Proposal for Protocol on a Quorum Blockchain with Zero Knowledge

Thomas Espel, Laurent Katz, Guillaume Robin
espel.thomas@gmail.com, robinguillaume.pro@gmail.com, laurentkatz01@gmail.com

Le Lab Banque de France, Paris

November 10, 2017

Abstract

In this paper, we present an implementation scheme of an RTGS[1] on Quorum using the Solidity language. It is heavily inspired by the Schnorr signature protocol to verify the identity of the participants. We have implemented a distributed ledger solution for Delivery vs Payment that promises to offer increased efficiency and resilience. Our architecture mimics current market structure. For such needs, we added an extra layer of security that allows our solution to comply with the requirements of the regulator while enabling competitive actors to collaborate using the shared registry. It also leaves room for regulation, while still running in a decentralised way with coordinating agents.

We use non-interactive zero-knowledge algorithms, which are cryptographic protocols with numerous applications in the fields of cryptocurrencies. They allow an agent to verify that another agent holds a specific information, while the latter never discloses this information.

For the sake of our experimentations, we had to use very small integers in our protocols. These integers are too small to comply with current security standards in finance, although the architectural principles can be easily transposed with better performing protocols. We present suggestions to improve our proof of concept and our architecture in the last part.

1 Introduction

1.1 Disclaimer

The present document reflects the research on blockains carried with the support of the Lab Banque de France , in application of the Banque de France policy to support the experimentation of innovative solutions based on new technologies, during summer 2017 by the team of the three authors.

It is not a position statement, nor an official approval or an endorsement by the Banque de France of any technology or protocol. The possible application for a Delivery versus Payment system, particularly for a mission critical system such as the RTGS, does not reflect the vision of the Banque de France regarding the future of such systems.

The names of the agents in the proposed scheme were chosen to facilitate the understanding of the reader, and do not reflect the vision of the Banque de France. These names only represent a theoretical representation and do not refer to any existing entity or person.

1.2 Background

The significant growth of the number of financial transactions and the imperious need to secure them involves a continuous rise of the cost of market infrastructures. Moreover, the

[1] Real Time Gross Settlement

historical fragmentation infrastructures where each actor uses to maintain its separate database has resulted in duplicated IT systems and costly reconciliation processes between them. In this context, several public and private financial institutions are exploring the opportunity to leverage blockchain technologies to build a more streamlined, efficient and robust common backbone.

On the one hand, current regulations in most countries requires an institution to monitor the system. It needs easy access to all transactions. However, in most blockchain systems all the information is made public to ensure the security of the ledger. This is a real issue regarding business secrecy and confidentiality since two competitors can have access to each other's transactions.

Finally, the reliability and the trust of the actors in blockchain systems should be based on the protocol itself, instead of relying on a monitoring institution as the current financial system works today. Hiding some information from the rest of the network must not undermine confidence in the system.

1.3 Security and secrecy

In this paper, we propose an architecture for an RTGS system which both complies with current regulations and protects business secrecy.

We aim at protecting the value of all balances and transaction amounts. The proof we created is inspired by the structure of the Schnorr signature protocol.

1.4 Zero Knowledge Proof

Zero Knowledge Proofs are cryptographic methods by which one party can prove another one it holds a secret information, without revealing any clue on the latter. Those methods are particularly interesting in the field of secured financial networks because they are resistant to a man in the middle attack. The parties are called "prover" and "verifier". Guillou and Quisquater proposed an interesting way to picture how Zero Knowledge Proofs work in their work [1].

Any Zero Knowledge Proof must satisfy the properties given below.

1. Completeness: If the protocol is followed with the appropriate inputs, the verifier always accepts the proof.
2. Soundness: There is only a low probability that a malicious prover can trick an "honest" verifier.
3. Zero-Knowledge: The verifier can not learn any information about the secret because of the proof, as long as the prover follows the protocol.

Any protocol which satisfies the first two properties is a proof of knowledge.

A class of protocols, called Σ-protocols, allow us to obtain *Honest Verifier Zero Knowledge* (HVZK). They are of the following 3-move form.

1. The prover sends a message h.
2. The verifier sends the prover a random e.
3. The prover answers the verifier according to e, and the verifier can either accept or reject the proof subsequently.

Σ-protocols allow us to create Honest Verifier Zero Knowledge since we suppose the verifier will not cheat when sending e to try to obtain information on the secret of the prover. "Pure" Zero Knowledge Proof are hard to find since many rely on the generation of random numbers.

Fiat and Shamir proposed in 1986 a technique to create Non-Interactive Zero Knowledge Proof (NIZK) based on Interactive Zero Knowledge Proof like Σ-protocols. We can use the Fiat-Shamir Heuristic to create a digital signature. This method relies on generating the random e by a *random oracle*. The *random Oracle model* relies on the existence of perfectly random oracles. Since it is not possible to create a perfectly random oracle with modern computers, we can use hash functions assuming they can perform as good as a random oracle. This assumption cannot be formally proved. However, the complexity of modern hashes allows us to make this heuristic assumption [2, 3].

2 Theoretical considerations

2.1 Implementing standard protocols

To implement an adequate proof, we experimented two protocols, which inspired us for our proposal.

2.1.1 Schnorr Protocol

The Schnorr protocol is one of the most famous Σ-protocols. It relied on the discrete logarithm problem and was introduced by Schnorr in 1991 [4].

2.1.2 Guillou-Quisquater Protocol

The Guillou-Quisquater Protocol was first introduced in 1988 by Louis Guillou and Jean-Jaques Quisquater [5]. It is a practical implementation of the scheme of Fiat-Shamir.

We used this protocol to implement our first Non-Interactive Zero-Knowledge utilising the scheme of Fiat-Shamir.

2.1.3 1-bit Schnorr Signature

The 1-bit Schnorr signature is a variant of the regular Schnorr protocol. The sole difference is that the choice of the random sent by the verifier is restricted to one byte. This version of the Schnorr protocol can be proven perfect Zero-Knowledge, but it offers inferior security since the probability for a corrupted prover to guess the random number is $\frac{1}{2}$.

It offers several advantages, as it is as easy to implement as the regular Schnorr protocol and can be easily converted to NIZK using the scheme of Fiat-Shamir. By repeating several times the proof, we can improve the security exponentially. However, repeating the proof leads to the same problem as in the conventional Schnorr protocol, which is that it can only be proven HVZK [6].

