Etude, état de l'art et implémentation de Smart Contracts pour Blockchain

Study, state of the art and implementation of Blockchain Smart Contracts

université de BORDEAUX

Master Thesis

Master in *Sciences and Technologies*,
Specialty in *Mathematics*,
Track *Cryptology and Computer Security*.

Author
Paul Hermouet <paul.hermouet@etu.u-bordeaux.fr>

Supervisor
Jérémy Métairie <j.metairie@catie.fr>

Tutor
Gilles Zémor <gilles.zemor@u-bordeaux.fr>

August 24, 2018

Declaration of authorship of the document

I hereby certify that this material, which I now submit for assessment on the programme of study leading to the award of the Master in *Sciences and Technologies*, Specialty in *Mathematics* or *Computer Science*, Track *Cryptology and Computer Security*, is entirely my own work, that I have exercised reasonable care to ensure that the work is original, and does not to the best of my knowledge breach any law of copyright, and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

Date and Signature

Abstract

Blockchain Smart Contracts are a young technology and are still misunderstood and subject to flaws. Blockchain-based networks being essentially cryptocurrencies network like the Ethereum network, these flaws can lead to huge money loss.
That is why building tools in order to analyze Smart Contracts is essential.
This thesis consists on a presentation of the Ethereum network and its Smart Contracts, an example of a decentralized protocol and a method for analyzing Smart Contracts.

Acknowledgments

I wish to thanks my supervisor Jérémy Métairie for the advises he gave me over my internship and the redaction of this thesis.

I also wish to thanks the CATIE employees, especially the SIDO team for their good mood all along the internship.

And of course, I wish to thanks my family and friends.

Contents

Introduction

Presentation of the organization

I made an internship at CATIE (*Centre Aquitain des Technologies de l'Information et Electronique*), in Bordeaux, France. CATIE is an organization whose goal is to help companies on working with new technologies.
This organization was created in 2014, has quickly grown up and now gathers 33 employees, mostly engineers and PhD's. CATIE is continually involved in many research projects, in partnership with companies and laboratories.
CATIE works on different fields and is divided in 3 main departments:

- HOMA which works on Human-computer interactions.
- SONU which works on numeric objects conception.
- SIDO which works on Big Data, AI and Blockchain.

I was an intern in SIDO's department in the blockchain team. This team set up Hubchain, a blockchain platform whose goal is to help companies to understand the new possibilities the blockchain technology offers and to get started with them.

Presentation of the internship

This internship was focused on the blockchain technology and its applications. Firstly, let's make a quick remainder of what is a blockchain.

The first blockchain was made by the so-called Satoshi Nakamoto with the creation of the Bitcoin.
The purpose of this technology is to set up a decentralized, peer to peer and secured network. In this network, each user owns a copy of the ledger (the state of the blockchain) and tries to achieve a consensus with the other users. In order to do this, each time a user wants to make a modification in the ledger, he broadcasts this modification to his neighbors and appends it to his "block". Simultaneously, he listens to the modifications of his neighbors, appends them to this block and propagates it to his neighbors. Simultaneously, he mines his block which means he tries to find the pre-image of the hash of this block. If a user find the pre-image of the hash of his block, he sends this pre-image to his neighbors. Thus, every users can quickly verify that the solution is correct and then agree on the block.

Note that the mining scheme (denoted Proof of Work) presented above is not the only consensus protocol used, there are some other consensus protocol like the Proof of Stake or the Proof of Authority. Yet the Proof of Work is currently the most used.

During this internship, I studied the Ethereum blockchain. Created in 2013, this blockchain is the first which allowed its users to publish Smart Contracts, algorithms made to automate contracts. These Smart Contracts can not be modified from the moment they are broadcast in the blockchain.
Because of this restriction, developing a Smart Contract is delicate and must not be done like developing a standard application, developers have to keep in mind that what they will not be able to patch or fix the contract. Therefore, they have to be as sure as possible that the contract they are developing is not vulnerable to malicious attackers.
My job at CATIE was to demonstrate the potential of Smart Contracts and blockchain by implementing STRAIN, a decentralized sealed-bidding protocol and to develop Mint, a tool for analysing Smart Contracts and detecting their vulnerabilities in order to fix development mistakes.

Part 1

Ethereum

CHAPTER 1

The Ethereum blockchain

The blockchain concept appeared in 2008 when a person or a group of persons under the name of Satoshi Nakamoto published a paper called *Bitcoin: A peer-to-peer electronic cash system* [5].
However, the idea of a blockchain does not really come from Nakamoto himself: Nick Szabo, an American computer scientist, was the first to express the idea of a digital currency ruled by a proof of work and a chain of transactions [6]. And the proof of work concept was imagined by Cynthia Dwork and Moni Naor in an article published in 1992 [3].
But Nakamoto was the first to publish an implementation of a cryptocurrency, with his paper came the Bitcoin, the first blockchain-based network, many others followed.

A blockchain-based network is a peer-to-peer network: the network does not have a central server but each node in the network directly communicates with his neighbors. In the following, we will present how the ethereum blockchain-based network works.

In 2013, Vitalik Buterin published a formal definition of Ethereum. In 2015, the first implementation of Ethereum is released. It is designed to provide a world-computer: the users can deploy programs (called Smart Contracts) in the blockchain and these programs will be executed by all the users of the network.
In the following, we present the properties and features of the ethereum blockchain.

1.1 Accounts

An ethereum account is given by:

- an **address**: a 20-bytes string which identifies the account
- a **balance**: the number of ethers (Ethereum currency) the account currently owns

- a **bytecode**: a sequence of instructions (one instruction is encoded by one byte) which can be executed by the **Ethereum Virtual Machine** (we will see how it works later)
- a **storage**: a 32-bytes to 32-bytes mapping

There are two kinds of accounts: the user's accounts and the contract accounts.
The **bytecode** and **storage** of users' accounts are empty. However they own a private key in order to send signed message unlike contract accounts.

Moreover, users' accounts can send **transactions** to other accounts.
To send a **transaction**, a user's account needs to broadcast it to his neighbors which will broadcast it to their own neighbors, *etc* until all the users' accounts in the network receive the transaction.

1.2 Ledger

The ledger in ethereum is the state of the network, *i.e.* the amount of ether of each account in the network. Each node eventually shares the same ledger.

