

An introduction to cryptography through time lock puzzles

Saturday 23rd January, 2021 - 12:18

Student: Issam Jomaa
University of Luxembourg
Email: issam.jomaa.001.student@uni.lu

Tutor: Răzvan Roşie
University of Luxembourg
Email: razvan.rosie@uni.lu

Abstract

This first semester BSP is an introduction to the world of cryptography through the concept of Time Lock Puzzles. Time lock puzzles are important primitives due to the belief that parallelism cannot be employed to speed up such computations. This means that they are viable for real-life applications in diverse domains that necessitate encryption methods that are bound by a factor of time. Our BSP will focus on the implementations of such time lock puzzles and the way to solve them using two different programming languages. We can then draw some conclusions after analyzing the data we collect from running experiments. To collect this data we will be using Amazon's cloud service (AWS).

1.. Introduction

Time lock puzzles are a way to send messages in the future by bidding the deciphering process to a parameter T that represents time. The first accepted time lock puzzle was proposed more than 25 years ago by Rivest, Shamir, and Wagner in their paper (cf. [4]). Time lock puzzles are a major creation in the domain of cryptography since it has a lot of different viable applications in the real world.

As Rivest et al. suggest in their paper, they can be used for different application. For example : future financial transaction as a bidder would want to seal his bid until T intervals of time have passed before opening again.

It could also be used for deferred payments like mortgages since the encrypted digital cash would only be available at the beginning of each month for example.

It could even be used for diaries or key-escrow schemes. We can clearly see that it has a multitude of uses.

The RSW is based on 2 key components to make it what it is today "the only accepted time lock puzzle"

- 1) Fast generation of the puzzle.
- 2) Cannot be parallelized.

The first point, "Fast generation of the puzzle" is important for the algorithm to be usable in the industry as the generation should take way less time than the puzzle itself especially when it comes to generating puzzles that should remain unsolved for years. The second point is equally, if not more important as it is the base foundation of the Time-lock puzzle. Since the calculation being parallelized means that we can reduce the time used to solve it by using an exponential amount of CPU's which makes the whole concept pointless.

2.. Project description

2..1. Domains

The aim of this project is to reproduce a time lock puzzle based on the RSW scheme and analyse the data we gather by experimenting with the newly created programs. Therefore we could say that this project will focus on two main scientific domains.

- 1) Cryptography
- 2) Optimization and analysis

We will also be using two different programming languages that are different in nature to do the implementation. We are doing this to complement our scientific objectives and provide a large panel of insights, thus letting us room to experiment.

Cryptography

Cryptography in the ancient ages was synonymous with encryption which meant to convert readable data into an unreadable mix of data and while this meaning

still holds in the general mind, cryptography as of today "is the practice and study of techniques for secure communication in the presence of third parties called adversaries" [2]. So in general Cryptography would be the science of creating algorithms or protocols to prevent third parties¹ to see the data your transmitting to your target. In a world where digitalization is becoming more and more prominent and where technology is thriving and computers are getting faster, cryptography plays a major role in protecting not only people's privacy but also the government's secrets and any kind of exchange of data on the internet wherever it is electronic commerce, password authentication, bank payments, or even military communication. All and all we can say that cryptography has four main points it needs to fulfill :

1) **Confidentiality :**

The data that is intended to be seen by an entity X should only be accessible to it at any point of time. And by any mean should never be accessible by any stranger entity.

2) **Integrity :**

When data is transferred between two entities the data should not be tempered with in any fashion and if tempering has occurred it should be detected and registered.

3) **Non-repudiation :**

Once data has started transferring between two entities it cannot deny the intention of sending data at a later stage.

4) **Authentication :**

The identity of both the entities that sends and receive should be confirmed before the transfer of data as well as the origin and the destination of the data.

Optimization and analysis

As mentioned previously in Section 2.1, an encryption algorithm should be as secure as possible to avoid having breaches and to avoid the possibility of it being deciphered, but this should not temper with the speed of the encryption as users may not have the time to wait even mere minutes as time is money. This shows the difficulty of Cryptography as a perfect balance needs to be found. That's why it is even more important to have tools to analyze the algorithm's performance and retrieve the performance data to see how the algorithm acts in different conditions and optimize it to its best. In time lock puzzles this is

1. Meaning other parties then the sender and receiver.

even more important as we need to make sure that the algorithm works as intended and is being solved in a precise time as we predicted so these tools will come even handier.

2..2. Programming Languages

2..2.1. Python. "Python is a high-level, interpreted, interactive and object-oriented scripting language" [3]. Compared to other programming languages, Python stands out as being highly readable thanks to it using more English words compared to other programming languages that would often use more punctuation as syntax. Python also features dynamically allocated memory which is especially useful for applications like ours since we will need variables of length that could attain thousand's of Bits. Python also has a wide variety of libraries that are easy to use. That's why it is one of the programming languages we chose to use as it will be a nice first contact with the world of programming.

2..2.2. C++. C++ is a statically typed, compiled, general-purpose, case-sensitive, free-form programming language that supports procedural, object-oriented, and generic programming. C++ is regarded as a middle-level language, as it comprises a combination of both high-level and low-level language features [1]. C++ doesn't allocate memory dynamically, which means that we will have to find a way to attribute space to the specific structures to be used, as well as it does not provide a garbage collector, meaning that we have to handle the memory de-allocation process.

2..3. Difference between Memory Management in both Languages.