2.2 Current implementation

We were inspired by the Schnorr signature and implemented the following protocol. The main idea is to perform two simultaneous non-interactive Schnorr signatures using the same random given by the same Oracle. Because the verifier must be considered honest, we have to add an extra layer of security.

A non-interactive scheme of the Schnorr Protocol First of all, here is the "standard" Schnorr protocol we decided to work with, inspired by the one described by CRAMER, RONALD and DAMGÅRD [7]. Let $(p, q) \in \mathbb{N}^2$ two prime integers where q divides $p - 1$. $g = p^q$. Let P be the prover and V the verifier. P has chosen a secret $s \in \mathbb{Z}_q^*$ at random and $h = g^s \mod p$ is public. Using the FIAT-SHAMIR scheme, we introduce O the oracle.

1. P chooses a random $r \in \mathbb{Z}_q^*$ and calculates $k = g^r \mod p$.
2. O chooses a challenge $e \in \mathbb{Z}_{2^t}$ at random. $2^t < q$.
3. P sends its proof $\Pi = r - es \mod q$ to V. V performs the following calculation : $k' = g^{\Pi} h^e \mod p$. Iff $k' = k'$ then V accepts the proof.

This protocol has been proven complete, sound and zero-knowledge in the random oracle model [8].

Our variant We decided to run two signature checks in parallel, with a shared random chosen by a single oracle. The generation of the random e requires three elements to improve security. We called these elements the three keys to the signature.

1. The first key is a random number generated by the oracle itself.
2. The second key is the time-stamp which ensures a unique seed to generate a random number.
3. The third key is the exact amount of the transaction.

The first key allows our protocol to remain in the random oracle model. The second key prevents agents from trying to re-use e later on. The third key is only held by the agents involved in the transaction and by the Market Authority(described later). Altogether, it makes it very hard for an enemy to correctly guess e.

Since our variant is essentially two simultaneous Schnorr signature protocols with a shared oracle, our proof verifies the soundness and completeness properties of ZKP. If the oracle's first key is generated in the random oracle model, our proposal is also zero-knowledge in the FIAT-SHAMIR scheme.

We will only assume, due to the fact that we program customers to perform a restricted set of actions, that our proposal is HVZK.

3 Implementation and architectural justification

3.1 Architectural choice

3.1.1 Ethereum Quorum

For our prototype we had the opportunity to choose from a large variety of blockchains protocols, but we settled for Ethereum Quorum (or Quorum). Quorum is a second generation blockchain protocol with permissions built from a fork of the original Ethereum source code. It uses proof of work for mining new blocks in the chain, as well as an advanced voting system which can force a specific signature to validate an action and commit it into the ledger. The features that have driven our decision are the ones that follow.

- Native smart contract
- The Ethereum virtual machine
- Constellation private channels between nodes

Another important reason for choosing Quorum is that it is built by JPMorgan Chase with the support of Microsoft. By choosing Quorum, we can not only do our implementation of a Zero Knowledge Proof for transactions but also study the maturity of a solution built by major actors of the blockchain ecosystem. Finally, Quorum uses the GNU Lesser General Public License v3.0, which enables us to work freely and fork the original protocol for our prototype.

3.1.2 Smart contracts

Smart contracts are a feature that was created to achieve greater Turing completeness and create value chain ecologies on networks where the chain is deployed. The use of smart contracts in our prototype is simple and enables us to do all the needed operations on the chain and not in a tied program.

By themselves, smart contracts are code instructions stored in the chain, and that can be called and executed on the Ethereum Virtual Machine. They can deploy an extensive range of features. An example is the ability to deploy watchers on the blockchain network. This way a regulatory authority can build a specific blockchain tool and have it deployed by private partners. This way the regulatory authority can still wholly monitor the transactions transiting within its tool as well as having the infrastructural cost of such a tool paid by the private sector.

3.1.3 Ethereum Virtual Machine

The Ethereum Virtual Machine or EVM is a distributed virtual machine that focuses on providing a safe and sound runtime environment where the smart contracts can be executed. For our model, the permissioned EVM[2] of Quorum enables us to execute smart contracts in an environment, we control, or at the least, we can keep an eye on.

3.1.4 Private channels

Using cryptography and segmentation, Quorum allows us to hide critical information to secure our proof system [9]. The private channels have been a fundamental element in the realisation of the prototype, and it would have been less secure without the possibilities they open.

3.2 Programming language used and library

3.2.1 Solidity

Solidity is a high-level language native to Ethereum; it is contract oriented. It aims specifically to be executed on the EVM.

3.2.2 JavaScript

JavaScript is a dynamic object-oriented interpreted language. We used it for most of the off chain development.

3.2.3 Keccak-256

Solidity uses Keccak-256 [10] hash function. It does not follow the FIPS-202 [11] because it has been finalized in August 2015, the same month of the first release of Solidity programming language.

[2]Ethereum Virtual Machine

4 Our implementation on Quorum

Our Proof system

The tokens which define the balance of each client are defined into a single smart-contract named **Bank**. It includes all the functions required to manage the zero-knowledge proof system and is designed to simplify exchanges between **MarketAuthority** and **Client** smart-contracts. This paper describes these functions and is inspired by the work of Christian Ludkvist [12]. The blockchain records the output of these functions as logs which can be later processed to ensure the security.

4.1 Constellation and Quorum

Quorum Quorum is a fork of the Ethereum open-source project to provide features which allow to make financial applications and taking advantages of Ethereum smart-contracts and blockchain principles. It has been made by J.P. Morgan Chase & Co., a U.S. banking and financial services holding company.

Constellation Constellation is a network of nodes designed to ensure the security of exchanged information. It implements two components named *Transaction Manager* and *Enclave*. *Transaction Manager* manages transaction privacy and the *Enclave* leverages cryptographic techniques for transaction authenticity, participant authentication, and historical data preservation.

Once these two technologies are gathered, they provide a permissioned implementation of Ethereum that supports transaction and contract privacy. In this paper, we used Quorum features to ensure the privacy of critical transactions which secures our proof system.

4.2 Algorithm Proposal

Goal In this section, we expose our proposal to implement a secured proof system which takes advantage of Quorum features to ensure the privacy of critical transactions.

First, we will define several algorithms and principles to implement our proof system in a RTGS[3].

