CRYPTOGRAPHIC APPLICATIONS OF NONABELIAN GROUPS

CHRISTOPHER BATES, NICOLAS MEYER, THOMAS PULICKAL

Abstract. Since the development of the RSA cryptographic system by Rivest, Shamir, and Adleman, in 1977, RSA has become a cornerstone of communication security. However, as computers gain in processing speed, it is imperative that stronger cryptographic schemes be implemented in order to withstand attacks using advanced mathematical techniques and cutting edge technology. In this project, we conduct an analysis of the Cayley-Purser Algorithm as a public key cryptoscheme. We implement the Cayley-Purser Algorithm with the GAP System for Computational Discrete Algebra. Also, we discuss a generic method of key exchange that requires nonabelian groups with specific properties. We investigate the braid group and its application to this key exchange protocol. Finally, we implement the protocol using the GAP System for Computational Discrete Algebra.

Contents

Part 1. Cayley-Purser Algorithm

1. Description

The Cayley-Purser algorithm is a public-key cryptographic algorithm developed by Sarah Flannery [4]. The algorithm makes use of the group $GL(2, \mathbf{Z}_n)$, where n is product of two large primes p and q. In particular, p and q are chosen to be safe primes of the form $p = 2p' + 1$ and $q = 2q' + 1$, where p' and q' are primes as well. In this scheme, the message to be communicated is converted to a matrix, encrypted by the sender, and decrypted by the receiver by matrix multiplication with the key (or its inverse).

Receiver:

When the receiver is initiated, he chooses two large safe primes p and q and sets $n = pq$. Next, he randomly chooses two noncommuting elements $A, C \in GL(2, \mathbf{Z}_n)$. Throughout their communication, C is an important matrix that is held secret by the receiver. The following assignments are made as well:

$$B = C^{-1}A^{-1}C$$
$$G = C^r, \text{where } r \text{ is chosen to be a random natural number.}$$

Subsequently, A, B, G, and n are published. Note that C cannot be easily determined from B and G.

Sender:

When the sender is initiated, she uses the aforementioned public parameters to make the following assignments:

$$D = G^s, \text{where } s \text{ is chosen to be a random natural number}$$
$$E = D^{-1}AD$$
$$K = D^{-1}BD.$$

K is used as the enciphering key. A message μ is enciphered as $\mu' = K\mu K$ and μ' and E are sent to the receiver.

For the receiver to decipher this message, he must be able to compute K^{-1}. In fact, only E and C are needed to compute K^{-1} as shown below:

$$\begin{aligned} K^{-1} &= (D^{-1}BD)^{-1} \\ &= D^{-1}B^{-1}D \\ &= D^{-1}(C^{-1}A^{-1}C)^{-1}D \\ &= D^{-1}(C^{-1}AC)D \text{ , } D \text{ and C commute since } D = C^{rs} \\ &= C^{-1}(D^{-1}AD)C \\ &= C^{-1}EC. \end{aligned}$$

Since C and E are both known to the receiver he can easily compute K^{-1} and so compute $K^{-1}\mu' K^{-1} = K^{-1}(K\mu K)K^{-1} = \mu$.

2. Security

2.1. **Overview.** The security of the algorithm was believed to revolve entirely around the difficulty of determining C. With knowledge of C, one could easily compute K^{-1} and decipher the message. In search of this secret parameter, one may either try brute force attacks or try solving for C from one of the public parameters. We will show that both of these methods are infeasible.

2.2. **Brute Force Attacks.** Brute force attacks are virtually useless because of the large order of $GL(2, \mathbf{Z}_n)$

Proposition 2.1. *Let p and q be distinct primes such that $n = pq$. Then, $|GL(2, \mathbb{Z}_n)| = n(\phi(n))^2(p+1)(q+1)$. (here, ϕ is Euler's Phi Function)*

Proof. First, consider the order of $GL(2, \mathbb{Z}_p)$. We have $p^2 - 1$ choices for the first column of an element as we subtract off the column that consists only of zeros. We have $p^2 - p$ choices for the second column. Here, we are subtracting off the p multiples of the first column. Therefore, $|GL(2, \mathbb{Z}_p)| = (p^2 - p)(p^2 - 1)$. Since $GL(2, \mathbb{Z}_n) \cong GL(2, \mathbb{Z}_p) \times GL(2, \mathbb{Z}_q)$ (proved in Appendix A), $|GL(2, \mathbb{Z}_n)| = |GL(2, \mathbb{Z}_p)| \cdot |GL(2, \mathbb{Z}_q)| = (p^2 - p)(p^2 - 1)(q^2 - q)(q^2 - 1) = n(p-1)(q-1)(p-1)(q-1)(p+1)(q+1) = n(\phi(n))^2(p+1)(q+1)$. □

2.3. **Solving for C.**

2.3.1. *Taking the rth-root modulo n.* Solving for C from the public parameters $G = C^r$ and $B = C^{-1}A^{-1}C$ is also shown to be difficult. To solve $G = C^r$ for C, even if r were known, one must take the rth-root modulo n, which is not believed to be possible without knowing $\phi(n) = (p-1)(q-1)$. Since this would require knowledge of p and q, we assume that taking the rth-root module n is at least as difficult as finding the factorization of n, which is known to be infeasible for computers today.

