Master 1 SFPN

Post Quantum Blockchain

TEAUDORS Mickaël
mickael.teaudors@etu.upmc.fr

VALERI Yoann
yoann.valeri@etu.upmc.fr

29 mai 2019

Contents

1. Introduction

Quantum computers, the perfect association between computer science and physics, were still at the stage of prototypes some years ago but their development is now progressing quickly. With this new technology on the brink of disrupting the current crypto-systems, computers scientists have to find an answer to this serious threat.

Our semester project, called ***Post-Quantum Blockchain***, is an approach towards the post-quantum world. It is centered around the concept of BlockChain, a recent technology used in 2008 by Satoshi Nakamoto for the Bitcoin crypto-currency[1]. Even though it was created in 2008, theories in 1991 described a way to create a cryptographically secured chain of blocks[7], each containing a single transaction data and a timestamp that couldn't be tampered with. At its highest point, on the 17th of December 2017, one bitcoin was worth 19,290 USD[1].

This technology, though very recent, has seen an increase in its use throughout the world, to effectively secure lists of transactions and exchanges. Especially, it is the basis of all cryptocurrencies nowadays, and is also used by companies to trade with their suppliers, partners and customers, because of its ease of use and efficiency. For these reasons, it is very important that the security of Blockchain is ensured, especially regarding the rise of quantum computers, requiring new cryptographic algorithms.

To ensure the security of BlockChain, two protocols are used : the *Proof-Of-Work (PoW)*, to check the validity and write transactions into the Blockchain, and the *Signature*, to authenticate the issuers of transactions. The implementation of these protocols varies for each usage, but some features must be present : the *Proof-Of-Work* should allow users to write to the Blockchain valid blocks, while safeguarding "against malicious parties who possess less than 50% of the computational power of the network from creating an alternative history of transactions"[1]. On the other hand, the *signature* protocol should fulfil the role of any signature algorithm : verify that a message (or in the case of the Blockchain, a transaction) was effectively issued by a specific user, hasn't been tampered with, and that it is still valid.

Therefore, it is primordial that we look at how these protocols are implemented, and how they fare against current computers, but mostly how they will hold against quantum computers. In section 3.2 and 3.3, we will give details about the current implementations of Blockchain and the way we secure them, including their mathematical footholds and how the quantum computers could break the security of Blockchain using the already existing algorithms [6, 11]. In section 4, we will talk more precisely about the subject we were assigned to, present the tools used and the objectives. The main objective is the incorporation of post-quantum signature algorithms in the already existing Blockchain manipulation tool *Hyperledger Fabric*. In spite of the fact that our work is based on one specific post-quantum program, we want the integration of another post-quantum program to be quick and easy. After, we will go through our work, what we modified and implemented. We managed to incorporate two post-quantum signature algorithms: GeMSS[2] and MQDSS[3]. Finally, we will talk about the results of these modifications, especially regarding the practical time and memory usage.

[1] *https://www.blockchain.com/stats* for more information regarding the bitcoin

2. Acknowledgements

We would like to thank Research Director Jean-Charles Faugère[2] and associate professor Ludovic Perret[3], both members of the PolSys[4] joint project-team between the INRIA (National Institute for Computer Science and Applied Mathematics), the CNRS (National Centre for Scientific Research) and Sorbonne University's LIP6, for their help as project supervisors.

Particular thanks to PhD student Jocelyn Ryckeghem[5], for his help in implementing various algorithms and his answers to our questions regarding these algorithms.

3. State of the art

3.1 Modern cryptography

3.1.1 Digital signature

A digital signature is a mathematical process which objective is to prove the authenticity and the integrity of a specific information. In order to do that, the person who signs a message must prove he knows a secret information, called *secret key*. The recipient can check the validity of the signature using a public information associated to the signatory, called *public key*. The *public key* is computed using the *secret key* so that the verification can be done by any peer. Both keys are generated by the same entity which publish or give its *public key* to the outside, while making sure to keep the *secret key* for itself.

A digital signature is then produced through the three following algorithms :

- Keys generation : Probabilistic function G which generates a secret key (generally, uniformly at random) and compute the associated public key such that $(sk, pk) = G()$ with sk the secret key and pk the public key associated.
- Signature : Probabilistic function S which produces a signature s, computed from a person's message m and his or her secret key sk such that $s = S(m, sk)$.
- Signature verification : Deterministic function V which verifies the authenticity of a signature s by using the message m and the signature owner's public key pk such that $a = V(m, s, pk)$ with $a \in \{accepted/rejected\}$.

Finally, the digital signature scheme is a triplet of these 3 functions (G, S, V).

[2]https://www-polsys.lip6.fr/ jcf/
[3]https://www-polsys.lip6.fr/ perret/
[4]https://www-polsys.lip6.fr/
[5]https://www-polsys.lip6.fr/ ryckeghem/

The security of digital signatures can be improved by using several signatures at the same time. We can therefore introduce the idea of an hybrid-model where two different signature algorithms are used. Both signatures are then concatenated to form the final signature. Let us denote "||" the concatenation operator, sk_1, sk_2 two secret keys such that $sk_1 \neq sk_2$ and S_1, S_2 two signature algorithms, then we have the following signature for a message m :

$$s = S_1(m, sk_1) \,||\, S_2(m, sk_2)$$

3.1.2 Hash functions

Hash functions are used to produce fixed-size (n) data from data having arbitrary length. The size of the output depends on the hash function used. We generally define a hash function along with a hash scheme h (SHA, MD, ...) by :

$$\begin{aligned} H\colon \{0,1\}^* &\to \{0,1\}^n \\ x &\mapsto h(x) \end{aligned}$$

This kind of functions are supposed to be one way functions and therefore ensure the following properties :

- **preimage resistance** : given one output y, it is hard to find x such that $H(x) = y$
- **second preimage resistance** : given one input x, it is hard to find x' such that $H(x) = H(x')$
- **collision resistance** : it is hard to find (x, x') such that $x \neq x' and H(x) \neq H(x')$

The above properties are mandatory to ensure that a hash function is good cryptographycally speaking. However, in the *PoW* algorithm we talked about earlier, the main property we want a hash function to assure is the preimage resistance.

3.2 Blockain

3.2.1 Structure

As indicated by his name, a Blockchain is a set of blocks, each block being linked to the previous one. The first block of the list, the so-called *genesis block* is the only one not being linked to another block. Concerning the composition of the blocks, we can distinguish two main parts in a block : the header containing the data identifying the container, called the metadata, and a list of transactions. Usually, the size of a block in the Bitcoin Blockchain is almost 1.09 Mb while the number of transactions is between 1500 and 2700 per block[6].

[6] *https://www.blockchain.com/stats* for more information regarding the bitcoin

Among the foremost information in the header, there are :

- A hash of the entire previous block of the chain, which is used as the link to the previous element of the chain, meaning that a block doesn't have a link to its successor
- A random number called nonce, used for the proof-of-work, and which utility is described below
- A timestamp, highlighting the date of the block's creation

3.2.2 Proof of work and Signature

The Proof-of-Work protocol is a mathematical problem whose result is easily verifiable whereas the computing process is energy-consuming because it only can be solved with brute force attacks. In the Bitcoin Blockchain, the difficulty is set up so that it takes an average of 10 minutes to solve[1]. The problem used in it is called hashcash and works as follow : find a block for which its hash value is under a defined threshold. As hash functions are one-way functions, then the only solution to solve the problem is to try all the possibilities. To find a block that satisfies the condition, the block must be modified between each attempt. This is why the header contains a random number which is incremented until the hash value produced is valid. The first person finding a header satisfying the condition shares its solution to the network and then the block is added to the chain.