4.2.1 Requirements

1. **MarketAuthority** is an Ethereum smart-contract,
2. **Client** is an Ethereum smart-contract,
3. *client* is an internal data member representation of a client in a given smart-contract,
4. **Bank** is an Ethereum smart-contract,
5. **Oracle** is an Ethereum smart-contract,

4.2.2 Market Authority

Goal **MarketAuthority** is a smart-contract designed to manage the entrance of new service's customers in the market. It will compute and store unknown parameters to run our proof system.

[3]Real Time Gross Settlement System

Market Authority A **MarketAuthority** is composed of a *client* list $Clients$, a list of secret numbers $Secrets$ and two proof system parameters, P and Q.

Entrance To manage entrance, the **MarketAuthority** will compute and store two specific numbers based on P and Q. The first number is named $Proof$ and the second one is named $Verifier$.

Client definition A *client* is composed of three properties. First, a $Secret$ number which will be used to compute a proof for a transaction. Then, $Proof$ and $Verifier$ computed by the **MarketAuthority**.

Adding a client To add a *client* in the market; we define the following algorithm.

Algorithm 1: AddClient

Data:
$ClientAddress$, which contains the Ethereum address of the new client.
$Balance$, which contains the initial balance of the new client.
Result: A new client added in the client list $Clients$
begin
  if $ClientAddress$ *not in* $Clients$ **then**
    $index \longleftarrow$ randomInt(0, $NumberOfSecrets$)
    $hashBalance \longleftarrow$sha3($Balance$)
    if $index < numberOfSecrets$ **then**
      $Clients(ClientAddress).Secret \longleftarrow Secrets(index)$
      $Clients(ClientAddress).Proof \longleftarrow P^Q$
      $Clients(ClientAddress).Verifier \longleftarrow$
       $Clients(ClientAddress).Proof^{Clients(ClientAddress).Secret}$
      delete ($Secrets(index)$)
      $numberOfSecrets \longleftarrow NumberOfSecrets - 1$

P, Q Let $(P, Q) \in \mathbb{N}^2$ two prime integers where Q divides $P - 1$. Client $Proof = P^Q$

Clients It is a map using $ClientAddress$ as a key and storing a structure which contains three member variables $Secret$, $Proof$ and $Verifier$.

$NumberOfSecrets$ It is the size of the network. It describes how many secret numbers are already computed in the **MarketAuthority** which will finally be the maximum number of *clients*.

Secrets It is an array of secret numbers known only by the Market Authority.

delete Removes a value in a given array at a specific $index$.

randomInt Let randomInt be a sampling function from the uniform distribution of support $|[a, b]|$.

sha3 Computes the Ethereum-SHA-3 (Keccak-256) hash of the arguments.

4.2.3 Bank

Goal **Bank** is a smart-contract designed to register in the public blockchain all the transactions between two service's customers. To keep the privacy of the transactions amounts and the balance of each *client*, it contains tokens instead of the real values.

Bank A **Bank** is composed by a *client* list $Clients$, to keep track of all the **Clients** in the market. It will perform the zero-knowledge verification on all the transactions before updating the balances.

Entrance To manage the entrance in the **Bank**'s *client* list $Clients$, the **MarketAuthority** will request to add a new *client*.

Client A *client* is composed of seven properties. First, a token, $hashBalance$, representing the balance before the transaction. Another token, $hashBalanceAfter$, representing the balance after a given transaction. Two proof numbers to perform zero-knowledge verification, $zkVerifier$ and $zkProof$. An array of proof numbers $proof$ and its size $proofSize$. Finally, a proof token $proofToken$.

Adding a client To add a new *client* we define the following algorithm.

Algorithm 2: AddClient

Data:
$ClientAddress$, contains the Ethereum address of the service's customer.
$Proof$, contains the proof number computed by **MarketAuthority**.
$Verifier$, contains the verifier number computed by **MarketAuthority**.
$HashBalance$, contains the initial hashed balance generated by **MarketAuthority**.
Result: A new *client* added in the *client* list $Clients$
begin
  if $ClientAddress$ *not in* $Clients$ **then**
    $Clients(ClientAddress).hashBalance \longleftarrow HashBalance$
    $Clients(ClientAddress).hashBalanceAfter \longleftarrow HashBalance$
    $Clients(ClientAddress).zkProof \longleftarrow Proof$
    $Clients(ClientAddress).zkVerifier \longleftarrow Verifier$

$Clients$ It is a map using $ClientAddress$ as a key and storing structures of which everyone describes a *client*.

Adding a proof Each **Client** has to provide an array containing his proof numbers to prove that the current transaction has been requested by himself. The **Bank** will verify if every **Client** taking part in the transaction can prove his identity using the proof system.

Computing a new balance We multiply the current $hashBalance$ with the first proof number, and we apply the **sha3** function to it. We obtain a new token which will be used to update the balance once the transaction has been verified.

The following algorithm describes how the proof numbers and the proof token of each *client* are stored.

Algorithm 3: AddProof

Data:
ClientAddress, contains the Ethereum address of the service's customer.
Proof, contains an array of computed proof numbers given by a **Client**.
ProofSize, contains the number of proof numbers in *Proof*.
ProofToken, contains a boolean which indicate if it is a proof token or not.
Result: Add a new proof to a given *client*.

```
begin
    if ClientAddress in Clients then
        if ProofToken is false then
            Clients(ClientAddress).Proof ⟵ Proof
            Clients(ClientAddress).ProofSize ⟵ ProofSize
            Clients(ClientAddress).hashBalanceAfter ⟵
            sha3(Clients(ClientAddress).hashBalance * Proof(0))
        else
            Clients(ClientAddress).ProofToken ⟵ Proof
```

Clients It is a map using *ClientAddress* as a key and storing a structure which describes a *client*.

sha3 Computes the Ethereum-SHA-3 (Keccak-256) hash of the arguments.

Verifying a transaction To perform the verification of a transaction, we use a protocol inspired by a non-interactive version of the Schnorr protocol. We use a "three key system" to unlock the right to perform a transaction. Each key is held by a group of agents and allows to generate a unique random.

- **Current time stamp** The timestamp of the transaction is accessible to every agent of the network. The use of the timestamp prevents the random number generated to be used later.
- **Transaction value** The exact transaction value is only known by the agents taking part in the transaction.
- **Random value** The random value, which performs as a salt, is generated by the oracle itself. It prevents any agent to try to predict the value of the random, even if it holds the other information.

Each one of these three keys allows the generation of the random to perform the verification of the identities.