1.3 Transactions

A **transaction** can be seen as a 5-uple $T = (T_{from}, T_{to}, T_{value}, T_{data}, T_{gas}) \in \mathbb{N}^5$ where:

- T_{from} is the address of the account who sends the transaction
- T_{to} is the address of the recipient's account of the transaction
- T_{value} is the amount of ether bound to the transaction, these ethers will be transferred from account T_{from} to account T_{to}
- T_{data} is the data which will be transferred with the transaction. It can be a message if T_{to} is a user's account or data which will be used during the bytecode execution if T_{to} is a contract account
- T_{gas} is the amount of gas bound to the transaction, the gas can be seen as a fuel needed for bytecode execution: the more complex the bytecode is, the more gas-expensive it is

1.4 Ethereum Virtual Machine (EVM)

The **Ethereum Virtual Machine** (EVM) is an important part of Ethereum. It is a virtual machine able to execute ethereum bytecode. It is a rather simple stack-based machine. When a user send a transaction to a contract account, all the nodes in the network will run contract's bytecode within the EVM. During a bytecode execution, the EVM has a volatile memory and a stack which are emptied at the end of the execution. It can also access the contract account's storage and the transaction which called it. The contracts are often use for transfers of ether

CHAPTER 2

Solidity

Ethereum developers obviously do not directly write bytecodes in order to develop a Smart Contract. Several programming languages have been developed in order to do this at a higher level. The languages the most used are Solidity (a language based on javascript syntax) and LLL (stands for Low-level Lisp-like Language, a language based on Lisp syntax).
We will not linger over the differences between these two languages and, since Solidity is (by far) the language the most used for Smart Contracts programming, we will now refer contracts as Solidity-developed contracts.

```
contract Bank {
    string name;
    mapping(address => uint) private balances;

    constructor (string _name) public {
        name = _name;
    }

    function getBalance () public view returns (uint) {
        return balances[msg.sender];
    }

    function store () public payable {
        //Stores a certain amount of ether
        balances[msg.sender] += msg.value;
    }


    function withdraw (uint _amount) public {
        require(_amount <= balances[msg.sender]);
        balances[msg.sender] -= _amount;
        msg.sender.transfer(_amount);
    }
}
```

A contract in Solidity is represented as a class, with attributes, methods and a constructor, making easy for a developer to develop a Smart Contract.
The purpose of this language is to make Smart Contracts source code as transparent as possible in order to avoid unexpected behaviours.
Throughout the history of Ethereum and its flaws, some modifications have been made to this language in order to avoid some common flaws. For example, it is now much harder for a developer to create a Smart Contract vulnerable to reentrancy (see section 6.1): by default, a transfer of ether to an account is made with a small amount of gas. Thanks to that, if the *fallback* function of this account is called, this *fallback* function will not be able to perform a lot of operations (such as the calls that are needed for reentrancy attacks).
However, these modifications obviously do not prevent all the flaws. That is why code review tools are decisive.

Part 2

STRAIN - Secure Auctions For Blockchains

One of the main benefits of a distributed network is not to rely on a Trusted Third Party (TTP). However, it can be difficult to turn an application relying on a TTP into the same application without TTP. Indeed, TTP allows a cheaters detection. The stake in distributed networks is designing protocols emulating TTP for real world usages.

In the following part, we present STRAIN [1], a protocol emulating a TTP in a sealed-bid-auction: an auction where each participant reveals his bid only to the auctioneer, once all bidders have bidden, the auctioneer reveals the winner: the one who bid the bigger amount.

During my internship, I made an implementation of STRAIN in Python for the client side and in Solidity for the Smart Contract.

This part is organized as follows:

- Firstly we present the Blum integers and the Golwasser-Micali's encryption scheme.
- Then we present a boolean circuit used to compare bids.
- Finally we sum up the protocol.

CHAPTER 3

Preliminaries

3.1 Blum Integers

Definition 1. *$n \in \mathbb{N}$ is a Blum integer if $n = p \times q$ where p and q are prime numbers such that $p \equiv q \equiv 3\ (mod 4)$.*

Proposition 1. *Let n be a Blum integer and a a quadratic residue in $\mathbb{Z}/n\mathbb{Z}$, then a has four square roots in $\mathbb{Z}/n\mathbb{Z}$ and only one of them is also a quadratic residue in $\mathbb{Z}/n\mathbb{Z}$.*

Proposition 2. *Let n be a Blum integer with prime factors p and q, and $(\frac{a}{n}) = 1$, then a is a quadratic residue modulo $\mathbb{Z}/n\mathbb{Z}$ if and only if $a^{\frac{(p-1)\times(q-1)}{4}} = 1\ (mod\ n)$.*

Proof.

- Let's assume that $a^{\frac{(p-1)(q-1)}{4}} = 1\ (mod\ n)$.
  Let $b = a^{\frac{(p-1)(q-1)+4}{8}}$.
  Then $b^2 = a \times a^{\frac{(p-1)(q-1)}{4}} = a\ (mod\ n)$
  Thus b is a square root of a.
- Let's assume that a is a quadratic residue in $\mathbb{Z}/n\mathbb{Z}$.
  Then $\exists b \in \mathbb{Z}/n\mathbb{Z}$ such that $b^2 = a\ (mod\ n)$.
  Then $a^{\frac{(p-1)(q-1)}{4}} = b^{\frac{(p-1)(q-1)}{2}} = \begin{cases} 1\ (mod\ p) \\ 1\ (mod\ q) \end{cases}$
  Thus $a^{\frac{(p-1)(q-1)}{4}} = 1\ (mod\ n)$

□

Proposition 3. *Let p be an odd prime number, and $a \in (\mathbb{Z}/p\mathbb{Z})^*$, then a is a quadratic residue in $\mathbb{Z}/p\mathbb{Z}$ if and only if $a^{\frac{p-1}{2}} = 1\ (mod\ p)$*

Proof.

- Let $a \in (\mathbb{Z}/p\mathbb{Z})^*$ a quadratic residue in $\mathbb{Z}/p\mathbb{Z}$.
  Then $\exists b \in (\mathbb{Z}/p\mathbb{Z})^*, b^2 = a\ (mod\ p)$.
  i.e. $a^{\frac{p-1}{2}} = b^{p-1} = 1\ (mod\ p)$ (Fermat's little theorem).

- Let $a \in (\mathbb{Z}/p\mathbb{Z})^*$ such that $a^{\frac{p-1}{2}} = 1\ (mod\ p)$.
  Let g be a generator of $(\mathbb{Z}/p\mathbb{Z})^*$,
  Then $\exists k \in \mathbb{N}$ such that $g^k = a\ (mod\ p)$
  i.e. $g^{k\frac{p-1}{2}} = 1(mod\ p)$
  Then $(p-1) \mid k\frac{(p-1)}{2}$.
  Then k is even *i.e.* $\exists l \in \mathbb{N}$ such that $k = 2l$.
  Then $a = g^{2l} = (g^l)^2\ (mod\ p)$.
  Thus a is a quadratic residue in $\mathbb{Z}/p\mathbb{Z}$.

