An introduction to PAKE'S through a J-PAKE-CRS Implementation

Wednesday 19th May, 2021 - 20:00

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

Tutor: Peter Y A RYAN
University of Luxembourg
Email: peter.ryan@uni.lu

Abstract

Our second BSP is an introduction to a special form of cryptographic key exchange protocol called PAKE which stands for Password Authenticated Key Exchange. As of today, PAKE's are amongst the most common cryptographic primitives as they permit two or more parties to establish cryptographic keys based on the knowledge of a secret. In this project, we will discuss how we implemented such PAKE called CRS-J-PAKE based on the J-PAKE. Throughout this paper, we will try to address all scientific and technical details to give an overview of all elements that take part in our protocol.

1. Introduction

The internet is the largest global communication network connecting millions of computers. And, as it is gaining in popularity throughout the world, its place is more and more important in the daily life of any human being across the globe.
This is not without consequences since the democratization of this communication device makes it of the utmost importance to make sure the private life of the user is actively being protected from any harm. For that Everyone on the internet uses the predominant method of using passwords due to their ease of use.But these passwords are usually not long and susceptible to brute-force guessing attacks.
There exist other authentication factors, but they require the usage of additional hardware that implies a financial implication. That is why the password method is the one usually deployed coupled with some protocols to protect it, but these methods require a Public Key Infrastructure (PKI) which also requires a financial investment.
That's where Password-Authenticated Key Exchange also known as PAKE tries to solve this problem. What these schemes try to do, is to make sure no password-driven data is being exchanged across an insecure channel. They do that by constructing a cryptographic key exchange that will make sure that both parties prove possession of the shared password without ever transmitting it to the other. Since the passwords are never exchanged through these insecure channels, they are therefore protected from attacks like offline brute-force dictionary attacks. In fact a PAKE protocol muss fulfill various security requirements such as :

1) Off-line dictionary attack resistance
2) Forward secrecy
3) Known-session security
4) On-line dictionary attack resistance

as described in [9].
The idea of such a protocol was first described by Bellovin and Merritt (cf. [1]). Since then, the idea became widely known, and there exists a multitude of PAKE-derived protocols.

2. Project Description

2.1. Domains

The main idea behind this project is to make sure the student learns about the important concept in cryptography that is PAKE's in detail by trying to implement them. At the end of the project, the student must have learned about various concepts revolving around the main topic such as the Diffie Hellman key exchange, Elliptic curves, Zero-Knowledge proofs, etc...
In this optic, the student, with the help of his tutors, will first try to implement different variants of the PAKE protocol, such as the J-PAKE and the CRS-J-PAKE. This implementation will be in both Python and C++.
Afterward, the student will try to optimize his work

by following the guidelines of the tutors. The optimization will focus on making the program faster and more secure against attacks such as Timing Attacks. Therefore, we could say that this project will focus on two main scientific domains.

1) Cryptography.
2) Optimization and analysis.

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" [3]. So in general Cryptography would be the science of creating algorithms or protocols to prevent third parties[1] 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.

In this project, we will focus more on security protocols. A security protocol " is an abstract or concrete protocol that performs a security-related function and applies cryptographic methods, often as sequences of cryptographic primitives. A protocol describes how the algorithms should be used. A sufficiently detailed protocol includes details about data structures and representations, at which point it can be used to implement multiple, inter operable versions of a program" [4]

1. Meaning other parties then the sender and receiver.

Optimization and analysis

As mentioned previously in Section 21, an encryption algorithm should be as secure as possible to avoid having breaches and to avoid the possibility of the data it transmits 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. In our case, it is of the utmost importance that there is no leakage of information and that it is perfectly invulnerable to attacks such as Timing Attacks as the whole privacy of the user is dependant on this protocol alone in most of the cases.

2.2. Programming Languages

Python

"Python is a high-level, interpreted, interactive and object-oriented scripting language" [10]. 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.

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 [2]. 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. 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.

The main deliverable

Our main deliverables will be our various implementation of the J-PAKE and CRS-J-PAKE variants of the PAKE protocol. More precisely, we can divide our deliverables into two categories. The ones written in Python, and the ones written in C++. In our python category, we will have an implementation of a basic J-PAKE and CRS-J-PAKE. In the C++ category, we will have an optimized version of our CRS-J-PAKE that we implemented in Python. This implementation will have as a goal to be efficient while being more secure than our previous implementation by using various techniques that we will discuss later on.

3. Background

3.1. Scientific Background

3.1.1. Elliptic curves. For the sake of understanding our project, it is important to understand one of its core components called elliptic curves. Elliptic curves are at the core of what we call Elliptic Curve Cryptography or ECC. Today, elliptic curve cryptosystems can be found in a multitude of cryptographic protocols and concepts, such as TLS, PGP, and SSH, as well as for cryptocurrencies like bitcoin.
What makes ECC stand out, is its ability to create smaller key sizes than other methods like the RSA while maintaining a high level of security. "ECC bases its approach to public-key cryptographic systems on how elliptic curves are structured algebraically over finite fields"([6]) which makes, in theory, the keys generated harder to crack.
To understand why choosing ECC is better than RSA performance-wise we need to take a look at the security yield of the encryption keys which is notable. For example, an ECC key of $521bit$ achieves the level of security a $15360bit$ RSA key offers. The difference is tremendous. This doesn't mean ECC is not vulnerable to anything. Several potential vulnerabilities need to be addressed when implementing it and we address those points in (insert ref to technical).

Weierstrass curves

In this introduction to the concept, we will be focusing on Weierstrass curves as this is the type of curve that we implemented in our project. We will define such a curve as a set of points described by the equation:

$$y^2 = x^3 + ax + b \tag{1}$$

Such that $4a^3 + 27b^2 \neq 0$.
Depending on the value of a and b, these curves will have different shapes on the plane, but they will all have in common the symmetricity about the $x-axis$. It should also be known that any non-vertical line will intersect the curve in three places or fewer. We also need a "point at infinity" to be part of our curve and that we denote with the symbol zero "0".

group law

With all these elements we can finally define a group law over the elliptic curves.