Algorithm 4: VerifyTransaction

Data:
SenderAddress contains the Ethereum address of the service's customer meant as the sender.
ReceiverAddress contains the Ethereum address of the service's customer meant as the receiver.
E contains a random number generated by the **Oracle** specifically for this transaction.
Result: Returns a boolean indicating if the proof is correct or not.
begin

```
i ⟵ 0
while i < Clients(ReceiverAddress).ProofSize do
    mask ⟵ (1 ≪ i)
    k ⟵ Clients(SenderAddress).zkProof^{Clients(SenderAddress).ProofToken}
    kp ⟵ (Clients(ReceiverAddress).zkVerifier^{(E∧mask)≫i}) *
      (Clients(ReceiverAddress).zkProof^{Clients(ReceiverAddress).Proof(i)})
    if k ≠ kp then
        return (false)
    i ⟵ i + 1
return (true)
```

Clients It is a map using *ClientAddress* as a key and storing a structure which describes a *client*.

Sending amount of money To perform a transaction between two **Clients**, we have to verify if the proofs given by the **Clients** are correct. If the proofs are verified, we can update the balances with the $hashBalanceAfter$ computed before and contained in each *client*. We can define the following algorithm.

Algorithm 5: SendAmount

Data:
SenderAddress contains the Ethereum address of the service's customer meant as the sender.
ReceiverAddress contains the Ethereum address of the service's customer meant as the receiver.
E contains a random number generated by the Oracle specifically for the current transaction.
Result: Updates the client's balances.
begin

```
if SenderAddress in Clients and ReceiverAddress in Clients then
    if VerifyTransaction(SenderAddress, ReceiverAddress, E) and
      VerifyTransaction(ReceiverAddress, SenderAddress, E) then
        Clients(SenderAddress).hashBalance ⟵
          Clients(SenderAddress).hashBalanceAfter
        Clients(ReceiverAddress).hashBalance ⟵
          Clients(ReceiverAddress).hashBalanceAfter
```

Clients It is a map using *ClientAddress* as a key and storing a structure which describes a *client*.

4.2.4 Client

Goal The **Client** smart-contract is the interface between the **Bank** and the service's customer Ethereum node. Its goal is to be the intermediary to perform transactions.

Client In this case, a *client* is defined by six properties. First its *Balance*, which is not encrypted and only available in the **Client** smart-contract. There are two numbers, mandatory to perform transactions, *Secret* and $Value$. The *Secret* is a critical number, unique to each *client*, to compute proof numbers and being able to perform a transaction. The $Value$ is the amount of money of the current transaction, only known by the two parties of the transaction (it can also be known by the **MarketAuthority**). Q and P are two numbers used to perform the transaction as well. Finally, N, the number of repetitions to do to compute a proof and being able to perform a transaction.

Computing a proof number To compute a proof number, a **Client** will use a random number E and its secret number *Secret*. It allows to prove the identity of the service's customer and being able to perform a transaction with another customer. In the end, the **Bank** will be able to verify the identity of each party and perform the requested transaction. To compute a proof, the following algorithm is applied.

Algorithm 6: ComputeProof

Data:
$ProofToken$, contains a proof token computed by the **Client**.
E, contains a random number generated by the **Oracle** specifically for the current transaction.
Result: Returns a proof token.
begin
  return $(ProofToken - E * Secret)$

Secret It is a secret number known only by the **MarketAuthority** and the current **Client**. It has been provided by the **MarketAuthority**.

Computing transaction proofs To perform a full transaction, a **Client** has to compute several proof numbers. The number of repetitions N has been defined by the **MarketAuthority** at the creation of the **Client** smart-contract. The following algorithm describes the procedure.

Algorithm 7: ComputeTransactionProof

Data:
$ProofToken$, contains a proof token computed by the **Client**.
E, contains a random number generated by the **Oracle** specifically for the current transaction.
Result: Returns an array of size N containing proof numbers.
begin
  $proofs \longleftarrow$ An empty array
  $i \longleftarrow 0$
  while $i < N$ **do**
    $mask \longleftarrow (1 \ll i)$
    $proofs(i) \longleftarrow$ ComputeProof($ProofToken$, $(E \wedge mask) \gg i$)
    $i \longleftarrow i + 1$
  return $(proofs)$

N It is the number of times a **Client** has to compute a proof number to prove its transaction.

Computing transaction proof token All the transactions need a unique proof token. The following algorithm describes how to compute it. It has to be randomly chosen between 2 and $Q - 1$.

Algorithm 8: ComputeTransactionProofToken

Result: Returns a random number between 2 and $Q - 1$.
begin
  return (randomInt(2, $Q - 1$))

Q Let $(P, Q) \in \mathbb{N}^2$ two prime integers where Q divides $P - 1$.

randomInt Let randomInt be a sampling function from the uniform distribution of support $|[a, b]|$.

4.2.5 Oracle

Goal The **Oracle** has to keep secret the function which generates random numbers for transactions. If the random function is known by someone or controlled by someone, he will be able to anticipate the generation of numbers and predict which number will be generated for a specific transaction. Knowing the next generated number, he will be able to pretend to be someone else.

Oracle An **Oracle** smart-contract knows its owners and can recognize if the **Client** who calls it is its owner or not. It also knows if all the owners have retrieved their random number. When the random number has been retrieved by both clients, The **Oracle** destroys itself.

Computing a random value As it has been said earlier, the random function is only known by the trusted party which has to be the **MarketAuthority**. According to this assumption, we can write the following algorithm.

Algorithm 9: ComputeRandomValue

Data:
$Timestamp$, contains the current timestamp of the transaction.
$Value$, contains the amount of the transaction.
Result: Returns a random value computed using $Timestamp$, $Value$ and a random value.
begin
  return (sha3(randomInt(1, $MaxInt$), $Timestamp$, $Value$)

$MaxInt$ It is the maximum integer on the current system architecture used to run the computation.

randomInt Let randomInt be a sampling function from the uniform distribution of support $|[a, b]|$.

sha3 Computes the Ethereum-SHA-3 (Keccak-256) hash of the arguments.

