# A trip between creation and destruction of non-commutative public key exchange protocols

Borja Gómez

November 7, 2019

**Abstract**

Conventional asymmetric key exchange protocols rely on computing elements in commutative groups, where the employed trapdoor-permutation function is commutative, allowing Alice and Bob to compute the same element in $G$ as changing the orders of the variables or elements doesn't alter the output. The research found in this paper is focused on the analysis of key exchange protocols found in non-commutative cryptography, sometimes called group-based cryptography. Variations of these schemes made by the author are also included. Concretely, four schemes are presented using matrices over finite fields and permutation groups containing all the theory to break these schemes along with its pseudo-code and implementations in Mathematica.

## 1 Introduction

Non-commutative cryptography arises from the research made by M. R. Magyarik and N. R. Wagner [1] where in 1985 they presented a public key cryptosystem based on the word problem. From there, a series of new cryptographic schemes where devised [2][3][4][5][9] Some of these methods have been partially or fully solved in [6][7][8] What is important to know is that non-commutative cryptographic schemes can be used to encipher/decipher, authenticate and exchange keys. However, at some point, they must use material from a commutative algebraic structure. This is why some people says that commutativity will always be present. The other relevant reason is that the security of the cryptosystems that use non-commutative protocols has not being studied as much as in commutative schemes.

### 1.1 Motivation

Non-commutativity is an interesting property as it is not found in the conventional cryptosystems like RSA, DH, ECC, DSA, Elgamal .... The author started his interest on non-commutative Cryptography back in 2016. Using the Symmetric Group as the platform group the trapdoor function composed the permutations of Alice and Bob with their private values into a third shared permutation that's equal to both parties. As a consequence of studying research from other authors more key exchange protocols were developed and analyzed. The field looks promising as Multivariate Public Key Cryptography, both directions have not been extensively analyzed like in cryptosystems based on integer factorization and discrete logarithm problems. Major credit is for the authors of [10] as the information they provide is what inspired this research.

## 2 Key Exchange protocols

There are multiple key exchange protocols found in literature. Normally these protocols are classified by the underlying problem where they are based. Almost every protocol has a modified version by changing the properties of subgroups $A, B \leq G$. These protocols are based on the following group-theoretic problems:

- (subgroup restricted) conjugacy-search problem
- decomposition problem
- factorization-search problem

These problems can be redefined to their decisional branch. Then instead of searching for an element that satisfies the desired properties, they decide whether or not such elements exist. Let's start by enumerating some of these protocols that later will be analyzed:

### 2.1 Protocols based on the conjugacy-search problem

#### 2.1.1 Koo, Lee et al key exchange protocol

The protocol developed by Ko, Lee et al [2] is a good introduction to show how non-commutative cryptographic protocols work. It is very inspired in the Diffie-Hellman key exchange. The protocol depends on the conjugacy search problem as the transformation function is the conjugation in $G$. Let $g^x$ denote the conjugation of $g$ by $x \in G$, thus $g^x = x^{-1} \cdot g \cdot x$. Select $A, B \leq G$ to commute elementwise, then to build their public keys Alice and Bob select private elements $a \in A, b \in B$.

$$\begin{aligned} pub_A &= g^a \\ pub_B &= g^b \\ shared_A &= (g^b)^a = a^{-1}(b^{-1}gb)a \\ shared_B &= (g^a)^b = b^{-1}(a^{-1}ga)b \end{aligned}$$

## 2.2 Protocols based on the decomposition-search

### 2.2.1 Twisted protocol

The main idea that can be extracted from [4] is that two subgroups $A, B$ of $G$ are selected both commuting elementwise. Alice chooses $a_1 \in A$ and $b_1 \in B$. Same does Bob for $a_2, b_2$. They compute their public key as follows:

$$\begin{aligned} g \in G \quad A \leq G, B \leq G \\ pub_A = a_1 g b_1 \\ pub_B = b_2 g a_2 \end{aligned}$$

To compute the shared key the same method as the previous protocol is utilized. Alice appends to the left $a_1$ and to the right $b_1$ in $pub_A$. Same does Bob on $pub_A$ but with his own private values $b_2, a_2$ in this order.

$$\begin{aligned} shared_A = a_1(b_2 g a_2)b_1 = a_1 b_2 g a_2 b_1 \\ shared_B = b_2(a_1 g b_1)a_2 = a_1 b_2 g a_2 b_1 \end{aligned}$$

it is a slight modification of the common decomposition problem which can be stated as:

$$\begin{aligned} pub_A &= a_1 g a_2 \\ pub_B &= b_1 g b_2 \\ shared_A &= a_1(b_1 g b_2)a_2 = a_1 b_1 g a_2 b_2 \\ shared_B &= b_1(a_1 g a_2)b_2 = a_1 b_1 g a_2 b_2 \end{aligned}$$

#### 2.2.1.1 Modified twisted with commutative subgroups

The author found a new variant where $A, B \leq G$ are selected to not commute elementwise. It results that this variant is included in [10] pp 45 point 4.2.3. Then $a_1 a_2 = a_2 a_1$ and $b_1 b_2 = b_2 b_1$ must commute. Publish the central element $g$ and subgroups $A, B \leq G$ such that $ab \neq ba$.

$$\begin{aligned} pub_A &= a_1 g b_1 \\ pub_B &= a_2 g b_2 \\ shared_A &= a_1(a_2 g b_2)b_1 = a_1 a_2 g b_1 b_2 \\ shared_B &= a_2(a_1 g b_1)b_2 = a_1 a_2 g b_1 b_2 \end{aligned}$$

### 2.2.2 Protocols based on factorization-search

This protocol is very similar to the twisted but with the exception that the central element $g$ is eliminated. Thus restriction on $A, B \leq G$ is mantained to commute elementwise. To compute their public keys, Alice selects $a_1 \in A, b_1 \in B$ same does Bob with $a_2, b_2$..

$$\begin{aligned} pub_A &= a_1 b_1 \\ pub_B &= a_2 b_2 \\ shared_A &= b_1(a_2 b_2)a_1 = a_2 a_1 b_1 b_2 \\ shared_B &= a_2(a_1 b_1)b_2 = a_2 a_1 b_1 b_2 \end{aligned}$$

#### 2.2.2.1 Using commutative subgroups not commuting elementwise

Select $A, B \leq G$ but with the condition that they do not commute elementwise. Let $A = C_G(a_1), B = C_G(b_1)$ then the protocol description is stated as:

$$\begin{aligned} pub_A &= a_1 b_1 \\ pub_B &= a_2 b_2 \\ shared_A &= a_1(a_2 b_2)b_1 = a_1 a_2 b_1 b_2 \\ shared_B &= a_2(a_1 b_1)b_2 = a_1 a_2 b_1 b_2 \end{aligned}$$

## 2.3 Stickel's protocol

The idea of the protocol [5] is to exponentiate two generators of two distinct subgroups of $G$, then multiply the result. Two generators $a \in A, b \in B$ are given such that $A, B \leq G$. The goal is that Alice and Bob compute the same element in $G$. For that, define $n_a = Ord_G(a), n_b = Ord_G(b)$ so Alice and Bob choose private positive integer values $a_1, b_1$ and $a_2, b_2$ to compute their respective public keys:

$$\begin{aligned} pub_A &= a^{a_1} b^{b_1} \quad a_1 < n_a, b_1 < n_b \\ pub_B &= a^{a_2} b^{b_2} \quad a_2 < n_a, b_2 < n_b \end{aligned}$$

Alice picks up $pub_B$ and appends $a^{a_1}$ to the left and $b^{b_1}$ to the right. Bob does the same but with his private pair $a_2, b_2$.

$$\begin{aligned} shared_A &= a^{a_1}(a^{a_2} b^{b_2})b^{b_1} = a^{a_1+a_2} b^{b_1+b_2} \\ shared_B &= a^{a_2}(a^{a_1} b^{b_1})b^{b_2} = a^{a_1+a_2} b^{b_1+b_2} \end{aligned}$$

Both users obtain the same element as powers of the same element commute. But since $ab \neq ba$ the attacker cannot obtain such expression by multiplying $pub_A pub_B = a^{a_1} b^{b_1} a^{a_2} b^{b_2}$. He must find out one of the two private exponent tuples utilized in the scheme. However, the implementation details are given in a different manner as seen in [10] pp 47.