□

Proposition 4. *If n is a Blum integer, then -1 is not a quadratic residue modulo n.*

Proof.
Let n be a Blum integer with prime factors p and q.
Then $p \equiv 3\ (mod\ 4)$
i.e. $\exists k \in \mathbb{N}$ such that $p = 4k + 3$.
Then $\frac{p-1}{2} = 2k + 1$ *i.e.* $\frac{p-1}{2}$ is odd.
Then $(-1)^{\frac{p-1}{2}} = -1\ (mod\ p)$.
Then -1 is not a quadratic residue in $\mathbb{Z}/p\mathbb{Z}$.
Thus -1 is not a quadratic residue in $\mathbb{Z}/n\mathbb{Z}$. □

Remark 1. *However, let's notice that if n is a Blum integer, $(\frac{-1}{n}) = 1$.*

3.2 Goldwasser-Micali cryptosystem

We present in the following the Golwasser-Micali's encryption scheme, the homomorphisms of this encryption system as well as a version of this system with a homomorphism for the **AND** operator. This system is asymmetric and relies on the quadratic residuals problem.

Classical cryptosystem

GenKeys algorithm generates a couple private key, public key. k is a security parameter ($\simeq$ number of bits of the keys).

Algorithm 1: GenKeys

Data: k: security parameter
Result: secret key, public key
begin
  $p \xleftarrow{\$} \{x \in [\![2^{\frac{k}{2}-1}, 2^{\frac{k}{2}}]\!], x\ prime, x \equiv 3\ (mod\ 4)\}$;
  $q \xleftarrow{\$} \{x \in [\![2^{\frac{k}{2}-1}, 2^{\frac{k}{2}}]\!], x\ prime, x \equiv 3\ (mod\ 4), x \neq p\}$;
  $pk \leftarrow p \times q$;
  $sk \leftarrow \frac{(p-1)(q-1)}{4}$;
  return sk, pk;
end

EncryptOneBit algorithm encrypts one bit b with the public key pk.

Algorithm 2: EncryptOneBit

Data: pk: public key, b: bit to encrypt
Result: c: ciphertext of b
begin
  $r \xleftarrow{\$} (\mathbb{Z}/n\mathbb{Z})^*$;
  $c \leftarrow r^2 \times (-1)^b \ (mod\ pk)$;
  return c;
end

Thus, the ciphertext of a message $m \in \mathbb{N}$ is the tuple containing the ciphertexts of each bit in m's binary expansion. Let's denote t the number of bits in m's binary expansion, and $m[i]$ the i^{th} bit of m, we get the following algorithm:

Algorithm 3: Encrypt

Data: pk: public key, m: message to encrypt
Result: c: ciphertext of m
begin
  $c \leftarrow \overbrace{[0, 0, ..., 0]}^{t}$;
  for i *from* 1 *to* $length(c)$ **do**
    $c[i] \leftarrow EncryptOneBit(pk, m[i])$;
  end
  return c;
end

DecryptOneBit algorithm decrypts a ciphertext c of one bit b with the secret key sk.

Algorithm 4: DecryptOneBit

Data: sk: private key, pk: public key, c: ciphertext of a bit b
Result: b
begin
  if $c^{sk} = 1 \ (mod\ pk)$ **then**
    $b \leftarrow 0$;
  end
  else
    $b \leftarrow 1$;
  end
  return b;
end

All it needs to find the binary expansion of a plaintext m from its ciphertext c is to decrypt all the components of c (which is a tuple).
In the following algorithm, we suppose that we can use a **binToInt** function which maps a bits' array A with the integer x such that the binary expansion of x is A.

Algorithm 5: Decrypt

Data: sk: private key, pk: public key, c: ciphertext of m (tuple)
Result: m: binary expansion of the plaintext
begin
  $m \leftarrow \overbrace{[0, 0, ..., 0]}^{length(c)}$;
  for i *from* 1 *to* $length(c)$ **do**
    $m[i] \leftarrow DecryptOneBit(sk, pk, c[i])$;
  end
  return binToInt(m);
end

Proposition 5. $\forall m \in \mathbb{N}, Decrypt(Encrypt(m)) = m.$

Proof. Proving this result comes down to prove that $\forall b \in \{0, 1\}$, $DecryptOneBit(EncryptOneBit(b)) = b$.
If $b = 0$, then $c = EncryptOneBit(pk, b) = r^2$ (with r randomly uniformly chosen in $(\mathbb{Z}/n\mathbb{Z})^*$)
c is a quadratic residue modulo pk so $c^{sk} = 1 \ (mod\ pk)$ (proposition 2).
If $b = 1$, then $c = EncryptOneBit(pk, b) = -r^2$ (with r randomly uniformly chosen in $(\mathbb{Z}/n\mathbb{Z})^*$)
Let's assume that c is a quadratic residue modulo pk, then $\exists s \in \mathbb{Z}/n\mathbb{Z}$ such that $s^2 = -r^2 \ (mod\ pk)$.
Then, as r is inversible modulo pk, $-1 = \frac{s^2}{r^2} \ (mod\ pk)$ and so -1 is a quadratic residue modulo pk $\rightarrow$ impossible (proposition 3).
Thus, c is not a quadratic residue modulo pk then $c^{sk} \neq 1 \ (mod\ pk)$

□

Homomorphisms

Classical Golwasser-Micali cryptosystem is homomorphic for addition modulo 2 (XOR / $\oplus$) and for bit flipping (NOT).

Proposition 6. $DecryptOneBit(EncryptOneBit(b_1) \times EncryptOneBit(b_2)) = b_1 \oplus b_2$

Proof.
$EncryptOneBit(b_1) = r_1^2 \times (-1)^{b_1}$
$EncryptOneBit(b_2) = r_2^2 \times (-1)^{b_2}$
$EncryptOneBit(b_1) \times EncryptOneBit(b_2) = (r_1 \times r_2)^2 \times (-1)^{b_1 \oplus b_2}$
Which is a ciphertext of $b_1 \oplus b_2$. □

Proposition 7. $DecryptOneBit(-EncryptOneBit(b)) = NOT(b)$

Proof.
$EncryptOneBit(b) = r^2 \times (-1)^b$
$-EncryptOneBit(b) = r^2 \times (-1)^{b \oplus 1}$
Which is a ciphertext of $b \oplus 1$ ie a ciphertext of $NOT(b)$. □

AND Goldwasser-Micali's version

In this section, we present a version of Glodwasser-Micali cryptosystem which supports homomorphism for AND law.
Let's denote λ a security parameter for this cryptosystem. We will see later how it impacts the decryption.
In this version, the keys generation is the same as in the classical version.

