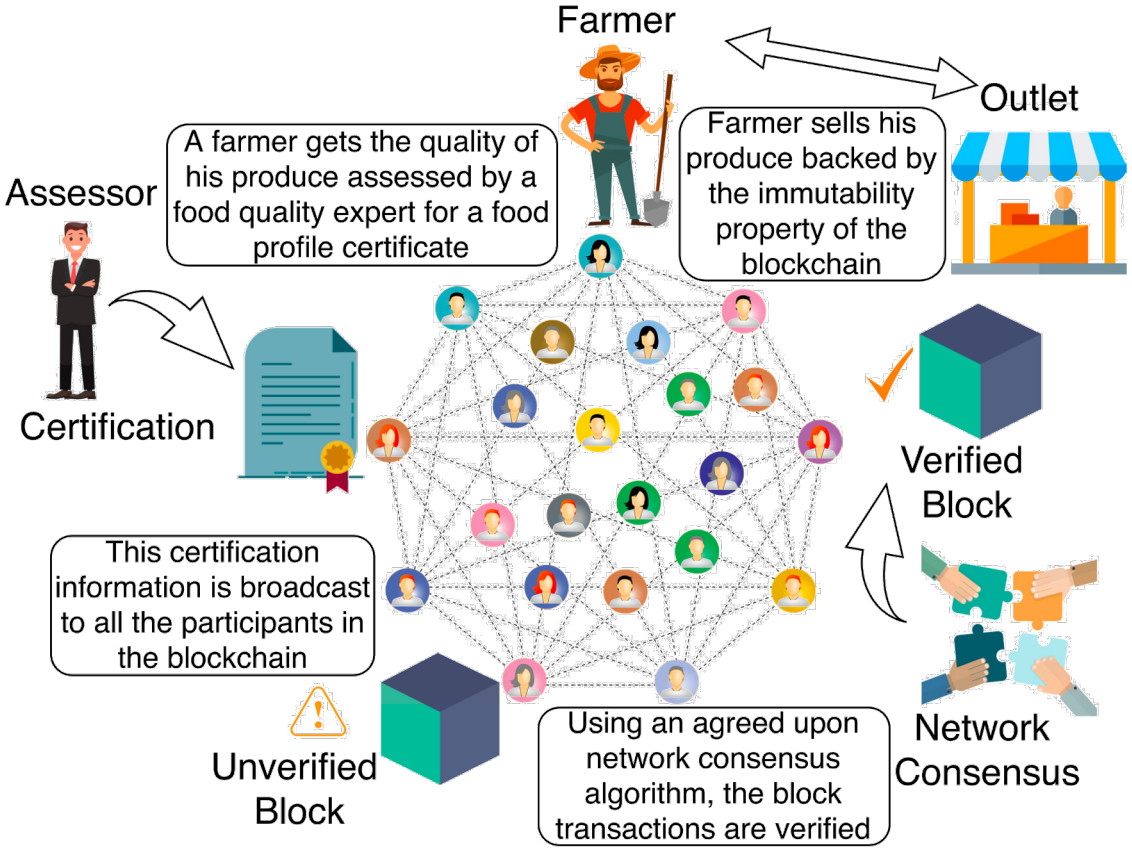
Supply chain logistics



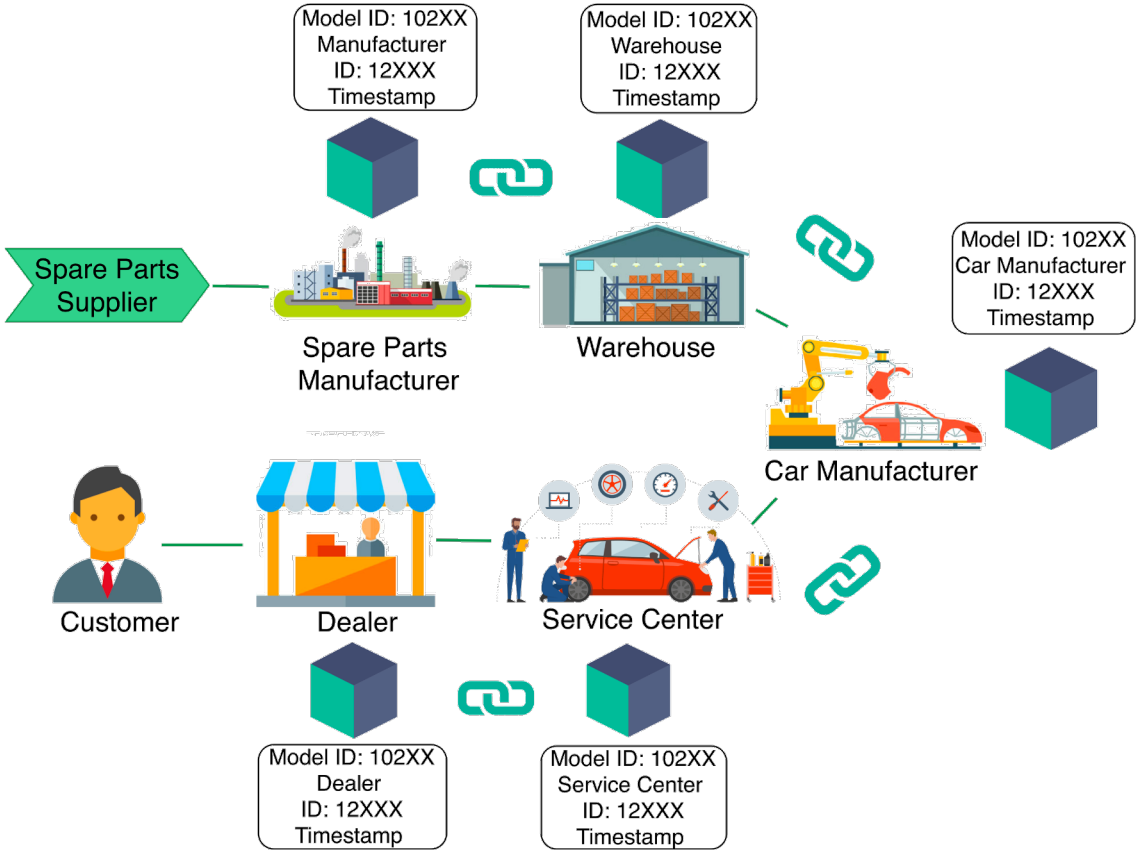
Power industry



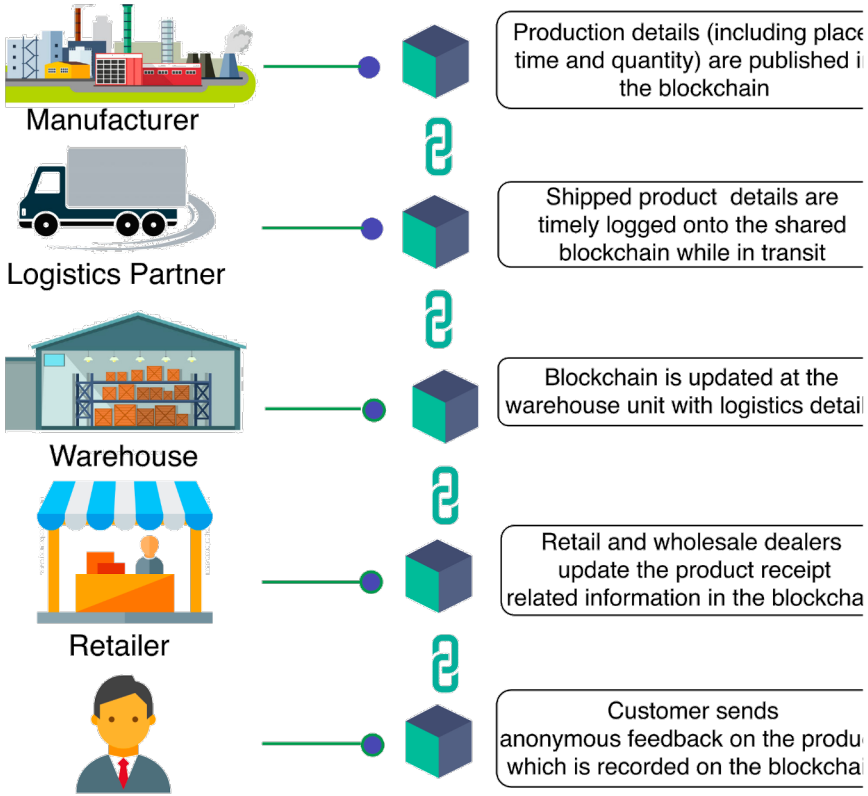
Agriculture industry



Manufacturing industry



E-commerce and retail industry

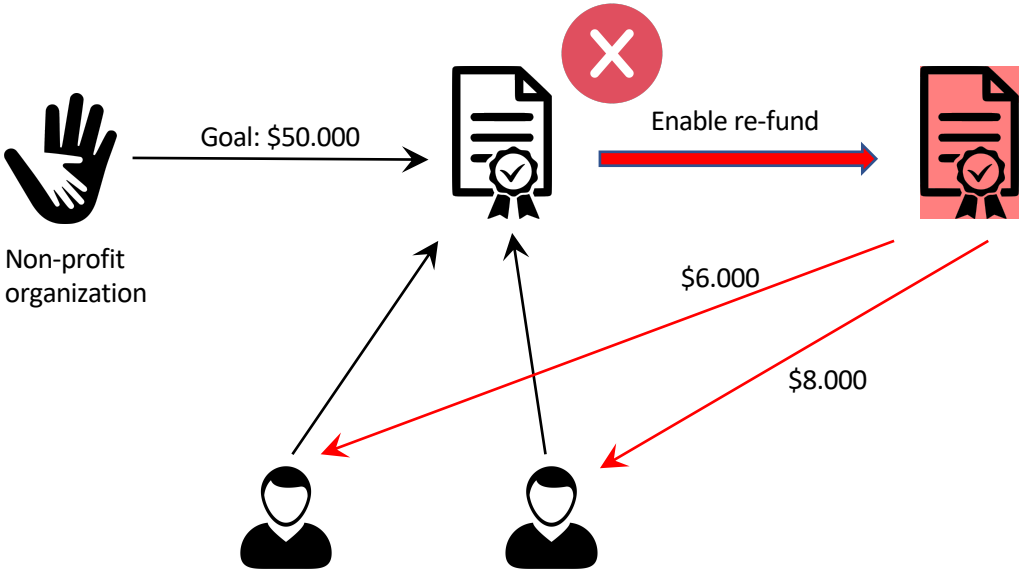


Problem

- Programs have bugs, but bugs in smart contracts might generate illegal gains and losses

- Bugs in smart contracts cannot be patched (blockchain transactions are irreversible)