### 2.3.1 Alternative definition

Let $w$ be public. In order to build their public keys, Alice chooses $c_1$ in the centralizer of the group. Bob does the same with $c_2$.

$$\begin{aligned} w \in G \quad c_1, c_2 \in C_G \\ pub_A = c_1 a^{a_1} w b^{b_1} \quad a_1 < n_a, b_1 < n_b \\ pub_B = c_2 a^{a_2} w b^{b_2} \quad a_2 < n_a, b_2 < n_b \end{aligned}$$

The shared key is computed in the same way appending to left-right but in this case with have an extra value that commutes with every element of the group. Thus both parties end up having:

$$\begin{aligned} shared_A = c_1 a^{a_1}(c_2 a^{a_2} w b^{b_2}) b^{b_1} = c_1 c_2 a^{a_1+a_2} w b^{b_1+b_2} \\ shared_B = c_2 a^{a_2}(c_1 a^{a_1} w b^{b_1}) b^{b_2} = c_1 c_2 a^{a_1+a_2} w b^{b_1+b_2} \end{aligned}$$

### 2.3.2 Twisted Stickel's using commutative subgroups not commuting elementwise

This variant is obtained when combining the modified twisted protocol with commutative subgroups seen in 2.2.1.1 and Stickel's protocol. Let $A =< a >, B =< b >$ such that $A, B \leq G$ and $ab \neq ba$. Then Alice and Bob choose private exponents $(a_1, b_1)$ and $(a_2, b_2)$ then setup their public keys as:

$$\begin{aligned} pub_A = a^{a_1} g b^{b_1} \quad a_1, b_1 < Ord_G(a) \\ pub_B = a^{a_2} g b^{b_2} \quad a_2, b_2 < Ord_G(b) \\ shared_A = a^{a_1}(a^{a_2} g b^{b_2}) b^{b_1} = a^{a_1+a_2} g b^{b_1+b_2} \\ shared_B = a^{a_2}(a^{a_1} g b^{b_1}) b^{b_2} = a^{a_1+a_2} g b^{b_1+b_2} \end{aligned}$$

And the attacker cannot obtain any information as multiplying $pub_A \cdot pub_B$ doesn't give the required expression in $shared_A, shared_B$ since elements of $A, B$ don't commute elementwise. Note that this protocol can be converted to the general approach using subgroups $A, B \leq G$ that commute elementwise.

## 3 Cryptanalysis of key exchange protocols

Four distinct key exchange protocols are presented . The first one is a working protocol based on the factorization-search using matrices over a finite field. The second presentation is based on the Koo Lee et al on permutation groups but instead of conjugation using exponentiation. The third one is the Stickel's protocol using permutation groups where the private exponents of $(a_1, b_1, a_2, b_2)$ can be recovered with a specific technique. The fourth case is a combination of the twisted protocol and Stickel's. Real case examples can be found in every scheme, along with pseudo-code.

## 3.1 Factorization-search protocol using matrices in $GL(n, q)$

The following key exchange algorithm was discovered and broken by the author of this research. It is based on the factorization-search using commutative subgroups, found in 2.3.1. Factorization-search depends on "factoring" an element into a product of two elements in $G$.

In this case Alice and Bob work with matrices in $GL(n, p)$ which is the general linear group over the finite field $F_q$ where $q = p^n$. The goal of Alice and Bob is to compute the same shared secret using matrices from two distinct commuting spaces but not commuting elementwise.

Alice chooses matrices $A, B \in F_p^{n\times n}$ where $AB \neq BA$. She computes the matrix $Q_1 = AX - XA, Q_2 = BY - YB$ which contains linear polynomials in each term $(i, j)$. This representation is used to extract and build the coefficient matrix of the system $Q_1 = 0, Q_2 = 0$. Computing the transpose of the nullspace yields $nsQ_1 = Null(Q_1)^T, nsQ_2 = Null(Q_2)^T, \in F_p^{n^2\times n}$ where both bases send a vector of size $n$ to a vector of size $n^2$ in $F_p$, which can be represented as a $n \times n$ matrix that satisfies $AX - XA = 0$ and inherently $AX = XA$ in the case of $nsQ_1$, same for $nsQ_2$ with $BY - BA = 0$. A formal presentation of the protocol is given below.

Let $\phi$ be the map that sends a vector of size $n^2$ to a $n \times n$ matrix:

$$\phi : F_p^{n^2\times 1} \mapsto F_p^{n\times n}$$

Alice chooses private matrices $A, B \in F_p^{n\times n}$. Computes $nsQ_1, nsQ_2$ and sends her public key $(AB, nsQ_1, nsQ_2)$ to Bob. Bob chooses $x, y \in F_p^n$ which are the private values used to construct $C, D$ using $nsQ_1$ and $nsQ_2$ respectively.

$$\begin{aligned}
nsQ_1 &= Null(AX - XA)^T \\
nsQ_2 &= Null(BY - YB)^T \\
pub_A &= (AB, nsQ_1, nsQ_2) \\
C &= \phi(nsQ_1 \cdot x) \\
D &= \phi(nsQ_2 \cdot y) \\
pub_B &= CD \\
shared_A &= A(CD)B = ACBD \\
shared_B &= C(AB)D = ACBD
\end{aligned}$$

Alice and Bob obtain the same expression but the attacker doesn't as $(AB)(CD) = ABCD$ and $(CD)(AB) = CDAB$ because $BC \neq CB, DA \neq AD$. At a first glance an attacker would put his eye on the input of $nsQ_1, nsQ_2$ as it is a vector of size $n$ over $F_p$ thus $q = p^n$ possible tuples for selecting the right matrix $C$ or $D$. The same construction can be applied in $K = \mathcal{R}$ but is results in an infinite space of commuting matrices with coefficients that are floating numbers, where in the author's opinion working in a finite field is a better option for precision and efficiency.

### 3.1.1 A complete example

To familiarize and ease the reading process an example is given with matrices of size $3 \times 3$ over $F_2$. These are matrix elements in $GL(3, 2)$:

Alice chooses her private matrices $A, B \in F_p^{3\times3}$:

$$A = \begin{pmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

$$B = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}$$

To build her public key $pub_A$ she computes the product $AB$ and the commuting spaces by calculating the transpose of the nullspace of $Q_1$ and $Q_2$, which she calls $nsQ_1, nsQ_2 \in F_2^{9\times3}$.

$$AB = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}$$

$$nsQ_1 = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

$$nsQ_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

Now she sends her public key $pub_A = (AB, nsQ_1, nsQ_2)$ to Bob. Bob selects two private tuples $x, y \in F_2^3$ and computes $C$ and $D$ by evaluating these vectors on $nsQ_1$ or $nsQ_1$ which output a $9 \times 1$ vector that needs to be stated as a $n \times n$ matrix, this is why the map $\phi$ is important.

$$x = (1,0,1), y = (1,1,0)$$

$$\phi(nsQ_1 \cdot x) = C = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix}$$

$$\phi(nsQ_2 \cdot y) = D = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}$$

Bob computes the product $CD$ and sends his public key $pub_B = (CD)$ to Alice.

$$CD = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

Alice and Bob compute the same shared value:

$$shared_A = A(CD)B = ACBD = \begin{pmatrix} 0 & 1 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

$$shared_B = C(AB)D = ACBD = \begin{pmatrix} 0 & 1 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

### 3.1.2 Complete Cryptanalyisis

As said before, the key exchange algorithm that's being described is completely broken using an attack based in linear algebra. The attacker can compute an alternative and distinct pair $C' \neq C, D' \neq D$ where its product is the same as Bob's public key thus $C'D = CD$. In the end, the attacker succeeds in the key exchange being able to impersonate Bob. Even using PKI, Bob is not able to modify the handshake but passively computes $pub_B = CD$ thus he can compute $shared_B$ using Alice's public key $pub_A$.

To succeed in the impersonation, the attacker knows that $CD = \phi(nsQ_1 \cdot x)\phi(nsQ_2 \cdot y)$. He writes up the algebraic expression of the product, which can be generalized for arbitrary parameters since it's linear. As the output of multiplying $nsQ_1 \cdot x$ gives a $n^2$ vector, every coordinate of $C$ is a $n$ linear combination of the $i$-th row of $nsQ_1$ with the coordinate vector $x$.