1) The elements of the group are the different and infinite points that constitute the elliptic curve.
2) The identity element is the point at infinity that we defined to be 0.
3) The inverse of a point X is the one symmetric on the x-axis.
4) Addition follow the rule that given three aligned, non-zero points A, Z, E their sum is $A+Z+E = 0$.

Finite field F_P

Now for the sake of implementation, we will restrict our elliptic curves to a finite field F_p . But before that, we must define what does x/y means in F_p. x/y would be equal to $x.y^-1$ which in plain words would mean x times the multiplicative inverse of y. This will be done using the extended euclidean algorithm to find the inverse, but we will talk about technical details later. We can now give a new definition to our elliptic curve that is now restricted and takes into account the infinity point.

$$\{(x, y) \in^2 \mid y^2 = x^3 + ax + b,\ 4a^3 + 27b^2 \neq 0\} \cup 0 \tag{2}$$

Algebraic sum

We will now define the equations to calculate Algebraic sum (point addition). For a point $P = (x_P, y_P) Q = (x_Q, y_Q) and R = (x_R, y_R)$ we can calculate $P + Q$ as $-R$:

$$x_R = (m^2 - x_P - x_Q) mod P \tag{1}$$

$$\begin{aligned} y_R &= [y_P + m(x_R - x_P)] mod P \\ &= [y_Q + m(x_R - x_Q] mod P \end{aligned} \tag{2}$$

If $P \neq Q$ the m is equal to:

$$m = (y_P - y_Q)(x_P - X_Q)^{-1} mod P \tag{3}$$

Else if $P = Q$ then m is equal to:

$$m = (3x_P^2 + a)(2y_P)^{-1} mod P \tag{4}$$

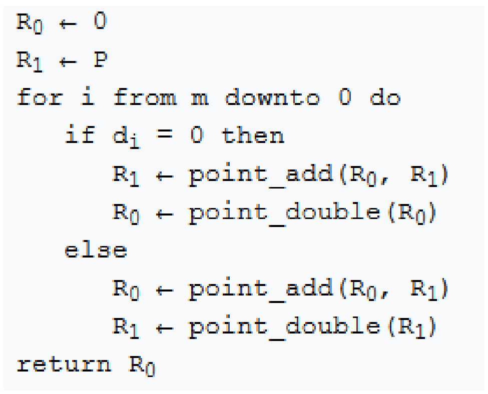
Where a is a parameter of the curve.

Scalar Multiplication

For scalar multiplication, it is even easier we just need to add the point to itself n times.

$$nP = P_1 + P_2 + ... + P_n \tag{1}$$

But since this approach is in itself very inefficient and THE double and add algorithm is susceptible to Timing attacks, We must learn about the Montgomery Ladder which is an algorithm that approaches point multiplication in a fixed amount of time. Here is the pseudo-code for such a Montgomery Ladder:



```
$R_0$ ← 0
$R_1$ ← P
for i from m downto 0 do
    if $d_i$ = 0 then
        $R_1$ ← point_add($R_0$, $R_1$)
        $R_0$ ← point_double($R_0$)
    else
        $R_0$ ← point_add($R_0$, $R_1$)
        $R_1$ ← point_double($R_1$)
return $R_0$
```

Fig. 1. Montgomery ladder. algorithm from [7]

3.1.2. Diffie-Hellman. Before explaining how both J-PAKE and CRS-J-PAKE work, we will explain how the Diffie-Helmann exchange works as it is the basis of both PAKE'S protocols. The Diffie-Hellman is a way to share a secret between two people without sharing enough information over the communication channels for a hypothetical listener to deduce this secret. This is the property that assures forwards secrecy for PAKE's as talked about in Section 1. This property is done totally through the mathematical properties that groups offer.
let Alice choose a prime number p and a number g such that $g < p$.
Alice can now choose a random number a and calculate A such that $A = g^a[mod p]$. Alice then send g,p, and A.
Bob now chooses a number b and calculates B such that $B = g^b[mod p]$ and then calculate $K = A^b[mod p]$ Finally, Bob sends B to Alice.
Alice can now calculates $K = B^a[mod P]$ and they both have the same Key K since $K = g^{ab}[mod p]$, and this without ever sending any valuable information over the communication channel. We have in the appendix Section 51, an image that uses paintings as a metaphor might help understanding this concept. As long as the prime number P is big enough this method is completely secure theoretically. However in practice the Diffie-Hellman method is susceptible to the man-in-the-middle attack.

3.2. Zero Knowledge Proofs

"In cryptography, a zero-knowledge proof or zero-knowledge protocol is a method by which one party (the prover) can prove to another party (the verifier) that they know a value x without conveying any information apart from the fact that they know the value x." [14]
This ZKP must satisfy three properties :

1) Completeness: If the statement is true the other party must be convinced completely.
2) Soundness: If the statement is false the Other party must be sure to detect it.
3) Zero-Knowledge: When the statement is true, the other party must not learn anything else but that the statement is true. It must not by any means find a way to find the secret.

Tons of protocols can be considered Zero-Knowledge but we will focus on the Schnorr protocol as it is the one we will use in our implementation. To understand how it works let's imagine we are in the Diffie-Hellman

context we just explained.
As we said Alice send A, but we need her to prove to Bob that she knows a. To do that, after Alice sends A along R such that $R = G^r$ where r is a random number smaller than p, Bob will pick a random number c such that $c < P$ and send it to Alice.
Bob will then know that Alice has a if $g^{pi} \times A^c = R$ because $g^{pi} \times A^c = g^{r-c\times a} \times g^{a\times c} = g^r$.
But as we can see this protocol as it is right now is not efficient at all we need to send and receive a lot of information. That is why using this schnorr protocol coupled with the Fiat-Shamir heuristic will permit us to construct a Non-interactive zero-knowledge proof that will need Alice to only send information once.
To do this, Alice will construct herself the challenge so that she sends to bob both the proof and the challenge he then just needs to do the calculation to check that the proof is right. We will take the same settings as before we will just add a hash function that we will call H that is a one-way function that will give the same output for the same input.
Alice will like last time pick R such that $R = g^r$ where r is a random number smaller than p. Alice will then calculate c such that $c = H(G||a||A)$ and pi such that $pi = r - c\,a\,[mod p - 1]$ Alice will then send pi with R, c, and A. if $R = g^{pi} \times A^c$ then like last time, Alice has proven that she knows a. The parameters we put in c might change depending on the implementation.