2.3.2. *Elements of Large Order.* Solving $B = C^{-1}A^{-1}C$ for C appears easier. However, [4] shows that the number of solutions to this is equal to the size of the $Cent(A)$. Since an element may commute with all of its powers, we note that $\langle A \rangle \subseteq Cent(A)$ (where $\langle A \rangle$ is the cyclic group generated by A) and therefore that $|Cent(A)| \geq |\langle A \rangle| = o(A)$. Thus by ensuring that $o(A)$ is nearly always large, we can have a level of confidence in the infeasibility of solving for C from the public parameter B. The following explains why [4] A probably has large order.

To do this we define a homomorphism as follows

$$\begin{aligned} \varsigma : GL(2, \mathbf{Z}_n) &\to \mathbf{Z}_n^* \\ A &\mapsto \det(A) \end{aligned}$$

. Then, note that

$$1 = \varsigma(I_{GL(2,\mathbf{Z}_n)}) = \varsigma(A^{o(A)}) = \det(A)^{o(A)}$$

, where $I_{GL(2,\mathbf{Z}_n)}$ denotes the identity matrix in $GL(2, \mathbf{Z}_n)$. Since, $\det(A)^{o(A)} = 1$, $o(\det(A)) \mid o(A)$. This in turn guarantees that the order of $A \in GL(2, \mathbf{Z}_n)$ is at least the order of $\det(A) \in \mathbf{Z}_n^*$. Thus by showing that the order of elements in $\mathbf{Z}_n^*$ is nearly always large, we can show that the same must be true for the elements of $GL(2, \mathbf{Z}_n)$.

To find the orders of the elements of $\mathbf{Z}_n^*$, we consider an isomorphic group $\mathbf{Z}_p^* \times \mathbf{Z}_q^*$ (proved in Appendix B). We note that for $(a, b) \in \mathbf{Z}_p^* \times \mathbf{Z}_q^*$, $o((a, b)) = [o(a), o(b)]$, where the bracket notation denotes the least common multiple. The possible orders of a and of b are determined as follows.

Since $p = 2p' + 1$ and $q = 2q' + 1$, where p' and q' are both odd primes, the order of the groups $\mathbf{Z}_p^*$ and $\mathbf{Z}_q^*$ are $2p'$ and $2q'$, respectively. Then by Sylow theorems, we may determine the order of elements and number of elements of each order for the groups $\mathbf{Z}_p^*$ and $\mathbf{Z}_q^*$.

$\mathbf{Z}_p^*$		$\mathbf{Z}_q^*$	
Order	Number of elements of this order	Order	Number of elements of this order
1	1	1	1
2	1	2	1
p'	$p' - 1$	q'	$q' - 1$
$2p'$	$p' - 1$	$2q'$	$q' - 1$

Recall that for any $x \in \mathbf{Z}_n^*$, $o(x) = [o(a), o(b)]$, where $a = x \mod p \in \mathbf{Z}_p^*$ and $b = x \mod q \in \mathbf{Z}_q^*$. From the previous table we know that $o(a) \in \{1, 2, p', 2p'\}$ and $o(b) \in \{1, 2, q', 2q'\}$. The following table contains all combinations of these orders and their respective least common multiples.

$o(a)$	$o(b)$	$[o(a), o(b)]$
1	1	1
1	2	2
2	1	2
2	2	2
p'	1	p'
$2p'$	1	$2p'$
p'	2	$2p'$
$2p'$	2	$2p'$
1	q'	q'
1	$2q'$	$2q'$
2	q'	$2q'$
2	$2q'$	$2q'$
p'	q'	$p'q'$
$2p'$	q'	$2p'q'$
p'	$2q'$	$2p'q'$
$2p'$	$2q'$	$2p'q'$