One of the largest differences between these two languages is how they manage memory. In Python, everything is considered an object, Numbers and modules are held in objects while the object of a class is the class itself and functions are considered an object. These objects are created in the memory and are directly managed by Python that keeps track of which objects are still in use and which can be freed. This is called reference counting. To do this Python employs a garbage collector. This on the other hand makes Python slower as it may take a longer amount of time to clear and manage the memory. C++ is a different case since you are supposed to manage your own memory, meaning that while it may take more

time to write the code and ensure everything is being taken into account, your program will produce faster response times. After compilation, your program won't have to worry about any kind of problem with the memory and calculations should be done faster.

2..4. Deliverables

Through the description of our objectives we can get a global view on what kind of deliverables we should expect having at the end of our project.

2..4.1. The main deliverable.

Our main deliverable is a program that will be an implementation of the RSW algorithm and as such will be generating the public parameters (having size N-Bits) and the puzzle that is supposed to be solved only after "T" intervals of time have passed. Such a program will also be simulating an attack on this scheme trying to find the solution. During this process, the time taken by the attack to solve the puzzle will be registered and recorded in a separate text file. This deliverable will be in two versions, one in Python and one in C++.

2..4.2. Second deliverable.

We will be using two different programming languages to code our algorithm. The second deliverable should be a program that interprets the data we collect automatically and create graphs to help us interpret it. This will help us not only improve our program but also optimize the way we use it by comparing the two versions of it. To collect a large array of data we will use Amazon cloud computing AWS so we can run the program 24/7 until we get a large enough data pool.

3.. Background

3..1. Scientific Background

3..1.1. Cryptography's sub-fields.

It is important before delving in Cryptography-related subjects to know more about cryptography and especially the three main types of Cryptography. These are:

1) Hash Functions :

A Hash function – also called a one-way functions – represents a type of cryptographic primitive that doesn't use an (undisclosed) key. Instead, it uses an algorithm to compute

a 'fixed-length hash' based on the plain text we give it. The plain text is then impossible to recover as the algorithm employed makes it so that it gives a digital fingerprint to the content of a file making sure that even a little change changes completely the result we have. Hash functions are used to make sure the file was not tampered with. They are also used to store passwords on systems. It is technically impossible to invert a hash as the functions are one-way. It is however possible to use database's to find the strings that produce the same hash using the same function used.

2) Symmetric-Key Cryptography :

Symmetric-Key Cryptography is a way to send data securely by encrypting it. This system uses a key that serves for both encryption and decryption of the data. This means that both sender and receiver of data must have the same key for the scheme to work. While this scheme is considered to be secure, its security is determined by how difficult it is to guess a key which means a key of appropriate length is supposed to even be quantum secure. This scheme is still vulnerable since the largest difficulty is that the parties must have the same key. Meaning that at one point in time the key must be shared with the receiver and that's where the flaw in this system resides.

3) Public-key cryptography :

A public-key cryptosystem or asymmetric cryptography is a way to send data securely by encrypting it. This system uses a pair of keys that have different functions.

- A public key which is known by both the sender and a receiver.
- A private key that should only be known by the receiver.

How this works, is that the Sender will use the public key known by both the users to encrypt the data through a mathematical algorithm. Once the data is encrypted, it can only be deciphered using the private key. This means that a regular user needs only to protect his private key that he doesn't need to send over through the internet meaning a less likely chance for it to be hacked. In summary :

Definition 3.1 (Public Key Encryption Scheme). A PKE scheme consists of these algorithms (Setup, Enc, Dec) with the following specification:

- Setup:** This takes as input the key length and return a public key and a decryption key,
- Encryption:** This is a randomized algorithm that takes as input the encryption key and the message and some randomness term and returns a cipher-text.
- Decryption:** This takes as input the decryption key and the cipher-text and returns the message

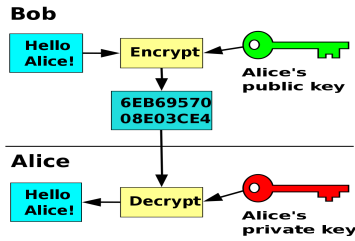


Fig. 1. An asymmetric key encryption scheme. Image from [2].

3.1.2. The RSW scheme.

The concept of Cryptographic or (Time lock) puzzles is to cipher some data in a way that the content is secured and can only be deciphered using a precise amount of time that the cipherer will choose and set up before creating the puzzle. To do that, the time lock puzzle must use an algorithm that generates the puzzle fast but more importantly, it must be not parallelizable. This is the major difficulty to be overcome when creating such algorithms as paralleled computing would render the puzzle completely useless since this would mean that the CPU time would be related to the type and the quantity of the hardware. This is not what we want as we need the puzzle to be universal.

Ronald L. Rivest, Adi Shamir, and David A. Wagner are the creators of the algorithm named after themselves RSW. It is one of the most well known Time Lock Puzzle. This is the algorithm we will be using as it checks all the conditions we have set up in the first paragraph.

Suppose we want to encrypt a message M such that the period taken for it to be deciphered will be T . We

need first to generate a composite modulus n such that q and p are secret prime numbers:

$$n = p \cdot q \quad (1)$$

We will also compute :

$$\phi(n) = (p-1)(q-1) \quad (2)$$

And

$$t = TS \quad (3)$$

where S is the number of squaring modulo n per second that can be performed by the solver.