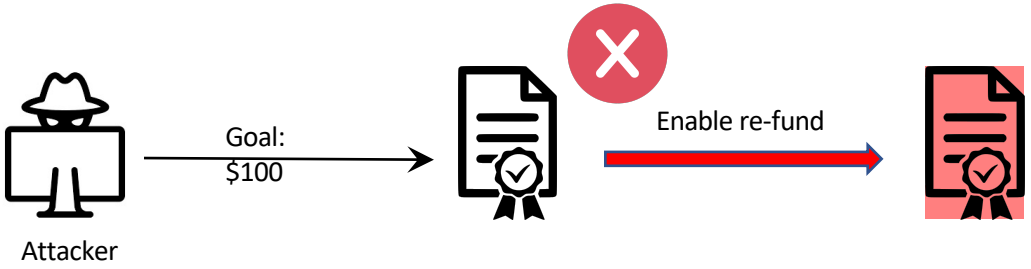
Smart contract for charity

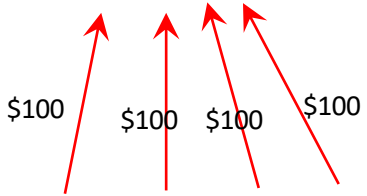


Smart contract for charity

```
function withdraw(){
    if (refundable) {
        uint userContribution = balance[msg.sender];
        bool success = msg.sender.call.value(userContribution);
        if (success) balance[msg.sender] = 0;
    }
}
```

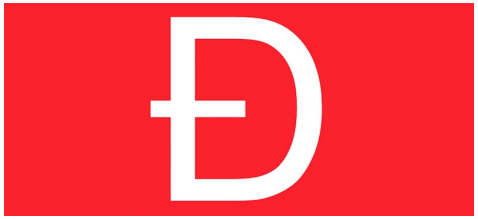
Attack






```
function withdraw(){
    if (refundable) {
        uint userContribution = balance[msg.sender];
        bool success = msg.sender.call.value(userContribution);
        if (success) balance[msg.sender] = 0;
    }
}
```

A prominent case



- DAO: Decentralized Autonomous Organization
- Decentralized venture capital
- Receive Ether for DAO tokens
- $150M

- June 17, 2016 a loophole was found in the smart contact
- A hacker could steal $70M in DAO tokens
- Ethereum hard fork required