If we consider the least common multiples of all 16 combinations of the possible orders of a and b, we find that $[a,b] \in \{1, 2, p', q', 2p', 2q', p'q', 2p'q'\}$. To find out how many elements in $\mathbf{Z}_n^*$ we have of each of these orders, we have to consider the number of different ways we can arrive at a particular order. For example, from the table we can see that if $o(n) = o((a,b)) = 2p'$, then either ($o(a) = 2p'$ and $o(b) = 1$) or ($o(a) = p'$ and $o(b) = 2$) or ($o(a) = 2p'$ and $o(b) = 2$). So there are three possibilities. From the previous table, we know that $o(a) = 2p'$ for $p' - 1$ elements and $o(b) = 1$ for 1 element. This gives $(p' - 1) * 1$ different ways for $o(a) = 2p'$ and $o(b) = 1$. Similarly, $o(a) = p'$ for $p' - 1$ elements and $o(b) = 2$ for 1 element. This also gives $(p' - 1) * 1$ different ways for $o(a) = p'$ and $o(b) = 2$. In the same way, there are $(p' - 1) * 1$ different ways for $o(a) = 2p'$ and $o(b) = 2$. Then in total we have $3(p' - 1) = 3p' - 3$ elements in $\mathbf{Z}_n^*$ or order $2p'$. The following table from [4] has been filled using this method. Thus, when p' and q' are large, most elements of $GL(2, \mathbf{Z}_n)$ will have large order, ensuring that A has large order.

Order	Reason	Number
1	$[1,1] = 1$	1
2	$[1,2] = [2,1] = 2,2] = 2$	3
p'	$[p',1] = p'$	$p' - 1$
q'	$[1,q'] = q'$	$q' - 1$
$2p'$	$[2p',1] = [p',2] = [2p',2] = 2p'$	$3p' - 3$
$2q'$	$[1,2q'] = [2,q'] = [2,2q'] = 2q'$	$3q' - 3$
$p'q'$	$[p',q'] = p'q'$	$p'q' - p' - q' + 1$
$2p'q'$	$[2p',q'] = [p',2q'] = [2p',2q'] = 2p'q'$	$3p'q' - 3p' - 3q' + 3$

2.4. **Cracking the Cayley-Purser Algorithm.** It was believed that the Cayley-Purser algorithm was secure because of the great difficulty in solving the equation $CB = A^{-1}C$ for the secret matrix C given the large order of the centralizer $Cent(A)$ in $GL(2, \mathbf{Z}_n)$. However, in light of the fact that the secret decryption key $L \in GL(2, \mathbf{Z}_n)$ is defined by $L = C^{-1}EC = K^{-1}$, it turns out that any unit scalar multiple $C' \in GL(2, \mathbf{Z}_n)$ of C is enough to form the decryption key L. Let t be any element of $\mathbf{Z}_n^*$, and let $C' = tC$. Then

$$\begin{aligned} C'^{-1}EC' &= (tC)^{-1}E(tC) \\ &= (t^{-1}C^{-1})E(tC) \\ &= C^{-1}EC \\ &= L. \end{aligned}$$

Thus, any multiple C' of C, along with the parameter E, which is sent to the receiver along with the cyphertext μ', can be used to form the decryption key L.

This scalar multiple of C can be obtained in the following way. The public parameter $G \in GL(2, \mathbf{Z}_n)$ is defined by $G = C^r$ for some random $r \in \mathbf{N}$. Let $c_C(x)$ denote the characteristic polynomial of matrix C. By the division algorithm in $\mathbf{Z}_n[x]$, there exist polynomials $q(x), r(x) \in \mathbf{Z}_n[x]$ such that $x^r = c_C(x)q(x) + r(x)$ and $0 \leq deg(r(x)) < deg((c_C(x)) = 2$. We will use the following theorem from linear algebra.

Theorem 2.2 (Cayley-Hamilton)**.** *Let $M \in GL(2, \mathbf{Z}_n)$, and let $c_M(x)$ be the characteristic polynomial of matrix M given by $c_M(x) = |M - xI|$. Then $c_M(M) = 0$.*

Applying the Cayley-Hamilton Theorem, we have

$$\begin{aligned} G &= C^r \\ &= c_C(C)q(C) + r(C) \\ &= r(C) \end{aligned}$$

Since $0 \leq deg(r(x)) < 2$, $r(x)$ is at most a linear polynomial in $\mathbf{Z}_n[x]$, there exist $\alpha, \beta \in \mathbf{Z}_n$ such that $r(x) = \alpha x + \beta$. Then $G = r(C) = \alpha C + \beta I$. If we can show that $\alpha \in \mathbf{Z}_n^*$ then we can solve the equation

$$G = \alpha C + \beta I$$

for the secret matrix C written as a linear combination of the of the public matrix G and the identity matrix I, giving us

$$C = \alpha^{-1} G - \alpha^{-1} \beta I.$$

As part of an investigation into the security flaws of the Cayley-Purser algorithm, Flannery [4] showed $\alpha \in \mathbf{Z}_n^*$ in most cases. In those rare instances when α is not a unit in $\mathbf{Z}_n$, Flannery showed that the secret factors p and q of n are revealed. In either case, the Cayley-Purser algorithm is cracked.