We then generate a key K with which we are going to cipher our message to obtain our cipher-text :

$$C_M = \text{EncryptionAlgorithm}(K, M) \quad (4)$$

finally we pick a random a (with $1 < a < n$), and encrypt K as :

$$C_k = K + a^{2^t} \pmod{n} \quad (5)$$

To make this computation faster and more efficient we will divide in two parts :

We first compute :

$$e = 2^t \pmod{\phi} \quad (6)$$

Then :

$$b = a^e \pmod{n}. \quad (7)$$

Our output that constitutes the Time Lock puzzle will be (n, a, t, C_k, C_M) while all the other variables will be erased. Solving the Puzzle is now only possible through one operation which is to find b so that we can deduce it from C_k and find the key to unlock the puzzle. To find b we must compute it like this :

$$b = a^{2^t} \pmod{n} \quad (8)$$

For our project we don't have the need to encrypt a message so we will only focus on producing the puzzle and timing how long does it take to solve it. So our project will need to focus on how to generate prime numbers efficiently and scripting the creation and solving of the puzzle while timing how long does the process takes so that we can analyze the data we have and proceed to draw our observations.

3.1.3. Miller-Rabin test.

As we said previously in (Section 3.1.2), we will need prime numbers for our RSW scheme. For this, we will need to know about the Miller-Rabin primality test. Using this test on any integer will tell us if the number is composite or not in which case it is considered a number that is probably prime. This

test is divided into three steps for an integer n :

- 1) Find $n - 1 = 2^k \cdot m$
- 2) Choose an integer a such that : $1 < a < n - 1$
- 3) Compute $b_0 = a^m \pmod n$
If b_0 is equal to 1 or -1 then n is probably a prime number. If not, then we need to repeat step 4 as much as needed :
- 4) Compute $b_n = b_{n-1}^2 \pmod m$
If b_n is equal to 1 then n is a composite number while if b_n is equal to -1 then n is probably a prime number.

In rare occasions the loop might be infinite without ever landing on 1 or -1 in which case the number would be considered as composite.

3..2. Technical Background

3..2.1. Python Modules.

PyCryptodome

PyCryptodome is one of the most well-known Packages for cryptography related content in Python. He is the continuation of the project Pycrypto and encapsulates lots of functions that will come in handy to facilitate the implementation of the time lock puzzle.

Matplotlib

matplotlib is a Python plotting library. It provides us a way to visualize data by creating graphs and displaying them through their API. It also provides a MATLAB-like interface as well as having the advantages of being open-source and free.

3..3. C++ Libraries

GNU MP

GNU Multi Precision (MP) or GMP is a library that will help us bypass the 64 BITS limit of the c++ language. Here is it's definition from their own site 'GMP is a free library for arbitrary precision arithmetic, operating on signed integers, rational numbers, and floating-point numbers. There is no practical limit to the precision except the ones implied by the available memory in the machine GMP runs on. GMP has a rich set of functions, and the functions have a regular interface.

The main target applications for GMP are cryptography applications and research, internet security applications, algebra systems, computational algebra research, etc.[5]

4.. Time Lock Puzzle Implementation's

4..1. Requirements

4..1.1. Non Functional requirements.

- **Easy to use and moduable**

Our program should be composed of as much function as possible to reduce the length of the main function and make it easy to modify, not only during the production stage but even after. This will help us improve the program in the future if necessary but it will also help us extract interesting functions from our program.

- **Well Documented**

Documentation is a must for any program, this will assure easy comprehension of the Code for others but also keeps track of its progress during the production stage. Comments will especially come in handy to explain functions using unusual libraries that are hard to understand without prior knowledge.

- **Simplicity**

Each function inside the code must be as simplified as possible, there should be no waste of lines and no useless operations. Simplicity is key.

- **Configurable**

For it to be intuitive for users, the program should make it easy to change parameters that change variables like "key-lengths" or the number of loops.

4..1.2. Functional requirements.

- **Fast generation of primes**

One major point of Time Lock Puzzles is the fact that they must be able to be generated fast for real-life applications. For that, as explained previously in Section 3.1.2 our program needs prime numbers, and it is fundamental for us to be able to generate them as fast as possible.

- **Generate precise Data**

One of our goals is to collect precise data about the time taken by the computer to find the key. So we must be able to only time the operations used to find the key and we must be sure that the allocated calculating power is the same.

- **Generate readable Data**

The data we generate must be in a readable format while also being in the correct format for it to be read by our second program.

4..2. Design

Since this implementation is done in two different Programming languages, we will talk first about design points that are valid for both implementation's while specifying elements that are specific to either of them.

4..2.1. Time Lock puzzle.

Our project first goal is to implement a Time Lock Puzzle based on the RSW scheme (Section 3.1.2). However our finality is not the encryption of a message. What we are looking for is timing how much does it take for a computer to find key's of specific length's. i.e. Timing how long does it take to solve the puzzle. For that we will tweak the RSW implementation to only hold what we want.

Our program should first find 2 primary numbers of a specified bit-length and calculate a composite modulus n such that :

$$n = pq \quad (9)$$

we also would normally calculate t (Section 3.1.2) however we will take a predefined array of t variables since the change's on the result made by this parameter is not interesting for us. Finally we will take a random integer a such that :

$$1 < a < n \quad (10)$$

Finally we will directly simulate an attack against this puzzle and try to find the key k . For that we must compute it as such :

$$k = a^{2^t} \pmod n \quad (11)$$

4..2.2. Data collect.

The program should start keeping track of the time at the start of the attack and this until it finishes. It must record all the data in a separate text file. The data recorded must be formatted in a specific way so that our second deliverable can read it. It must present the bite size of the key that was found alongside the time taken to solve the puzzle for a specific parameter t .