Algorithm 6: EncryptOneBit^{AND}

Data: pk: public key, b: bit to encrypt
Result: c: ciphertext of b
begin
  if $b = 0$ **then**
    $(a_1, a_2, ..., a_\lambda) \xleftarrow{\$} \{0,1\}^\lambda$;
    return $(EncryptOneBit(a_1), ..., EncryptOneBit(a_\lambda))$
  end
  else
    return $\overbrace{(EncryptOneBit(0), ..., EncryptOneBit(0))}^{\lambda\ times}$
  end
end

Algorithm 7: DecryptOneBit^{AND}

Data: sk: private key, pk: public key, c: ciphertext of a bit b
Result: b
begin
  for *i from* 1 *to* λ **do**
    if $DecryptOneBit(sk, pk, c[i]) = 1$ **then**
      return 0
    end
  end
  return 1
end

Same as for the classical version, to encrypt a plaintext $m \in \mathbb{N}$, we encrypt each bit of the m's binary expansion.

Proposition 8. *Let* $n \in \{0, 1\}$.

1. $Pr(\textit{wrong decryption}|b = 1) = 0$.
2. $Pr(\textit{wrong decryption}|b = 0) = 2^{-\lambda}$.

Proof.

1. $EncryptOneBit^{AND}(1) = (EncryptOneBit(0), ..., EncryptOneBit(0))$
   And $DecryptOneBit(EncryptOneBit(0)) = 0$
   Then $DecryptOneBit^{AND}(1) = 1$
2. $EncryptOneBit^{AND}(0) = (EncryptOneBit(a_1), ..., EncryptOneBit(a_\lambda))$
   with $a_i \in \{0, 1\}\ \forall i \in [\![1, \lambda]\!]$
   $Pr(\text{wrong decryption}) = Pr(a_i = 0\ \forall i \in [\![1, \lambda]\!]) = 2^{-\lambda}$.

□

Homomorphisme AND

Proposition 9. *Let* $b_1, b_2 \in \{0, 1\}^2$,
$EncryptOneBit^{AND}(b_1) = (c_{1,1}, ..., c_{1,\lambda})$
and $EncryptOneBit^{AND}(b_2) = (c_{2,1}, ..., c_{2,\lambda})$
Then $DecryptOneBit^{AND}((c_{1,1} \times c_{2,1}, ..., c_{1,\lambda} \times c_{2,\lambda})) = b_1\ AND\ b_2$ *(with probability equal to 1 if* $b_1 = b_2 = 1$ *and else with probability equal to* $2^{-\lambda}$*).*

Proof.

- If $b1 = b2 = 1$,
  Then $\forall i \in [\![1, \lambda]\!],\ c_{1,i} = EncryptOneBit(0),\ c_{2,i} = EncryptOneBit(0)$
  Then $c_{1,i} \times c_{2,i} = EncryptOneBit(0 \oplus 0) = EncryptOneBit(0)$
  And then $DecryptOneBit^{AND}((c_{1,1} \times c_{2,1}, ..., c_{1,\lambda} \times c_{2,\lambda})) = 1 = b_1\ AND\ b_2$
- If $b1 = b2 = 0$,
  Then $\forall i \in [\![1, \lambda]\!],\ Pr(c_{1,i} = EncryptOneBit(0)) = 1/2 et Pr(c_{1,i} = EncryptOneBit(1)) = 1/2$, same for $c_{2,i}$
  Then $Pr(c_{1,i} \times c_{2,i} = 0) = 1/2,\ Pr(c_{1,i} \times c_{2,i} = 1) = 1/2$.
  Then $(c_{1,1} \times c_{2,1}, ..., c_{1,\lambda} \times c_{2,\lambda}) = EncryptOneBit^{AND}(0)$
- If $b_1 \neq b_2$ (we can assume that $b_1 = 0$ et $b_2 = 1$ without loss of generality).
  Then $\forall i \in [\![1, \lambda]\!],\ Pr(c_{1,i} = EncryptOneBit(0)) = 1/2$ and $Pr(c_{1,i} = EncryptOneBit(1)) = 1/2$, and $c_{2,i} = EncryptOneBit(0)$
  Then $Pr(c_{1,i} \times c_{2,i} = 0) = 1/2,\ Pr(c_{1,i} \times c_{2,i} = 1) = 1/2$.
  Then $(c_{1,1} \times c_{2,1}, ..., c_{1,\lambda} \times c_{2,\lambda}) = EncryptOneBit^{AND}(0)$

□

Algorithm 8: ClassicalToAND

Data: γ: ciphertext of a bit b (classical version)
Result: $(c_1, ..., c_\lambda)$: ciphertext of a bit b (AND version)
begin
  $(a_1, a_2, ..., a_\lambda) \xleftarrow{\$} \{0,1\}^\lambda$;
  $c \leftarrow \overbrace{[0, ..., 0]}^{\lambda \ times}$;
  for i *from* 1 *to* λ **do**
    if $a_i = 1$ **then**
      $c[i] \leftarrow EncryptOneBit(0)$;
    end
    else
      $c[i] \leftarrow -EncryptOneBit(0) \times \gamma$;
    end
  end
  return c
end

Proposition 10. *The **ClassicalToAND** algorithm (above) turns a classical ciphertext of a bit b into a AND ciphertext of b.*

Proof.

- If $b = 0$, γ is a quadratic residue modulo pk.
  So $-EncryptOneBit(0) \times \gamma$ is not a quadratic residue modulo pk.
  Then $DecryptOneBit^{AND}(-EncryptOneBit(0) \times \gamma) = 0$ (with probability equal to $2^{-\lambda}$).
- If $b = 1$, it is the opposite.

□

CHAPTER 4

Comparison

The STRAIN protocol emulates a trusted third party which would have gathered the bids of all participant, computed the comparisons between these bids and finally sent the result to the participants.

4.1 Yao's Millionaire problem

In this section, we make a summary of the Millionaire problem.

The Yao's Millionaire problem introduces Alice and Bob, two millionaires who want to compare their wealth without telling the other how much money they have.

Solving this problem can be done using a XOR-AND-NOT homomorphic cryptosystem and a logical gate using only XOR($\oplus$), AND ($\wedge$) and NOT ($\neg$) operators.
The Golwasser-Micali encryption scheme gives us such a cryptosystem, but we still have to introduce the logical gate (let's denote it F).