3.2.1. J-PAKE. J -PAKE is, as described by the RFC8236 [9] a "PAKE protocol called Password-Authenticated Key exchange by juggling, designed by Feng Hao and Peter Ryan in 2008". J-PAKE is original in that it has security proofs, and that it is built upon a "Well established Zero-knowledge proof (ZKP) primitive" Schnorr NIZK proof" as well as optimizing the use of theses for practical use. J-PAKE is designed to be working with both RSA and ECC initialization but we will talk about the ECC version of it. The J-PAKE protocol we will describe is taken from the RFC8236 [9].

Protocol setup

Let $E(F_p)$ be an elliptic curve defined over a finite field F_p, where p is a large prime. Let G be a generator for the subgroup over $E(F_p)$ of prime order n. We will define Alice and Bob as being the two communicating parties. We first need them to have one shared secret called s such as $1 < s < n - 1$.

1^{st} round

Alice will first select two keys called $x1$ and $x2$ that are both chosen randomly from $[1, n-1]$. Bob does the same thing but his keys will be called $x3$ and $x4$. Alice will then send to bob $G1$ and $G2$ such that

$$\begin{aligned} G1 &= G \times [x1] \\ G2 &= G \times [x2] \end{aligned} \tag{2}$$

With that, Alice will send the ZKP for both $x1$ and $x2$ Bob will do the same with $G3$ and $G4$ such that

$$\begin{aligned} G3 &= G \times [x3] \\ G4 &= G \times [x4] \end{aligned} \tag{3}$$

Bob will then send the ZKP for $x3$ and $x4$. The ZKP's contain $(UserID, V = G \times v, r = v - d * c[modn])$. v is taken randomly from $[1, n-1]$ and $c = H(G + V + GX + userID)$, with H being a hash protocol and GX being G times the secret we want to prove that we have. both Alice and Bob should check the ZKP before going to the next round otherwise the session is aborted.

2^{nd} round

Alice will now send A,

$$A = (G1 + G3 + G4) \times [x2 \times s] \tag{4}$$

and a ZKP for $[x2 \times s]$.
Bob will send B

$$B = (G1 + G2 + G3) \times [x4 \times s] \tag{5}$$

and a ZKP for $[x4 \times s]$.
THE ZKP is computed the same way it's just G that changes and becomes (G1 + G3 + G4) for Alice and (G1 + G2 + G3) for Bob. as usual, both parties should check the ZKP before proceeding to the next round.

Final round

Finally Alice will compute

$$Ka = (B - (G4 \times [x2 \times s])) \times [x2] \tag{6}$$

And, Bob will compute

$$Kb = (A - (G2 \times [x4 \times s])) \times [x4] \tag{7}$$

both Ka and Kb should be equal and as such they both have the same key. As much as J-PAKE is revolutionary in certain aspects the detractors of such a protocol point out the huge amount of exponentiations taken by the ZKP which directly impacts the efficiency of the algorithm. It's this problem that the next protocol will try to address CRS-J-PAKE.

3.2.2. CRS-J-PAKE.

The CRS-J-PAKE first described in is a protocol derived from the J-PAKE protocol by Jean Lancrenon, Marjan Skrobot, and Qiang Tang. Addresses the problem of too many exponentials in J-PAKE by introducing two novels algorithms the RO-J-PAKE and the CRS-J-PAKE. In this paper, we will focus on the CRS-J-PAKE as it is the algorithm we will implement in detail later on. CRS-J-PAKE enjoy's two less ZKP than the J-PAKE while maintaining a similar level of security which improves tremendously the performance of the protocol. They are also more efficient in terms of communication as they need fewer elements to be sent over the communication channel. The other particularity of CRS-J-PAKE is the Common Reference String (CRS) that is set in the Protocol when implemented and must be known by all parties that intend to use a particular implementation. This CRS will serve to erase the values $x1$ and $x3$ described in the J-PAKE protocol as we don't need them anymore. The setup round is almost the same as the J-PAKE we just need to define the CRS that we will denote as U and that both parties must

$1^{st} round$

Alice will select a key called $x1$ chosen randomly from $[1, n-1]$. and calculate $G1$ such that:

$$G1 = G \times [x1] \tag{8}$$

Bob does the same thing with $x1$ also chosen randomly from $[1, n-1]$. and calculate $G2$ such that:

$$G2 = G \times [x2] \tag{9}$$

Then both Alice and Bob will send the value they computed with ZKP for x1 and x2. The ZKP will almost be the same as the one used in J-PAKE, $(UserID, V = G \times v, r = v - d * c[modn])$ only difference will be that we will add U in the hash to compute c such that $c = H(G + V + GX + userID + U)$. Both Alice and Bob will check the ZKP before going to the next round otherwise the session will be aborted.

2^{nd} round

Alice will now send α such that:

$$\alpha = UG2 \times x2s \tag{10}$$

and the ZKP for x2s.
While Bob will send β such that:

$$beta = UG1 \times x1s \tag{11}$$

and the ZKP for x2s.
this time both ZKP won't include U in their hash. Obviously, Alice and Bob will once again check the ZKP before going to the next round and abort the session in case of incompatibility.

Final round

Once the last ZKP proved, Alice will now calculate K such that:

$$K = (\beta X2 \times (-x1 \times s) \times x1 \tag{12}$$

and use a hashing algorithm on this K to get sk which will be the final key obtained through this protocol. Bob will also calculate K but with a different equation:

$$K = (\alpha X1 \times (-x2 \times s) \times x2 \tag{13}$$

then he will also hash K with the same hashing algorithm to also get sk and as such obtain the same key as Alice finish the protocol and obtaining the desired result. A detailed figure taken from the paper [12] is available in the annex.

3.3. Technical Background

3.3.1. Python modules.

the secp256k1 is a file taken from the website "asecuritysite.com"[11] written by William J Buchanan, it is an implementation of the secp256k1 elliptic curve well known for being used in the Bitcoin's public-key cryptography. We will use this file for our Python implementations of the PAKE protocol and it will also serve as a basis for our C++ implementation of the secp256k1.