5 Architecture Proposal

5.1 Simplified Transaction Schematics

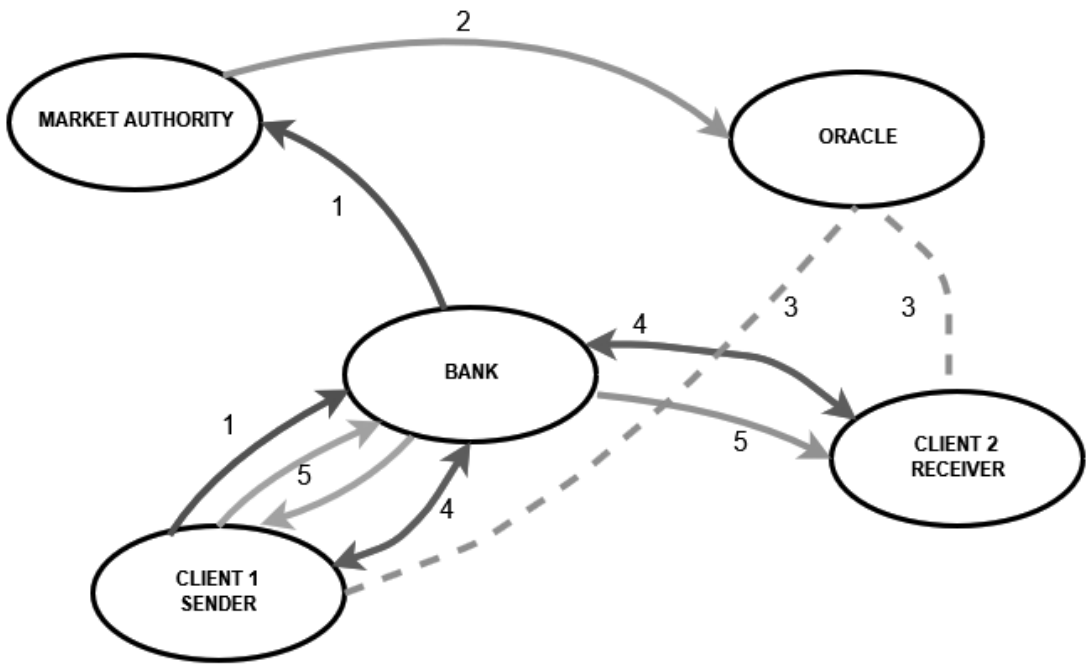

Figure 1: Simplified Transaction Scheme

We can make the analogy for our transaction scheme of an unforgeable form with signatures, official stamps, and receipts for the parties involved in the proceedings.

5.2 Full Transaction Schematics

5.2.1 Requires

Assumptions Each schematics below assumes that each service's customer has a node in the Quorum network and knows its own Constellation public key to perform private transactions. Also, we assume that each service's customer that wants to make a transaction can communicate with the other party using an external communication channel. This external channel has to be secured in order to not compromise critical information like the amount of the transaction and the two parties identities.

Security **Client**, **MarketAuthority** and **Oracle** smart-contracts have to be created as private smart-contracts using PRIVATE FOR Quorum functionality. The **Bank** smart-contract has to be created in the public blockchain. A useful thing to do is to add a Constellation node, owned by a trusted party, in each PRIVATE FOR declaration in order to allow the trusted party to keep an eye on each transaction. A trusted party could be a central bank for instance.

Registration Each **Client** has to be registered in the **MarketAuthority** by the trusted party. The trusted party should register each **Client** in the **Bank** allowing them to perform transactions.

5.2.2 Proof Token Generation

Step 1 Each service's customer asks its **Client** smart-contract to create a proof token. As the **Client** smart-contract is PRIVATE FOR themselves, all the Ethereum transactions between the service's customer Quorum node and the **Client** smart-contract will be encrypted. The proof token will be used in the next transaction as a unique number to compute a new proof using our proof system.

Step 2 Each transaction needs a random number as defined in our proof system which has to be generated under trusted conditions. A **Client** will ask the **Bank** to create a new **Oracle** smart-contract to generate a new random number.

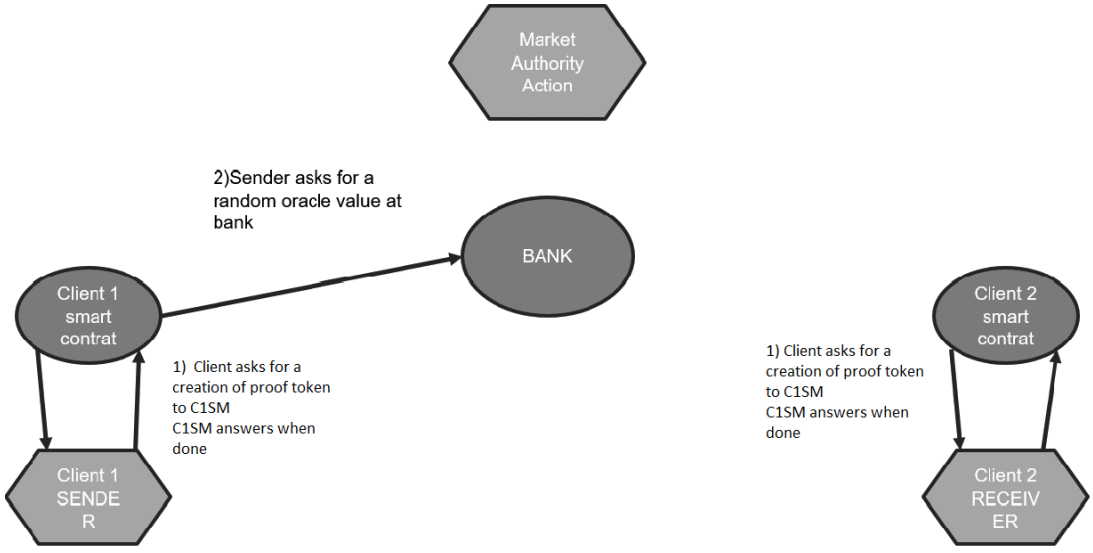

Figure 2: Proof Token Generation

5.2.3 Oracle Creation

Step 3 The **Bank** smart-contract, which received the **Oracle** creation request, sends a [4]global event to create an **Oracle** for the given service's customer Ethereum addresses. The **MarketAuthority** will catch the event and create a new **Oracle** smart-contract PRIVATE FOR the two given **Clients** taking part in the transaction.

Step 4 The **MarketAuthority** is always listening on **Oracle** creation event. Once it receives this event, it creates a new **Oracle** smart-contract which is PRIVATE FOR the service's customers taking part in the current transaction. When the **Oracle** is correctly mined, the **MarketAuthority** will send back the Ethereum address of the smart-contract to the **Bank**.