Let x be Alice's fortune and y Bob's fortune, we will note x_k the i^{th} bit of x such that $x = \sum_{k=0}^{n} x_k \cdot 2^k$. Same goes for y and $(y_k)_{1 \leq k \leq n}$, with n being the number of bits of x's and y binary expansion (let's suppose for simplicity that they have the same number of bits).
Let $F_i = x_i \wedge \neg y_i \wedge \bigwedge_{j=i+1}^{n} (x_j = y_j)$.
Then our gate will be $F = \bigvee_{i=0}^{n} F_i$.
Notice that "$a = b$" is equivalent to "$\neg(a \oplus b)$".

Proposition 11. *F is true if and only if $x > y$.*

Proof.
F is true $\iff$ $\exists i \in [\![0, n]\!]$ such that F_i is true.
$\iff$ $x_j = y_j \forall j > i$, $x_i = 1$ and $y_i = 0$.
$\iff$ $x - y = \sum_{k=0}^{i-1}((x_k - y_k) \cdot 2^k) + 2^i > 0$.
$\iff$ $x > y$. □

We will use F to compare the bids of each couple of users in the bidding protocol.

4.2 Comparison between two semi-honest users

Now we assume that each user can broadcast data to the blockchain.

At the beginning of the bidding, every users compute a couple private key/public key and broadcast the public key.
Then every users choose an amount to bid, encrypt it and broadcast the ciphertext.

In the following, we present how two users can compare their bids without giving their values.
We first present the comparison scheme (Fishlin's protocol [4]) which provide a secure comparison in a model of semi-honest users (users who can share information with other users but who follow the protocol).

In the following, we will note $(S_i)_i$ the set of the participants, sk_i and pk_i the couple private key/public key of S_i and v_i his/her bid.

When S_i wants to compare his/her bid with S_j's bid, he/she follows the following protocol:

1. S_i encrypts v_i with pk_j. Let's denote c_{ij} the resulting ciphertext.
2. S_i computes, thanks to Golwasser-Micali cryptosystem's homomorphisms, a ciphertext for each $(v_{i,k} = v_{j_k})$ as well as an ciphertext for each $(\neg v_{j,k})$, for $1 \leq k \leq n$.
3. S_i embeds these ciphertexts as well as c_j into Goldwasser-Micali$^{\text{AND}}$ ciphertexts using the $ClassicalToAND$ algorithm.
4. S_i computes, thanks to Golwasser-Micali$^{\text{AND}}$ cryptosystem's homomorphism, a ciphertext for each F_k.
5. Finally, S_i randomly shuffles these resulting ciphertext and broadcasts this mixing.

Then, if S_j wants to know if his/her bid is greater than S_i's bid, he just has to get the mixing and to decrypt it. If all the decoded bits are equals, then his/her bid is greater than S_i's bid, else, it's not.
Note that the shuffling is done to prevent S_j to know the index k such that $F_k = 1$. This knowledge would give an information about v_i.

4.3 Comparison between two malicious users

In this section, we present a way to secure the previous scheme in a model of malicious users (users who can share information and cheat).

We make the assumption that the judge of the bidding is honest or semi-honest.

A malicious user could cheat by using another value to compute c_{ij} than his/her bid.

Thus, we want a user S_i to prove to a verifier V that c_{ij} is valid, *i.e.* verifies $Decrypt(sk_i, pk_i, c_i) = Decrypt(sk_j, pk_j, c_{ij})$ *i.e.* that S_i did use v_i to compute c_{ij}.

Notation 1. *In the following sections, we will use the following notations:*

- $(v_i)_k$ *is the* k^{th} *bit of* v_i.
- *A coin of an Goldwasser-Micali encryption is the random number used to compute the encryption (eg the coin of* $Encrypt(pk, b) = r^2 \cdot (-1)^b \ mod \ pk$ *is* r*).*

Proof Protocol

Commitment
1. S_i draws $(\delta_k)_{1 \leqslant k \leqslant n} \in \{0,1\}^n$
2. S_i computes $\gamma_k = Encrypt(pk_i, \delta_k)$ and $\gamma_k^{'} = Encrypt(pk_j, \delta_k)$ for $1 \leqslant k \leqslant n$
3. S_i computes $\Gamma_k = \gamma_k \cdot (c_i)_k$ and $\Gamma_k^{'} = \gamma_k^{'} \cdot (c_{ij})_k$ for $1 \leqslant k \leqslant n$
   Notice that, δ_k, γ_k and $\gamma_k^{'}$ being Goldwasser-Micali ciphertexts, $\Gamma_k = Encrypt(pk_i, (v_i)_k \oplus \delta_k)$ and $\Gamma_k^{'} = Encrypt(pk_j, (v_i)_k \oplus \delta_k)$
4. S_i sends $(\gamma_k, \gamma_k^{'}, \Gamma_k, \Gamma_k^{'})_{1 \leq k \leq n}$ and c_{ij} to the verifier V

Challenge
5. V sends n bits b_k to S_i

Reply
6. For each b_k, if $b_k = 0$, S_i sends the plaintext (δ_k) and the coins of γ_k and $\gamma_k^{'}$ to V, else if $b_k = 1$, S_i sends the plaintext $((v_i)_k \oplus \delta_k)$ and the coins of Γ_k and $\Gamma_k^{'}$ to V

Verification
7. V verifies that $\Gamma_k = \gamma_k \cdot (c_i)_k$ and $\Gamma_k^{'} = \gamma_k^{'} \cdot (c_{ij})_k$
8. V recomputes γ_k and $\gamma_k^{'}$ if $b_k = 0$ or Γ_k and $\Gamma_k^{'}$ if $b_k = 1$ given the plaintexts and coins sent by S_i and verifies that the results correspond to the values sent by S_i.
9. V verifies that the plaintexts sent by S_i are the same (they are supposed to be either δ_k or $\delta_k \oplus (v_i)_k$).

In practice, this proof protocol is done multiple times: doing this proof $\lambda^{''}$ times brings the probability of failing the verification to $2^{-\lambda^{''}}$.

CHAPTER 5

Protocol

At the beginning of the auction, each user broadcasts his public key on the blockchain. To ensure that each user broadcast a valid public key, each user S_i computes a ZK-proof that pk_i is a Blum integer (proof can be found below). Then, participants commit to a bid as we saw above and start comparing their bids.

5.1 Protocol

We sum up in the following the different steps of the protocols.