Smart contracts

Smart contract: an *autonomous agent* stored in the blockchain by a "creation" transaction, with a *state* consisting of *balance* and private *storage*, and whose *code* is stored as Ethereum Virtual Machine bytecode

```
1 contract Puzzle{
2   address public owner;
3   bool public locked;
4   uint public reward;
5   bytes32 public diff;
6   bytes public solution;
7
8   function Puzzle() //constructor{
9     owner = msg.sender;
10    reward = msg.value;
11    locked = false;
12    diff = bytes32(11111); //pre-defined difficulty
13  }
14
15  function(){ //main code, runs at every invocation
16    if (msg.sender == owner){ //update reward
17      if (locked)
18        throw;
19      owner.send(reward);
20      reward = msg.value;
21    }
22    else
23      if (msg.data.length > 0){ //submit a solution
24        if (locked) throw;
25        if (sha256(msg.data) < diff){
26          msg.sender.send(reward); //send reward
27          solution = msg.data;
28          locked = true;
29        }}}}
```

- When a "contract creation" transaction is executed, all miners modify the blockchain state adding the new contract:
  - the contract is assigned a new address
  - a private storage is created and initialized by running the constructor
  - the EVM bytecode is associated with the contract
- The contract owner invokes a transaction to update the reward
- Other users invoke a transaction to submit their solution to the puzzle

Gas system

- Each EVM instruction needs a pre-specified amount of gas to be executed
- Users sending a transaction specify gasPrice and gasLimit
- Miners who execute the transaction receive gasPrice multiplied by the actually consumed gas, up to gasLimit
- If the execution exceeds gasLimit, it is rolled back and cancelled, but the sender has still to pay gasLimit to the miner

Types of security bugs

- Transaction-Ordering Dependence (TOD)
- Timestamp dependence
- Mishandled exceptions
- Reentrancy vulnerability

Transaction-Ordering Dependence

If a block contains two transactions T_i, T_j invoking the same contract, the order of execution is unknown until the miner who mines the block decides it. If the final state depends on the transaction order, it is unknown at the time of transaction submission.

```
1 contract Puzzle{
2   address public owner;
3   bool public locked;
4   uint public reward;
5   bytes32 public diff;
6   bytes public solution;
7
8   function Puzzle() //constructor{
9     owner = msg.sender;
10    reward = msg.value;
11    locked = false;
12    diff = bytes32(11111); //pre-defined difficulty
13  }
14
15  function(){ //main code, runs at every invocation
16    if (msg.sender == owner){ //update reward
17      if (locked)
18        throw;
19      owner.send(reward);
20      reward = msg.value;
21    }
22    else
23      if (msg.data.length > 0){ //submit a solution
24        if (locked) throw;
25        if (sha256(msg.data) < diff){
26          msg.sender.send(reward); //send reward
27          solution = msg.data;
28          locked = true;
29        }}}}
```

- If owner and user submit a transaction at the same time, the user might receive a reward different from the reward observed when the transaction was submitted
- A malicious owner might listen to the network and when a solution is submitted, a transaction to reduce the reward is also submitted, possibly with a high `gasPrice` to incentivise miners to include it in the next block

Transaction-Ordering Dependence

```
1 contract MarketPlace{
2   uint public price;
3   uint public stock;
4   /.../
5   function updatePrice(uint _price){
6     if (msg.sender == owner)
7       price = _price;
8   }
9   function buy (uint quant) returns (uint){
10    if (msg.value < quant * price || quant > stock)
11      throw;
12    stock -= quant;
13    /.../
14  }}
```

- If there are multiple buy requests, some might be cancelled even if `quant <= stock` at the time of transaction submission
- Buyers may have to pay higher than the price observed at transaction submission time if an `updatePrice` transaction is executed before the buy transaction

Timestamp dependence

A contract may use the block timestamp to execute critical operations (e.g., sending money), but the block timestamp is set by the block miner.

```
1 contract theRun {
2   uint private Last_Payout = 0;
3   uint256 salt = block.timestamp;
4   function random returns (uint256 result){
5     uint256 y = salt * block.number/(salt%5);
6     uint256 seed = block.number/3 + (salt%300)
7                    + Last_Payout +y;
8     //h = the blockhash of the seed-th last block
9     uint256 h = uint256(block.blockhash(seed));
10    //random number between 1 and 100
11    return uint256(h % 100) + 1;
12  }}
```

- A random number is used to assign a jackpot
- The miner can set the block timestamp within a margin (~900s) of the current local time
- Since all parameters involved in the computation of random are known, the miner can predict the result for each timestamp and can choose the timestamp that awards the jackpot to any player she pleases