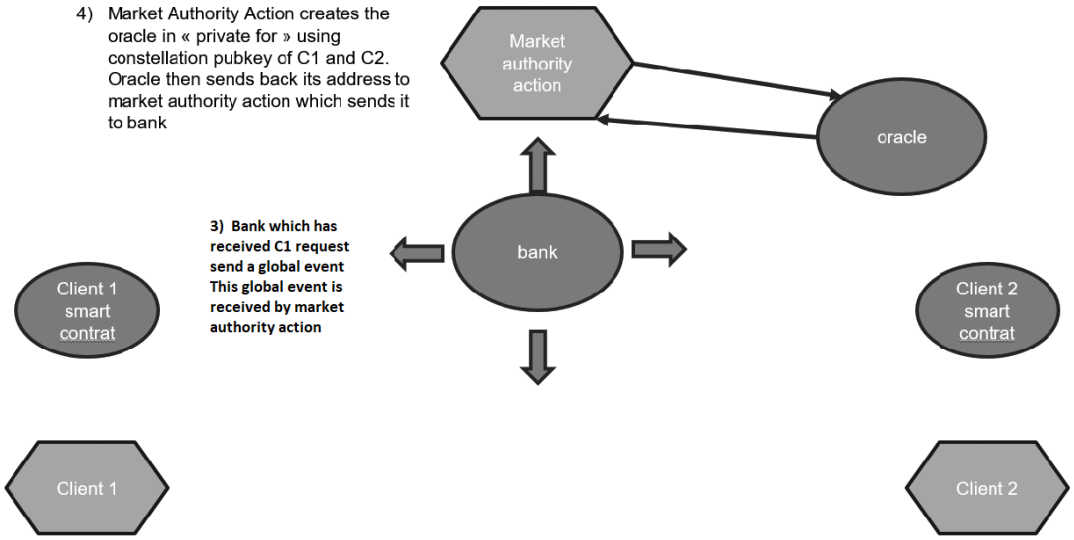

Figure 3: Oracle Creation

5.2.4 Random Number Generation

Step 5 Once the **Oracle** has been created, the **Bank** will broadcast the Ethereum address using a global event. As the **Oracle** smart-contract is PRIVATE FOR two specific service's customers, only them can interact with it. The **Clients** have to listen to this event to catch the Ethereum address up.

Step 6 Each **Client** sends a request to the **Oracle** and gets a new random number. During this process, critical information is exchanged. The amount of the transaction is needed to compute the random number. As the amount of the transaction is a critical information, the **Clients** perform a PRIVATE FOR transaction with the **Oracle** smart-contract. Only the two service's customers will have access to this information in clear; otherwise, it will be encrypted.

Step 7 Each **Client** requests the **Oracle** to get his unique random number and finally computes his transaction proofs. Once both of them got the random number, the **Oracle** automatically destroys itself.

[4]Means that all the ethereum nodes listening this event will receive it

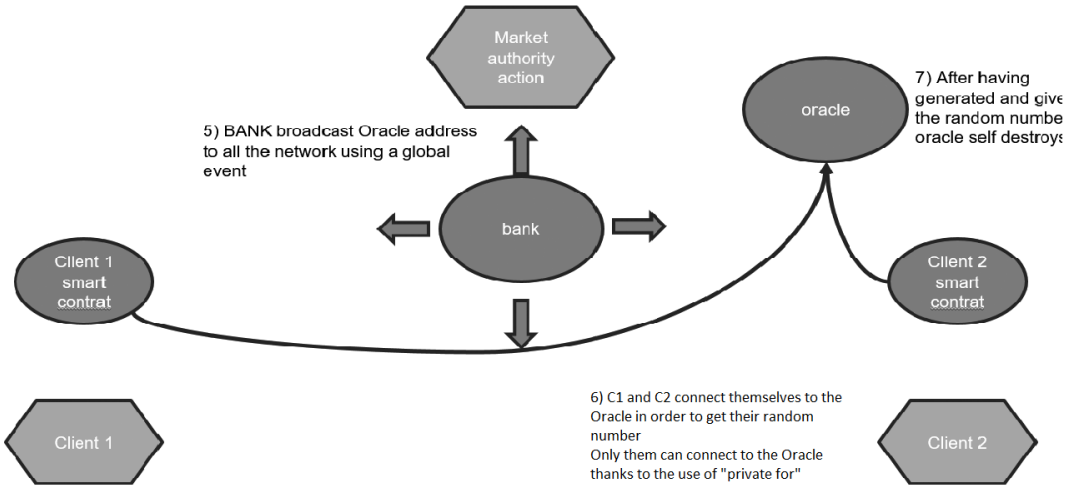

Figure 4: Random Number Generation

5.2.5 Proof Token Submission

Step 8 To perform a transaction between two service's customers, each one has to compute a proof token. To do that, they have to request it from their **Client** smart-contract and finally submit it to the **Bank**. This step of the procedure is also a private Ethereum transaction.

Step 9 Once a **Client** got its proof token, it has to declare it to the **Bank**. It will use the proof tokens to verify the proof of each **Client** and perform the requested transaction. All the Ethereum transactions with the **Bank** are public. This enables every service's customer on the network to verify the proofs by himself.

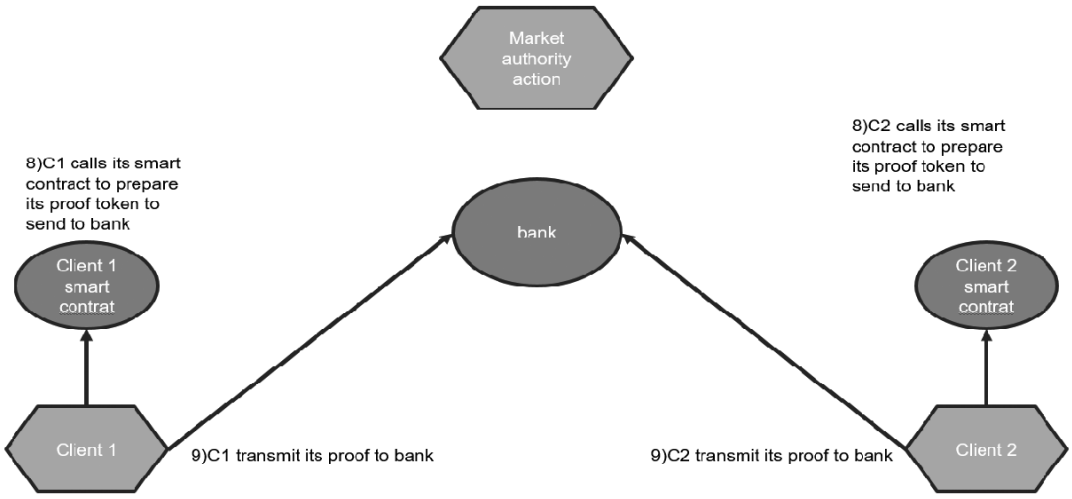

Figure 5: Proof Token Submission

5.2.6 Transaction Proof Submission

Step 10 As for the Proof Token Submission, each service's customer will request proof numbers to their **Client** smart-contract and declare them to the **Bank** as well.