3.3.2. 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.[13]

OpenSSL

OpenSSL is one of the most popular cryptographic Libraries that offer a wide range of cryptographic tools. This library is written in the C language and was first released in 1998. In our case, we will be using OpenSSL to hash data using the Sha-256 which is a one-way function of the algorithm Sha-2 developed by the NSA. This algorithm can take a message up to 2^{64} bit and produce a hash of 256bits.

3.4. CRS-J-PAKE

In this section, we will specifically focus on our technical implementation of the PAKE protocol in C++. More precisely, our CRS-J-PAKE as it englobes all that we have learned during this project. We will first talk about our implementation of the seckp256k1 and all related arithmetical operations. We will also talk about how we designed functions to make our implementation less susceptible to timing attacks by avoiding conditional statements and other implementation choices.

A key feature in our code is the use of the GNU library. To Be able to use it in our code we must add the header : $\#include < gmpxx.h >$.
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(Section 332).

Elliptic curve

To start our implementation of the elliptic curve, we first define a $Point$ structure that will have two parameters x and y that will represent the coordinate of a point on the curve.

```
struct Point
{
    mpz_class x;
    mpz_class y;
};
```

We will also define a second structure called $Elliptic_Curve$ that will hold six parameters, one of them being a point.

```
struct Elliptic_Curve
{
    mpz_class p;
    mpz_class a;
    mpz_class b;
    struct Point g;
    mpz_class n;
    mpz_class h;
};
```

These six parameters represent all we need to initialize the Weierstrass curve of our choice. In our program, we will be initializing those parameters with the values of the secp256K1 curve.

```
Elliptic_Curve initialize_curve() {
    Elliptic_Curve ECC;
    ECC.p = ("115792089237316195423570985008687907853269984665640564039457584007908834671663");
    ECC.a = ("0");
    ECC.b = ("7");
    ECC.n = ("115792089237316195423570985008687907852837564279074904382605163141518161494337");
    ECC.h = ("1");
    ECC.g.x = ("55066263022277343669578718895168534326250603453777594175500187360389116729240");
    ECC.g.y = ("32670510020758816978083085130507043184471273380659243275938904335757337482424");
    return ECC;
}
```

p being the finite field F_p, a and b are the parameters in the Weierstrass equation, $g.x$ and $g.y$ are the coordinates of the base point, and finally, n is the order n of G, and h is the co-factor.

Now that we have our elliptic curve, we need to define all the arithmetic-related operations that we need. We will first start with the $inverse\_mod()$ function. It takes as a parameter two variables k and p both mpz_class and returns the multiplicative inverse of n modulo p as an mpz_class.

To do that, we first implemented an extended Euclidean algorithm to get a three tuple (gcd, x, y) such that $n * x + p * y == gcd$ of n and p. We then assert that the gcd we found is equal to 1 otherwise we abort the program. We then return the result of $x\%p$ which is the multiplicative inverse of n modulo p.

```
mpz_class inverse_mod(mpz_class k, mpz_class p){
    mpz_class temp ;
    if (k == 0) {
        throw "Division by zero condition!";
    }

    if (k < 0) {
        return p - inverse_mod(-k, p);
    }

    mpz_class s = 0 ;
    mpz_class old_s = 1;
    mpz_class z = 1;
    mpz_class old_z = 0;
    mpz_class r = p;
    mpz_class old_r = k;

    while (r!= 0) {
        mpz_class quotient;
        mpz_fdiv_q(quotient.get_mpz_t(),
old_r.get_mpz_t(), r.get_mpz_t());
        temp = old_r - quotient * r;
        old_r = r;
        r = temp;
        temp = old_s - quotient * s;
        old_s = s;
        s = temp;
        temp = old_z - quotient * z;
        old_z = z;
        z = temp;
    }

    mpz_class gcd = old_r;
    mpz_class x = old_s;
    mpz_class y = old_z;

    assert (gcd == 1);
    return x % p ;
}
```

We also implemented a boolean function called $is\_on\_curve()$ that will serve as a check method for future functions. This function takes as parameters the point we want to check if it is on the curve and the parameters of the said curve. If the point has coordinate (0,0) the point is on the curve as it is the point at infinity we defined. Otherwise, we just use a regular calculation method to check the condition.

```
bool is_on_curve(Point a, Elliptic_Curve curve){
    if (a.x == 0 && a.y == 0) {
        return true;
    }
    mpz_class x = a.x;
    mpz_class y = a.y;

    if (((y * y - x * x * x - curve.a * x
- curve.b) % curve.p) == 0) {
        return true;
    }

    else {
        return false;
    }
}
```

We will now talk about the $Point\_add()$ function that does the addition of two points by taking the said two points as a parameter with the elliptic curve and returns the result as a Point. The algorithm used is the one described in the scientific section of this paper(Section 311).

```
Point point_add(Point point1, Point point2,
Elliptic_Curve curve) {
    assert (is_on_curve(point1, curve));
    assert (is_on_curve(point2, curve));

    if (point1.x == 0 && point1.y == 0) {
        return point2;
    }
    if (point2.x == 0 && point2.y == 0) {
        return point1;
    }
    mpz_class x1 = point1.x;
    mpz_class y1 = point1.y;

    mpz_class x2 = point2.x;
    mpz_class y2 = point2.y;

    if (x1 == x2 && y1 != y2) {
        Point point0;
        point0.x = 0;
        point0.y = 0;
        return point0;
    }

    mpz_class m;

    if (x1 == x2) {
        m = (3 * x1 * x1 + curve.a) *
inverse_mod(2*y1, curve.p);
    }
    else {
        m = (y1 - y2) *  inverse_mod(x1 -x2,
curve.p);
    }

    mpz_class x3 = m*m - x1 -x2 ;
    mpz_class y3 = y1 + m * (x3 - x1);
    Point result;
    result.x = x3 % curve.p;
    result.y = -y3 % curve.p;

    assert (is_on_curve(result, curve));

    return result;
}
```

Now to discuss the scalar multiplication which will be a huge part of the CRS-J-PAKE protocol we need to discuss how we optimized this function to avoid it being susceptible to Timing attacks. For that, we had to avoid two things. Conditional statements and modulo operations. The reason to avoid conditional statements is that the if part and the $else$ part might not take the same amount of CPU cycles to execute, and this poses a threat to us as it makes the program susceptible to timing attacks.
If we used a regular point and add function based on the Montgomery ladder we talked about earlier (Section 311), We would have to do an if-else condition depending on if a bit is 0 or 1.
But if we examine the operations we are doing we see that the operations are almost mirroring, so if we could achieve a conditional swapping of the variables that are constant in time we could erase the conditional statement. And, that is exactly what we are going to do by introducing a conditional swapping function called $CSWAP()$. This $CSWAP()$ function will swap the content of the two variables that we pass as pointers to the function depending on the value of the variable b. if $b = 1$ the swap will occur if not the swap won't occur. We implemented the function according to this pseudo code from [8].