The other protocol that interests us more in this project is the signature one. The signature is created using one's secret key, and verified by the use of that same entity's public key. It allows for a person to identify the source of a document (in our case a transaction or a message), ensure it wasn't modified by a third-party and that the signatory can't deny signing the document later on. Using signatures on the transactions, their validity is questioned before adding a block on the chain, and if the transaction is legitimate, the transaction is added to the list of that block. The transaction is considered legitimate if the issuer owns the money he wants to send.

3.2.3 Blockchain technology : *Hyperledger Fabric*

Blockchain is a new way to store information and is commonly used in modern applications. For that purpose platforms exist for which purpose is to provide an API[7] for blockchain-based application. Among the most prominent, there is one open-source project named *Hyperledger* supported by the Linux foundation and IBM.

Hyperledger[8] gathers a lot of different frameworks, all related to blockchain manipulation and development. These tools are mainly developed using the GO, a compiled programming language released in 2009 by Google. It is performance-oriented, built in a way that takes advantage of the modern multi-core architectures. Instead of using threads, GO is based on "goroutines", which work in the same way, but use only 2Ko, contrarely to threads that generaly consume 1Mo. This is a low-level programming language that incorporates the concept of objects, with a syntax close to C. To summarize, the GO combines effectiveness and easiness of understanding.

[7] Application Programming Interface

[8] To know more about hyperledger projects : *https://www.hyperledger.org/projects*

For this project we will use the *Fabric*[9] framework, mostly implemented for smart contracts and companies.

3.3 Quantum threat

Nowadays, our digital security and privacy relies on the cryptographic concepts aforementioned. Considering the computing power and architecture of current computers, the modern protocols suffice to guarantee the confidentiality, integrity and authenticity, which are the main principles a security protocol aims to ensure.

However, quantum computers are emerging. Estimation based on the pace of development of quantum computers are published [1], and experts think that the signature schemes commonly used will be broken in the next decade. The underlying explanation is the existence of many algorithms requiring quantum machines to be executed. A non exhaustive list of these algorithms is given below.

- Grover algorithm[6] :
  Lov Grover, an Indian-American computer scientist, presented in 1996 a theoretical quantum algorithm to perform database research quickly. His probabilistic algorithm allows to find an element in an N-entries database with a complexity $O(\sqrt{N})$. In cryptography, this algorithm can be used to carry out brute-force attacks faster than with current computers, and is a way to reverse hash functions in a reasonable amount of time. This has a huge impact on Blockchain, especially on the Proof-Of-Work process since it consists in reversing a hash function.

- Shor algorithm[11] :
  Peter Shor, mathematic professor at MIT[10], formulated in 1994 an algorithm to solve one of the hardest mathematical problems known today : the factorization of large integers. The problem consists of finding prime factors of a large integer N. A lot of modern cryptographic algorithms base their security on this problem, the most prominent being RSA. However, using a quantum computer, a solution to this problem can be found in polynomial time, that is with a complexity $O(size(N))$, given the function *size* returning the bit size of the input. Shor also devised an algorithm to solve the discrete logarithm problem in $O(E + F)$ with E the complexity of a modular exponentiation (which he also described in his paper) and F the complexity of a quantum Fourier transformation (also described in the same paper).

3.4 The NIST competition

Considering the impending arrival of quantum computers, the whole cryptographic system must be redesigned. A global competition organized by the NIST[11] called "Post-Quantum Cryptography"[12] is ongoing (currently at round 2), with the objective of confronting propositions of quantum-resilient cryptographic signature algorithms. In the scope of this competition,

[9] *https://www.hyperledger.org/projects/fabric*
[10] Massachusetts Institute of Technology
[11] National Institute of Standards and technology
[12] https://csrc.nist.gov/Projects/Post-Quantum-Cryptography

researchers of the PolSys are involded in the collaborative[13] project GeMSS [2], a C/C++ program offering the three main cryptographic functions, namely keys generation, signature and signature verification.

3.5 Post-quantum algorithms

There are currently 9 signature algorithms still in the NIST competition. The algorithms that we were focused on are GeMSS, introduced in the previous section, and MQDSS[3][14]. In this paragraph we present the functioning of these algorithms since we will base an important part of our work on it.

The GeMSS[2] algorithm is based on a former signature scheme called Quartz [9], and improves its security and efficiency on the matter, by producing shorter signatures with a faster algorithm. Quartz uses the *Hidden Field Equations* (*HFE*), a set of multivariate polynomials with a number of variables associated to the security level chosen. Moreover, Quartz uses a variant of *HFE* equations : *HFEv-* for *HFE vinegar and minus*, which are ways to introduce more randomness into the equations. The vinegar corresponds to adding variables, and the minus corresponds to removing equations. GeMSS implements 3 levels of security : 128, 192, 256.

In GeMSS, the keys are derived from a multivariate polynomial $F \in \mathbb{F}_{2^n}$. The public key takes the form of a set of quadratic equations $\{p_1, ..., p_m\}$ and the secret key has the form of a pair of inversible matrix, both with coefficients in $\mathbb{F}_2$. For the lowest level of security(128), the length of the public key is 417408 bytes and the length of the secret key is 14208 bytes. The process of signing is roughly equivalent to computing the roots of the polynomial F. The efficiency of GeMSS lies in the short length of the signatures produced (only 48 bytes for GeMSS 128) and the ability to perform quick signatures verification.

The digital signature proposed by MQDSS is also based on multivariate polynomials over a finite field $\mathbb{F}_q$, with q a prime. Thoroughly, the problem used is called *Multivariate Quadratic*[13] and consists in solving a polynomial equations system over a finite field. Denoting the security level of the algorithm k, the secret key is randomly chosen and is composed of k bits. Then the public key is derived from the secret key and has a length of $k + n\lceil log_2(q)\rceil$, with n the number of equations and q the order of the finite field $\mathbb{F}_q$. The program propose two reference versions for the parameters, the one we will use contains a secret key made of 32 bytes, a public key of 62 bytes and a signature length of 32882 bytes [3][15]. Moreover The best known attack, on classical computers, can be performed with 2^{159} operations in $\mathbb{F}_q$.

4. Subject presentation

As presented above, the current state of the reshaping of the modern cryptosystem is the choice of the new algorithms to use. Through this project, we perform the next step. Considering one quantum-resiliant cryptographic algorithm, we test its compatibility with modern softwares.

[13]Laboratories involved : Polsys, Inria, CNRS, CS, Orange, LVM
[14]MQDSS specification Version 1.1
[15]MQDSS specifications page 53

As we can't test for all the software which use cryptographic functionalities, the choice has been made for us. The software we will work on is the open-source blockchain framework *Hyperledger Fabric*[16] [5]. *Hyperledger Fabric* represents one framework among others available on the Hyperledger platform. Its purpose is to provide basic tools for designing and implementing distributed ledgers.

Our project goal is to modify *Fabric* to incorporate a post-quantum signature scheme. Since we can't work with all post-quantum signature algorithms in the competition, we mainly base our work on GeMSS, provided by our supervisors, and MQDSS, retrieved on the official website. Thoroughly, the modifications we carry out consist in replacing the three functions associated with the signature scheme in *Hyperledger Fabric*. An underlying objective is to find a solution to interface GO with C, since those algorithms are too big to be translated into another language.