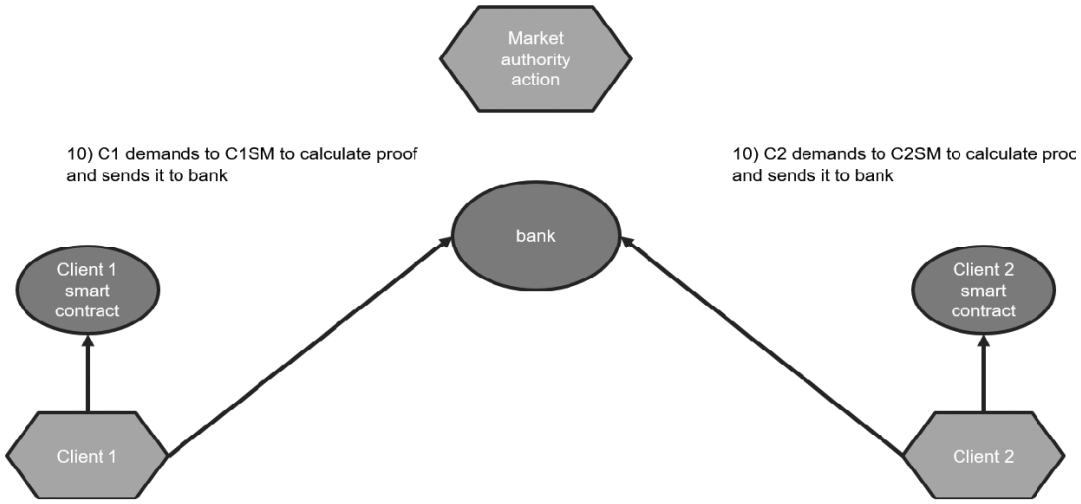

Figure 6: Transaction Proof Submission

5.2.7 Performing Transaction

Step 11 The *sender* service's customer will send a request to the **Bank** to perform the transaction and update the balances.

Step 12 Once the **Bank** receives the request from *sender*, it verifies the proofs of each service's customer taking part in the transaction.

Step 13 If all the proofs are correct the **Bank** will the send a global event to broadcast the updating of the balances. Each **Client** taking part in the transaction will catch the event up.

Step 14 All **Clients** receiving the **Bank**'s global event and taking part in the transaction will update their balance. The difference with the **Bank** balances is that the **Client** balances are not hashed. Each **Client** knows its exact balance. The **Bank** doesn't know either the amount of the transaction nor the balance of each **Client**. It only notifies the current hashed state of the **Clients** balances.

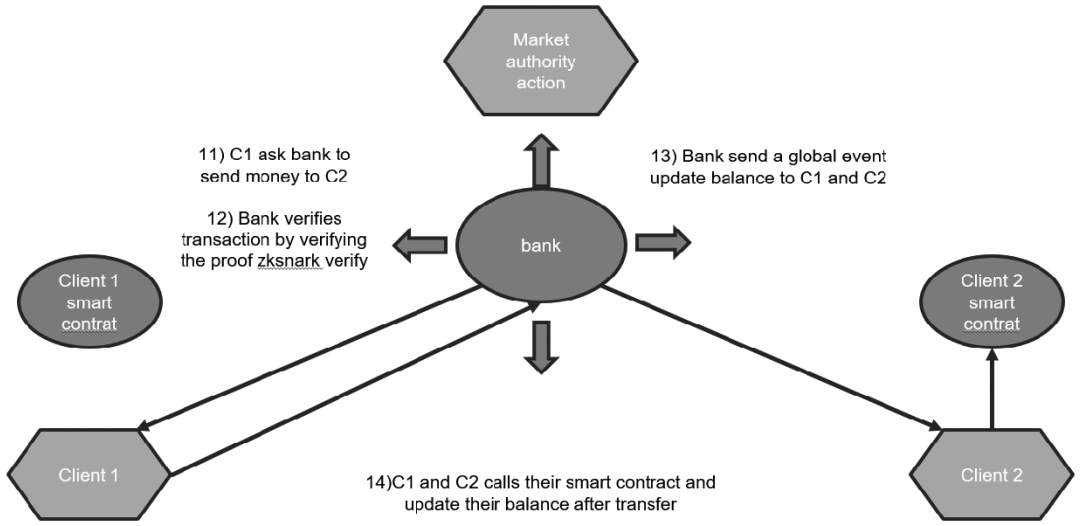

Figure 7: Performing a Transaction

5.2.8 Conclusion

Integrity In our proof system, even if neither the service's customers nor the bank know about the balances of each other and the amount of the transaction, they can prove that each of these is correct. Unrolling the whole blockchain, each service's customer can verify if the current state of another one is valid or not.

Security The security is ensured by our proof system. Indeed, no critical information is exchanged between the public space of the network and the private space. Only authorised smart-contracts can interact with the **Clients** and exchange critical information.

Business anonymity Finally, this proof system permits to keep the private aspect of each transaction in a business meaning. Any service's customer can do business with another service's customer without exposing the transaction amount it is applying for its [5]customer. Keep in mind that all the smart contracts are created by a trusted party which owns the **MarketAuthority** smart-contract.

6 Proposed improvements

6.1 Clients management

Multiple bank scheme

To be able to deal with multiple banks in our system we need to build another layer of nodes dedicated to the banks. Such a layer will do the usual work of routing queries from the client to the bank that will then transfer such queries to the central authority and start the zero knowledge system as described in our architecture.

6.2 Security improvement

6.2.1 Larger prime numbers

Due to the limited calculation power at our disposal, we used small prime numbers for our implementation. In a more advanced build, our protocol will need larger prime numbers.

6.2.2 More efficient protocols

Our implementation is using the method of the discrete logarithm to put both parts of the transaction to the test. The more recent zero-knowledge proof implementation is using the elliptic curve method as first described by Miller [13] and Koblitz [14] .

New methods, such as zk-SNARKs who is using elliptic curve method are supposed to have a better performance in both security and protocol speed [15] [16].