3. Implementation

3.1. **Creating a Test Environment.** We implemented the Cayley-Purser algorithm in GAP. To simulate having public and private information, we used a permission based directory structure. After using the algorithm, the relevant portions of the file system will look like this:

/cp:

permissions	owner	group	file	description
-rw-r–r–	root	users	encodecode.g	for converting text to a matrix
-rwxr——	receiver	users	public.key	created by ./receiver/init.g; contains A, B, G, n
-rwxr——	sender	users	public.msg	created by ./sender/encipher.g; contains mu', E
drwx——	receiver	users	receiver	
drwx——	sender	users	sender	

/cp/receiver:

permissions	owner	group	file	description
-rwx——	receiver	users	decipher.g	
-rwx——	receiver	users	init.g	
-rwx——	receiver	users	secret.key	created by ./init.g; contains C

/cp/sender:

-rwx——	sender	users	encipher.g	
-rwx——	sender	users	init.g	
-rwx——	sender	users	message.txt	a plain-text message
-rwx——	sender	users	secret.key	created by ./init.g; contains Ee, K

3.2. **Code. /cp/receiver/init.g:**

```
safePrimes :=[
  49943, 50147, 50459, 51047, 51203, 51287, 51347, 51407, 51599, 51683,
  51827, 51839, 52223, 52379, 52919, 53003, 53147, 53267, 53699, 53759,
  53783, 53987, 54287, 54563, 54959, 55079, 55103, 55163, 55487, 55547,
  55619, 55787, 55967, 56003, 56039, 56807, 56999, 57119, 57143, 57287,
  57503, 57587, 57719, 57803, 57899, 57923, 58043, 58067, 58403, 58679,
  58727, 58907, 58967, 59063, 59447, 59747, 60539, 60647, 60779, 60899,
  61343, 61379, 61547, 61703, 61967, 62039, 62303, 62507, 62639, 62939,
  63299, 63443, 63587, 63599, 63719, 64007, 64019, 64283, 64319, 64763,
  65063, 65123, 65147, 65267, 65543, 65579, 65687, 65867, 66047, 66107,
  66239, 66359, 66383, 66923, 66959, 67043, 67139, 67247, 67427, 67499,
  67547, 67619, 67883, 68507, 68567, 68639, 68879, 69263, 69767, 69827,
  69899, 70139, 70163, 70199, 70223, 70583, 71147, 71663, 71867, 71987,
  71999, 72167, 72383, 72503, 72707, 72767, 72959, 73127, 73259, 73523,
  73583, 73643, 73847, 73859, 74027, 74099, 74279, 74363, 74507, 74759,
  75239, 75707, 75743, 75983, 76079, 76367, 76379, 76403, 76463, 76607,
  76667, 76907, 76919, 77003, 77279, 77339, 77447, 77723, 77747, 77783,
  77867, 78179, 78467, 78479, 78839, 78887, 79043, 79103, 79139, 79319,
  79559, 79907, 79943, 79967, 79979, 80387 ];;


    p := Random(safePrimes);;
    q := Random(safePrimes);;
    n := p*q;;
    g := GL(2, ZmodnZ(n));;
    #Print("p=",p," q=", q, " pq=", n, " size(g)=", Size(g),"\n");;
    repeat
         C := PseudoRandom(g);;
         A := PseudoRandom(g);;
    until C*A<>A*C;;
    r := 0;;
    #Print("Computing r...\n");;
    digits := Reversed([0..LogInt(Size(g)/2,10)-1]);;
    for i in digits do
         r := r + Random([0..9])*10^i;;
     od;;
    B := C^-1*A^-1*C;;
    Print("C^", r, ":\n");;
    G := C^r;;

    OutputLogTo("/cp/receiver/secret.key");;
    Print("C:=", C, ";\n");;
    OutputLogTo();;

    OutputLogTo("/cp/public.key");;
    Print("A:=", A, ";\n");;
```

```
     Print("B:=", B, ";\n");;
     Print("G:=", G, ";\n");;
     OutputLogTo();;
quit;;
```

/cp/receiver/decipher.g:

```
Read("/cp/receiver/secret.key");;
Read("/cp/public.msg");;
Read("/cp/encodecode.g");


L := C^-1*Ee*C;;
mu := L*mup*L;;
lis := [mu[1][1], mu[1][2], mu[2][1], mu[2][2]];
Print(decode(lis));;

quit;;
```

/cp/sender/init.g:

```
Read("/cp/public.key");
s := Random([1..1000]);;
D := G^s;;
Ee := D^-1*A*D;;
K := D^-1*B*D;;

OutputLogTo("/cp/sender/secret.key");
Print("Ee := ", Ee, ";;\n");
Print("K := ", K, ";;\n");
OutputLogTo();
```