$$C_{i,j} = \sum_{k=1}^{n} nsQ_{1((i-i)j+j,k)} x_k$$

$$D_{i,j} = \sum_{k=1}^{n} nsQ_{2((i-i)j+j,k)} y_k$$

Now he computes the product of the algebraic expressions of $C, D$:

$$(CD)_{i,j} = \sum_{k=1}^{n} C_{i,k} D_{k,j}$$

As every element in $C$ or $D$ is a linear combination on $nsQ_1$ or $nsQ_2$ the attacker obtains a summation of distinct products in $CD_{i,j}$ where every product is a linear combination itself on one of these nullspaces As a result, every position in $CD$ combines both unknown tuples $x = (x_1, \cdots, x_n), y = (y_1, \cdots, y_n)$. The attacker wants to obtain alternative matrices $C', D'$ that satisfy the equivalence $C'D' = CD, C' \neq C, D' \neq D$. With all the aforementioned information he knows that any position $(CD)_{i,j}$ may carry all the combinations of products $x_i y_j$ thus every arbitrary position is decomposable into $n$ unknown linear factors and $n$ linear coefficients. The attacker is versed on Linear Algebra, realizing that he can mount an attack to recover the product $x_i y_j$with $n$ terms of the product $CD$. Moreover, take the previous example case. Attacker defines $x \in F_2^3$ with binary integer values and recover the associated binary integer tuple $y \in F_2^3$. Let's evaluate the whole algebraic procedure that an attacker does to mount such an attack:

$$C' = \phi(nsQ_1 \cdot (x_1, x_2, x_3)) = \begin{pmatrix} x_1 + x_2 & x_3 & x_2 \\ x_2 + x3 & x_1 + x_2 & x_3 \\ x_3 & x_2 & x_1 \end{pmatrix}$$

$$D' = \phi(nsQ_2) \cdot (y_1, y_2, y_3) = \begin{pmatrix} y_1 & y_2 & y_2 \\ y_2 & y_1 & y_3 \\ y_2 & 0 & y_1 \end{pmatrix}$$

$$C'D' = \begin{pmatrix} y_1(x_1 + x_2) + x_2 y_2 + x_3 y_2 & y_2(x_1 + x_2) + x_3 y_1 & y_2(x_1 + x_2) + x_2 y_1 + x_3 y_3 \\ y_2(x_1 + x_2) + y_1(x_2 + x_3) + x_3 y_2 & y_1(x_1 + x_2) + y_2(x_2 + x_3) & y_3(x_1 + x_2) + y_2(x_2 + x_3) + x_3 y_1 \\ x_1 y_2 + x_2 y_2 + x_3 y_1 & x_2 y_1 + x_3 y_2 & x_1 y_1 + x_2 y_3 + x_3 y_2 \end{pmatrix}$$

The attacker setups $x' = (1, 1, 1)$ where $x \neq x'$ then the coefficient matrix $A \in F_2^{9 \times 3}$ is obtained by substituting $x'$ in the algebraic expression of $C'D'$. As we are in linear algebra, it's possible to express the whole attack in a system involving $y'$ as the unknown tuple in the LHS, the product matrix $CD$ with binary integer coefficients as a $n^2$ vector in the RHS and the aforementioned coefficient matrix $A$ in the LHS multiplying by the left, this is $A.y' = b$. Note that $A$ is a $9 \times 3$ matrix thus the system is overdetermined as there are more equations than variables. It has a solution by Rouche-Capelli's Theorem and with any CAS software it's easily solvable.

$$A.y' = \phi^{-1}(CD)$$

$$A = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix}$$

$$b = \phi^{-1}(CD) = \{0, 0, 1, 1, 0, 0, 0, 1, 0\}$$

Solving the system gives $y' = (0, 1, 1)$. Now the attacker knows that this new pair $x = (1, 1, 1)', y = (0, 1, 1)'$ satisfies that $C'D' = CD$ obtaining both matrices $C'D'$ using $nsQ_1, nsQ_2$ as follows:

$$C' = \phi(nsQ_1 \cdot x') = \begin{pmatrix} 0 & 1 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

$$D' = \phi(nsQ_2 \cdot y') = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}$$

$$C'D' = CD = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

$$shared_{ATTK} = C'(AB)D' = shared_A = shared_B$$

The algebraic property where this attack is based can be stated as:

$$\begin{aligned} shared_{ATKK} = shared_A = shared_B \iff \exists x', y' \in F_p^n : \\ C'D' = CD = \phi(nsQ_1 \cdot x') \cdot \phi(nsQ_2 \cdot y') \wedge C'A = AC' \quad D'B = BD' \end{aligned}$$

It is not sufficient to find a pair $C'D' = CD$ since the shared key requires that $C'$ commutes with $A$ and $D'$ with $B$. The attacker could also solve the system by selecting an arbitrary vector $x'$ and matrix $C'$, where $C'$ is invertible thus $D' = C'^{-1}(CD)$, solving for $y'$ on $nsQ_2 \cdot y' = \phi^{-1}(D')$. However, sometimes the values of $x', y', C', D'$ exists but $BD' \neq D'B$ and this cases are frequent when solving by the inverse method. Thus is better to stick with the system derived from the algebraic expansion of the product of $C'D'$, because it grants a solution when the overdetermined system can be solved, this is, when the selected $x'$ by the attacker makes the system solvable.

### 3.1.3 Conclusion

Creating a protocol that enables Alice and Bob to compute a shared secret is easily made when you know general Math and Cryptographic theory but crypt-analysing it is not trivial as there could be multiple ways to break it into smaller problems that can be solvable, depending on their time-space complexity. In this case the cryptographic protocol was based on the Factorization-Search where the elements of the platform group are matrix elements from the general linear group of dimension $n$ over $F_2$. It's broken since every aspect of the scheme is linear resulting thus the public key setup procedure can be reversed and tracked down into a linear system of equations giving the private values $x, y \in F_2^n$.

## 3.2 A protocol based in the Discrete Logarithm Problem using permutations

The author broke an own made scheme based on Diffie-Hellman using elements of a permutation group. It results that the authors of [9] released a cryptosystem using the symmetric group as the platform group, which they claim that's secure enough.

The protocol defined in this section shares the main characteristics with [9]. However, the scheme in [9] enciphers and the protocol explained below doesn't, as it is a key exchange protocol. The key to break both schemes is how an attacker solves the discrete logarithm in permutation groups.

### 3.2.1 Permutation groups and Cryptography

The problem with permutation groups is that the number of symbols make computations harder to perform. Moreover, it's unfeasible, for example, to represent a cyclic group of the typical Diffie-Hellman KE protocol, where $p = 2q + 1$, being $p$ a safe-prime and $q$ a Sophie-Germain prime of at least 1024 bit. The permutation group $P$ represented by $g$ would be cyclic of $q$ elements and $q$ symbols. Imagine a permutation element composed of 1 cycle of approximately $2^{1024}$ elements. Then permutation groups are not useful in the cryptnalatytic part of the standard Diffie-Hellman definition. However, elements of permutation groups have very nice conditions when operating with them, for example: multiplication, exponentiation and conjugation can be decomposed to extract information. In addition, either we can obtain the permutation representation of an arbitrary group $G$ or the representation of $G$ as a subgroup of $GL(n, F_q)$, which can be helpful to build simple examples that may help. Now the question is: why are permutation groups used here? Well, the following protocol satisfies some properties that could make it secure, but it works with permutation elements that are representable, which leads the attacker to discover some attacks.

### 3.2.2 Protocol Description

The protocol consist on the Koo Lee et. al. version using permutation elements of $P \leq S_n$, but exponentiation is used instead of conjugation. Then it reduces to the classic Diffie-Hellman approach using $P$ as a platform group.

#### 3.2.2.1 Building the permutation group $P$ and its generator $\sigma$

Before proceeding to construct an example of a permutation group with enough order using $n$ symbols, those with experience in group-theory can find a more general definition of $\sigma =< P >$ in the point **3.3.1** or in Algorithms $1, 2$.. The definition explained here is thought to help the reader to build an efficient permutation group $P$ whose rate $Deg(P)/|P|$ is optimal.

A permutation group $P$ that has big order but a small quantity of symbols is the same as having $|P|$ big and $Deg(P)$ small. The reader may be familiarized with the concept that the order of an element $\sigma \in P$ is the least common multiple of the length of each cycle contained in it. This is for a permutation $\sigma \in P$ with $k$ cycles:

$$Ord_P(\sigma) = lcm(|c_1|, \cdots, |c_k|)$$

We can exploit this definition to create a permutation generator of $P$ that has big order but small degree order, this is, the $lcm$ is maximal for a given degree. Let $\pi(L)$ denote the number of primes below $L$, construct $\sigma$ with $\pi(L)$ cycles each of prime length $p_i$. Since prime factors don't share any factor in common, this is $\gcd(p_i, p_j) = 1$ for any pair of prime numbers, thus maximal $lcm$ in the order formula. Then define the degree and order of $\sigma$ as follows:

$$|P| = lcm(p_1, \cdots, p_{\pi(L)}) = \prod_{i=1}^{\pi(L)} p_i \quad Deg(P) = \sum_{i=1}^{\pi(L)} p_i$$

Note that the permutation generator $\sigma =< P >$ is composed of $\pi(L)$ cycles of prime length:

$$\sigma = c_1, \cdots, c_{\pi(L)}, \quad |c_i| = p_i, \quad 1 \leq i \leq \pi(L)$$

#### 3.2.2.2 Building public keys:

Alice selects private $x < |P|$, Bob does the same for $y < |P|$. Their public key is calculated under exponentiation as follows:

$$pub_A = \sigma^x$$
$$pub_B = \sigma^y$$

#### 3.2.2.3 Computing the shared secret:

As this protocol is based on the classic Diffie-Hellman approach there is no mystery on how the secret material is computed:

$$shared_A = shared_B = (\sigma^x)^y = (\sigma^y)^x$$

### 3.2.3 Cryptanalysis

As said before, using representable permutation groups arise certain problems, such that DLP turns out to be a solvable problem with the help of the Chinese Remainder Theorem and some basic concepts of algebraic combinatorics.