Mishandled exceptions

A contract may raise an exception (e.g., if there is not enough gas or the call stack limit = 1024 is exceeded), but the error might be propagated to the caller either as an exception or as boolean value `false` (e.g., `send` returns `false` upon error).

```
1 contract KingOfTheEtherThrone {
2   struct Monarch {
3     // address of the king.
4     address ethAddr;
5     string name;
6     // how much he pays to previous king
7     uint claimPrice;
8     uint coronationTimestamp;
9     }
10  Monarch public currentMonarch;
11  // claim the throne
12  function claimThrone(string name) {
13    /.../
14    if (currentMonarch.ethAddr != wizardAddress)
15      currentMonarch.ethAddr.send(compensation);
16    /.../
17    // assign the new king
18    currentMonarch = Monarch(
19        msg.sender, name,
20        valuePaid, block.timestamp);
21  }}
```

- If `ethAddress` is a contract address (or a dynamic address) instead of a normal address, more gas may be required
- A contract may call itself 1023 times before calling `claimThrone`
- In both cases, if `send` fails, king looses throne without compensation

Reentrancy vulnerability

When a contract calls another contract, the current execution waits for the call to finish in an intermediate, possibly inconsistent state. The callee may call back the caller in such inconsistent state.

```
1 contract SendBalance {
2  mapping (address => uint) userBalances;
3  bool withdrawn = false;
4  function getBalance(address u) constant returns(uint){
5   return userBalances[u];
6  }
7  function addToBalance() {
8   userBalances[msg.sender] += msg.value;
9  }
10 function withdrawBalance(){
11  if (!(msg.sender.call.value(
12    userBalances[msg.sender])())) { throw; }
13  userBalances[msg.sender] = 0;
14  }}
```

- The default function `value` of the sender may call `withdrawBalance` again, causing a double transfer of money.
- The recent `TheDao` hack exploited a reentrancy vulnerability to steal around 60 M$.

Fixing smart contracts vulnerabilities

- Guarded transactions (for TOD)
- Deterministic timestamp
- Better exception handling

However, to deploy these solutions all clients in the Ethereum network should be upgraded.

Guarded transactions

- Objective: contract invocation either returns the expected output or fails
- The contract is called with the expected condition

```
1 contract Puzzle{
2   address public owner;
3   bool public locked;
4   uint public reward;
5   bytes32 public diff;
6   bytes public solution;
7
8   function Puzzle() //constructor{
9     owner = msg.sender;
10    reward = msg.value;
11    locked = false;
12    diff = bytes32(11111); //pre-defined difficulty
13  }
14
15  function(){ //main code, runs at every invocation
16    if (msg.sender == owner){ //update reward
17      if (locked)
18        throw;
19      owner.send(reward);
20      reward = msg.value;
21    }
22    else
23      if (msg.data.length > 0){ //submit a solution
24        if (locked) throw;
25        if (sha256(msg.data) < diff){
26          msg.sender.send(reward); //send reward
27          solution = msg.data;
28          locked = true;
29        }}}}
```

msg.data = <solutuion>
condition = “reward=<r>”

Deterministic timestamp

- Instead of using the (easy to manipulate) **block timestamp**, contracts should use **block index**
  - Block index always increase by 1
  - No flexibility at attacker side

timestamp - lastTime > 24 hours

blockNumber - lastBlock > 7200

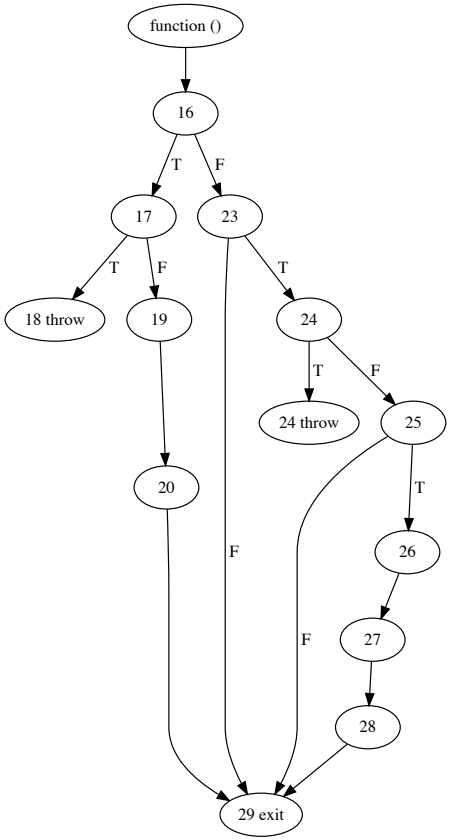
Better exception handling

- Automatically propagate exceptions (at EVM level) from callee to caller
- Adding explicit **throw** and **catch** statements