The first objective we want to reach at the end of the project is having both *Fabric* native keys and GeMSS keys working in parallel. In this implementation, GeMSS generated keys are in charge of the signature generation and verification protocols while the original computed keys of *Fabric* keep performing the rest of their initials tasks. Then, once this is working correctly, the optimal objective over this project is to reach an hybrid cryptography. That means having the original keys and signature functions of *Fabric* working in parallel of GeMSS's ones. In this model, both signatures are concatenated and both must be checked. The final step would be to utterly replace the original keys with the GeMSS ones but this work requires more time than we have for our project (as they are used for credential creations, nym signatures and other functions).

Once the implementation objectives are reached, our task consists in comparing the ratio between the original performances and the new ones. Adding one key-pair generation will unavoidably increase the execution time and this information must be measured and compared to the original time. On the other hand, we expect the signature verification process to be much faster than their original counterparts. Indeed, GeMSS was created with the goal of verifying signatures almost instantly, and we have to verify that this property is kept even with the interfacing of GO with C.

5. Our work

5.1 Document research

Our subject gathers a lot of mathematical notions we have to understand to carry out the project correctly. Our work is linked with the following concepts : blockchains, signature algorithms, quantum computing and the underlying mathematical principles. In order to organize our thought and our research, we designed a mindmap[17] gathering the main keywords related to our subject.

[16] *https://www.hyperledger.org/projects/fabric*
[17] See appendix 4

Having our subject well defined, we gradually built our bibliography taking into account all the different notions. In order to guide us, we have access to one initial article [1], dealing with the Bitcoin Blockchain and explaining its functioning and limits. This is a recent article, published in 2017, and the concepts are still in use today. The reading requires a strong mathematical background to go through the technical parts, which represent the majority of the document. Especially, it explains and gives mathematical proofs of the weakness of the modern signature algorithms against quantum computers. This document has been a good point of start for the elaboration of our bibliography, since it broaches all the main problematics we discuss through this project. Moreover, it contains a lot of references, that inspired and guided us along this semester.

5.2 Working environment setup

After getting a glimpse over all the notions related to the project, we started by setting up our working environment. We basically have two programs, and we want to use one in the other. Our first task has been to setup the environment for *Hyperledger Fabric*. The program is written in GO, so we installed the newest version at the moment, which is the version 11.5. Concerning *Fabric*, it was enough to make the program works. We cloned the git repository [5] and we were able to run the tests.

On the other hand, we retrieved the source code of GeMSS. The first version we got was made of a blending of C and C++. Making the program execute for the first time has been a bit longer than for *Fabric*, since we had much more dependencies to install. Indeed, GeMSS references different libraries, which are *openSSL*, completed with the *libcrypto* library, the *keccak* library contained in the project *XKPC (Extended Keccak Code Package)* [14] and NTL for the C++ part. Once all the dependencies installed, the environment for GeMSS was done and we could start thinking about the integration with *Hyperledger Fabric*.

The first problematic we faced was the interfacing : how to make a program written in GO and another in C/C++ work together, assuming that none of them can't be translated into the other language in a reasonable amount of time. Fortunately, the GO language includes a package called "CGO" that allows to call C procedures within a GO-written program. Therefore, we can use this GO functionality to make the link between *Fabric* and GeMSS. However, after solving that problem, we faced the problematic that GeMSS is written in C/C++, and the C++ part turns out to be a serious obstacle. Indeed, CGO is compatible with C code but not C++. After some days, we got a new version of the GeMSS program, totally written in C, along with a Makefile to produce a static library from the source code.

From this moment, our environment was appropriate for the smooth run of the project and we could start working on the source code of *Hyperledger Fabric*. In order for our supervisors to have access to our work as we go along, they created a git repository on the platform *Inria*.

5.3 Cryptography in *Hyperledger Fabric*

After installing *Hyperledger Fabric*, we started to read the official documentation of the program along with the source code. We found the part regarding security in the folder called *Idemix*,

which contained various files for key generation, message signature, credential and NymSignature (signatures associated to a particular credential) creation. Reading the documentation and analyzing the code, we find out that the signature scheme used is Schnorr[10], coupled with the Fiat-Shamir transformation[4] and elliptic curves[8]. This protocol is not resilient against quantum computers because it relies on the discrete logarithm problem, for which Shor already made an algorithm, and therefore must be changed with new algorithms.

To give some details about the functioning of the cryptographic side of *Fabric*, it is organized as shown in *Figure 1* and explained below:

- the files we modified are located in the $Idemix$ folder. This folder contains the implementation of the main cryptographic functions, especially the ones related to the signature scheme. The variables used in this package are represented with redefined types, slightly different from the rest of the application.

- the folder responsible for making the link between $Idemix$ and the rest of the program is called $Bridge$. Concretely, its role is to translate the parameters from the program into the corresponding types of the package $Idemix$. Then the $Idemix$'s desired function is called and the results is sent back to the caller.

- $Bridge$ fetches the translated types in the $Handlers$ folder. The files of this folder are used to convert parameter's values into $Idemix$'s types using dedicated functions. The files's classes all implement an interface declaring the functions, taking in variables of type handlers.xxx, with xxx the type name, and giving back the corresponding value of type xxx defined in $Idemix$ (for instance, a type handlers.IssuerKey is defined in $Handlers$, there is a function taking a variable of that type and returning the corresponding variable of type IssuerKey, defined in $Idemix$).

For example, to sign a message, the signature function of the file in $Bridge$ is called, with the corresponding parameters. These parameters are translated through $Handler$'s functions and then $Idemix$'s signature function is called to perform the concrete signing process. Finally, the signature is directly sent back.

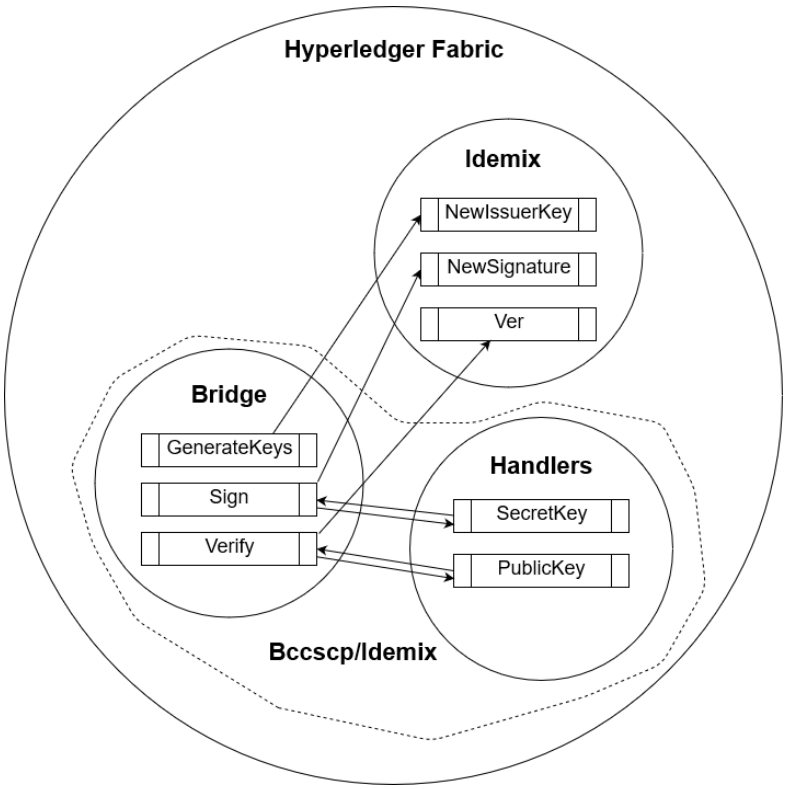


Figure 1: Structure of Hyperledger Fabric

5.4 Modifications performed

5.4.1 Files modified

As we have already mentioned, the core of the modifications we brought to the source code of *Fabric* is the folder named *Idemix*. It contains all the cryptographic functions and is composed, among others, of the following files :

- **idemix.pb.go** : in this file are declared the different classes and their methods.
- **issuerKey.go** : it contains the keys generation function.
- **signature.go** : here are the functions to generate and verify the signatures.
- **idemix_test.go** : this last one contains the functions to test the validity of the cryptographic algorithm, executed with the command `go test`.