4..2.3. Scripting.

What we are looking for in this project is to generate a lot of data with different bit-sized keys and for different parameters t . To attain this result, we will make it so that our programs loops in a way that for each bit-size we specify, our program will decipher 10 puzzles with ten different values of t . Before incrementing again a fixed value to the bit-size of the keys and deciphering another ten puzzles with the same 10 different values of t . This automatizing will help us run the script during long hours without the necessity of a human presence to control everything.

4..2.4. Memory in C++.

When designing a program in c++ it is really important to take into account the fact that c++ can only hold variables of 64 bits. However when dealing with subjects such as Cryptography usually there is a need for variables that exceed this limit by a very big margin. To overcome this problem we will need to use the GNU library Section 3.3.

4..3. Production

4..3.1. C++ Code.

The key feature of our code is the use of the GNU library. To Be able to use it in your code you must add the line : `#include <gmpxx.h>` .

GNU is a super big library that has multiple features that we won't all explore for our project. We will be mainly using the `mpz_class` and functions related to it. This will make it possible for us to store very big variables inside them and use them.

Randomness

Randomness is an essential component of any cryptographic related device. Attackers shouldn't be able to guess a key by watching pattern's in the generating of it. Even if "true" randomness is not achievable on a computer when this document is being written, it doesn't mean we can't approach it.

We also need to take into account the fact that the random number generated must be of any length that will be assigned to it before generation meaning that we will have to bypass the c++ limitation of variables. Generally, to generate random numbers, computers will use what's called a seed. Each time the function to generate a random number runs, the functions must first be fed a seed which will be the base of our future generated number. The function then uses a complex mathematical algorithm to build upon that seed and transform it into a number that conforms to what we expect from the function. This means that using the same seed will give us the same number, meaning that our seed must come from somewhere attackers can't have access to. This also means we need to ensure that the probability that the function is given the same seed is low so that it doesn't slow down our calculations.

It seems pretty logic then to have the first function called 'getRandomInt()' in our program to generate this seed.

```
unsigned long getRandomInt(){
    std::random_device rd;
    return rd();
}
```

For that, we will use an inbuilt function in c++ called `std::random_device` and make it return to us an unsigned long int. This function should be good enough to generate the seed as we need it to be a big number to avoid the chances of getting the same seed, while not being too complex. It is common to use the time as a seed for random functions but in our case, it is a terrible choice as it will limit us to a single generation of integer per second. It is also quite common to use the entropy generated by the computer as a seed but this fails on Virtual Machines as they don't generate enough of it. That's why we will stick to our function.

Now Onto the main part, for our Time lock puzzle we will need two random functions, A function we will call 'fonctionrandom' it will take as a parameter the number of bits the outputted number should be. This is the functions that will generate the prime candidates.

```
mpz_class fonctionrandom (int bits) {
    gmp_randclass X(gmp_randinit_default) ;
    X.seed(getRandomInt());
    mpz_class z = X.get_z_bits(bits);
    return z ;
}
```

the first line in the function is setting the algorithm that will generate the random number from the seed. We chose the default one that comes in the library. The second line will generate a seed `X` using `getRandomInt()`. Finally the 3rd line will assign to the variable `z` that is an `mpz_class` a random number of the specified bits length using the seed `X` and the function `.get_z_bits()`.

The second function will be called `fonctionarandom()`, it will generate for us the `a` variable that we discussed about in Section 3.1.2. It will do this by taking as a parameter an `mpz_class n` and output an `mpz_class z` such that $2 \leq z \leq n-2$.

```
mpz_class fonctionarandom (mpz_class n) {
    gmp_randclass X(gmp_randinit_default) ;
    X.seed(getRandomInt());
    mpz_class z = X.get_z_range(n-2);
    if (z == 0 | z==1 ) ; {
        return 2 ;
    }
    return z ;
}
```

The two first lines are the same as the `fonctionarandom()` function. In the third line an `mpz_class z` will be assigned a random number such that $2 \leq z \leq n-2$, using the function `get_z_range`. To avoid having a 0 or a 1 we write an if statement line 4 and 5 in the function to return a 2 in case `z == 0` or `z == 1`.

Prime number generation

Before explaining how prime generation functions are done in our program, we are going first to show and explain two functions that we are going to write to make exponential calculation's easier for us to write. The goal is to reduce the number of parameters to give to do an exponential calculation with an `mpz_class` from 4 to 2 in case of a power of 2 and to 3 parameters in case of a power to an `x` number. the first function called `exp_one()`

```
mpz_class exp_one(mpz_class a, mpz_class b,
    mpz_class c){
    mpz_class r;
    mpz_powm(r.get_mpz_t(), a.get_mpz_t(),
    b.get_mpz_t(), c.get_mpz_t());
    return r;
}
```

This will help us do exponential for any number with 3 parameters instead of 4. In the first line of the function we create a variable `mpz_class r` and then we use the function of the library GNU `mpz_powm()` that takes 4

parameters : r will be the output result of the function , a will be the number that will be risen to the power of $b \bmod c$. The second function will be for raising numbers to the power of two by only specifying the number to raise and the mod.

```
mpz_class exp_two(mpz_class a, mpz_class b){
    mpz_class r, c;
    c = 2;
    mpz_powm(r.get_mpz_t(), a.get_mpz_t(),
    c.get_mpz_t(), b.get_mpz_t());
    return r;
}
```

the principle is the same we just set a variable c that will be the power to always be equal to 2.