ALGORITHM 9: CSWAP using logic-arithmetic.

```
Input: $(b, x, y)$, where $b \in \{0,1\}$; and $x, y$ are $n$-bit strings.
Output: $(x', y') = \begin{cases} (x,y), & \text{if } b = 0; \\ (y,x), & \text{otherwise.} \end{cases}$
1 $m \leftarrow 0 - b$            ( Using two's complement. )
2 $d \leftarrow m \wedge (x \oplus y)$
3 $x' \leftarrow x \oplus d$
4 $y' \leftarrow y \oplus d$
5 return $(x', y')$
```

Fig. 2. CSwap algorithm. algorithm from [8]

How this code works is simple and take advantage of the properties of Bitwise operators, if b is 0 we initialize m to be 0. if b is 1, then we want m to be -1 so that its binary representation is an array of $1's$. By performing an XOR operation on both variables, we are storing in z a binary representation of a number where all $1's$ represents the digits that are different between the binary representation of x and y. We then perform an AND operation between m and z. This is the most important operation as it what makes the conditional swap take a constant amount of time, if m is an array of $1's$ then we want the swap to occur this AND operation will make sure that $d = z$, otherwise if m is equal to 0 then we don't want the swap to occur and this operation will give as a result 0. we then do the xor operation for x and y with z. If z is 0 then nothing will change as anything xor false to, but if z is not zero then z binary representation has one's in all indexes that are different between x and y and this is what makes the swap possible as the xor operation makes it such that $False + True = True$ and $True + True = False$. Here is our implementation of this algorithm:

```
void CSWAP(mpz_class b, mpz_class *x,
mpz_class *y) {

    mpz_class m = 0 - b;
    mpz_class z = (*x ^ *y);
    mpz_class d = m & z;
    mpz_class xp = *x ^ d;
    mpz_class yp = *y ^ d;
    *x = xp;
    *y = yp;
}
```

Now that we have taken care of the need to use a conditional statement, we need to look for a way to make our modulo operation also constant in time for defensive reasons. For that, we will implement a function called $reduction$ that does the operation $a \% curve.n$ where a is the number we want to reduce and $curve.n$ is the parameter n of the curve we have implemented. In this case, we make sure that the number a is below 256-bit. This function will be based on this pseudo-code, which is an implementation of the Barrett reduction. This pseudo code does have an if statement so we will have to tweak our implementation to use the $CSWAP()$ function.
We also had to tweak it to avoid having divisions as the division operation is not constant-time.To do that, we took to our advantage the fact that it is a division by a power of two, and as such we used bitshifts.

Algorithm 2: The Barrett reduction [16].

```
Input: $0 \leq c < 2^{2N}$; $\lambda = \lfloor 2^{2N}/P \rfloor$.
1: $q = \lfloor \frac{c \cdot \lambda}{2^{2N}} \rfloor$
2: $r = c - q \cdot P$
3: if $r \geq P$ then
4:     $r = r - P$, $q = q + 1$
5: end if
Output: $q = \lfloor c/P \rfloor$, $r = c \bmod P$.
```

Fig. 3. The Barrett reduction. algorithm from [8]

Here is our implementation:

```
mpz_class reduction(mpz_class a,
Elliptic_Curve curve) {
    mpz_class u;
    u = ("46316835694926478169428394003475163141
48096202088248947852400194972323911421 46");
    mpz_class t = (a * u);
    t = t >> 514;
    t = a - (t * curve.n);
    mpz_class b = (((curve.n+((~t)+1)) >> 514)
& 1);
    u = t;
    t = t - curve.n;
    CSWAP(b, &t, &u);
    return u;
}
```

And so we can finally combine everything to get our scalar multiplication function that takes as parameter the scalar k, the Point $point$ we want to multiply, and the parameters of the curve.

```
Point scalar_mult(mpz_class k, Point point,
Elliptic_Curve curve) {
    assert (is_on_curve(point, curve));

    Point R0;
    R0.x = 0;
    R0.y = 0;

    if ((k % curve.n) == 0 || (point.x == 0
&& point.y == 0)) {
        return R0;
    }

    if (k<0) {
        return scalar_mult(-k,
point_neg(point,curve), curve);
    }

    k = reduction(k, curve);
    int size = mpz_sizeinbase(k.get_mpz_t(),
256);   // 256-bit = 32-byte
    unsigned char scalarbyte[size] = { 0 };
// initialize scalar byte array
    // export scalar k to 32-byte array
    mpz_export(scalarbyte, NULL, -1, 1, 1, 0,
k.get_mpz_t());

    Point R1;
    R1.x = point.x;
    R1.y = point.y;

    // Montgomery ladder
    for (int i = 255; i >= 0; i--) {
        int sbit = (scalarbyte[i>>3] >> (i & 7))
& 1; // obtain the current scalar bit
        int sbit1 = (scalarbyte[(i+1)>>3] >>
((i+1) & 7)) & 1;
        int b = sbit ^ sbit1;
        CSWAP(b, &R0.x, &R1.x);
        CSWAP(b, &R0.y, &R1.y);
        R1 = point_add(R0, R1, curve);
        R0 = point_add(R0, R0, curve);
    }

    int sbit = (scalarbyte[0>>3] >> (0 & 7))
& 1;
    CSWAP(sbit, &R0.x, &R1.x);
    CSWAP(sbit, &R0.y, &R1.y);
    assert (is_on_curve(R0,curve));
    return R0;
}
```

Since the Montgomery algorithm wants us to start from the most significant bit we had to obtain the reverse of the binary representation of the number k. That's what this part of the code is doing:

```
int size = mpz_sizeinbase(k.get_mpz_t(), 256);
// 256-bit = 32-byte
unsigned char scalarbyte[size] = { 0 };
// initialize scalar byte array
// export scalar k to 32-byte array
mpz_export(scalarbyte, NULL, -1, 1, 1, 0,
k.get_mpz_t());
```

Thanks to the Function $mpz\_export()$ taken from the library GNU_MP. By changing the third parameter, we can obtain the reverse of the binary representation as an array. We can then obtain a specific bit from this array using this line of code where i is the index of the bit we want to retrieve:

```
int sbit = (scalarbyte[i>>3] >> (i & 7)) & 1;
```

With this we finished introducing all functions related to Elliptic Curves.