5.4.2 Structures

We first decided to modify the *idemix.pb.go* file and create new structures associated with our use of GeMSS. In the original *Fabric*, we are interested in 3 structures : *IssuerPublicKey*, *IssuerKey* and *Signature*[18]. The first one is used to represent the public key, the second one contains the secret key and a reference to the first structure (so it contains the key pair), and the

[18]See Appendix 1

last one contains the information regarding the signature of a message. Therefore, we decided to create 3 new structures, imitating the original ones, in order to keep some parts of the internal logic of the program. However, considering the NIST interface (and so the prototypes of the GeMSS functions), these structures were fairly simple, each one containing a single array of bytes, and a reference of the public key in the structure containing the secret key. We didn't include the length of the arrays, as GO provides a function to get the length of an array, contrary to C.

5.4.3 Functions

Similarly to the structures, we created new functions performing the same tasks as the original ones, but responsible of calling the GeMSS functions. To create these functions, we had to check the documentation for CGO and understand how we can call C functions from GO, but mostly, how we can give the parameters and get the results. Fortunately, the CGO allows us to directly do mallocs from the GO side (which we must free afterwards) and convert C types to GO types and vice versa. That way, we just have to create the containers for each arguments of the GeMSS functions[19], call the functions, and convert the containers back to bytes array. Finally, we create the structures, assign the converted arrays and return the new structures.

5.4.4 Files created

From the GO-written program, we get access to the C functions through the use of static libraries. Since GeMSS's implementation is done with the use of various macros for function definitions and lengths, we created a file gathering these macros, entailing only one header file to include. So we added a file called *header.h* containing the different macros for the algorithms and security levels (128, 192, 256), along with the functions to call, depending on a single macro. We can easily swap between the algorithms used for encryption, since it's only one macro to modify, named $ALGORITHM$, taking values in {1,2,3,4} and referencing the three versions of GeMSS and MQDSS.

For these same reasons, we also had to do this part for MQDSS, by modifying the Makefile used by this algorithm, along with the creation of a script to generate a header file.

5.4.5 New signature algorithm integration

As mentioned previously, we base our work mainly on two post-quantum algorithms whereas there are 9 available at the NIST competition. Each of them respects the same interface, meaning that every function has the same prototype in every program. Considering this fact, we modified the *header.h* aforementioned so that it would be very easy to add and test the other algorithms, as they must respect the NIST's interface. To actually add another algorithm, we would just have to update :

- the macros : add a new possible value for the macro $ALGORITHM$, and update the macros referring to the sizes of keys and signatures.

[19] See Appendix 2

– the functions by adding a condition on the macro $ALGORITHM$ to call the corresponding functions of the new algorithm

– the main file by importing the library generated from the new algorithm

The final step is to actually generate the library, and afterwards, it is possible to use this algorithm instead of GeMSS or MQDSS.

5.5 Advanced modifications : hybrid cryptography

However, using the 2 set of structures is impossible in the rest of the program, as we can't easily get the structures we want in *Bridge*, as shown in *Figure 2*. Making use of these new structures would mean heavy modifications on the rest of *Fabric* to replace the old structures by the new ones. Therefore, after talking with our supervisors, we were asked to move on to our second objective which is an hybrid crypto-system in the core of *Fabric* : instead of the 3 separate structures already added, we would actually modify the 3 already-in-use structures to directly insert GeMSS information inside. Doing this means we would have an hybrid cryptography, relying on multiple algorithms to ensure its security. Therefore, we would use on the one hand the old algorithms (Schnorr, elliptic curves and Fiat-Shamir heuristic) and on the other hand, a post-quantum one.

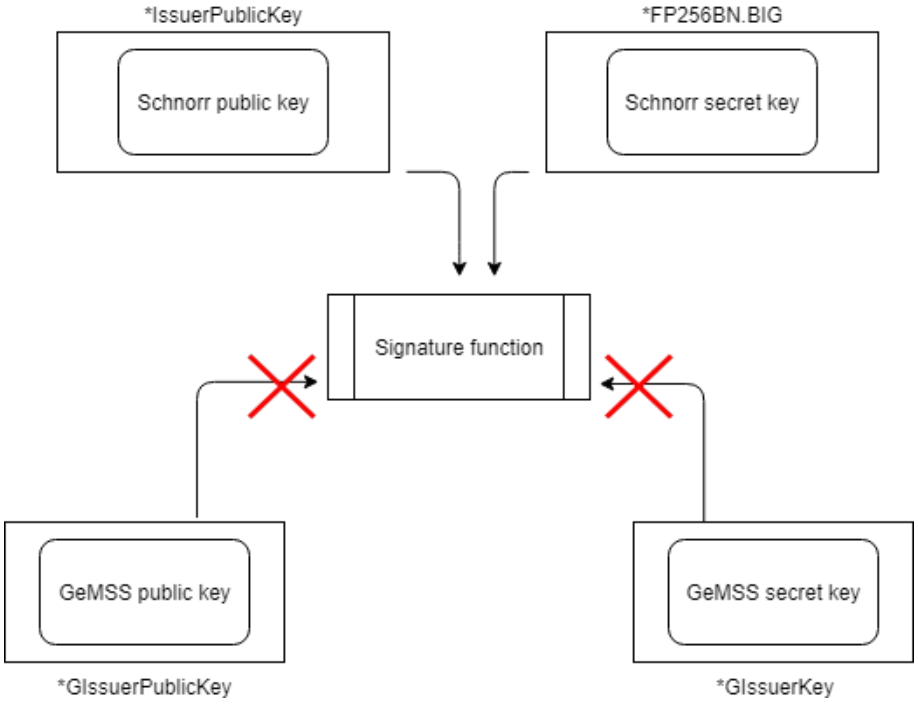


Figure 2: Signature function can't access GeMSS's public and secret keys in separate

For that purpose, we modified the original structures discussed earlier (IssuerKey, IssuerPublicKey and Signature) and added the byte arrays (for GeMSS keys and signatures) to each structures[20]. We also modified some functions. Especially, we modified the original functions for

[20]See Appendix 1

key generation, signature generation and signature verification just so that they would also call our new functions and fill the new field of the structures created. Subsequently, we realized that there was one big problem with our modifications : the function for creating signature needs the secret key. However, the secret key given as parameter in the old function is in the form of a big integer (*FP256BN.BIG*[21]), and not as an instance of the IssuerKey structure, containing a secret key, so we can't use the field we need to call the GeMSS functions, as shown in *Figure 3*.