Now the first step for us is to write a function that check's if a prime is number using the Miller-Rabin test Section 3.1.3. This function which return a Boolean value is named *millerTest()* and takes two *mpz_class* variables, n which is the number to test and d which is equal to m in the equation :

$$n-1 = 2^k \cdot m \quad (1)$$

We will show how to get the variable m later.

```
bool millerTest ( mpz_class d, mpz_class n){
    mpz_class a = fonctionrandom(n) ;
    mpz_class x = exp_one(a,d,n) ;

    if (x == 1 || x == n-1){
        return true;
    }
    while (d != n-1){
        x = (x*x) % n ;
        d *= 2 ;
        if (x==1){
            return false ;
        }
        if (x == n-1){
            return true ;
        }
    }
    return false;
}
```

In line 1 of the function we generate the variable a with a random number such that $2 \leq z \leq n-2$. Then we proceed to the miller-Rabin test. So we raise a to the power of $d \bmod n$, and assign that value to x . If x is equal to 1 or $n-1$ then the number is probably prime(return *true*), if not then we will do step 4 of the miller-Rabin test. We will square $x \bmod n$ but this time if x is = 1 then the number is composite (return *false*), and if it is = $n-1$ then it is probably prime(return *true*). We will insert this step 4 in a while loop to avoid getting stuck indefinitely. Each time, d will be multiplied by 2 and if it becomes equal to $n-1$, then we will consider n to be composite and return *false*.

By now we can arise the question, if the number is probably prime how can we insure it is totally prime ? For that we will create another function that will be the backbone of our primality test. It's name is *isPrime* and it's a Boolean function that takes as parameters two *mpz_class* n and k with n being the number we want to test and k an integer that will talk about later.

```
bool isPrime(mpz_class n, mpz_class k ){
    if ( n<= 1 || n==4) return false ;
    if ( n<= 3 ) return true ;
    mpz_class d = n -1 ;
    while (d %2 == 0 ) {
        d /= 2 ;
    }
    for ( mpz_class i = 0 ; i<k ; i++){
        if (!millerTest(d,n)){
            return false ;
        }
    }
    return true ;
}
```

The first two lines of our function are test-corners to gain time in case $n \leq 4$ Then we assign to a variable d $n-1$. Now we need to make some changes to d for it to be useful for our Miller-Rabin test. For that we will divide it by 2 until it is not even anymore. When it is not we take the last value when it was even and assign it to d . Finally we can run the miller Test function we describes earlier, and this is where the variable k comes in handy. Since the Miller-Rabin test can only tell us if a number n is probably Prime we can run it a multiple of time until the probability of it being not primes is acceptable by the user standard's. So this is what the for loop does, it redo the miller test k times. If the miller-Rabin Test is *false* the function *isPrime()* returns *false* and if it is not after the looping then the function *isPrime()* return's *true*. Finally We just need to mix all the functions to get a functions that generates Prime numbrs. This function will be named *generateprime()* and will take as a parameter an int u that is the size of the prime number to generate in bits.

```
mpz_class generateprime(int u) {

    mpz_class k = 2 ;
    mpz_class i ;

    while (k != true) {
        i = fonctionrandom(u) ;
        if ( isPrime(i,k) == true ){
            k = true ;
        }
    }
    return i ;
}
```


We first specify the variable k that we talked about previously it is to be changed by the user to choose the degree of probability that the number generated is prime. Then we created a while loop that will generate a random number of size u bits with the function we explained earlier *fonctionrandom()*. Then we test the number generated with the *isPrime()* functions and if the number is probably prime then the loop stop's and the functions returns the prime number.

Time lock puzzle solving.

As we explained in Section 3.1.2, to solve the time lock puzzle we must find the variable b by performing the equation :

$$b = a^{2^t} \pmod{n} \quad (1)$$

That's what is done in our function *rsu()* that takes as parameters : g that represents a in the equation, t that is t in the equation and n which is also n in the equation.

```
mpz_class rsu(mpz_class t, mpz_class g ,
mpz_class n){
    mpz_class s = g ;
    for ( mpz_class i = 0 ; i<t ; i++){
        s = (s*s) % n ;
    }
    return s ;
}
```

In the first line we create an *mpz_class* variable named s that will be equal to g . After that we do a for loop to mimic the a^{2^t} , we also use this for loop to do the $(\text{mod } n)$ step after each calculation which makes our calculations faster.

Main Function.

Our Main function is supposed to put in together all the Requirements and make use of all our functions we defined previously and that is exactly what it does:

```
int main() {
    ofstream myfile ;
    myfile.open("data.txt") ;
    int T [] = {14,15,16,17,18,19,20,21,22,
23};
    int i = 0 ;
    int modulus_size = 81 ;
    while ( i <= 100 ) {
        myfile << modulus_size << endl ;
        mpz_class p, q, n ;
        p = generateprime(modulus_size) ;
        q = generateprime(modulus_size) ;
        n = p*q ;
        mpz_class g = fonctionarandom(n) ;
        if (g == 0) {
            g = 1 ;
        }
        for (int z = 0 ; z < 10 ; z++){
            mpz_class t = pow(2,T[z]) ;
```

```
std::clock_t start ;
double duration ;
start = std::clock() ;
mpz_class s = rsu(t,g,n) ;
duration = (std::clock() - start) /
(double) CLOCKS_PER_SEC ;
myfile << duration << endl ;
}
modulus_size += 81 ;
i++ ;
cout << "finished"<< endl ;

}
myfile.close() ;
return 0 ;
}
```

The two first lines opens the file we are going to record data in. The name can be changed according to the user's preference. T is an array of the different t parameters we are going to use for the generation of key's. Int i will just be a counter for our loops. The variable *modulus_size* is an int that will represent the size of the key we are generating in bits.