7 Author's proposed application: Delivery versus Payment

In order to build a system similar to a real Delivery versus Payment system the protocol still needs some upgrade.

- We need to be able to differentiate between cash and securities accounts. To do this, a possible solution is to bound two separate smart contracts to each node on the network. Each one of those nodes can be specialised in securities or cash.

[5]Meant as a business customer

- In order to be able to store the various types of cash and securities, we should use mapping types of structures as they exist in [6]Solidity. Furthermore, we can reuse that solution for storing our settlement system
- Another important feature if we want to build a real delivery versus payment system is the implementation of a liquidity saving mechanism. In such a system it is possible by design to delay the full payment of securities. Such a system can be built by adding to every agent a smart contract responsible for keeping the historicity of the soon to be settled transaction. Such a smart contract would be, obviously, inaccessible by any other entity than it interfaces thanks to private for specifications.
- By reducing the gas cost of the various operation on the core system and implementing a voting system to reach consensus as defined in the block voting function, it will be possible to speed up the process even more and remove the gas limitation existing on the prototype. In such a system only a majority of Market authority nodes (which are typically run by central banks) can vote and validate a block and therefore execute the zero-knowledge verify command.
- Lastly a soon to be released feature into quorum detailed in issue 142 of the Quorum repository is supposed to add the ability to add new members to a private smart contract. Once this feature is released some of the architecture will have to be reworked

However, the impact of a greater sophistication of the distributed system and its abilty to scale remains uncertain. Moreover, from a business perspective, the benefits, risks and required transformations of business processes are still not clearly evaluated would require more in-depth studies.

8 Conclusion

The costs of IT infrastructure in the financial sector are rising due to the significant increase in the number of transactions and the rise of security standards. Modern technologies, such as blockchain and zero-knowledge proof protocols may help address this issue.

At the Lab Banque de France, we implemented an experimental RTGS architecture on a Quorum platform based on HVZK proofs to demonstrate the potential of such methods. Our architecture is highly inspired by the current structure of financial markets and may meet the regulations needs currently enforced regarding the control of the markets and the role of central banks.

Although we could not use the most efficient Zero-Knowledge protocols due to limited time and computing power, our architecture is easily upgradable and scalable. With our proposed improvements, such an architecture could be used for an efficient Delivery versus Payment systems such as Target2Securities in the Eurozone.

The writer's opinion is that Zero-Knowledge proofs will be of growing importance in the next years and that blockchain -based architectures will allow the development of safe and reliable banking networks. Indeed, the research and development of such architectures will probably constitute a decisive element for the understanding and the regulation of markets by Central Banks and financial institutions shortly.

[6]The Ethereum programming language for smart-contracts

9 Acknowledgements

We would like to thank Mr Thierry Bedouin and Mr Guillaume André for their support.

We would also like to thank Jeanne Schpilberg Katz, Philippe Azoulay and Marc Fasquelle for their help during the review of this paper.

We would like to thank our supervisors (alphabetically): Mr Jean-Baptiste Baraton, Mr Jean-Marc Boucher, Mr Kamal Bouhouch and Mr Alexandre Le Douaron at the Banque de France.

References

[1] J.-J. Quisquater, M. Quisquater, M. Quisquater, M. Quisquater, L. Guillou, M. A. Guillou, G. Guillou, A. Guillou, G. Guillou, and S. Guillou, *How to Explain Zero-Knowledge Protocols to Your Children*, pp. 628–631. New York, NY: Springer New York, 1990.

[2] A. Fiat and A. Shamir, "How to prove yourself: Practical solutions to identification and signature problems," pp. 186–194, Springer-Verlag, 1987.

[3] M. Bellare and P. Rogaway, "Random oracles are practical: A paradigm for designing efficient protocols," in *Proceedings of the 1st ACM Conference on Computer and Communications Security*, CCS '93, (New York, NY, USA), pp. 62–73, ACM, 1993.

[4] C. P. Schnorr, "Efficient signature generation by smart cards," *J. Cryptol.*, vol. 4, pp. 161–174, Jan. 1991.

[5] L. C. Guillou and J.-J. Quisquater, *A Practical Zero-Knowledge Protocol Fitted to Security Microprocessor Minimizing Both Transmission and Memory*, pp. 123–128. Berlin, Heidelberg: Springer Berlin Heidelberg, 1988.

[6] I. Damgård, "On σ-protocols," *Lecture Notes, University of Aarhus, Department for Computer Science*, 2002.

[7] R. Cramer, I. Damgård, and B. Schoenmakers, "Proofs of partial knowledge and simplified design of witness hiding protocols," in *Advances in Cryptology—CRYPTO'94*, pp. 174–187, Springer, 1994.

[8] Y. Seurin, "On the exact security of schnorr-type signatures in the random oracle model," *Advances in Cryptology–EUROCRYPT 2012*, pp. 554–571, 2012.

[9] J. M. Chase, "Quorum whitepaper." `https://github.com/jpmorganchase/quorum-docs`, 2016. Online; Accessed: 2017-09-24.

[10] N. I. of Standards and Technology, "Universal mobile telecommunications system (umts);lte; specification of the tuak algorithm set: A second example algorithm set for the 3gpp authentication and key generation functions f1, f1*, f2, f3, f4, f5 and f5*; document 1: Algorithm specification." `http://www.etsi.org/deliver/etsi_ts/135200_135299/135231/12.01.00_60/ts_135231v120100p.pdf`, 2014. Online; Accessed: 2017-09-24.

[11] N. I. of Standards and Technology, "Sha-3 standard: Permutation-based hash and extendable-output functions." `http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf`, 2015. Online; Accessed: 2017-09-24.

[12] C. Lundkvist, "Introduction to zk-snarks with examples." `https://media.consensys.net/introduction-to-zksnarks-with-examples-3283b554fc3b`, 2017. Online; Accessed: 2017-08-18.

[13] V. S. Miller, *Use of Elliptic Curves in Cryptography*, pp. 417–426. Berlin, Heidelberg: Springer Berlin Heidelberg, 1986.

[14] N. Koblitz, "Elliptic curve cryptosystems," *Mathematics of computation*, vol. 48, no. 177, pp. 203–209, 1987.

[15] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza, *SNARKs for C: Verifying Program Executions Succinctly and in Zero Knowledge*, pp. 90–108. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013.

[16] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza, *Scalable Zero Knowledge via Cycles of Elliptic Curves*, pp. 276–294. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014.