#### 3.2.3.1 Discrete logarithms on permutation groups

The author found a deadly point in the protocol description, concretely in the public key generation part. By examining exponentiation on the permutation generator $\sigma$ by the private value $x$ in the following way:

$$\begin{aligned} \sigma &= c_1, \cdots, c_{\pi(L)} \\ \sigma^x &= c_1^x, \cdots, c_{\pi(L)}^x \\ \sigma^x &= c_1^{x \equiv_{|c_1|} x_1}, \cdots, c_{\pi(L)}^{x \equiv_{|c_{\pi(L)}|} x_{\pi(L)}} \end{aligned}$$

From the last definition of $\sigma^x$ we recover a system of residues in distinct prime fields $F_{p_i}$ where this system is solved using the Chinese Remainder Theorem.

$$\begin{cases} x \equiv_{p_1} x_1 \\ \cdots \\ x \equiv_{p_{\pi(L)}} x_{\pi(L)} \end{cases}$$

This is the rediscovery of the Pohllig-Hellman method but for permutation groups, which recovers the private exponent $x$ working on distinct cyclic groups of prime order, so distinct $x_i$ that later come up together via CRT to form $x$. As the reader may notice, for every cycle in $\sigma$, the private exponent $x$ is reduced modulo each cycle length thus $x \equiv x_i \pmod{p_i}$ as $p_i$ is the length of the $i$-th cycle on $\sigma$.

#### 3.2.3.2 Solving DLP when exponent is coprime to order

When $x$ is selected to be coprime with the order of $P$ as $\gcd(x, |P|) = 1$, the cycles of $\sigma$ don't suffer any modification on their length, as $x \not\equiv_{|c_i|} 0$. This condition is very important to analyze discrete logarithms as it's trivial to compare $\sigma$ with $\sigma^x$ to check which displacement has been made between symbols of each cycles, retrieving every $x_i$ and ending up mounting the aforementioned system of residues on different prime fields.

#### 3.2.3.3 Solving DLP when exponent is not coprime to order

When $gcd(x, |P|) = d \neq 1$ the thing changes a bit, using Lagrange's theorem we know that $d \mid |P|$ so $x$ must be either a single prime found in the factorisation of $|P|$ or a product of these. The number of cycles of length 1 in $\sigma^x$ denote the number of symbols that have been sent to themselves under exponentiation, thus this number is the sum of the cycle lengths of cycles sent to the identity. The number of cycles of length 1, $n_{c_1}$ is calculated as follows:

$$n_{c_1} = \sum_{\forall p_i \in x} p_i$$

For finding $x$ the attacker only must analyze the 1-cycles. This is trivial and he finds $x$ just by multiplying the prime length cycles that have been sent to the identity in $\sigma^x$. The author finds that this case is easier to solve than the case where the exponent is taken to be coprime with the order, as this case doesn't imply running an instance of CRT since obtaining the 1-cycles would reveal their prime length, being deterministically solved.

#### 3.2.3.4 An implementation case for cryptanalysis

Let's construct a permutation group $P$ using the limit $L = 100$, thus $\pi(L) = 25$ primes under 100. The degree and order are given as:

$$Deg(P) = \sum_{i=1}^{\pi(L)} = 1060$$

$$|P| = \prod_{i=1}^{25} p_i = 2305567963945518424753102147331756070$$

The pseudocode to retrieve such partition of $\pi(L)$ primes, its order and degree can be expressed as follows:

```
Input: The parameter $limit$ where primes will be selected
Output: Degree $n$, Order $ord$ and the integer partition vector $\lambda$
Function GetRndPartitionDegree(limit) is
    $n \leftarrow 0$
    $ord \leftarrow 0$
    $nprimes \leftarrow \pi(limit)$
    for $i \leftarrow 1$ to $nprimes$ do
        $\lambda[i] \leftarrow NthPrime[i]$
        $n \leftarrow n + \lambda[i]$
        $ord \leftarrow ord \cdot lambda[i]$
    end
    return $(n, ord, \lambda)$
end
```

**Algorithm 1:** Generating a random integer partition of prime numbers under *limit*

Print $\lambda$ to check that these primes are correct

$$\lambda = (2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97)$$

Let $\sigma$ be the permutation generator of $P$, it has approximately

$$2^{\log_2(230556796394551842475310214733175607 0)} = 2^{120.79453}$$

possible exponents but it is just composed of 1060 symbols and 25 cycles. Then $\sigma$ is easily representable as a cycle permutation using the provided pseudocode that returns a permutation $\sigma$ of degree $n$ and cycle type $\lambda$:

```
Input: Degree $n$ and an integer partition $\lambda \vdash n$
Output: Permutation $\sigma$ of degree $n$ whose cycle type is $\lambda$
Function GenRndPerm($\lambda$, $n$) is
    $symbols \leftarrow RandomSample[n]$
    $offset \leftarrow 1$
    $\sigma \leftarrow \emptyset$
    for $i \leftarrow 1$ to $|\lambda|$ do
        $\lambda_i \leftarrow \lambda[i]$
        $c_{\lambda_i} \leftarrow List[\lambda_i]$
        $k \leftarrow 1$
        for $j \leftarrow offset$ to $p + offset$ do
            $c_{\lambda_i}[k] = symbols[j]$
        end
        $\sigma \leftarrow \sigma \cdot c_{\lambda_i}$
        $offset \leftarrow offset + \lambda_i$
    end
    return $\sigma$
end
```

**Algorithm 2:** Generating random permutation of degree $n$ and cycle type $\lambda$

**Let's write the obtained representation of $\sigma$.**

$\sigma = (1, 2), (3, 4, 5), (6, 7, 8, 9, 10), (11, 12, 13, 14, 15, 16, 17), (18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28),$
$(29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41), (42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58),$
$(59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77), (78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89,$
$90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100), (101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114,$
$115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129), (130, 131, 132, 133, 134, 135, 136,$
$137, 138, 139, 140, 141, 142, 143, 144, 145, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156, 157, 158,$
$159, 160), (161, 162, 163, 164, 165, 166, 167, 168, 169, 170, 171, 172, 173, 174, 175, 176, 177, 178, 179,$
$180, 181, 182, 183, 184, 185, 186, 187, 188, 189, 190, 191, 192, 193, 194, 195, 196, 197), (198, 199, 200,$
$201, 202, 203, 204, 205, 206, 207, 208, 209, 210, 211, 212, 213, 214, 215, 216, 217, 218, 219, 220, 221,$
$222, 223, 224, 225, 226, 227, 228, 229, 230, 231, 232, 233, 234, 235, 236, 237, 238), (239, 240, 241, 242,$
$243, 244, 245, 246, 247, 248, 249, 250, 251, 252, 253, 254, 255, 256, 257, 258, 259, 260, 261, 262, 263,$
$264, 265, 266, 267, 268, 269, 270, 271, 272, 273, 274, 275, 276, 277, 278, 279, 280, 281), (282, 283,$
$284, 285, 286, 287, 288, 289, 290, 291, 292, 293, 294, 295, 296, 297, 298, 299, 300, 301, 302, 303,$
$304, 305, 306, 307, 308, 309, 310, 311, 312, 313, 314, 315, 316, 317, 318, 319, 320, 321, 322,$
$323, 324, 325, 326, 327, 328), (329, 330, 331, 332, 333, 334, 335, 336, 337, 338, 339, 340, 341,$
$342, 343, 344, 345, 346, 347, 348, 349, 350, 351, 352, 353, 354, 355, 356, 357, 358, 359, 360, 361,$
$362, 363, 364, 365, 366, 367, 368, 369, 370, 371, 372, 373, 374, 375, 376, 377, 378, 379, 380, 381),$
$(382, 383, 384, 385, 386, 387, 388, 389, 390, 391, 392, 393, 394, 395, 396, 397, 398, 399, 400, 401,$
$402, 403, 404, 405, 406, 407, 408, 409, 410, 411, 412, 413, 414, 415, 416, 417, 418, 419, 420, 421,$
$422, 423, 424, 425, 426, 427, 428, 429, 430, 431, 432, 433, 434, 435, 436, 437, 438, 439, 440),$
$(441, 442, 443, 444, 445, 446, 447, 448, 449, 450, 451, 452, 453, 454, 455, 456, 457, 458, 459,$
$460, 461, 462, 463, 464, 465, 466, 467, 468, 469, 470, 471, 472, 473, 474, 475, 476, 477, 478,$
$479, 480, 481, 482, 483, 484, 485, 486, 487, 488, 489, 490, 491, 492, 493, 494, 495, 496, 497$
$, 498, 499, 500, 501), (502, 503, 504, 505, 506, 507, 508, 509, 510, 511, 512, 513, 514, 515,$
$516, 517, 518, 519, 520, 521, 522, 523, 524, 525, 526, 527, 528, 529, 530, 531, 532, 533,$
$534, 535, 536, 537, 538, 539, 540, 541, 542, 543, 544, 545, 546, 547, 548, 549, 550, 551,$
$552, 553, 554, 555, 556, 557, 558, 559, 560, 561, 562, 563, 564, 565, 566, 567, 568),$
$(569, 570, 571, 572, 573, 574, 575, 576, 577, 578, 579, 580, 581, 582, 583, 584, 585, 586,$
$587, 588, 589, 590, 591, 592, 593, 594, 595, 596, 597, 598, 599, 600, 601, 602, 603, 604,$
$605, 606, 607, 608, 609, 610, 611, 612, 613, 614, 615, 616, 617, 618, 619, 620, 621, 622,$
$623, 624, 625, 626, 627, 628, 629, 630, 631, 632, 633, 634, 635, 636, 637, 638, 639), (640,$
$641, 642, 643, 644, 645, 646, 647, 648, 649, 650, 651, 652, 653, 654, 655, 656, 657, 658, 659,$
$660, 661, 662, 663, 664, 665, 666, 667, 668, 669, 670, 671, 672, 673, 674, 675, 676, 677, 678,$
$679, 680, 681, 682, 683, 684, 685, 686, 687, 688, 689, 690, 691, 692, 693, 694, 695, 696, 697,$
$698, 699, 700, 701, 702, 703, 704, 705, 706, 707, 708, 709, 710, 711, 712), (713, 714, 715,$