The first while loop will specify how many different key sizes we want to generate, by default it is 100 meaning that starting from 81 which is the first *modulus_size* we will have 100 different key lengths. At the end of each loop we double the modulus size variable so that we are not stuck with the same one. Inside the while loop, the program will first write the modulus_size in our file. We then generate two prime numbers with the function *generateprime()* and we calculate n wich is $= pq$. We also generate a random number bigger then 0 with the function *fonctionarandom()* and assign it to an *mpz_class* g .

Inside the while loop we will have a for loop that will calculate what represents the variable e in the RSW scheme. For that we will pick the first integer inside the array T and calculate 2^T and assign it to t .

Now before starting the *rsu()* function which will solve the time lock puzzle by finding the key, we will start a counter using the internal clock of the computer. We will assign to *start* a value that represents the time when the line was executed. The program will then solve the puzzle and directly after he is done will subtract from the current time when the actions is done the value of *start*. We then assign to the variable *duration* our result and we have the time it took for our puzzle to be solved in seconds. This duration is then recorded in the data file. This loop will circle ten times each time using a different integer from the

T array. When both loops are finished We will close the data file to avoid buffer problems and stop the program. To use all these commands we must import some libraries using these commands.

```
#include <bits/stdc++.h>
#include <cmath>
#include <gmp.h>
#include <gmpxx.h>
#include <iostream>
```

4.3.2. Python 3. Our program in Python 3 is way simpler as the use of dedicated and performant libraries will reduce considerably the length of the code. We will only write one function before the main one which will be the *rsw()* function. It is exactly the same as the one we have in our previous code so there is not a need to explain it.

```
def rsw(t,g,n) :
    s = g
    for i in range(t):
        s = s*s %n
    return s
```

Our test function is also line by line equivalent to the main function in the c++ program, but with function names change since we will be using libraries instead of our own functions. He is the main function :

```
def test() :
    with open("data.txt","w") as datafile :
        T = [14,15,16,17,18,19,20,21,22,23]
        z=0
        i=0
        modulus_size = 81
        while i<100 :
            datafile.write(str(modulus_size)+
                "\n")
            p = number.getPrime(modulus_size)
            q = number.getPrime(modulus_size)
            n = p * q
            g = random.randint(1, n)
            compt = 0
            for z in range(0,10) :
                t = 2**(T[compt])
                start = time.time()
                s = rsw(t, g, n)
                end = time.time()
                datafile.write(str(end-start)
                    "\n")
                compt += 1
            datafile.flush()
            modulus_size = modulus_size+81
            i +=1
```

The first change is that we use the function *number.getPrime()* to get our prime numbers. This function is taken from the library *Crypto.Util* and that we import using this command

```
from Crypto.Util import number
```

We also use a library to get the random number g for that we use the function *random.randint()* from the library *random* that we import using this command:

```
import random
```

We also must import the library *time* to use the function *time.time()* using the command :

```
import time
```

We just have to run this function inside the main function of the Python program and that's it. To do that we write the function main like this :

```
def main():
    test()

if __name__ == '__main__':
    main()
```

4.4. Assessment of the requirements.

Easy to use and moduable

- We've put a lot of effort into fragmenting our program as much as possible, especially the one written in c++. We created a lot of functions to make it moduable and even created functions to simplify calculations. Since we used a lot of libraries in the python program and due to its nature of having lots of specialized libraries that are simple to use, we don't have a lot of functions but the program remains super simple and concise. So we can affirm that this point holds.

• Well Documented and Simple

Our program should be well documented, as each function has a line that explains the principal utility of it. Each function is designed to keep it to a minimum of lines while breaking down all the calculations. Each function should be easily understandable line by line by any user with a minimum of knowledge. We can also say that this point holds.

• Configurable

For the program to be able to give us data formatted in a certain way so that can be read by the second deliverable, we had to sacrifice some

practicality. The user will have to change the code because by default the program will only work in a certain way without taking into account the user's preference. We can say that this point is partially not held.

- **Fast generation of primes**

Our prime generation function is probably the fastest we can achieve with knowledge limited to a first-semester project. It can generate primes in matters of seconds to minutes depending on how big the number is. The point is as such considered as held.

- **Generate precise Data**

Using the computer's internal clock we precisely record the amount of time it takes to do a specif task. So the data is as precise as it gets.