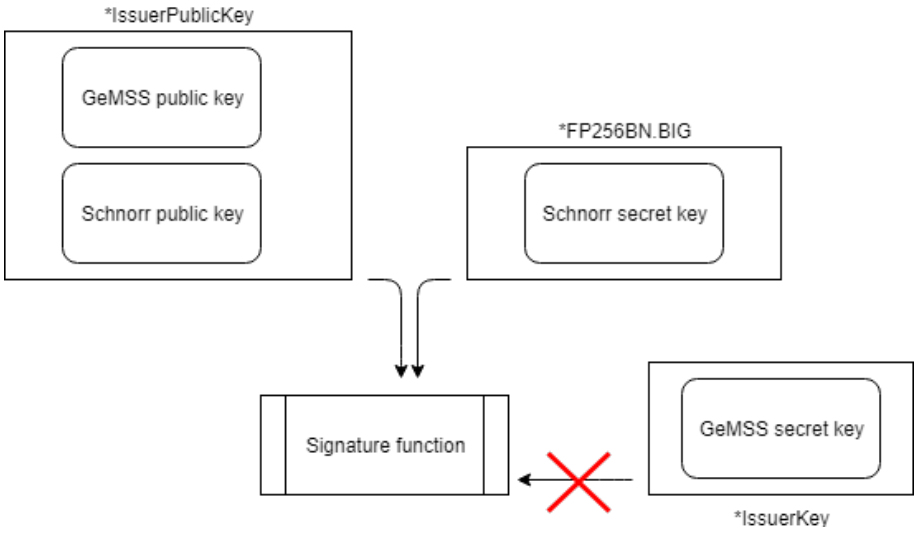


Figure 3: Hyperledger doesn't allow to add a parameter

To solve this problem, we tried to add a parameters to the signature function so that the secret key generated by GeMSS algorithm is accessible. This solution worked when running our personal test functions. We were able to use both signature algorithms, meaning that each of them must be checked to validate the signature. Even though this method seems to be the correct approach, it turns out that it is not viable when using in the *Fabric* program. We face compilation errors because the signature function must respect some interfaces and therefore the prototypes of the functions can't be modified. To actually be able to have the secret key as we want in *Bridge*'s signature function, we would have to change the interface, meaning modifying all the other classes implementing this interface, and everything related to modified functions. It was clearly not possible for the amount of time we had.

So here we are stuck, we need to access the GeMSS secret key while the *Fabric*'s one is provided in the form of a pointer toward a *FP256BN.BIG* structure. We discussed this problem with our supervisors and alternative solution was evoked. They proposed us to concatenate GeMSS secret key with *Fabric*'s one. In other words, it means to allocate enough memory to contains both keys and divide this buffer in two parts, the first one containing the *FP256BN.BIG* structure, which size is 40 bytes, and the second one containing GeMSS bytes array representing the secret key, having a size depending on the version (128,192,256). This whole buffer is then considered as a *FP256BN.BIG* pointer so that it can be given as parameter to the signature function. In the signature function we need to separate the buffer to retrieve both keys, and perform the two signatures.

[21]https://github.com/hyperledger/fabric-amcl/blob/master/amcl/FP256BN/BIG.go

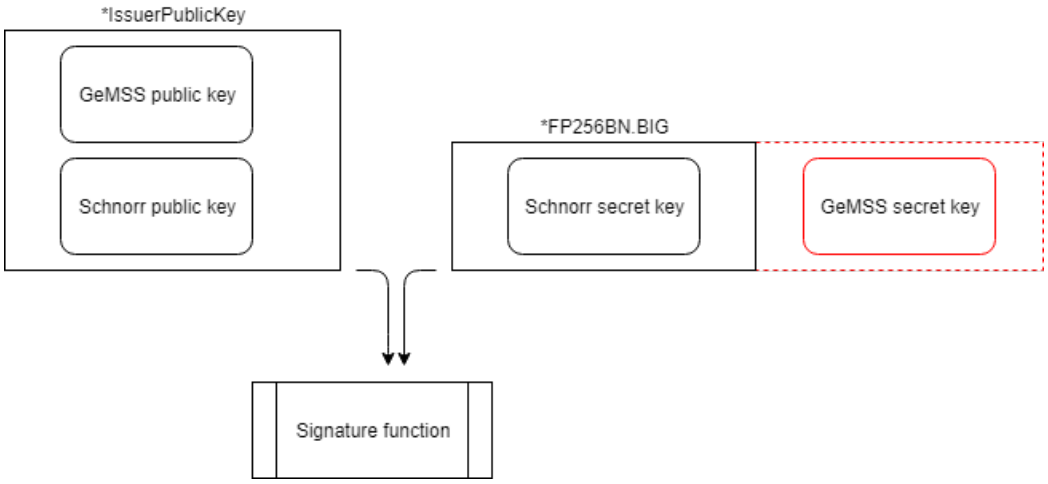


Figure 4: Concatenate GeMSS key after the original parameter

We managed to implement this solution [22], meaning that *Fabric* original signature function internally performs GeMSS signature too. The prototype of the function hasn't been changed but now we consider that the *FP256BN.BIG* pointer is as mentioned above. For now we didn't modify the rest of *Fabric* to fully reach an hybrid cryptography model but local tests are conclusive. Indeed the verification function is also adapted to this model and carry out both verifications. If one signature is not correct then an error is returned.

5.6 Test environment setup

For testing purposes, *Hyperlerger Fabric* provides multiple unit test. Especially, as mentioned before, the file called *idemix_test.go* contains tests for the cryptographic functions. This file contains nothing but a single function, calling all the functions defined in the other files. We managed, with this peculiar function, to test all of our early modifications, especially the ones regarding the 3 new structures and corresponding functions.

After validating the functions, we decided to create ways to test our new functions more than once, over a very short period of time. For that purpose, the GO provides a benchmark feature[23]. To use this, we have to create new functions with particular names and containing a particular `for` loop. That loop is set to that the content runs for at most a certain time (default is 1 second, can be changed with a flag). Therefore, we test our functions during that time, and we can see afterward the number of time the loop has been run, and the average time taken on each round. To ease and automate this process of testing, we created a script running all the commands we want and redirecting the results in a file.

It is worth mentioning that upon using our new benchmark functions, we managed to find a segmentation fault in GeMSS 192. After ensuring that is not the fault of our code, we reported it to our supervisors, and GeMSS's development team. They managed to find that the problem was coming from the fact that, in GeMSS, the functions manage memory as if it was aligned.

[22] See Appendix 2,3

[23] For more information regarding the benchmark feature : https://golang.org/pkg/testing/

However, the GO malloc function doesn't allow for memory alignment, and so that would cause the segmentation fault.

5.7 Performance tests

After changing the algorithms and making sure there wasn't any bug, we focused on checking the performance results obtained from the different tests. These tests have been done for two main purposes : evaluate the program's behaviour under an intensive environment and compare the performances with the initial program to quantify the impact of post-quantum cryptography.

We decided to use the benchmark functionality offered by the GO programming language (see 5.6). We compare the execution time of each cryptographic function in relation to the algorithm used between GeMSS128, GeMMS192, GeMSS256 and MQDSS. Actually, we have three benchmark functions. Each of them tests one function among key generation, signature and signature verification, for a defined amount of time, as explained in the above section. We run the tests on an Ubuntu virtual machine. First of all, we measure the execution time of each algorithm.

It is worth mentioning that under intensive circumstance, MQDSS signature function crashes almost every time and we didn't manage to understand the reason. These crashes also have been reproduced by one of GeMSS's developers, who couldn't even run normal tests.