$716, 717, 718, 719, 720, 721, 722, 723, 724, 725, 726, 727, 728, 729, 730, 731, 732, 733, 734, 735, 736, 737, 738, 739, 740, 741, 742, 743, 744, 745, 746, 747, 748, 749, 750, 751, 752, 753, 754, 755, 756, 757, 758, 759, 760, 761, 762, 763, 764, 765, 766, 767, 768, 769, 770, 771, 772, 773, 774, 775, 776, 777, 778, 779, 780, 781, 782, 783, 784, 785, 786, 787, 788, 789, 790, 791), (792, 793, 794, 795, 796, 797, 798, 799, 800, 801, 802, 803, 804, 805, 806, 807, 808, 809, 810, 811, 812, 813, 814, 815, 816, 817, 818, 819, 820, 821, 822, 823, 824, 825, 826, 827, 828, 829, 830, 831, 832, 833, 834, 835, 836, 837, 838, 839, 840, 841, 842, 843, 844, 845, 846, 847, 848, 849, 850, 851, 852, 853, 854, 855, 856, 857, 858, 859, 860, 861, 862, 863, 864, 865, 866, 867, 868, 869, 870, 871, 872, 873, 874), (875, 876, 877, 878, 879, 880, 881, 882, 883, 884, 885, 886, 887, 888, 889, 890, 891, 892, 893, 894, 895, 896, 897, 898, 899, 900, 901, 902, 903, 904, 905, 906, 907, 908, 909, 910, 911, 912, 913, 914, 915, 916, 917, 918, 919, 920, 921, 922, 923, 924, 925, 926, 927, 928, 929, 930, 931, 932, 933, 934, 935, 936, 937, 938, 939, 940, 941, 942, 943, 944, 945, 946, 947, 948, 949, 950, 951, 952, 953, 954, 955, 956, 957, 958, 959, 960, 961, 962, 963), (964, 965, 966, 967, 968, 969, 970, 971, 972, 973, 974, 975, 976, 977, 978, 979, 980, 981, 982, 983, 984, 985, 986, 987, 988, 989, 990, 991, 992, 993, 994, 995, 996, 997, 998, 999, 1000, 1001, 1002, 1003, 1004, 1005, 1006, 1007, 1008, 1009, 1010, 1011, 1012, 1013, 1014, 1015, 1016, 1017, 1018, 1019, 1020, 1021, 1022, 1023, 1024, 1025, 1026, 1027, 1028, 1029, 1030, 1031, 1032, 1033, 1034, 1035, 1036, 1037, 1038, 1039, 1040, 1041, 1042, 1043, 1044, 1045, 1046, 1047, 1048, 1049, 1050, 1051, 1052, 1053, 1054, 1055, 1056, 1057, 1058, 1059, 1060)$

The reader can check himself that $\sigma$ has in fact 1060 symbols and 25 cycles each of prime length, i.e, the last cycle of $\sigma$ has 97 symbols (the last prime under 100, thus cycle length 97).

Now we are in the position of working in the cryptanalysis of the key exchange protocol as the previous construction of $P$ and $\sigma$ is completed. Let's try to retrieve the private exponent $x$ used by Alice. For that Alice selects $x < |P|$ in an interval $[start, ord]$ which translated to pseudocode looks like:

```
**Input:** Interval start and multiplicative order of the permutation $\sigma$
**Output:** An exponent $x$ that's coprime with $ord$
**Function** *GenRndExp(start, ord)* **is**
    $found \leftarrow false$
    $x \leftarrow ord$
    **while** *not* $Coprime(x, ord)$ **do**
        $x \leftarrow Rand(start, ord)$
    **end**
    **return** x
**end**
```

**Algorithm 3:** Generating random exponents coprime with $Ord_G(\sigma)$

i.e $x = 1285784155774121654618741 = 1879 \cdot 708892241 \cdot 965297245219$ which none of those primes are in the prime factorization of $|P|$ thus $\gcd(x, |P|) = 1$. Then Alice builds her public key via exponentiation as $\sigma^x = \sigma^{1285784155774121654618741}$ which results in another permutation composed of 25 cycles maintaining their prime length, as $d_i \mid x \wedge d_i \nmid \ |P|$. To retrieve $x$ the trick is for each cycle $c_{i,\sigma}$ on $\sigma$ and $c_{i,\sigma^x}$ in $\sigma^x$ calculate the distance from the first symbol of $c_{(i,\sigma)}$ to the image of that symbol in $c_{(i,\sigma^x)}$, this is for two cycles of same length one in $\sigma$ and the other in $\sigma^x$ the exponent $x_i$ is recovered measuring the distance from $sym_{c_{(i,\sigma)}}$ to the image $\sigma^x(sym_{c_{(i,\sigma)}})$ in the cycle $c_{(i,\sigma)}$. It is very simple and the worst case is to iterate on $p_i$ symbols as $p_i$ is the cycle length of both $c_{(i,\sigma)}$ and $c_{(i,\sigma^x)}$. Then conclude that the security of this protocol is not bounded by the possible private exponent candidates, this is by $|P|$ but by the permutation group's degree $Deg(P)$ as the worst case would be to iterate on $\sum_{i=1}^{\pi(L)} p_i$ symbols. Let $c_{(25,\sigma)}$ be the 25-th cycle on the generator $\sigma$ and $c_{(25,\sigma^x)}$ be the 25-th cycle on Alice's public key permutation $\sigma^x$. We want to retrieve the exponent $x_{25}$ that contributes to the system of residues that hides $x$. First present these two cycles as following:

$$c_{(25,\sigma)} = (964, 965, 966, 967, 968, 969, 970, 971, 972, 973, 974, 975, 976, 977, 978, 979, 980, 981, 982, 983, 984, 985, 986, 987, 988, 989, 990, 991, 992, 993, 994, 995, 996, 997, 998, 999, 1000, 1001, 1002, 1003, 1004, 1005, 1006, 1007, 1008, 1009, 1010, 1011, 1012, 1013, 1014, 1015, 1016, 1017, 1018, 1019, 1020, 1021, 1022, 1023, 1024, 1025, 1026, 1027, 1028, 1029, 1030, 1031, 1032, 1033, 1034, 1035, 1036, 1037, 1038, 1039, 1040, 1041, 1042, 1043, 1044, 1045, 1046, 1047, 1048, 1049, 1050, 1051, 1052, 1053, 1054, 1055, 1056, 1057, 1058, 1059, 1060)$$

$$c_{(25,\sigma^x)} = (964, 970, 976, 982, 988, 994, 1000, 1006, 1012, 1018, 1024, 1030, 1036, 1042, 1048, 1054, 1060, 969, 975, 981, 987, 993, 999, 1005, 1011, 1017, 1023, 1029, 1035, 1041, 1047, 1053, 1059, 968, 974, 980, 986, 992, 998, 1004, 1010, 1016, 1022, 1028, 1034, 1040, 1046, 1052, 1058, 967, 973, 979, 985, 991, 997, 1003, 1009, 1015, 1021, 1027, 1033, 1039, 1045, 1051, 1057, 966, 972, 978, 984, 990, 996, 1002, 1008, 1014, 1020, 1026, 1032, 1038, 1044, 1050, 1056, 965, 971, 977, 983, 989, 995, 1001, 1007, 1013, 1019, 1025, 1031, 1037, 1043, 1049, 1055)$$