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.

The second function will be called *random_number_range*(), It will be taking as a parameter an *mpz_class* n and output an *mpz_class* z such that $0 \le z \le n$.

```
mpz_class random_number_range (mpz_class n) {
    gmp_randclass X(gmp_randinit_default) ;
    X.seed(getRandomInt());
    mpz_class z = X.get_z_range(n);
    return z ;
}
```

Hashing

As described in Section 34, we need a hashing function at the end of our protocol. That's why we implemented the *sha*256() function using the library OpenSSL. it takes a string as a parameter and has it using the sha256

```
string sha256(const string str)
{
    unsigned char hash[SHA256_DIGEST_LENGTH];
    SHA256_CTX sha256;
    SHA256_Init(&sha256);
    SHA256_Update(&sha256, str.c_str(),
str.size());
    SHA256_Final(hash, &sha256);
    stringstream ss;
    for(int i = 0; i < SHA256_DIGEST_LENGTH;
i++)
    {
        ss << hex << setw(2) << setfill('0') <<
(int)hash[i];
    }
    return ss.str();
}
```

Since the hash we obtain is in hexadecimal, we implemented a function that transforms that hash in decimal using stringstream which is a feature of the C language.

```
mpz_class hash_to_int(string s){
    std::stringstream ss;
    mpz_class x;
    s = sha256(s);
    ss << std::hex << s;
    ss >> x;
    return x;
}
```

Utilities

we will now talk about the utility function that we need. The first function is *point_to_string*() which will concatenate the x coordinate and the y coordinate of a point on the curve, and then transform it to a *String* using *stringstream*.

```
string point_to_string(Point point) {
    std::stringstream ss;
    string result1;
    string result2;
    ss << point.x;
    ss >> result1;
    std::stringstream().swap(ss);
    ss << point. y;
    ss >> result2;
    return result1+result2;
}
```

We also have the $point\_to_mpz$() function that will also make a point concatenate its coordinate and will return them as an *mpz_class* using first the *point_to_string* and then *stringstream*.

```
mpz_class point_to_mpz(Point point) {
    string str = point_to_string(point);
    std::stringstream ss;
    ss << str;
    mpz_class m ;
    ss >> m;
    return m;
}
```

And finally, the *stm*() function that will convert an *mpz_class* to a string using *stringstream*.

```
string stm(mpz_class x) {
    string str;
    std::stringstream ss;
    ss << x;
    ss >> str;
    return str;
}
```

Schorr Zero knowledge proof

We will now talk about how we implemented the Non-interactive Zero-Knowledge Proofs wt talked about in Section 32 in our protocol. As seen in Section 34, we need to create two different Zero-Knowledge proofs with two other functions to verify theses. Since we can't output two values at the same time using C++, we will pass with the other parameters two pointers for the generators of the ZKP. The only difference between both generating functions is the parameters that the variable c take in. The process in itself identical. *generate_nizk_proof* generates the first ZKP while *generate_nizk_proof_2* generates the second.

```
void generate_nizk_proof(Point G, mpz_class x,
Point XS, Point ID, string U,
Elliptic_Curve curve,
Point *r1, mpz_class *r2) {
    mpz_class v = random_number_range(curve.n);
    Point vS = scalar_mult(v, G, curve);
    string chal = point_to_string(G) +
point_to_string(XS) + point_to_string(ID) + U +
point_to_string(vS);
    mpz_class c = hash_to_int(chal);
    mpz_class pi = reduction((v-c*x), curve);
    *r1 = vS;
    *r2 = pi;
}
```

```
void generate_nizk_proof_2(Point G, mpz_class x,
Point XS, string L, Elliptic_Curve curve,
Point *r1, mpz_class *r2) {
    mpz_class v = random_number_range(curve.n);
    Point vS = scalar_mult(v, G, curve);
    string chal = point_to_string(G) +
point_to_string(XS) + L + point_to_string(vS);
    mpz_class c = hash_to_int(chal);
    mpz_class pi = reduction((v-c*x), curve);
    *r1 = vS;
    *r2 = pi;
}
```

We then have functions that check the ZKP generated by the previous functions. Each Generating function has a check function.

```
void check_nizk_proof(Point G, mpz_class pi,
Point X, Point ID, string U, Point Vs,
Elliptic_Curve curve) {
    string chal = point_to_string(G) +
point_to_string(X) + point_to_string(ID) + U
+ point_to_string(Vs);
    mpz_class c = hash_to_int(chal);
    Point check =  point_add(
scalar_mult(pi,G,curve),
scalar_mult(c, X,curve), curve);
    if (check_two_points(check, Vs)) {
        cout << ("It has been proved") << endl;
    }
    else {
        cout << ("not proved") << endl;
    }
}
```

```
void check_nizk_proof_2(Point G, mpz_class pi,
Point X, string  L, Point Vs,
Elliptic_Curve curve) {
    string chal = point_to_string(G) +
point_to_string(X) + L + point_to_string(Vs);
    mpz_class c = hash_to_int(chal);
    Point check = point_add(
scalar_mult(pi,G,curve),
scalar_mult(c, X,curve), curve);
    if (check_two_points(check, Vs)) {
        cout << ("It has been proved") << endl;
    }
    else {
        cout << ("not proved") << endl;
    }
}
```