1. S_i generates a couple private key, public key sk_i, pk_i and computes a proof $P^{Blum}(pk_i)$ that pk is a Blum integer.
2. S_i choses an amount to bid v_i and computes $c_i = Encrypt(pk_i, v_i)$.
3. S_i sends $pk_i, P^{Blum}(pk_i), c_i$ to the blockchain.
4. S_i waits for the other users to send their keys, proofs and bids $pk_j, P^{Blum}(pk_j), c_j$ to the blockchain, then gets them
5. S_i checks if all the $P^{Blum}(pk_j)$ are valid. If not, the protocol stops.
6. S_i compares the bids using the encryption protocol seen above and computes the corresponding P^{eval}.
7. S_i sends the results of the comparisons and the P^{eval} to the blockchain.
8. S_i waits for the other users to send their comparisons and P^{eval}, then gets them.
9. S_i checks that all the P^{eval} are valid.
10. S_i decrypts comparisons, and broadcasts them on the blockchain.
11. The winner(s) of the bid is (are) declared.

5.2 Proof Knowledge of Blum integer P^{Blum}

The idea of this proof [2] for a prover P is to prove to a verifier V that $n \in \mathbb{N}$ is a Blum integer without giving information about the prime factorization of n.

1. P picks randomly uniformly $x \in (\mathbb{Z}/n\mathbb{Z})^*$.
2. P sends $x^2 \ mod \ n$ to V.
3. V sends a challenge $\epsilon \in \{-1, 1\}$ to P.
4. P sends y to V such that $y^2 = x^2 \ mod \ n$ and $(\frac{y}{n}) = \epsilon$.
5. V verifies that $y^2 = x^2 \ mod \ n$ and $(\frac{y}{n}) = \epsilon$.

Part 3

Smart Contracts analysis

One of the main benefits of the blockchain, the fact that it is impossible to modify a Smart Contract after it is broadcast on the blockchain is also an important constraint. This is especially problematical when one finds, after some time that the contract does not exactly work as expected.
Thus it is primordial for a contract designer to be as sure as possible that his contract is not subject to bugs or flaws. This check can be done by analyzing the contract. There are different ways to do it, we will see some of these way in the following.

During my internship, I built MINT, a tools for analysing Smart Contracts. I implemented this tool in Python. To run the analysis, it runs a symbolic execution, a method explained in the following.

This part is organized as follows:

- Firstly, to put things in context, we present some of the most important flaws in Ethereum's history.
- Then we introduce the SMT problems and the symbolic analysis.

CHAPTER 6

Ethereum Flaws

Since its release in 2015, the Ethereum network has encountered some important issues, not in the network itself but in some important Smart Contracts.

6.1 The DAO

The DAO (stands for Decentralized Autonomous Organization) was one of the most important Smart Contracts on Ethereum. Released in 2016, it was supposed to allow ethereum users to show and finance projects. This contract quickly gathered a large amount of user and ethers. Unfortunately, one month after its released, it has been attacked: three million of ethers (about 50 million dollars at the time) have been taken from it. We present below a piece of code that contains the flaw which took The DAO down.

```
1  contract DAO {
2
3      // maps each address with its tokens
4      mapping(address => uint) public balances;
5
6      function store() public payable {
7          // Stores a certain amount of tokens
8          balances[msg.sender] += msg.value;
9      }
10
11     function withdraw(uint _amount) public {
12         // Allows a user to withdraw its tokens
13         require (balances[msg.sender] >= _amount);
14         msg.sender.transfer(_amount);
15         balances[msg.sender] -= _amount;
16     }
17
18 }
```

The contract above has one attribute (*balances*) and two functions (*store* and *withdraw*).
Its usage is simple: a user calls *store* and appends some ether to the call to store these ethers in the contract, the balance of the user is therefore increased by the amount of ethers he appended. A user can recover some of the ethers he stored by calling *withdraw*: this function first checks if he has enough balance, and then sends him the ether and decreases his balance by the number of ether he recovered.

The attack uses the *fallback* features, the attacker creates a contract having these functions:

Algorithm 9: StartAttack
Call *store* function of *TheDAO* contract with 100 ethers Call *withdraw* function of *TheDAO* contract with 100 ethers as argument.

Algorithm 10: *fallback*
Call *withdraw* function of *TheDAO* contract with 100 ethers as argument.

The attacker then transfers 100 of his ethers to this contract and calls the *StartAttack* function, let us see step by step what this call do:

1. The attacker's contract stores 100 ethers in *TheDAO* contract.
2. The attacker's contract calls *TheDAO*'s *withdraw* function in order to recover its 100 ethers.
3. The test (line 13 on the code above) passes.
4. *TheDAO* contract sends 100 ethers to the attacker's contract.
5. The *fallback* function of the attacker's contract is automatically called: it calls *TheDAO*'s function *withdraw* again.
6. The test (line 13) passes again because the line 15 decreasing the user's balance has not been reached yet.
7. The process starts over and over and the balance of the attacker's contract keeps increasing until all the ethers of *TheDAO* contract are gone.
8. The attacker can now transfer these ethers from his contract to his account.

This kind of flaw is called reentrancy. In this case, the problem could have been avoided if the decreasing of the user's balance would have been made before the transfer (swapping the lines 14 and 15 on the code above).

6.2 ERC20 Tokens

ERC20 is a standard for token creation in ethereum. It has become common that ethereum services do not use directly ethers inside their contracts but that they create a new currency, a token. ERC20 was created to provide a safe and generic way to create token. It provides a list of functions to implement for a token to be certified as an ERC20 token.
However, many contracts using ERC20 tokens implements other functions to make tokens manipulations easier, this is the case of the *batchTransfer* function which was meant to ease the transfer of tokens to several accounts and which was implemented on several contracts:

```
1    function batchTransfer(
2        address[] _recipients,
3        uint _amount) public {
4
5        uint totalAmount = _recipients.length * _amount;
6        require(totalAmount <= balances[msg.sender]);
7        balances[msg.sender] -= totalAmount;
8        for (uint i = 0; i < _recipients.length; i++) {
9            balances[_recipients[i]] += _amount;
10       }
11   }
```

This function is vulnerable to an overflow attack, here is a way for an attacker to get as many tokens as he wants:

- Let's note $account1, account2$ two accounts owned by the attacker.
- $account1$ calls *batchTransfer* with $[account1, account2]$ as *_recipients* and 2^{255} as *_amount*.
- $\_recipients.length = 2$, so $totalAmount = 2^{256} = 0$ (integers operations in Solidity are made modulo 2^{256})
- Then the test line 6 passes
- The balance of $account1$ is decreased by 0
- The balances of $account1$ and $account2$ are increased by 2^{255}

In this case, there was no attack: the developers spotted the flaw before anyone can exploit it and froze the contracts.