/cp/sender/encipher.g:

```
Read("/cp/sender/secret.key");;
Read("/cp/encodecode.g");
input := InputTextFile("/cp/sender/message.txt");
txt := ReadAll(input);
txt := txt{[1..Size(txt)-1]};
data := encode(txt, ModulusOfZmodnZObj(K[1][1]));
mu := [[0,0], [0,0]];

#Note: Only takes as much of the message as can fit in our matrix
for i in [1..2] do
     for j in [1..2] do
          if 2*(i-1)+j<=Size(data) then
               mu[i][j] := data[2*(i-1)+j];;
          else
               mu[i][j] := 0;;
          fi;
     od;
od;
```

```
Print("Computing mu'...");
mu := K^0*mu;;
mup := K*mu*K;;
Print("done.\n");
OutputLogTo("/cp/public.msg");;
Print("mup := ", mup, ";\n");
Print("Ee := ", Ee, ";\n");
OutputLogTo();
quit;
```

Part 2. A Generic Method Using Groups for Key Establishment

[1] defines a generic method for using groups to establish a common secret key between two parties. In fact, the method is generic enough even to work with monoids. However, we will only be interested in groups.

4. Generic Requirements for a Protocol

The protocol for establishing a shared key requires that for two groups U, V, we define the following:

$$\beta : U \times U \to V \gamma_1 : U \times V \to V \gamma_2 : U \times V \to V$$

such that:
i)

$$\forall x, y_1, y_2 \in U, \beta(x, y_1 \cdot y_2) = \beta(x, y_1) \cdot \beta(x, y_2).$$

ii)

$$\forall x, y \in U, \gamma_1(x, \beta(y, x)) = \gamma_2(y, \beta(x, y)).$$

iii) It is generally infeasible to determine a secret element x given that $y_1, ..., y_k$ and $\beta(x, y_1), ..., \beta(x, y_k)$ are publicly known (where $y_1, ..., y_k \in U$).

5. The Protocol

First, we publicly assign person A and person B subgroups of U. Let S_A and T_B be subgroups of U generated by $s_1, ..., s_m$ and $t_1, ..., t_n$ respectively.

Next, person A chooses a secret element $a \in S_A$ and sends $\{\beta(a, t_1), ..., \beta(a, t_n)\}$ to person A. Person B similarly chooses a secret element $b \in T_B$ and sends $\{\beta(b, s_1), ..., \beta(b, s_m)\}$ to person A. With the information sent by person B, person A can compute $\beta(b, a)$. For example, suppose $a = s_1 s_4 s_2$. Then, by the properties of β, $\beta(b, s_1)\beta(b, s_4)\beta(b, s_2) = \beta(b, s_1 s_4 s_2) = \beta(b, a)$. With $\beta(b, a)$, person A can also compute $\kappa := \gamma(a, \beta(b, a))$.

In a similar manner, B can compute $\beta(a, b)$, with which he can also compute $\gamma_2(b, \beta(a, b))$. By the properties of γ_1 and γ_2, $\gamma_2(b, \beta(a, b)) = \gamma_1(b, \beta(b, a)) = \kappa$. Thus, κ becomes the established key.

6. An Example

In their paper [1], Anshel et al. provide an example of a specific $(\beta, \gamma_1, \gamma_2)$ that works for any group $U = V$ where it is infeasible to solve the conjugacy problem.

In particular, we define them as follows:

$$\beta(x, y) = x^{-1}yx$$
$$\gamma_1(u, v) = u^{-1}v$$
$$\gamma_2(u, v) = v^{-1}u$$

Then, it follows that $\beta(x, y_1)\beta(x, y_2) = (x^{-1}y_1x)(x^{-1}y_2x) = x^{-1}y_1(xx^{-1})y_2x = x^{-1}(y_1y_2)x = \beta(x, y_1y_2)$. Likewise, $\gamma_1(x, \beta(y, x)) = x^{-1}(y^{-1}xy) = (x^{-1}yx)^{-1}y = (beta(x, y))^{-1}y = \gamma_2(y, \beta(x, y))$.

7. Choosing a Group

The idea is to choose a group that is complicated enough. Near the end of their paper, Anshel et al. suggest looking into the braid group. In the next part, we will look into using the braid group for key establishment.

Part 3. Braid Group

8. A Finite Presentation

The braid group B_n is defined by the following.

$$B_n = \langle \sigma_1, \sigma_2, ..., \sigma_{n-1} | \sigma_i\sigma_j = \sigma_j\sigma_i \text{ for } |i - j| > 1, \sigma_i\sigma_j\sigma_i = \sigma_j\sigma_i\sigma_j \text{ for } |i - j| = 1 \rangle$$

This presentation is motivated from the crossings or braids one can make with a set of n strands. In particular, σ_i signifies the crossing of strands i and $i + 1$ such that the ith strand goes beneath $(i + 1)$th strand, and σ_i^{-1} signifies the crossing of the same strands such that the $(i + 1)$th strand goes beneath the ith strand. The identity element signifies making no crossings. B_n is of infinite order since one can take two strands and continue crossing them infinitely many times without ever returning to the identity.