Main function

With this, we finished talking about all the functions, and we just need to implement our protocol following what we described in Section 34. There is not much to explain, at the start of the protocol we decided that the secret will be "IssamJomaatest" and we record the duration of the protocol and display it at the end. At the end of the protocol, we display the key that A and B have. If it is the same the program display that the test was a success.

```
int main() {
    Elliptic_Curve curve = initialize_curve();
    double duration;
    std::clock_t start;
    start = std::clock();
    string secret = "IssamJomaatest";
    mpz_class s = hash_to_int(secret);

    Point G = curve.g;
    Point U = generation_CRS(curve);
    string strU = point_to_string(U);

    mpz_class x1 = random_number_range(curve.n);
    mpz_class x2 = random_number_range(curve.n);

    Point X1 = scalar_mult(x1, curve.g, curve);
    Point X2 = scalar_mult(x2, curve.g, curve);

    Point A = point_add(X1, X2, curve);
   A = scalar_mult(x1*s, A, curve);

    Point B = point_add(X1, X2, curve);
   B = scalar_mult(x2*s, A, curve);

    Point Vs;
    mpz_class pi_1;
    generate_nizk_proof(G, x1, X1, A, strU,
curve, &Vs, &pi_1);
    check_nizk_proof(G, pi_1, X1, A, strU,
Vs, curve);

    Point Vg;
    mpz_class pi_2;
    generate_nizk_proof(G, x2, X2, B, strU,
curve, &Vg, &pi_2);
    check_nizk_proof(G, pi_2, X2, B, strU,
Vg, curve);

    Point beta = point_add(U, X1, curve);
    beta = scalar_mult(x2*s, beta, curve);

    Point alpha = point_add(U, X2, curve);
    alpha = scalar_mult(x1*s, alpha, curve);

    Point UX1 = point_add(U, X1, curve);
    Point UX2 = point_add(U, X2, curve);

    string la = point_to_string(A) +
point_to_string(B) +
point_to_string(X1) + point_to_string(X2) + strU;
    string lb = point_to_string(B) +
point_to_string(A) +
point_to_string(X2) + point_to_string(X1) + strU;
```

```
    Point Valpha;
    mpz_class pi_alpha;
    generate_nizk_proof_2(UX2, x1*s, alpha,
la, curve, &Valpha, &pi_alpha);
    check_nizk_proof_2(UX2, pi_alpha, alpha,
la, Valpha, curve);

    Point Vbeta;
    mpz_class pi_beta;
    generate_nizk_proof_2(UX1, x2*s, beta, lb,
curve, &Vbeta, &pi_beta);
    check_nizk_proof_2(UX1, pi_beta, beta, lb,
Vbeta, curve);

    Point ka = scalar_mult(-x1*s, X2, curve);
    ka = point_add(beta, ka, curve);
    ka = scalar_mult(x1, ka, curve);

    Point kb = scalar_mult(-x2*s, X1, curve);
    kb = point_add(alpha, kb, curve);
    kb = scalar_mult(x2, kb, curve);

    mpz_class ska = hash_to_int(stm(ka.x));
    mpz_class skb = hash_to_int(stm(kb.x));

    duration = (std::clock() - start) /
(double) CLOCKS_PER_SEC;
    cout << duration << endl;
    cout << "This is the result of the Client A:
" << ska << endl;
    cout << "This is the result of the Client B:
" << skb << endl;

    if (ska == skb) {
        cout << "This test was a success."
<< endl;
    }
}
```

3.5. Assessment

The program is working as intended and we completed the main goal of the project which is the implementation of a PAKE. We can nonetheless give an assessment on certain specific points that are important.

- **Modulable**

  We've put a lot of effort into fragmenting our program as much as possible, especially the one written in c++. The goal was to make it modulable in an effort to reduce the length of the main function. This also makes the program easy to modify as we were constantly improving it during the duration of the whole project. This will also help us extract interesting functions for future use in other programs.

- **Security**

  As we said in Section 1, PAKE's have different security requirements that all need to hold. And as such, we had to make sure that PAKE's are not susceptible to timing attacks which are easy to do and very effective. In an effort to make our program more secure we improved the scalar multiplication function to make completely secure by erasing conditional statement and creating our own modulo function. We failed however to extend this property to all the functions as we ran out of time. We nonetheless exceeded to goals of this project as it was not in the goal of the project to make the program secure against timing attacks.

4. Acknowledgment

I would first like to thank Prof. Dr. Peter Y. A. Ryan 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 Webex but also via email things that I did not understand. I would also like to thank Hao Cheng and Aditya Damodaran for taking the time to answer all my questions and helping me improve my program.

5. Conclusion

During this project, we approached the concept of the PAKE protocol through a direct implementation of the J-PAKE and a CRS-J-PAKE variant. We learned through this project about different subjects that are all related to cryptography such as the Elliptic curves concept and the maths behind it, both Interactive and not-interactive Zero-knowledge proofs, Hashing functions. We also learned about the potential risks when implementing cryptography-related functions and how to avoid and thwart flaws that could be used for attacks like timing attacks.

References

[1] Steven Michael Bellovin and Michael Merritt. "Encrypted key exchange: Password-based protocols secure against dictionary attacks". In: (1992).

[2] *Cpp_overview*. URL: https://www.tutorialspoint.com/cplusplus/cpp_overview.htm (visited on 05/13/2021).

[3] *Cryptography*. URL: https://en.wikipedia.org/wiki/Cryptography (visited on 05/13/2021).

[4] *Cryptography Protocols*. URL: https://en.wikipedia.org/wiki/Cryptographic_protocol (visited on 05/13/2021).

[5] *Diffie-Hellman wikipedia*. URL: https://en.wikipedia.org/wiki/DiffieHell-man_key_exchange (visited on 05/16/2021).

[6] *Elliptic Curve Cryptography*. URL: https://avinetworks.com/glossary/elliptic-curve-cryptography/ (visited on 05/14/2021).

[7] *Elliptic curve point multiplication*. URL: https://en.wikipedia.org/wiki/Elliptic_curve_point_multiplication (visited on 05/19/2021).

[8] Armando Faz-Hernández, Julio Lopez, and Ricardo Dahab. "High-performance implementation of elliptic curve cryptography using vector instructions". In: *ACM Transactions on Mathematical Software (TOMS)* 45.3 (2019), pp. 1–35.

[9] *J-PAKE*. URL: https://eprint.iacr.org/2010/190.pdf (visited on 05/15/2021).