Take the first symbol in $c_{25,\sigma}$, this is, $sym_{c_{(25,\sigma)}} = 964$. Now calculate the image of that symbol in $\sigma^x$, this is $\sigma^x(sym_{c_{(25,\sigma)}}) = \sigma^x(964) = 970$. The technique is to calculate how many positions are from 964 to 970 in $c_{(25,\sigma)}$, which inherently is 6. Doing it for every cycle in $\sigma, \sigma^x$ retrieves the complete list of residues $x_i$, starting from $x_1$ to $x_{25}$ over the prime fields $F_2$ to $F_{97}$. The following pseudo-code illustrates the technique employed to solve the Discrete Logarithm Problem in permutation groups:

```
**Input:** Public generator $\sigma$ and permutation power $\sigma^x$
**Output:** The private exponent $x$
**Function** *SolveDLP($\sigma$,$\sigma^x$)* **is**
    **for** $i \leftarrow 1$ **to** $n_c(\sigma)$ **do**
        $c_\sigma \leftarrow \sigma[i]$
        $c_{\sigma^x} \leftarrow \sigma^x[i]$
        $len_{c_\sigma} \leftarrow |c_\sigma|$
        $modlist[i] \leftarrow len_{c_\sigma}$
        $sym_{c_\sigma} \leftarrow c_\sigma[1]$
        $sym_{c_{\sigma^x}} \leftarrow \sigma^x(sym)$
        **for** $j \leftarrow 1$ **to** $len_{c_\sigma}$ **do**
            **if** $c_\sigma[i] = sym_{c_{\sigma^x}}$ **then**
                $residues[i] \leftarrow j - 1$
                break;
            **end**
        **end**
    **end**
    $x \leftarrow CRT(residues, modlist)$
    **return** $x$
**end**
```

**Algorithm 4:** Solving DLP on permutations groups

It is self-explanatory, using all the aforementioned concepts, ending on the resolution of $x$ applying the CRT. The list *residues* contains the 25 residues that have been found.:

$$residues = (1, 2, 1, 4, 4, 7, 3, 16, 15, 11, 4, 26, 37, 22, 30, 11, 40, 37, 12, 6, 47, 73, 40, 13, 6)$$

where the $i$-th residue is taken in $F_{p_i}$ i.e the 5-th residue is 4 since $x \equiv_{11} x_5 \equiv 4$ where 11 is the 5-th prime.

In the end the real $x$ is recovered, being the space of $x$ composed of approximately $2^{120}$ candidates, which in the beginning of the analysis *seemed good enough.* When $\gcd(x, |P|) \neq 1$ the DLP is deterministcally solved just multiplying the cycle prime length of cycles that have been sent to the identity.

### 3.2.4 Conclusion

We've seen how to solve the discrete logarithm problem in permutation groups when the order of the group is composed by an increasing product of primes. The trade-off between the order and degree is necessary to be able to represent elements in $P$. But this approach causes a deadly impact on the probability of breaking the scheme. The conclusion made is that this protocol cannot be enhanced in any form as its security depends entirely on the number of symbols, this is, bounded by $Deg(P)$ and not by the order $|P|$. This is equal to say that the Discrete Logarithm Problem in cyclic groups is bounded by the minimal number of symbols of its permutation group representation. The number of symbols and cycles depends on the order of $G$, so it's crucial that the factorization of $|G|$ does not contain a big quantity of factors.

Thus, demonstrating why in Diffie-Hellman is recommended that $p = 2q+1$ as the order of $G$ is $2q$ a generator $g$ is selected of order $q$ that generates a cyclic group of big order $q$, this is the same as having a permutation generator of 1 cycle of prime order $q$ thus indecomposable in smaller cycles and rendering this cryptanalysis useless since $Deg(P) = |P|$ which is not representable. Then as the final conclusion, **DLP is bounded by** $Deg(P)$.

## 3.3 Stickel's Protocol on permutation groups

The author thought that one way to enhance the previous protocol would be to change the way of computing public and shared keys in the key negotation-exchange. Stickel's protocol found in 2.2.1 fits well here because it's based on exponentiation where this depends on the DLP in permutation groups. As seen before, this is fully deterministic and solvable, but Stickel's combines two exponentiations via multiplication, thus, *it may hide* the structure of both exponentiations making it harder to recover one or more private keys. Nonetheless, in the cryptanalysis section it is proven to be broken by an approach based on set theory and combinatorics, a novel method that may have cryptanalytic application in other schemes.

### 3.3.1 Protocol Description

The platform group now is $G = S_n$ the symmetric group on $n$ symbols. Two permutations $\alpha, \beta$ are selected such that generate subgroups $A, B \leq G$. To construct an arbitrary generator, select a partition $\lambda \vdash n$ and construct a cycle of length $\lambda_i$ for each partition symbol $\lambda_i$, each cycle containing unique symbols from the set of $n$ symbols. Head to 3.2.3.4 or in [9] for an example on how to obtain such permutation. There's a slight modification when computing shared-public key when comparing this protocol to the previous one:

$$\begin{aligned} \sigma \in A \quad \rho \in B \quad a_1, a_2 < Ord_G(\sigma) \quad b_1, b_2 < Ord_G(\rho) \\ pub_A = \sigma^{a_1}\rho^{b_1} \quad pub_B = \sigma^{a_2}\rho^{b_2} \\ shared_A = \sigma^{a_1}(\sigma^{a_2}\rho^{b_2})\rho^{b_1} = \sigma^{a_1+a_2}\rho^{b_1+b_2} \\ shared_B = \sigma^{a_2}(\sigma^{a_1}\rho^{b_1})\rho^{b_2} = \sigma^{a_1+a_2}\rho^{b_1+b_2} \end{aligned}$$

Adding an extra non-commutative operation, the multiplication-composition in $S_n$. This is, two elements that lie in different subgroups $A, B \leq G$ but their multiplication lies in $G$. And this non-commutativity is what prevents the attacker of obtaining $shared_A, shared_B$ as multiplying public keys doesn't give any information to the attacker. This is explained in the Stickel's protocol description in [TODO REF].

### 3.3.2 Cryptanalysis

The goal is to recover either $(a_1, b_1)$ or $(a_2, b_2)$ as one of these pairs makes the attacker succeed in the shared secret computation. We know that the DLP is present here but hidden by multiplication on $G$. The public keys, *apparently*, don't give information on the structure of exponentiation in both generators $\alpha, \beta$. In the previous section only one generator was used, revealing how the exponent $x$ affects and shift symbols in cycles and this is crucial to recover $x$. Now the attacker faces a harder scenario to solve, where first he must take out the multiplication envelope to be able to start recovering any private exponent. As we are in permutation groups it's very important to understand how permutation composition (multiplication) works.

#### 3.3.2.1 Definition of right-to-left multiplication

The *right-to-left* multiplication for permutations $\alpha\beta = \delta$ can be written for every symbol $i$ as $\alpha\beta(i) = \delta(i)$ i.e $\beta(i) = j \rightarrow \delta(i) = \alpha(j)$. It is important to analyze it as multiplication *hides* two exponentials that rely on the DLP. Attacker knows $(\alpha, \beta, (\alpha^{a_1}\beta^{b_1}), (\alpha^{a_2}\beta^{b_2}))$. It results that multiplication can be decomposed to recover either $(a_1, b_1)$ or $(a_2, b_2)$ using simple set theory and algebraic combinatorics.

#### 3.3.2.2 Algebraically obtaining private exponents

A good practice when dealing with cryptnalaysis is to identify the reversal procedure of setting up public keys. The mathematical problem found in the reversal procedure can believed to be hard as integer factorization or DLP in certain groups, but that doesn't prevent an analyst to know which algebraic expressions *break or recover* private information. In this case, we have $pub_A = \delta = \alpha^{a_1}\beta^{b_1}$, so if we recover $b_1$, by the inverse of the exponentiation on $\beta$ by $b_1$ we obtain $\alpha^{a_1}\beta^{b_1}\beta^{-b_1} = \alpha^{a_1}$. As we have $\alpha, \alpha^{a_1}$, then $a_1$ is easily recoverable applying the DLP solving method on permutation groups as shown in the previous protocol section 3.2.3.4, Algorithm 4. Summarizing, if $b_1$ is recoverable then $a_1$ is immediately obtained as DLP is trivial to solve when permutations can be represented, as this is the case. Further, in this section, these algebraic expression are helpful in the cryptanalytic part as $b_1$ or $b_2$ is proven to be recoverable under specific circumstances. The author focus on recovering $(a_1, b_1)$ in Alice's public key $\delta = \alpha^{a_1}\beta^{b_1}$.