9. Key Establishment

9.1. **Anshel, Anshel, and Goldfeld.** One method we can use for key establishment in the braid group is the protocol developed by Anshel et al. mentioned in the previous part.

9.2. **An Alternate Method.**

9.2.1. *Overview.* An alternate method was proposed in [5]. In this scheme, we choose some l and r and take the subgroups LB_l and RB_r of $B_{(l+r)}$, where LB_l is generated by $\{\sigma_1, ..\sigma_{(l-1)}\}$ and RB_r is generated by $\{\sigma_{(l+1)}, ..., \sigma_{(r+l-1)}\}$. Note that LB_l and RB_r both do not affect the lth strand. Algebraically, this means that $\forall\sigma_i, \sigma_j(\sigma_i \in B_l, \sigma_j \in B_r), |i - j| > 1 \Rightarrow \sigma_i$ and σ_j commute.

9.2.2. *Protocol.* After choosing an l and r, the next step is to choose a public $x \in B_{(l+r)}$. (There are criterion mentioned in [5] that ensure that x is a good choice, but they are beyond the scope of this paper.)

After this, person A chooses a secret $a \in LB_l$ and sends $y_1 = axa^{-1}$ to person B. Person B, in the same way, chooses a secret $b \in RB_r$ and sends $y_2 = bxb^{-1}$ to person A.

Now person A can compute $\kappa = ay_2a^{-1}$. Person B can likewise compute by_1b^{-1}. Since a and b commute, $by_1b^{-1} = b(axa^{-1})b^{-1} = a(bxb^{-1})a^{-1} = ay_2a^{-1} = \kappa$. And so κ becomes the shared key.

10. Word Reduction

10.1. **Motivation.** Word reduction refers to the rewriting of a word in a simpler way. These algorithms are critical for hiding information. For example, suppose we have secret $a = \sigma_1\sigma_3^{-1}\sigma_2$ and public $x = \sigma_5\sigma_4\sigma_3\sigma_2^{-1}$. If the protocol requires us to broadcast $a^{-1}xa$, it is obvious that sending $\sigma_2^{-1}\sigma_3\sigma_1^{-1}\sigma_5\sigma_4\sigma_3\sigma_2^{-1}\sigma_1\sigma_3^{-1}\sigma_2$ (the plain concatenation) does not conceal a. However, it is difficult to find a from the equivalent word $\sigma_3\sigma_2\sigma_5\sigma_4\sigma_3\sigma_4^{-1}\sigma_2\sigma_1^{-1}\sigma_2^{-1}\sigma_3^{-1}$.

It is possible to reduce words in such a way that any two equivalent braids reduce to the same braid word. This braid word is called the braid's *canonical form*. Consider the previous two protocols. Once two parties have established a key $\kappa \in B_n$, the next step would be to somehow encrypt and decrypt messages. We typically do not want to use κ directly to encrypt or decrypt our message, since this would require a one-to-one mapping that converts each message into a braid. And so it becomes desirable to use κ to establish an encryption/decryption key that can directly act upon the message. A common solution to this is to consider κ no longer as an element of B_n but as some word, that is, as a string of "letters". We can then map this string into a sequence of bits for example (or whatever is useful for encryption/decryption) by applying some hash function to the string. While this is a good solution, it opens up a particular problem. While the aforementioned methods assure that both parties will share knowledge of the element κ, there is no guarantee that they will be represented by the same words on both sides. Since we convert κ to something useful for encryption/decryption based upon its word representation, both parties may end up with two different encryption keys. This motivates defining a canonical form for braid words such that any braid has exactly one braid word of canonical form.

10.2. **A Canonical Form.** [5] mentions a canonical form, which we will briefly examine. To understand the form, we must first loosely define a *permutation braid* and the *fundamental braid*.

Definition 10.1. *A permutation braid is a braid such that all crossings are positive and no two strands cross more than once.*

A permutation braid is the unique representation in the braid group of a given permutation. Intuitively, one can find the permutation braid for a given permutation π by take each ith strand (one at a time from left to right) and connecting it directly to the i^πth position.

Definition 10.2. *The fundamental braid is the permutation braid that corresponds to the permutation* $(1,n)(2,n-1)...(\lfloor \frac{n}{2} \rfloor, \lceil \frac{n}{2} \rceil + 1)$.

By Theorem 2.9 in [3], any braid can be uniquely represented by $(u, \pi_1, \pi_2, ..., \pi_p)$ where u is an exponent applied to the fundamental braid and π_1,...,π_p are permutation braids that are multiplied with the uth power of the fundamental braid.

In this paper, we will not consider algorithms that reduce words to this canonical form.