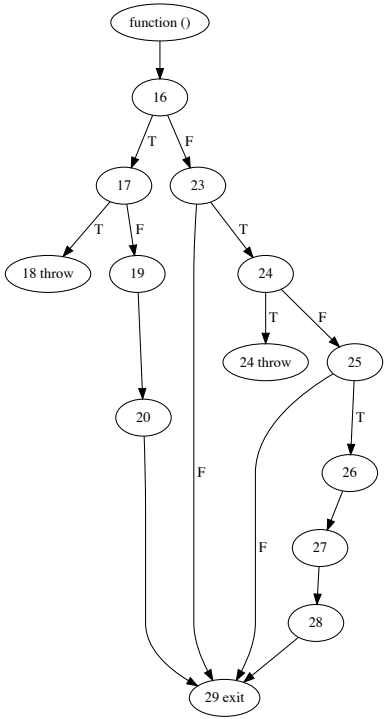
Static analysis

Control Flow Graph

```
15  function(){ //main code, runs at every invocation
16    if (msg.sender == owner){ //update reward
17      if (locked)
18        throw;
19      owner.send(reward);
20      reward = msg.value;
21    }
22    else
23      if (msg.data.length > 0){ //submit a solution
24        if (locked) throw;
25        if (sha256(msg.data) < diff){
26          msg.sender.send(reward); //send reward
27          solution = msg.data;
28          locked = true;
29        }}}}
```

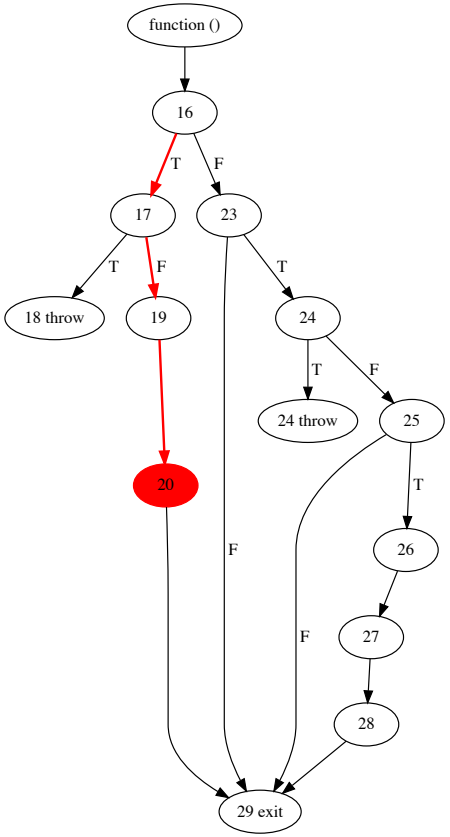


Symbolic Execution




```
15 function(){
16   if (msg.sender == owner){
17      if (locked)
18         throw;
19      owner.send(reward);
20      reward = msg.value;
21   }
22   else
23      if (msg.data.length > 0){
24         if (locked) throw;
25         if (sha256(msg.data) < diff){
26            msg.sender.send(reward);
27            solution = msg.data;
28            locked = true;
29         }}}
```

Symbolic Execution




```
15 function(){
16   if (msg.sender == owner){
17      if (locked)
18         throw;
19      owner.send(reward);
20      reward = msg.value;
21   }
22   else
23      if (msg.data.length > 0){
24         if (locked) throw;
25         if (sha256(msg.data) < diff){
26            msg.sender.send(reward);
27            solution = msg.data;
28            locked = true;
29         }}}
```

Symbolic Execution

$mgs.sender
$msg.value
$msg.data

```
function(){//main code, runs at every invocation
   if (msg.sender == owner){//update reward
      if (locked)
         throw;
      owner.send(reward);
      reward = msg.value;
   }
   else
      if (msg.data.length > 0){ //submit a solution
         if (locked) throw;
         if (sha256(msg.data) < diff){
            msg.sender.send(reward); //send reward
            solution = msg.data;
            locked = true;
```

$mgs.sender == owner

$locked_i == false$

$Reward_{i+1} == \$msg.value$

feasible

infeasible

$(\$mgs.sender == owner) \text{ AND } (locked_i == false) \text{ AND } (Reward_{i+1} == \$msg.value)$

Symbolic Execution

$mgs.sender
$msg.value
$msg.data

```
function(){//main code, runs at every invocation
   if (msg.sender == owner){//update reward
      if (locked)
         throw;
      owner.send(reward);
      reward = msg.value;
   }
   else
      if (msg.data.length > 0){ //submit a solution
         if (locked) throw;
         if (sha256(msg.data) < diff){
            msg.sender.send(reward); //send reward
            solution = msg.data;
            locked = true;
   }}
```