#### 3.3.2.3 Recovering $b_1$ from the product $\alpha^{a_1} \cdot \beta^{b_1}$

It has been theoretically proven that recovering the exponent $b_1$ immediately gives the remaining exponent $a_1$, hence it is important to analyze the permutation product of both exponentiations in $\alpha, \beta$ because multiplication *hides* both $(a_1, b_1)$. The attacker knows $pub_A = \delta = \alpha^{a_1}\beta^{b_1}$. Then for an arbitrary symbol $i < Deg(S_n)$ let $\delta(i) = k$, such that $\alpha^{a_1}(j) = k$ and $\beta^{b_1}(i) = j$. Thus there exists a cycle $c_{(k,\alpha)}$ on $\alpha$ that contains symbols $j$ and $k$. Moreover, there exists a cycle

$c_{(i,\beta)}$ on $\beta$ that contains $i$ and $j$. Inherently define the intersection $I_{(c_{(k,\alpha)},c_{(i,\beta)})}$ of cycles $c_{(k,\alpha)}$ and $c_{(i,\beta)}$ with cardinality $r$ as:

$$I_{(c_{(k,\alpha)},c_{(i,\beta)})} = c_{(k,\alpha)} \cap c_{(i,\beta)} = (j_1, \cdots, j_r)$$

The intersection contains all the possible $j$'s such that $c_{(i,\beta)}^{D_{i,j}}(i) = j \quad c_{(k,\alpha)}^{D_{j,k}}(j) = \delta(i) = k.$, giving $r$ exponents $D_{i,j}$, which are the distances taking $c_{(i,\beta)}$ from $i$ to every $j$ in the intersection. When $r = 1$ we know that there exists only one $j$ in $c_{(i,\beta)}$ that goes to $c_{(k,\alpha)}$ and satisfies the aforementioned cycle-image relation. The distance $D_{i,j}$ from $i$ to $j$ in $c_{(i,\beta)}$ determines the congruence

$$b_1 \equiv D_{i,j} \pmod{|c_{i,\beta}|}$$

As the reader may know, an attack can be mounted where applying CRT retrieves the original $b_1$ once the attacker recovered all the distances modulo each cycle length in $\beta$. Because the exponentiation $\beta^{b_1}$ can be viewed as exponentiating every cycle by separate, thus reducing the exponent $b_1$ modulo each cycle length.

It's been proven that the protocol is broken solving two system of congruences, one for $b_1$ and the other for $a_1$ if and only if every cycle $c_{(i,\beta)}$ has an intersection on cycles $c_{(\delta(sym),\alpha)}$ for each symbol $sym \in c_{(i,\beta)}$ where $r = 1$. But when $r > 1$ the thing changes a bit. It could happen that a particular cycle $c_{(i,\beta)}$ intersects with $r > 1$ on $c_{(\delta(sym),\alpha)}$ for each symbol $sym \in c_{(i,\beta)}$ . Then for every cycle containing the symbol $i$ in $\beta$, this is $c_{(i,\beta)}$, take the minimum intersection cardinality, this is, the least $r$, and this would give us $r$ possible distances $D_{i,j}$. Eventually, when solving for $b_1$ it would require

$$\prod_{i=1}^{n_c(\beta)} r_i$$

systems of congruences where $n_c(\beta)$ denotes the number of cycles in $\beta$ and $r_i$ is the **minimum cardinality** found from intersecting $c_{(i,\beta)}$ and $c_{(\delta(sym),\alpha)}$ for each symbol $sym \in c_{(i,\beta)}$. The complete algorithm to recover $(a_1, b_1)$ when $r = 1$ is given in pseudo-code that is latter found in the Appendix which contains its Mathematica code version.

```
Input: Public generators $\alpha, \beta$ and Alice's public key: $\delta = \alpha^{a_1}\beta^{b_1}$
Output: The private exponent tuple $(a_1, b_1)$
Function SolveStickel's($\alpha$,$\beta$, $\delta$) is
    for $i \leftarrow 1$ to $n_c(g)$ do
        $c_\beta \leftarrow \beta[i]$
        $len_{c_\beta} \leftarrow |c_\beta|$
        $modlist[i] \leftarrow len_{c_\beta}$
        for $j \leftarrow 1$ to $len_{c_g}$ do
            $sym_{c_\beta} \leftarrow c_\beta[j]$
            $k \leftarrow \delta(sym_{c_\beta})$
            $c_\alpha \leftarrow FindSymInCycle(k, \alpha)$
            $I_{(c_\beta, c_\alpha)} \leftarrow c_\beta \cap c_\alpha$
            $r \leftarrow |I_{(c_\beta, c_\alpha)}|$
            if $r = 1$ then
                $sym_{I_{(c_\beta, c_\alpha)}} \leftarrow I_{(c_\beta, c_\alpha)}[1]$
                $found \leftarrow false$
                $residue \leftarrow 0$
                while $not\ found$ do
                    $img \leftarrow \beta(i)$
                    $residue \leftarrow residue + 1$
                    if $img = sym_{I_{(c_\beta, c_\alpha)}}$ then
                        $residues[i] \leftarrow residue$
                        $found \leftarrow true$
                    end
                end
                break;
            end
        end
    end
    $b_1 \leftarrow CRT(residues, modlist)$
    $\alpha^{a_1} \leftarrow \alpha^{a_1}\beta^{b_1}\beta^{-b_1}$
    $a_1 \leftarrow SolveDLP(\alpha, \alpha^{a_1})$
    return $(a_1, b_1)$
end
```

**Algorithm 5:** Recovering $(a_1, b_1)$ from $\alpha^{a_1}\beta^{b_1}$

It does what has been explained: if for every cycle in $\beta$ we can found an intersection with $r = 1$ then we know the reduced exponent modulo $|c_{(i,\beta)}|$. Obtaining such reduced exponents for every cycle in $\beta$ gives a system of residues that's solved applying CRT and returning $b_1$. Then $a_1$ is immediately retrieved as a consequence of obtaining $b_1$. So we can compute the shared key as $shared_{ATTK} = \alpha^{a_1}(\alpha^{a_2}\beta^{b_2})\beta^{b_1}$.

## 3.4 Twisted Stickel's Protocol on permutation groups

### 3.4.1 Protocol Description

Last protocol included in this paper is a combined version of the Twisted Protocol and Stickel's, included in 2.2.3, where two subgroups $A, B \leq G$ are taken not commuting elementwise thus $ab \neq ba \quad a \in A, b \in B$. The definition of the entire protocol is given as:

$$\begin{aligned} pub_A &= \alpha^{a_1} g \beta^{b_1} \\ pub_B &= \alpha^{a_2} g \beta^{b_2} \\ shared_A = \alpha^{a_1}(\alpha^{a_2} g \beta^{b_2})\beta^{b_1} &= \alpha^{a_1+a_2} g \beta^{b_1+b_2} \\ shared_B = \alpha^{a_2}(\alpha^{a_1} g \beta^{b_1})\beta^{b_2} &= \alpha^{a_1+a_2} g \beta^{b_1+b_2} \end{aligned}$$

The principle is the same as in Stickel's, multiplying both exponentiations *hides* the exponents, but now, introduces a central publicly known element $g$. How does this parameter really affect the security comparing to Stickel's approach using permuation groups?

### 3.4.2 Cryptanalysis

It can be demonstrated that this modified twisted protocol is more vulnerable that the previous protocol since every intersection found has cardinality $r = 1$ making it trivial to solve as only two systems of congruences are needed for recovering $(a_1, b_1)$ or $(a_2, b_2)$. The proof can be given following the *right-to-left* multiplication rule. Let $\delta = pub_A = \alpha^{a_1} g \beta^{b_1}$. Then for an arbitrary symbol $i$ decompose the image in $\delta$ as $\delta(i) = r \quad \alpha^{a_1}(k) = r \quad g(j) = k \quad \beta^{a_1}(i) = j$. Exploiting this decomposition to our favor is what does this protocol vulnerable. Take the cycle $c_{(i,\beta)}$, now find all the cycles $c_{(k,g)}$ that contains any symbol $k$ of $c_{(r,\alpha)}$ One of these cycles must intersect with $c_{(i,\beta)}$ and there is only one right candidate as $g(j) = k$, since $g$ is not taking any exponentiation. From here, we know that $r = 1$ thus extract the distance $D_{i,j}$ modulo $|c_{(i,\beta)}|$ that goes from $i$ to $j$ as seen also in 3.3.2.3. This is a tricky analysis, but decomposition is what makes it understandable, viewing it as a triple intersection with the condition $g(j) = k$ and $r = 1$. For the rest of the analysis, it's enough to reuse the attacks presented in previous sections. Recover all distances to mount a system of residues that's solved by applying CRT, thus recovering $b_1$. Now recover $g^{a_1}$ and finally retrieve $a_1$ by a simple system of congruences obtained with the original DLP solving technique explained in 3.2.3.4, Algorithm 4. As in the previous cases, an example is given to prove that it is solvable despite of the magnitude of the used parameters.