10.3. **Dehornoy's Reduction.** Dehornoy's reduction (as see in [2]) does *not* guarantee that any two braid words that represent the same element will be reduced to the same braid word. As such, it does not consider the notion of a canonical form. It does, however, guarantee that any word representing the identity element will be reduced to the empty string (making it easy to determine if two words are equivalent). Yet the reason it is especially useful in key establishment is because Dehornoy's algorithm allows that any sufficiently complicated word can be rewritten quickly; whereas, reducing a word to canonical form is known to be slow in general. Recall that rewriting a word is useful in concealing its original composition.

10.4. **Dehornoy's Algorithm.** We will now explain Dehornoy's method for reducing braid words. So we will begin by giving a definition of a *reduced* braid word.

Definition 10.3. *The braid word w is reduced either if w is the nullstring, or if the main generator of w, defined as the generator with lower index occurring in w, occurs only positively or negatively.*

For example $\sigma_2\sigma_3\sigma_4^{-1}\sigma_2$ is a reduced braid word. On the other hand, $\sigma_2\sigma_3\sigma_4^{-1}\sigma_3\sigma_2^{-1}$ is not reduced. We will now need to introduce the idea of a *handle* in order to continue with our discussion of reduction of braid words.

Definition 10.4. *A σ_j-handle is a braid word of the form $\sigma_j^e v \sigma_j^{-e}$, where $e = \pm$ and the word v contains only generators $\sigma_k^{\pm 1}$ with $k < j-1$ or $k > j$. A main handle of w is a subword of w that is a σ_i-handle, where σ_i is the main generator of w.*

For example, the braid word $\sigma_3\sigma_1\sigma_4^{-1}\sigma_3^{-1}$ is a $\sigma_3 - handle$. Now we will give a definition of a *permitted* $\sigma_j - handle$.

Definition 10.5. *A $\sigma_j - handle$ $\sigma_j^e v \sigma_j^{-e}$ is permitted if it includes no $\sigma_{j+1} - handle$.*

The braid word $\sigma_3\sigma_4\sigma_3^{-1}$ is a *permitted* $\sigma_3 - handle$. However, the $\sigma_2 - handle$ $\sigma_2\sigma_3\sigma_4\sigma_3^{-1}\sigma_2^{-1}$ is not *permitted*

We are now in a position to present a method for reduction of braid words.

We can rewrite subwords of the form $\sigma_i^{\pm 1}\sigma_j^{\pm 1}$ when $|i-j| = 1$ using the relation $\sigma_i\sigma_j\sigma_i = \sigma_j\sigma_i\sigma_j$. Each of the identities in the following table can be obtained by manipulating these relations when $|i-j| = 1$.

Equivalent words for $\sigma_i^{\pm 1}\sigma_j^{\mp 1}$ when $\lvert i-j\rvert = 1$	
$\sigma_i^{\pm 1}\sigma_j^{\mp 1}$	Equivalent word
$\sigma_i^{-1}\sigma_j$	$\sigma_j\sigma_i\sigma_j^{-1}\sigma_i^{-1}$
$\sigma_i\sigma_j^{-1}$	$\sigma_j^{-1}\sigma_i^{-1}\sigma_j\sigma_i$

Theorem 10.6. *Let*

$$h := \sigma_j^e v_0 \sigma_{j+1}^d v_1 \sigma_{j+1}^d v_2 \cdots v_{m-1}\sigma_{j+1}^d v_m \sigma_j^{-e}$$

with $d, e \in \{-1, +1\}$ be a permitted σ_j-handle. Then the word

$$h' = v_0\sigma_{j+1}^{-e}\sigma_j^d\sigma_{j+1}^e v_1\sigma_{j+1}^{-e}\sigma_j^d\sigma_{j+1}^e v_2 \cdots v_{m-1}\sigma_{j+1}^{-e}\sigma_j^d\sigma_{j+1}^e v_m$$

obtained by the following map

$$\phi_{j,e} : \begin{cases} \sigma_j^{\pm} \mapsto \epsilon \\ \sigma_{j+1}^{\pm} \mapsto \sigma_{j+1}^{-e}\sigma_j^d\sigma_{j+1}^e \\ \sigma_k^{\pm} \mapsto \sigma_k^{\pm} \text{ for } k \neq j, j+1 \end{cases}$$

is equivalent to h.