$mgs.sender != $owner_i$

$locked_i$ == FALSE

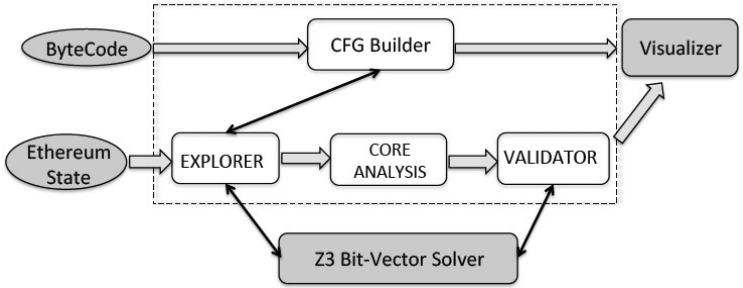
Sha256($msg.data) < $diff_i$

$Solution_{i+1}$ == $msg.data

$locked_{i+1}$ == TRUE

($mgs.sender != $owner_i$) AND ($locked_i$ == FALSE) AND (Sha256($msg.data) < $diff_i$) AND ($Solution_{i+1}$ == $msg.data) AND ($locked_{i+1}$ == TRUE) AND $locked_{i+1}$ == TRUE

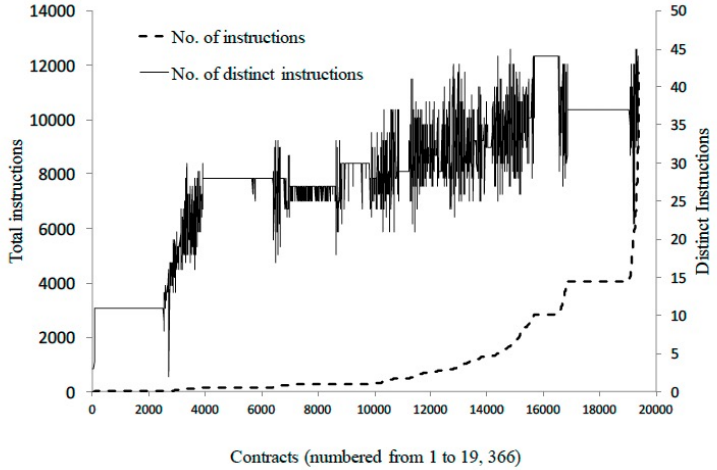
The Oyente tool



- The tool reports any vulnerabilities found in the input contract; 4k LOC of python; Z3 solver
- **CFG Builder** extracts CFG from EVM bytecode (dynamic jumps are left unresolved; they are determined during symbolic execution)
- **Explorer** performs symbolic execution of paths in depth first order; paths proved unfeasible by the solver are discarded
- **Core analysis** checks if vulnerabilities are present:
  - *TOD detector* checks if ether flows differ when order of transactions is changed
  - *Timestamp detector* uses a symbolic variable to propagate timestamp
  - *Mishandled exception detector* checks if call is followed by `ISZERO` check
  - *Reentrancy detector* checks if path condition for call is satisfiable after state changes
- **Validator** eliminates false positives by checking the feasibility of the path conditions involved in the discovered vulnerabilities by means of the Z3 solver

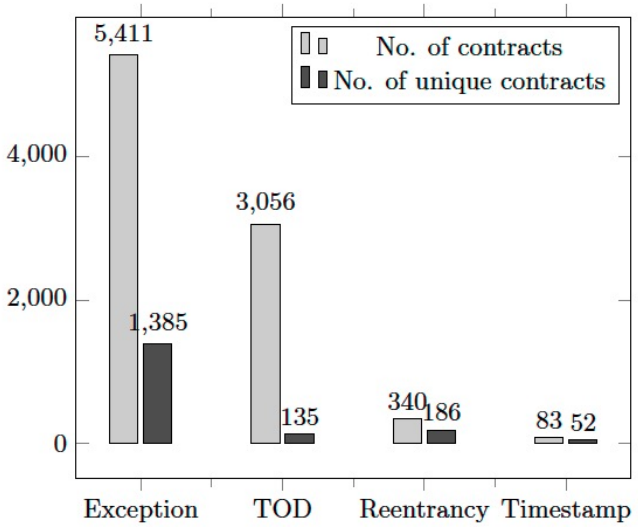
Experimental Results

Benchmark



- 19,366 smart contracts holding 3M Ethers (30 M$)
- Average balance = 318.5 Ethers (4,523 $)
- 366,213 feasible paths, found by Oyente in 3,000h on Amazon EC2