[10] *Python_overview*. URL: https://www.tutorialspoint.com/python3/python_overview.htm (visited on 05/13/2021).

[11] *secp256k1*. URL: https://asecuritysite.com/encryption/python_secp256k1ecdh (visited on 05/16/2021).

[12] Marjan Skrobot, Jean Lancrenon, and Qiang Tang. "Two More Efficient Variants of the J-PAKE Protocol". In: *ACNS 2016* (2016).

[13] *The GNU MP Bignum Library*. URL: https://gmplib.org/ (visited on 05/16/2021).

[14] *Zero-knowledge proof*. URL: https://en.wikipedia.org/wiki/Zero-knowledge_proof (visited on 05/19/2021).

5.1. Appendix

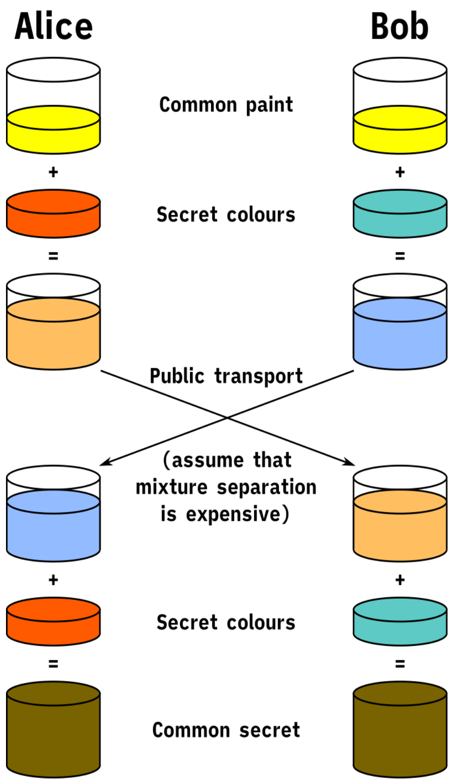
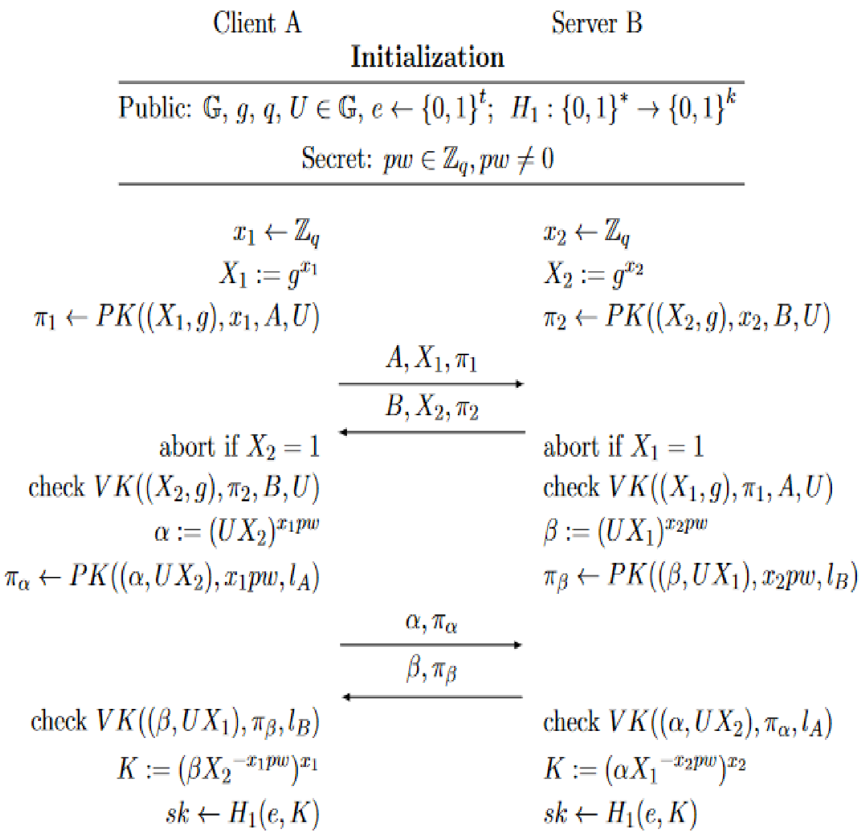


Fig. 4. Illustration of the concept behind Diffie–Hellman key exchange. image from [5]



Client A | Server B

Initialization

Public: $\mathbb{G}, g, q, U \in \mathbb{G}, e \leftarrow \{0,1\}^t;\ H_1 : \{0,1\}^* \rightarrow \{0,1\}^k$

Secret: $pw \in \mathbb{Z}_q, pw \neq 0$

Client A		Server B
$x_1 \leftarrow \mathbb{Z}_q$		$x_2 \leftarrow \mathbb{Z}_q$
$X_1 := g^{x_1}$		$X_2 := g^{x_2}$
$\pi_1 \leftarrow PK((X_1, g), x_1, A, U)$		$\pi_2 \leftarrow PK((X_2, g), x_2, B, U)$
	$\xrightarrow{A, X_1, \pi_1}$	
	$\xleftarrow{B, X_2, \pi_2}$	
abort if $X_2 = 1$		abort if $X_1 = 1$
check $VK((X_2, g), \pi_2, B, U)$		check $VK((X_1, g), \pi_1, A, U)$
$\alpha := (UX_2)^{x_1 pw}$		$\beta := (UX_1)^{x_2 pw}$
$\pi_\alpha \leftarrow PK((\alpha, UX_2), x_1 pw, l_A)$		$\pi_\beta \leftarrow PK((\beta, UX_1), x_2 pw, l_B)$
	$\xrightarrow{\alpha, \pi_\alpha}$	
	$\xleftarrow{\beta, \pi_\beta}$	
check $VK((\beta, UX_1), \pi_\beta, l_B)$		check $VK((\alpha, UX_2), \pi_\alpha, l_A)$
$K := (\beta X_2^{-x_1 pw})^{x_1}$		$K := (\alpha X_1^{-x_2 pw})^{x_2}$
$sk \leftarrow H_1(e, K)$		$sk \leftarrow H_1(e, K)$

Fig. 5. The CRS-J-PAKE protocol. image from [12]