You can find in Annexes the result of the analysis of an overflow-vulnerable contract by Mint, the Smart Contract analysis tools i have made during this internship.

CHAPTER 7

Symbolic Execution

The symbolic execution is a method to analyze a code and highlight its flaws. It basically consists on replacing all the variables in the code by symbolic values. Then running a static analysis (traversal of the code, the code is not executed) and get all the possible path with their constraints. We will describe this method in the following.

7.1 SMT - Satisfiability Modulo Theories

Tools that perform symbolic execution use SMT solvers, let's quickly present what the SMT problem is.

The SMT problem *is a decision problem for logical formulas with respect to combinations of background theories expressed in classical first-order logic with equality* (from Wikipedia) [7].

Less formally, a SMT instance is a sequence of predicates linked with AND ($\wedge$) logical operator.
e.g. $(3x > 0 \wedge 2x < 5)$ is an instance of a SMT problem with $3x > 0$ and $2x < 5$ as predicates.

Let $p = (p_i)_{1 \leqslant i \leqslant k}, p_i : E^n \longrightarrow \{true, false\}$ a set of predicates.
The SMT instance (p, o) is satisfiable if there exists a n-uple $x = (x_i)_{1 \leqslant i \leqslant n} \in \mathbb{N}^n$ such that $p_1(x) \wedge p_2(x) \wedge ... \wedge p_{n-1}(x) \wedge p_n(x)$ is True.

Example

Let $(3x > 0 \ \wedge 2x < 5)$ be a SMT instance, the predicates are given by the formulas $3x > 0$ and $2x < 5$. This instance is satisfiable: $x = 1$ is a solution.
However, the instance $(3x > 0 \ \wedge 2x < 2)$ is not satisfiable.

Over the years, many SMT solvers had been developed. These solvers checks whether a formula is satisfiable or nor. If it is, the solver returns concrete inputs that satisfy the formula.

7.2 Symbolic Execution

A symbolic execution builds a tree in which the nodes are the state of the program (*i.e.* the current variables and their symbolic values) and the edges are the constraints leading to these states.
A symbolic execution tool has a SMT solver to make sure that the constraints leading to a node are satisfiable.

This tree is built recursively during the static execution of the program:

- We start the execution with an empty node.
- If the execution reaches a new variable, it maps it either with a concrete value if it is possible or with a symbolic value (we will see how in the example below) and appends it to the current node.
- If the execution reaches an instruction which modifies the value of a variable, the value of this variable is also modifies in the current node.
- If the execution reaches a fork (*e.g.* a *if* statement), two sons of the current node are created, both sharing the current node's state. The edge from the current node and each one of its sons is the constraint leading from the father node to the son node. If the conjunction of all the constraints leading to a node is declared unsatisfiable by the SMT solver, the node is remove, else, the execution goes on from the sons.

During the static execution:
If the execution reaches a new variable, it maps it either with a concrete value if it is possible or with a symbolic value (we will see how in the example below).
If the execution reaches a fork in the program (*e.g.* *if* statement), the state of the execution is duplicated and a new execution starts with this state from each new path following the fork and the new constraint leading to this path is appended to the constraints sequence.
Thus, at the end of the execution, all the paths the program can take are identified with, for each path, the set of constraints that variables given in input must satisfy to reach it.

Example

```
1 int foo(int a, int b, int m) {
2     int c = m * a;
3     if (a > 0)
4         if (c < b)
5             vulnerable();
6         return 1;
7     return 0;
8 }
```

```
9
10
11 int main() {
12     int a, b;
13     scanf("%d", &a);
14     scanf("%d", &b);
15     int m = 3;
16     foo(a, b);
17 }
```

Let's see how a symbolic execution works in this example.

- The program starts with the *main* function and gets the values of *a* and *b* from the user. So we can not give a concrete value to *a* and *b*, then we map *a* with a symbolic value a_0 and *b* with a symbolic value b_0.
- The execution reaches the variable *m* which is instantiated so we replace *m* by the concrete value 3.
- We enter the *foo* function.
- The execution reaches the variables *c*, which is instantiated, then we map *c* with the symbolic value $a_0 + 3$.
- The execution reaches a fork, then two executions are ran in parallel: one with the constraint $a_0 <= 0$, the other with the constraint $a_0 > 0$.
- The first execution returns 0 and stops.
- The second one reaches another fork, it splits into two execution: one with the constraints $(a_0 > 0), (a_0 + 3 < b_0)$ and the other one with the constraints $(a_0 > 0), (a_0 + 3 \geqslant b_0)$.
- The first one calls the *vulnerable* function and stops.
- The second one returns 1 and stops.

A symbolic execution on this program gives us all the possible paths taken by the program. This analysis method is frequently associated with pattern matching and a SMT solver to highlight the flaws in the code. Pattern matching is a method which consists on defining several patterns which lead to flaws and identifying them in the code. In this example, let's suppose that the *vulnerable* function leads to a flaw, then the pattern matching will look in the paths given by the symbolic execution all the calls to this function. In our case, there is one occurrence. Once these occurrences are found, a SMT solver is used with the constraints of each path to find input values that lead to the flaw. In our case the constraints leading to the *vulnerable* function are $(a_0 > 0), (a_0 + 3 < b_0)$ and a SMT solver could give $a_0 = 1, b_0 = 5$.

7.3 Limitations

This method is not infallible. Indeed, there is some case where it can not efficiently analyze a program.

Firstly the number of path that can be taken in a program can be subject to an explosion: this number can be in the worst case 2^n where n is the number of forks in the program. Then it can be impossible to explore all the paths for large programs

Secondly the symbolic execution can not handle loops in which break conditions are symbolic. Let's look at the following piece of code:

```
1 int main() {
2     int n;
3     scanf("%d", &n);
4     for (int i = 0; i < n; i++) {
5         // ...
6     }
7 }
```

In this example, the variable n would be mapped to a symbolic value, which would lead to an infinite loop.
In this case, the symbolic execution tools generally substitute n with a concrete value. It is called a lower bounding of the loop: all the possible paths will not be taken into account.

There can also be limitations due to the SMT solver. Indeed, a SMT solver will not be able to solve all formulas. For example, solving the constraint $if\ (hash(a_0) = 1234)$... is suppose to be a difficult problem, as well as discrete modulo or other open problems.

Conclusion

The STRAIN protocol is a good example to show the benefits of a blockchain and how to use them. It shows how powerful can be some cryptographic methods, in this case Goldwasser Micali encryption scheme and zero-knowledge proofs but at the same time, it also shows that even in a rather simple example, the protocol to achieve the "anti-cheaters" security in a decentralized network is much longer and harder to implement than in a classical server-clients network. From the moment one wants to develop a Smart Contract a little more complex than the simple contracts shown as example, it could need a strong knowledge in cryptography.