The results are the following :

	Hyperleger	GeMSS128	GeMSS192	GeMSS256	MQDSS
GenKey	0.002 ms	13 ms	60 ms	191 ms	1 ms
Sign	31 ms	220 ms	560 ms	905 ms	33 ms*
VerSign	60 ms	0.05 ms	0.2 ms	0.5 ms	22 ms

Figure 5: Execution time with processor *Intel(R) Core(TM) i5-7600K CPU @ 3.80GHz*, *time taken over a few runs

	Hyperleger + GeMSS 128	Hyperleger + GeMSS 192	Hyperleger + GeMSS 256	Hyperleger + MQDSS
GenKey	14 ms	66 ms	207 ms	1.1 ms
Sign	227 ms	619 ms	1180 ms	61 ms*
VerSign	60 ms	62 ms	62 ms	85 ms

Figure 6: Hybrid execution time with processor *Intel(R) Core(TM) i5-7600K CPU @ 3.80GHz*, *time taken over a few runs

These results highlight the differences between the algorithms we implemented. In separate execution, Hyperledger's times and MQDSS ones are relatively the same, except for the verification of signatures, going 3 times faster than Hyperledger. This shows that using MQDSS instead of Hyperledger would not only be more secure, but faster overall. In the case of GeMSS, we see the verification is almost instantaneous in all cases, whereas the signature process takes up almost a second to complete in the highest security level. However, even when using the lowest security level, the differences are still noticeable, with the generation process taking much longer than MQDSS and Hyperledger alone.

These results are the same when using Hybrid execution. The generation process is much faster for Hyperledger + MQDSS in all cases, and the verification process is one-third slower. In the case of GeMSS, when using level of security 256, the signature process takes more than a second, and the time differences in generating keys is also noticeable. On the other hand, as expected, the signature verification times don't vary much between each level.

	GeMSS 128	GeMSS 192	GeMSS 256	MQDSS
GenKey	434 KB	1351 KB	3137 KB	0.288 KB
Sign	0.232 KB	0.264 KB	0.288 KB	49.288 KB
VerSign	96 B	96 B	96 B	96 B

Figure 7: Memory consumption for one function call with a 40 bytes message

We see that regarding the generation of keys, the memory consumption of MQDSS is only 288 bytes, meaning that this algorithm is more suited to embedded systems. On the other hand, GeMSS keys need more memory to be generated, up to more than 3 MB for the highest level of security. This experimental results can be explained by the difference in the size of keys and signature between the two algorithms (section 3.5).

These experimentation are useful to emphasize the differences between post-quantum signature algorithms. We notice that time and memory needs differ greatly from one algorithm to another. The choice of the algorithm has to be made considering the constraints of the encompassing application. For instance, if we consider an application that sign a large quantity of messages, then the signature process must be quick and an algorithm similar to MQDSS would be better than GeMSS. Conversely, if the application is more prone to verify signatures than produce them, GeMSS and similar algorithms would be a better choice. The same reasoning can applied to the memory usage. We notice that GeMSS requires more memory, especially to generate the keys and this aspect has to be considered when choosing the post-quantum signature algorithm.

5.8 Mini application test

Towards the end of the project, we decided to try and create a mini-test application to test our modifications in real conditions. When we started working on it, GeMSS functions were already available from *Fabric* and we were trying to combine the signatures with the original

ones already in place. Since we faced some complication during the fusion phase, we didn't manage to get relevant results on this part.

As presented previously, we had to find a way to circumvent the problem of the secret key structure (see section 5.5) to access the perfect hybrid model. Our solution works when running local tests but we didn't have time to implement it in real conditions, since we were close to the end of the semester. That's why we couldn't setup the mini-test application.

However we discussed a lot about this idea and we defined a real-condition environment that can be used once *Hyperledger Fabric* is compatible with the hybrid-cryptography model. Indeed, we found out that *Hyperledger* provides an environment emulating real transactions between peers. The simulation is performed through scripts offered in the open-source project *Fabric-Sample*[24]

[24] *https://github.com/hyperledger/fabric-samples*. The documentation is available on *https://hyperledger-fabric.readthedocs.io/en/release-1.4/tutorial/commercial_paper.html*

6. Conclusion

The project's main objective was to implement a quantum-attack resilient signature algorithm, in the open-source project *Hyperledger Fabric*. Initially, our intention was to completely replace the signature process (originally based on Schnorr and Elliptic curves) with GeMSS.

As we went along the project, the initial objective gradually transformed into replacing those algorithms with hybrid-cryptography. Instead of replacing the already in place Schnorr signature with GeMSS's one, we just strengthen the program security by keeping both signatures. Logically, these modifications substantially affect the program performances. Therefore, it is necessary to seek a steady balance between strong security and acceptable performances. For instance, we pointed out that using Schnorr signature completed with GeMSS's highest security level signature plummets the program's performances. The question now is to estimate if the security benefits of this hybrid-model are worth somewhat substantial performance losses.

For now, *Hyperledger Fabric*, in his Post-Quantum version, doesn't fulfill his main role as a Blockchain manipulation tool. However, we are only one step away from reaching this goal. As presented in section 5.5, we managed to obtain conclusive local results when testing our hybrid signature function. The final step is to make the necessary modifications in the core of *Hyperledger Fabric*, everywhere a signature is performed, to fully incorporate post quantum attacks resilience.

This project is just one example illustrating the impact quantum computers will have on our society. It is not only one software that has to be adapted for this revolution but every modern system that is based on security principles ensured by cryptography. However, by completely changing *Hyperledger Fabric*, the security of many Blockchain-based applications would be drastically improved, without needing any modifications from the people using this tool.

Quantum computers are close to revolutionizing our world the same way classical computers and the Internet did the past century. On the first hand, their fast-progressing development opens new possibilities in many important fields including high-performance computing, artificial intelligence and health care. As almost every technological breakthrough before, quantum computers are being developed in order to make our lives easier and more enjoyable. On the other hand, theses opportunities offered by quantum-computers can also represent a serious threat for our security and privacy if used with malicious intents. Indeed, the most complex mathematical problems for classical computers, implemented in broadly-used security protocols today, can be solved efficiently and quickly by any quantum computer, using already existing algorithms like Shor's factorization of big integers. Being aware of this new kind of threat, it is mandatory to consider the impending post-quantum world today. Through this project, we experimented the complexity of adapting the current tools for the post-quantum world, along with the necessity of this work.

7. Appendix

7.1 Appendix 1 : Issuer.pb.go

```
// IssuerKey specifies an issuer key pair that consists of
// ISk - the issuer secret key and
// IssuerPublicKey - the issuer public key
// SK - the secret key generated with GeMSS
type IssuerKey struct {
    Isk                  []byte           ``
    Ipk                  *IssuerPublicKey
    SK                   []byte     /* GeMSS Secret key */
    XXX_NoUnkeyedLiteral struct{}
    XXX_unrecognized     []byte
    XXX_sizecache        int32
}

// IssuerPublicKey specifies an issuer public key that consists of
// attribute_names - a list of the attribute names of a credential
// issued by the issuer
// h_sk, h_rand, h_attrs, w, bar_g1, bar_g2 - group elements
// corresponding to the signing key, randomness, and attributes
// proof_c, proof_s compose a zero-knowledge proof of knowledge of the secret key
// hash is a hash of the public key appended to it
type IssuerPublicKey struct {
    AttributeNames       []string
    HSk                  *ECP
    HRand                *ECP
    HAttrs               []*ECP
    W                    *ECP2
    BarG1                *ECP
    BarG2                *ECP
    ProofC               []byte
    ProofS               []byte
    Hash                 []byte
    PK                   []byte    /* GeMSS Public key */
    XXX_NoUnkeyedLiteral struct{}
    XXX_unrecognized     []byte
    XXX_sizecache        int32
}
```

```
// Signature specifies a signature object that consists of
// a_prime, a_bar, b_prime, proof_* - randomized credential signature values
// and a zero-knowledge proof of knowledge of a credential
// and the corresponding user secret together with the attribute values
// nonce - a fresh nonce used for the signature
// nym - a fresh pseudonym (a commitment to to the user secret)
type Signature struct {
APrime               *ECP
ABar                 *ECP
BPrime               *ECP
ProofC               []byte
ProofSSk             []byte
ProofSE              []byte
ProofSR2             []byte
ProofSR3             []byte
ProofSSPrime         []byte
ProofSAttrs          [][]byte
Nonce                []byte
Nym                  *ECP
ProofSRNym           []byte
RevocationEpochPk    *ECP2
RevocationPkSig      []byte
Epoch                int64
NonRevocationProof   *NonRevocationProof
SM                  []byte                  /* GeMSS Signature */
}
```