## 4 Final Conclusion

We have seen distinct key exchange protocols that initially seemed good enough to work with in Cryptography. The weaknesses found using permutation groups have a common factor: platform groups where elements are given as permutations introduce a clear risk factor of being entirely solved. Operations like: multiplication, exponentiation and conjugation are imminently decomposable. Also, it's been said that the discrete logarithm problem is bounded by the degree of the platform group where the congruence lies. Hence obtaining a permutation representation must not be possible for an attacker to solve the DLP.

Besides, the presented protocol based on factorization-search using matrices over $F_p^n$ is proven to be vulnerable as every element presented in public keys is decomposable into linear factors that make the private values recoverable. Using matrices can be advantageous if the reversal procedure of setting up public keys cannot be tracked down to a **solvable** system of linear equations. Which was not the case, as the resulting system is overdetermined, having a solution when $x \in F_p^n$ is selected correctly.

The short conclusion of this paper is that protocols based on decomposition, factorization-search and conjugation-search are interesting candidates to research in non-commutative cryptography. But note that these protocols are vulnerable if the platform group is isomorphic to a permutation group, where the isomorphism must be known and computable. In addition, special care must be taken when dealing with linearity and matrices.

## 5 Appendix: Implemented functions on Mathematica

All the provided pseudo-code was first implemented in Mathematica then transformed to symbolic pseudo-code. Mathematica's language works with 1-index instead of 0-index, that's why in the pseudo-code it starts counting from 1. The choice of Mathematica is obvious as it is a complete CAS that has all the number theoretic and group theoretic related functions, which in C++ would be easy to implement using alternative libraries like NTL, GMP between others. However, this research is not focused on reinventing the wheel in another language. Mathematica is enough to demonstrate and proof that these schemes are vulnerable, being another strong point that the pseudo-code can be implemented in whatever language of choice, thus being open for any researcher to test on any platform-language.

### 5.1 Solving Stickel's and permutation Diffie-Hellman

The following code solves the underlying problems where 3.2 and 3.3 are based. Moreover it can be adapted to solve 3.4 as well as the underlying problem is very similar.

First construct the list of primes so that permutation generators $\alpha, \beta$ have the same cycle type, thus same order, but not necessarily generate the same subgroup. From here we obtain the order and degree. Note that you can use any integer partition in modlist.

```
In[1]:= modlist = Table[Prime[n],{n,25}]
        {2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,89,97}
        deg = Sum[modlist[[i]],{i, 1,25}]
        1060
        ord = Product[modlist[[i]],{i,1,Length[modlist]}]
        2305567963945518424753102147331756070
```

Now define the primitives GenRndExp that returns a random exponent coprime with the order in the range $[2^6, ord]$ and GenRndPerm that returns a random permutation with cycle type consisting of the prime elements in modlist.

```
In[2]:=  GenRndExp[ord_]:=(
        While[True,
        num = RandomInteger[{2^64,ord}];
        If [CoprimeQ[num,ord],Return[num];];
        ];
        );
        GenRndPerm[cyclen_,deg_]:=(
        perm = Cycles[{}];
        offset=1;
        symbols = RandomSample[Range[deg]];
        For[i=1,i<=Length[cyclen],i++,
        p = cyclen[[i]];
        l = Array[0&,p];
        k=1;
        For[j=offset,j<(p+offset),j++,
        l[[k]]=symbols[[j]];
        k++;
        ];
        offset = offset + p;
        perm = PermutationProduct[perm,Cycles[{l}]];
        ];
        Return[perm];
        );
```

Generate permutations $\alpha, \beta$ and private exponents $x, y$ such that $\delta = \alpha^x \beta^y$

```
In[3]:= x =GenRndExp[ord]
        2301666390105283089435372891658364533
        y = GenRndExp[ord]
        258716250145449892560235270619190979
        alpha=GenRndPerm[modlist,deg]
        beta = GenRndPerm[modlist,deg]
        delta = PermutationProduct[PermutationPower[beta,y],PermutationPower[alpha,x]]
```

For obvious reasons, permutations are not entirely included here as it would take too much space. Now we are in the position to solve the Stickel's protocol, this is recover $(x, y)$ from $\delta = \alpha^x \beta^y$. For that call the method SolveStickels which first retrieves $y$, recovers $\alpha^x$ and solves for $x$ calling the method SolveDLP. Pseudo-code for SolveStickels can be found in 3.3.2.3, Algorithm 5. In the same way, pseudo-code for SolveDLP can be found in 3.2.3.4 Algorithm 4.

Then first define SolveDLP method as SolveStickels rely on this one for recovering $x$ once $y$ is known.

```
In[4]:= SolveDLP[perm_,powperm_] := (
        newmodlist = Array[0&,Length[modlist]];
        residues=Array[0&,Length[modlist]];
        For[i=1,i <= Length[modlist],i++,
        csigma = perm[[1]][[i]];
        csigmax = powperm[[1]][[i]];
        cyclen = Length[csigma];
        newmodlist[[i]] = cyclen;
        imgsymsigmax = csigmax[[2]];
        For[j=1,j<=Length[csigma],j++,
        If[csigma[[j]] == imgsymsigmax, Break[];];
        ];
        residues[[ i]]=j-1;
        ];
        Print[ChineseRemainder[residues,newmodlist]];
        );
        FindCycSym[sym_,perm_]:=(
        For[k=1,k<=Length[perm[[1]]],k++,
        pcyc = perm[[1]][[k]];
        For[m=1,m<=Length[pcyc],m++,
        psym = pcyc[[m]];
        If[psym == sym, Return[pcyc];];
        ];
        ];
        );
```

Now define SolveStickels, as said it will recover both $x, y$.

```
In[5]:= SolveStickels[perm1_,perm2_,delta_,deg_]:=(
betalen = Length[perm2[[1]]];
newmodlist = Array[0&,betalen];
residues = Array[0&,betalen];
onelinedelta = PermutationList[delta,deg];
For[i=1,i<=betalen,i++,
cbeta = perm2[[1]][[i]];
newmodlist[[i]] = Length[cbeta];
For[j=1,j<=Length[cbeta],j++,
symbeta = cbeta[[j]];
imgsymdelta =onelinedelta[[symbeta]];
calpha = FindCycSym[imgsymdelta,perm1];
intcycalphabeta = Intersection[calpha,cbeta];
interlen = Length[intcycalphabeta];
If [interlen == 1,
intsym = intcycalphabeta[[1]];
onelinecbeta = PermutationList[Cycles[{cbeta}],deg];
img=symbeta;
residue=0;
While[True,
img = onelinecbeta[[img]];
residue++;
If[img == intsym, residues[[i]]=residue; Break[];];
];
Break[];
];
];
];
```

Now let's recover both $x, y$ by passing to SolveStickels the permutations $\alpha, \beta, \delta$ which are publicly known in the scheme.

```
In[6]:= SolveStickels[alpha,beta,delta,deg]
        25871625014544989256023527061919O979
        2301666390105283089435372891658364533
```

It has recovered both $x, y$ thus we are able to compute the same shared values that Alice and Bob compute.

## References

[1] Neal R. WagnerMarianne R. Magyarik *A Public-Key Cryptosystem Based on the Word Problem* CRYPTO 1984: Advances in Cryptology pp 19-36

[2] K. H. Ko, S. J. Lee, J. H. Cheon, J. W. Han, J. Kang and C. Park *New Public-Key Cryptosystem Using Braid Groups* CRYPTO 2000: Advances in Cryptology pp 166-183

[3] Vladimir Shpilrain and Alexander Ushakov *A new key exhcange protocol based on the decomposition problem* `https://eprint.iacr.org/2005/447.pdf`

[4] Vladimir Shpilrain, Alexander Ushakov *Thompson's group and public key cryptography* `https://arxiv.org/abs/math/0505487`

[5] E. Stickel *A New Method for Exchanging Secret Keys* Third International Conference on Information Technology and Applications `https://ieeexplore.ieee.org/document/1488999/`

[6] Vladimir Shpilrain *Cryptanalysis of Stickel's Key Exchange Scheme* JCSR 2008: Computer Science – Theory and Applications pp 283-288

[7] Adi Ben-Zvi, Arkadius Kalka, and Boaz Tsaban *Cryptanalysis via algebraic spans* `https://eprint.iacr.org/2014/041.pdf`

[8] D. Garber, S. Kaplan, M. Teicher, B. Tsaban, U. Vishne *Length-based conjugacy search in the Braid group* `https://arxiv.org/abs/math/0209267v2`

[9] Adi Ben-Zvi, Arkadius Kalka, and Boaz Tsaban *Cryptanalysis via algebraic spans* `https://eprint.iacr.org/2014/041.pdf`

[10] Alexei Myasnikov, Vladimir Shpilrain and Alexander Ushakov *Non-commutative Crtpyography and Complexity of Group-theoretic Problems* Mathematical Surveys and Monographs VOL 177