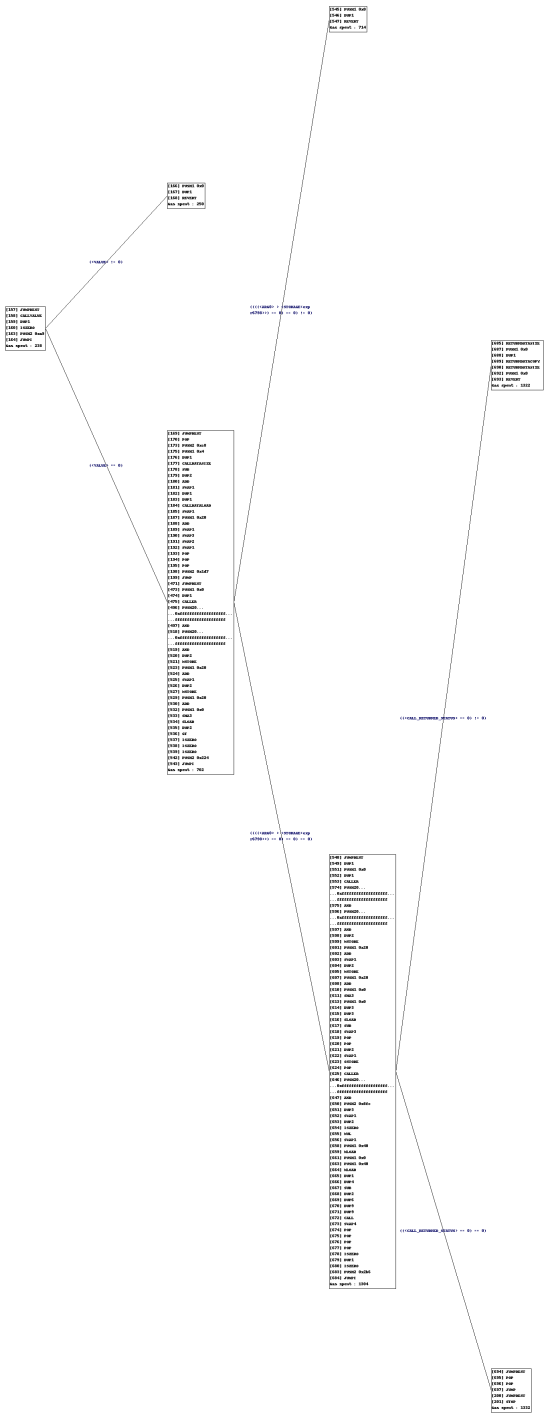
This internship at CATIE gave me the opportunity to study blockchains, and all the ecosystem around the different blockchain, especially Ethereum but also some special blockchain made for the Internet of Things. During the building of Mint, I saw all the impact software flaws can have in this kind of environment. Indeed, Ethereum can not be considered as mature yet and this special environment sometimes leads Smart Contracts developers to underestimate the actual impact of flaws.
As Ethereum blockchain is young, there are still many modifications in Solidity and in the development good practices and it was exciting to work in this constantly changing domain.

Annexes

.1 Mint analysis of a vulnerable contract

We present below the result of the analysis by Mint of an overflow-vulnerable Smart Contract. Mint draws the execution tree of each function and publish a report on the possible flaws.

```
pragma solidity ^0.4.21;

contract Vulnerable {

    mapping(address => uint) private balances;

    function getBalance() public view returns (uint) {
        return balances[msg.sender];
    }

    function store() public payable {
        balances[msg.sender] += msg.value;
    }

    function withdraw (uint _amount) public {
        require(_amount <= balances[msg.sender]);
        balances[msg.sender] -= _amount;
        msg.sender.transfer(_amount);
    }

    function transfer(address _to, uint _amount) public {
        require(_amount <= balances[msg.sender]);
        balances[msg.sender] -= _amount;
        balances[_to] += _amount;
    }

    function batchTransfer(
        address[] _recipients,
        uint _amount) public {

        uint totalAmount = _recipients.length * _amount;
        require(totalAmount <= balances[msg.sender]);
        balances[msg.sender] -= totalAmount;
        for (uint i = 0; i < _recipients.length; i++) {
            balances[_recipients[i]] += _amount;
        }
    }
}
```



Execution tree of the *withdraw* function

```
===== Contract Vulnerable =====


### TRANSLATER ###
Version readable of opcodes written in results/opcode_readable.txt
Json file of opcodes written in results/opcodes.json


### TREE BUILDER ###

Building the execution trees...

-- Warning : some loops had to be bounded - program counters [869, 88
4]
Building the execution trees : OK


### VIEW ###

Drawing the trees...
The tree of function batchTransfer is too large and will not be displ
ayed.
Drawing the trees : OK

4 trees drew in /results/Vulnerable/img/


### SECURITY ###

Security Alert - code 2 - Contract Vulnerable could be prodigal ! (fu
nction withdraw)
Security Alert - code 7 - Contract Vulnerable - function batchTransfe
r is vulnerable to overflow/underflow tests.
2 alert(s) found in contract Vulnerable, see ./results/security.log f
or more information.


===== Summary =====

Execution ended with 2 security alert(s), see ./results/security.log
for more information
```

Logs from the analysis

The analysis returns two flags, one for the overflow flaw as expected and a warning on the *withdraw* function. "prodigal" means that a user can get ether from the contract without restrictions through *withdraw* function. In this case, there is no flaw, it is a kind of false positive.

Bibliography

[1] Erik-Oliver Blass and Florian Kerschbaum. Strain: A secure auction for blockchains. Cryptology ePrint Archive, Report 2017/1044, 2017. `https://eprint.iacr.org/2017/1044`.

[2] Manuel Blum. Coin flipping by telephone. In *Advances in Cryptology: A Report on CRYPTO 81*, pages 11–15, 1981.

[3] Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. pages 139–147. Springer-Verlag, 1992.

[4] Marc Fischlin. A cost-effective pay-per-multiplication comparison method for millionaires. In David Naccache, editor, *Topics in Cryptology — CT-RSA 2001*, pages 457–471, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[5] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system," http://bitcoin.org/bitcoin.pdf, 2008.

[6] Nick Szabo. Bit gold, http://unenumerated.blogspot.com/2005/12/bit-gold.html, 2008.

[7] Wikipedia contributors. Satisfiability modulo theories — Wikipedia, the free encyclopedia, 2018.