7.2 Appendix 2 : Issuerkey.go

```
/*
* Function to generate private and public keys using the GeMSS algorithm.
* The structure to store the result is Gissuerkey instead of issuerkey.
* The new structure is defined in idemix.pb.go.
*/
func NewGIssuerKey() (*GIssuerKey, error) {

    ISK, IPK, _ := completeKey()
    key := new(GIssuerKey)
    key.SK = ISK
    key.lengthSK = C.GeMSS_SK_SIZE

    key.IPK = new(GIssuerPublicKey)
    key.IPK.PK = IPK
    key.IPK.lengthPK = C.GeMSS_PK_SIZE

    return key, nil
}

func completeKey() ([]byte, []byte, error) {
    // Memory allocation using the C functions
    tempPK := C.malloc(C.sizeof_uchar * C.GeMSS_PK_SIZE)
    tempSK := C.malloc(C.sizeof_uchar * C.GeMSS_SK_SIZE)

    // Call to GeMSS to compute the keys
    C.sign_keypair((*C.uchar)(tempPK), (*C.uchar)(tempSK))

    // Copy the result from C allocated memory to GO allocated memory
    ISK := C.GoBytes(tempSK, C.GeMSS_SK_SIZE)
    IPK := C.GoBytes(tempPK, C.GeMSS_PK_SIZE)

    // Free memory
    C.free(unsafe.Pointer(tempPK))
    C.free(unsafe.Pointer(tempSK))
    return ISK, IPK, nil
}

func CompleteGIssuerKey(key *IssuerKey) error {
    ISK, IPK, _ := completeKey()
    key.SK = ISK
    key.Ipk.PK = IPK
    return nil
}
```

```
/*
 * Merge keys
 */
func HybridKeys(idemixSK unsafe.Pointer, GeMSSSK []byte) unsafe.Pointer {
GeMSSSKC := C.CBytes(GeMSSSK)
bothKeys := C.malloc(C.sizeof_uchar*C.GeMSS_PK_SIZE + 40)
C.fusion(   (unsafe.Pointer)(idemixSK), \\
            (unsafe.Pointer)(GeMSSSKC), \\
            (unsafe.Pointer)(bothKeys))
return bothKeys
}

/*
 * Separate keys
 */
func InitialKeys(bothKeys unsafe.Pointer) (unsafe.Pointer, []byte) {
idemixSK := C.malloc(40)
GeMSSSK := C.malloc(C.GeMSS_SK_SIZE)
C.separate( (unsafe.Pointer)(idemixSK), \\
            (unsafe.Pointer)(GeMSSSK), \\
            (unsafe.Pointer)(bothKeys))
GeMSSSKbytes := C.GoBytes(GeMSSSK, C.GeMSS_SK_SIZE)
C.free(unsafe.Pointer(GeMSSSK))
return idemixSK, GeMSSSKbytes
}
```

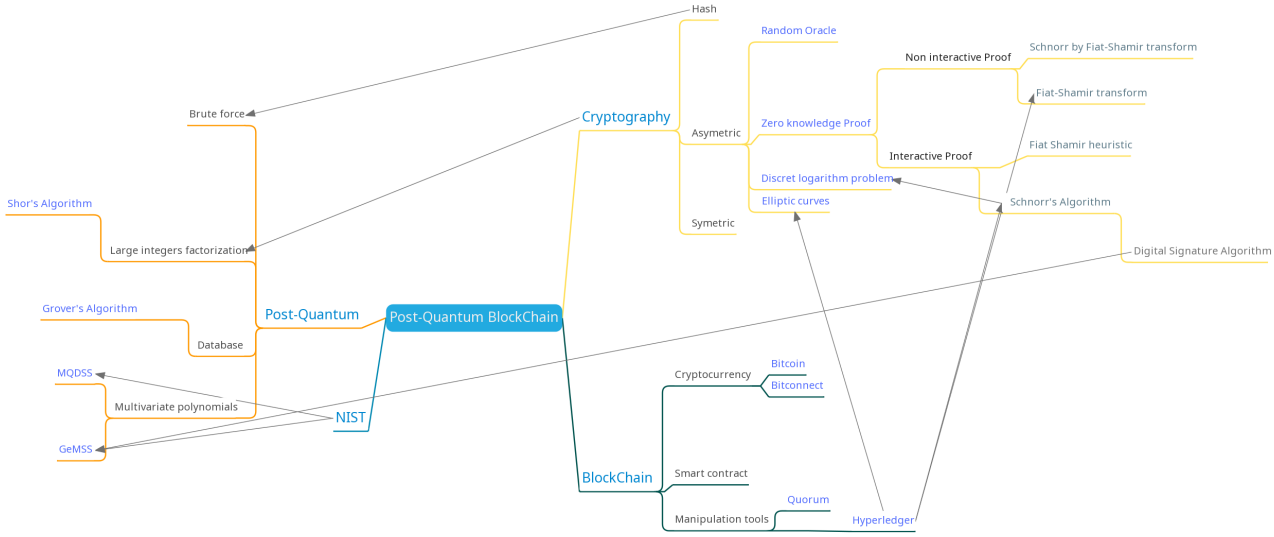
7.3 Appendix 3 : header.h

```
// 1 : GeMSS 128
// 2 : GeMSS 192
// 3 : GeMSS 256
// 4 : MQDSS

#define ALGORITHM 1

#define GeMSS_PK_SIZE (ALGORITHM == 1 ? \
   GeMSS_128U_CRYPTO_PUBLICKEYBYTES : ALGORITHM == 2 ? \
   GeMSS_192U_CRYPTO_PUBLICKEYBYTES : ALGORITHM == 3 ? \
   GeMSS_256U_CRYPTO_PUBLICKEYBYTES : CRYPTO_PUBLICKEYBYTES)

#define GeMSS_SK_SIZE (ALGORITHM == 1 ? \
   GeMSS_128U_CRYPTO_SECRETKEYBYTES : ALGORITHM == 2 ? \
   GeMSS_192U_CRYPTO_SECRETKEYBYTES : ALGORITHM == 3 ? \
   GeMSS_256U_CRYPTO_SECRETKEYBYTES : CRYPTO_SECRETKEYBYTES)
```

```
#define SIGN_SIZE (ALGORITHM == 1 ? \
   GeMSS_128U_CRYPTO_BYTES : ALGORITHM == 2 ? \
   GeMSS_192U_CRYPTO_BYTES : ALGORITHM == 3 ? \
   GeMSS_256U_CRYPTO_BYTES : CRYPTO_BYTES)

#define GeMSS_SIGN_KEYPAIR(ALGORITHM) \
   (ALGORITHM == 1 ? GeMSS_128U_crypto_sign_keypair : \
   ALGORITHM == 2 ? GeMSS_192U_crypto_sign_keypair : \
   GeMSS_256U_crypto_sign_keypair)

#define GeMSS_SIGN(ALGORITHM) \
   (ALGORITHM == 1 ? GeMSS_128U_crypto_sign : \
   ALGORITHM == 2 ? GeMSS_192U_crypto_sign : \
   GeMSS_256U_crypto_sign)

#define GeMSS_SIGN_VERIF(ALGORITHM) \
   (ALGORITHM == 1 ? GeMSS_128U_crypto_sign_open : \
   ALGORITHM == 2 ? GeMSS_192U_crypto_sign_open : \
   GeMSS_256U_crypto_sign_open)

static int sign_keypair(unsigned char *pk , unsigned char *sk) {
   #if ALGORITHM == 4
        return crypto_sign_keypair(pk,sk);
   #else
     return GeMSS_SIGN_KEYPAIR(ALGORITHM)(pk , sk);
   #endif
}

static int sign( unsigned char *sm, unsigned long long *smlen,
                 unsigned char *m, unsigned long long mlen,
                 unsigned char *sk) {
   #if ALGORITHM == 4
       return crypto_sign(sm, smlen, m, mlen, sk);
   #else
       return GeMSS_SIGN(ALGORITHM)(sm, smlen, m, mlen, sk);
   #endif
}

static int sign_verif( unsigned char *m, unsigned long long *mlen,
                       unsigned char *sm, unsigned long long smlen,
                       unsigned char *pk) {
   #if ALGORITHM == 4
       return crypto_sign_open(m , mlen , sm , smlen , pk);
   #else
       return GeMSS_SIGN_VERIF(ALGORITHM)(m , mlen , sm , smlen , pk);
   #endif
```

```
}

static void fusion(void *idemixKey , void *GeMSSKey , void *bothKeys) {
      memcpy(bothKeys , idemixKey , 40);
      memcpy(bothKeys + 40 , GeMSSKey , GeMSS_SK_SIZE);
}

static void separate(void *idemixKey , void *GeMSSKey , void *bothKeys) {
      memcpy(idemixKey , bothKeys , 40);
      memcpy(GeMSSKey , bothKeys + 40 , GeMSS_SK_SIZE);
}
```