Quantitative results



- False Positives (FP) were checked on 175 contracts, for which source code was available
- FP rate = 6.4% (10 / 175)
- **Exception prevalence**: due to small call stack depth (< 50) in benign runs
- **Reentrancy FP**: use of `send` instead of `call`; the latter sends all remaining gas to callee, who can use it to perform additional calls, while the former limits such amount

Qualitative analysis

```
1 function lendGovernmentMoney(address buddy)
2     returns (bool) {
3     uint amount = msg.value;
4     // check the condition to end the game
5     if (lastTimeOfNewCredit + TWELVE_HOURS >
6         block.timestamp) {
7         msg.sender.send(amount);
8         // Sends jacpot to the last creditor
9         creditorAddresses[creditorAddresses.length - 1]
10            .send(profitFromCrash);
11        owner.send(this.balance);
12
13        // Reset contract state
14        lastCreditorPayedOut = 0;
15        lastTimeOfNewCredit = block.timestamp;
16        profitFromCrash = 0;
17        creditorAddresses = new address[](0);
18        creditorAmounts = new uint[](0);
19        round += 1;
20        return false;
21    }}
```

Ponzi (pyramid scheme)

- new investments are used to pay previous investors and to add to the jackpot
- after 12h with no investments, the last investor and the contract owner share the jackpot

- An attacker may call the contract after 1023 self-calls, to make `send` instructions fail
- A second call to the contract results in the owner receiving the entire balance, because the contract state has been reset and `creditorAddresses.length` is zero

Qualitative analysis

```
1 function lendGovernmentMoney(address buddy)
2     returns (bool) {
3     uint amount = msg.value;
4     // check the condition to end the game
5     if (lastTimeOfNewCredit + TWELVE_HOURS >
6         block.timestamp) {
7         msg.sender.send(amount);
8         // Sends jacpot to the last creditor
9         creditorAddresses[creditorAddresses.length - 1]
10            .send(profitFromCrash);
11        owner.send(this.balance);
12
13        // Reset contract state
14        lastCreditorPayedOut = 0;
15        lastTimeOfNewCredit = block.timestamp;
16        profitFromCrash = 0;
17        creditorAddresses = new address[](0);
18        creditorAmounts = new uint[](0);
19        round += 1;
20        return false;
21    }}
```

Ponzi (pyramid scheme)

- new investments are used to pay previous investors and to add to the jackpot
- after 12h with no investments, the last investor and the contract owner share the jackpot

- An attacker may pick a timestamp ahead 12h to favour the last investor or before 12h to allow for additional investors to join the contract

Qualitative analysis

```
1// ID on sale, and enough money
2if(d.price > 0 &&  msg.value >= d.price){
3    if(d.price > 0)
4        address(d.owner).send(d.price);
5    d.owner = msg.sender;// Change the ownership
6    d.price = price;     // New price
7    d.transfer = transfer;  // New transfer
8    d.expires = block.number + expires;
9    DomainChanged( msg.sender, domain, 0 );
10}
```

EtherId

- create, buy and sell Ether Ids

- If `send` fails, id ownership is changed, but the initial id owner does not receive the payment. To force `send` to fail, an attacker may:
  - provide insufficient gas for the owner's address (e.g., when the owner's address is a contract address, instead of a normal address)
  - call itself 1023 times before calling EtherId

Conclusions

- Smart contracts are a **new kind of software**, with very specific features, such as:
  - distributed execution semantics
  - peculiar transaction model
  - time dependency
  - peculiar error handling model
  - peculiar reentrancy model
- **Bugs** in smart contracts may be subtle and difficult to detect; yet they may have major, impactful consequences
  - contract users are expected to be proficient in code comprehension, since code is the norm (“code is law”)
  - there is no liability for bugs
  - bugs cannot be patched (transactions are irreversible)
  - deficiencies in current contract execution model can be fixed only if all clients upgrade to a new version of the protocol

References

- Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, Aquinas Hobor: *Making Smart Contracts Smarter*. ACM Conference on Computer and Communications Security (CCS) 2016: 254-269
- Alladi, Tejasvi, Vinay Chamola, Reza M. Parizi, and Kim-Kwang Raymond Choo. *Blockchain applications for industry 4.0 and industrial IoT: A review*. IEEE Access 7 (2019): 176935-176951.