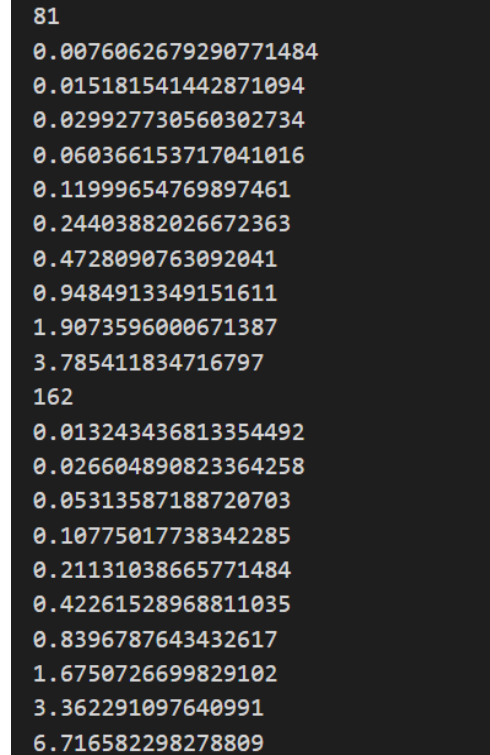
4..5. Data graphic's

In this section, we will talk about our second technical deliverable, it will be a short section as the program in itself is simple, written in Python 3, and supposed to create graphs for us so that we can analyze the data that we get from running our main technical deliverable. Our program will only take into account the ten first lengths of keys in the data file. We can't display the code in the report since it is very long but to explain it we can show a screen of our data file. This is a screen for only two lengths of keys respectively 81 bits and 162 bits. For each size of a key, our program tried to decipher ten time lock puzzles with different t parameters. That's why each length will have 10 different lines of data that represent in seconds how much time did the computer take to resolve the puzzle. Our program stores the first 10 lengths of keys in a list. It then creates a list for each length and record in it each line of data it has. Now the most important part once we have all theses lists is plotting them using the Library matplotlib we will first import what we need using the command

```
import matplotlib.pyplot as plt
```

We then use the function `plt.plot` to draw one of the 10 lines that will constitute our graph. We will dissect one of theses lines :

```
plt.plot(liste_of_t, list1, label="module is " +  
str(list_of_modules[0]), color='green')
```



```
81
0.0076062679290771484
0.015181541442871094
0.029927730560302734
0.060366153717041016
0.1199654769897461
0.24403882026672363
0.4728090763092041
0.9484913349151611
1.9073596000671387
3.785411834716797
162
0.013243436813354492
0.026604890823364258
0.05313587188720703
0.10775017738342285
0.21131038665771484
0.42261528968811035
0.8396787643432617
1.6750726699829102
3.362291097640991
6.716582298278809
```

Fig. 2. Screen from our data files

The first parameter is the x axis and that's the list of t parameters that were used. The second parameter is the y axis it's the list of the data that was recorded for a key-length which means this parameter must be changed for each line according to its key-length. The third parameter is a label for the line and the forth is a color for it. We repeat this line for the number of key-length we want to take into account. Finally we must label the whole graph we use the function `plt.xlabel()` to label the x axis of the graph and the function `plt.ylabel()` to label the y axis of the graph. finally we give a title to the graph with the function `plt.title()`.

4..6. Analysis

After running our main technical deliverable for hours we get two data files, one for the program in Python and one for the c++ one. We then use our second technical deliverable to create the two graphs we can find in the appendix of this report Section 6.1 . The first detail we can observe is that there is an exponential relation when solving our Time lock puzzle between the argument t and the key length.

Meaning that each time we increase our key length the parameter t will have an even bigger influence over the time taken to solve the puzzle. This helps us prove an already stated fact, the bigger the key the more secure it is. But this also proves that we need to find a good balance between security and time taken since the usage of time lock puzzles in real-life applications might need faster times especially for transactions etc... That's why RSW key sizes are generally of size 1024, 2048, or 4096 bits. There is another thing that we can note, on the y axis of both graphs if we compare them, the Python program seems to take much longer than the c++ one to solve the puzzle. On these graphs, we can observe that the Python program takes close to 10 times more to decipher keys of the same length with the same t parameter. This gets even more shocking when we talk about really big key lengths for example a time lock puzzle with a key of length 6399 bits and with a $t = 23$ will be solved in 3257.985 seconds by the Python program while it will be solved in only 210.558 by the c++ program, it's almost 16 times faster. We could try to explain this with the use of what we said in Section 2.3. We showed that c++ is much more efficient than Python when it comes to memory management even if less practical. This could be exactly why the c++ solving is being done faster, While c++ is just using the *mpz_class* to allocate the memory needed for big sized variables ahead of time, Python with its dynamical allocation of memory is reallocating memory after each calculation That is all the more true, especially when our program is dealing with exponentially growing numbers. We can then deduct that this is the reason C++ is way faster than Python, We can ask our self if it's possible to make Python faster in theses type of calculation by implementing a way to have it have a static memory but this is not in the scope of this BSP. This also shows while We can try to Make puzzles that would take years to be solved, this prediction is only valid if the parameter S which is the number of squaring modulo n per second that can be performed by the solver is constant. But this same parameter is generally based on the technology available which is not a good thing in a world where technology thrives. This means, that the Time lock puzzle we created, while mathematically correct is still unsafe when created for a long amount of time since technological improvement is not taken into consideration.

5.. Acknowledgment

I would first like to thank Răzvan Roşie for being my PAT and supporting me throughout the entirety of my first BSP at the Luxembourg University. He answered all my questions during the semester while taking time to explain to me carefully via Web-ex but also via email things that I did not understand. He also gave me lots of advice outside of the BSP project for my future as a computer scientist.

6.. Conclusion

During this project, we approached the concept of cryptography through a direct implementation of a time lock puzzle with the RSW scheme. We learned a lot about the mathematics behind cryptography, but we also grew as computer scientists since we learned how to plan the construction of a program, professionally written the code, and confront problems that arose throughout the project. We learned how to improve our programs and optimize calculations and this with two completely different programming languages gave us a peek at how vast the world of computing science is.

References

- [1] *Cpp_overview*. URL: https://www.tutorialspoint.com/cplusplus/cpp_overview.htm (visited on 12/11/2020).
- [2] *Cryptography*. URL: <https://en.wikipedia.org/wiki/Cryptography> (visited on 11/29/2020).
- [3] *Python_overview*. URL: https://www.tutorialspoint.com/python3/python_overview.htm (visited on 12/11/2020).
- [4] Ronald L Rivest, Adi Shamir, and David A Wagner. "Time-lock puzzles and timed-release crypto". In: (1996).
- [5] *The GNU MP Bignum Library*. URL: <https://gmplib.org/> (visited on 12/12/2020).