7.4 Appendix 4 : Mindmap



References

[1] Divesh Aggarwal, Gavin K. Brennen, Troy Lee, Miklos Santha, and Marco Tomamichel. "Quantum attacks on Bitcoin, and how to protect against them". In: *Ledger* 3 (Oct. 17, 2018). ISSN: 2379-5980. DOI: 10.5195/ledger.2018.127. arXiv: 1710.10377. URL: http://arxiv.org/abs/1710.10377 (visited on 02/05/2019).

[2] Antoine Casanova, Jean-Charles Faugère, Gilles Macario-Rat, Jacques Patarin, Ludovic Perret, and Jocelyn Ryckeghem. *GeMSS: A Great Multivariate Short Signature*. report. UPMC - Paris 6 Sorbonne Universités ; INRIA Paris Research Centre, MAMBA Team, F-75012, Paris, France ; LIP6 - Laboratoire d'Informatique de Paris 6, Dec. 11, 2017, pp. 1–4. URL: https://hal.inria.fr/hal-01662158 (visited on 03/09/2019).

[3] Ming-Shing Chen, Andreas Hülsing, Joost Rijneveld, Simona Samardjiska, and Peter Schwabe. "From 5-pass MQ-based identification to MQ-based signatures". In: *Advances in Cryptology – ASIACRYPT 2016*. Ed. by Jung Hee Cheon and Tsuyoshi Takagi. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2016, pp. 135–165. ISBN: 978-3-662-53890-6.

[4] Amos Fiat and Adi Shamir. "How To Prove Yourself: Practical Solutions to Identification and Signature Problems". In: *Advances in Cryptology — CRYPTO' 86*. Ed. by Andrew M. Odlyzko. Vol. 263. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 186–194. ISBN: 978-3-540-18047-0. DOI: 10.1007/3-540-47721-7_12. URL: http://link.springer.com/10.1007/3-540-47721-7_12 (visited on 03/28/2019).

[5] The Linux Foundation. *Hyperledger Fabric*. original-date: 2016-08-25T16:05:27Z. Mar. 28, 2019. URL: https://github.com/hyperledger/fabric (visited on 03/28/2019).

[6] Lov K. Grover. "A fast quantum mechanical algorithm for database search". In: *arXiv:quant-ph/9605043* (May 29, 1996). arXiv: quant-ph/9605043. URL: http://arxiv.org/abs/quant-ph/9605043 (visited on 03/09/2019).

[7] Stuart Haber and W. Scott Stornetta. "How to time-stamp a digital document". In: *Journal of Cryptology* 3.2 (Jan. 1, 1991), pp. 99–111. ISSN: 1432-1378. DOI: 10.1007/BF00196791. URL: https://doi.org/10.1007/BF00196791 (visited on 05/31/2019).

[8] Neal Koblitz, Alfred Menezes, and Scott Vanstone. "The State of Elliptic Curve Cryptography". In: *Designs, Codes and Cryptography* 19.2 (Mar. 1, 2000), pp. 173–193. ISSN: 1573-7586. DOI: 10.1023/A:1008354106356. URL: https://doi.org/10.1023/A:1008354106356 (visited on 03/28/2019).

[9] Jacques Patarin, Nicolas Courtois, and Louis Goubin. "QUARTZ, 128-Bit Long Digital Signatures". In: *Topics in Cryptology — CT-RSA 2001*. Ed. by David Naccache. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2001, pp. 282–297. ISBN: 978-3-540-45353-6.

[10] C. P. Schnorr. "Efficient Identification and Signatures for Smart Cards". In: *Advances in Cryptology — CRYPTO' 89 Proceedings*. Ed. by Gilles Brassard. Lecture Notes in Computer Science. New York, NY: Springer New York, 1990, pp. 239–252. ISBN: 978-0-387-34805-6. DOI: 10.1007/0-387-34805-0_22.

[11] Peter W. Shor. "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer". In: *SIAM Journal on Computing* 26.5 (Oct. 1997), pp. 1484–1509. ISSN: 0097-5397, 1095-7111. DOI: 10.1137/S0097539795293172. arXiv: quant-ph/9508027. URL: http://arxiv.org/abs/quant-ph/9508027 (visited on 03/09/2019).

[12] Caleb Spare. *Format Go's benchmarking output. Contribute to cespare/prettybench development by creating an account on GitHub*. original-date: 2014-01-24T02:18:34Z. Apr. 9, 2019. URL: https://github.com/cespare/prettybench (visited on 04/25/2019).

[13] Christopher Wolf and Bart Preneel. *Taxonomy of Public Key Schemes Based on the Problem of Multivariate Quadratic Equations*. 2005.

[14] XKCP. *eXtended Keccak Code Package. Contribute to XKCP/XKCP development by creating an account on GitHub*. original-date: 2013-06-30T09:12:10Z. Apr. 23, 2019. URL: https://github.com/XKCP/XKCP (visited on 04/26/2019).