6..1. Appendix

graph of the time the computation took by the argument t for ten different modulus

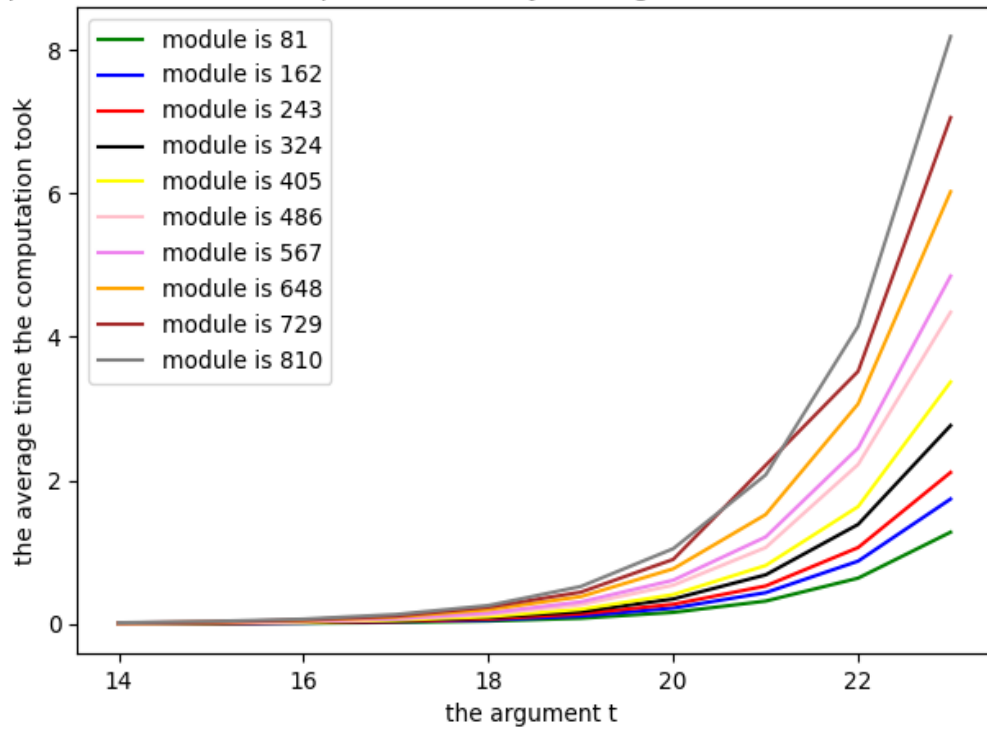


Fig. 3. Graph of the c++ data

graph of the time the computation took by the argument t for ten different modulus

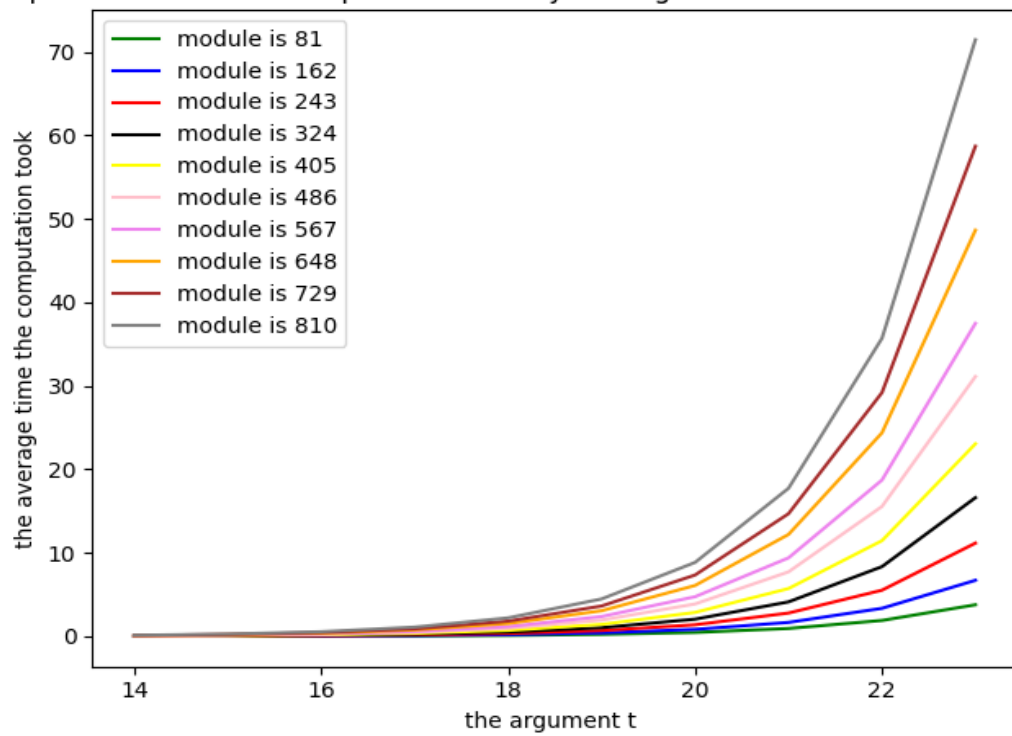


Fig. 4. Graph of the Python 3 data