Proof. We will begin in the case that $e = +1$ and $d = -1$. Now, v_0 contains $\sigma_k^{\pm}$ where $k \leq j-2$ and $k \geq j+2$. It follows that v_0 commutes with σ_j. So we can replace $\sigma_j v_0$ with $v_0\sigma_j$ in our expression for h. So we can rewrite h as the following:

$$v_0\sigma_j\sigma_{j+1}^{-1}v_1\sigma_{j+1}^{-1}\cdots v_{m-1}\sigma_{j+1}^{-1}v_m\sigma_j^{-1}$$

Using the table above, we can rewrite $\sigma_j\sigma_{j+1}^{-1}$ by its equivalent subword $\sigma_{j+1}^{-1}\sigma_j^{-1}\sigma_{j+1}\sigma_j$. So we can rewrite the above expression as:

$$v_0\sigma_{j+1}^{-1}\sigma_j^{-1}\sigma_{j+1}\sigma_j v_1\sigma_{j+1}^{-1}\cdots v_{m-1}\sigma_{j+1}^{-1}v_m\sigma_j^{-1}$$

So what we have essentially done is replaced σ_{j+1}^{-1} with $\sigma_{j+1}^{-1}\sigma_j^{-1}\sigma_{j+1}$ and moved σ_j to the right, next to v_1. In a similar fashion to the situation we had before, v_1 contains only $\sigma_k^{\pm}$ where $k \leq j-2$ and $k \geq j+2$. It follows that v_1 commutes with σ_j. So we can rewrite our above expression as:

$$v_0\sigma_{j+1}^{-1}\sigma_j^{-1}\sigma_{j+1}v_1\sigma_j\sigma_{j+1}^{-1}\cdots v_{m-1}\sigma_{j+1}^{-1}v_m\sigma_j^{-1}$$

Once again, we have an instance of $\sigma_j\sigma_{j+1}^{-1}$ in our expression. So we can apply the same rewriting process as before. Repeating this process of commuting σ_j with the v_i elements and replacing instances of $\sigma_j\sigma_{j+1}^{-1}$ with $\sigma_{j+1}^{-1}\sigma_j^{-1}\sigma_{j+1}\sigma_j$ moves σ_j through the word to the right, and eventually we are left with the following:

$$v_0\sigma_{j+1}^{-1}\sigma_j^{-1}\sigma_{j+1}v_1\sigma_{j+1}^{-1}\sigma_j^{-1}\sigma_{j+1}v_2\cdots v_{m-1}\sigma_{j+1}^{-1}\sigma_j\sigma_j^{-1}$$

Since $\sigma_j\sigma_j^{-1} = \epsilon$, we have reduced our handle to

$$v_0\sigma_{j+1}^{-1}\sigma_j^{-1}\sigma_{j+1}v_1\sigma_{j+1}^{-1}\sigma_j^{-1}\sigma_{j+1}v_2\cdots v_{m-1}\sigma_{j+1}^{-1}$$

In the case that $e = -1$ and $d = 1$ we apply a similar process as above. The only difference in the process is that we replace subwords by using the identity $\sigma_i^{-1}\sigma_j = \sigma_j\sigma_i\sigma_j^{-1}\sigma_i^{-1}$. In the case that $e = d = \pm 1$, we use essentially the same process as above (but starting from the right rather than the left) to move the final generator σ_j^{-e} in the word to the left. In the end, these processes for reduction can be realized to be the same as sending σ_j to ϵ, σ_{j+1} to $\sigma_{j+1}^{-1}\sigma_j^{-1}\sigma_{j+1}$, and v_i to v_i. □

11. Implementation

11.1. **Description.** To find σ_j-handles, our program uses a hash table, where the index and sign of a generator serve as its key and a stack of the positions of its occurrences within the braid word becomes its value. The size of the hash table is $2n + 1$, where n represents the number of generators in the group. The hashing function is defined by $f(\sigma_i^e) = e \cdot i + (n + 1)$, where i is the index of the generator and $e = \pm 1$.

We define the *p-left-closest* σ_j to be the right-most instance of some generator σ_j in the word such that its position within the word is less than a position p.

As the program scans through the braid word (a "concatenation" of generators and their inverses) from left to right, it first checks for free reductions by comparing the generator at the current position (we call it σ_j) with the previous generator. If a free reduction can be done, the two generators are removed from the word.

If not, it takes the current position p and pushes it on to σ_j's stack within the hash table. It then checks the stacks of σ_j^{-1}, σ_{j-1}, and σ_{j-1}^{-1} to see if there is an opportunity for a handle. For there to be a handle, σ_j^{-1}'s stack must be nonempty and the p-left-closest σ_{j-1} or σ_{j-1}^{-1} must not lay between the current position and the p-left-closest σ_j^{-1}. Note that the positions for the p-left-closest σ_j^{-1}, σ_{j-1}, and σ_{j-1}^{-1} can be found at the top of their respective stacks in the hash table.

Once a handle is found, it is passed to the *reduceHandle* function which simply reduces the handle according to the algorithm and returns the reduced handle.

11.2. **Code.**

```
reducedHandle := function(handlex)
     local handle, j, e, pos;
     handle := ShallowCopy(handlex);

     j := AbsoluteValue(handle[1]);
     e := handle[1]/j; #sign of j
     Print("handle: ", handle, " replaced with ");
     handle := handle{[2..Size(handle)-1]}; #trim j and -j from the ends
     pos := 1;
     while(pos<=Size(handle)) do
          if handle[pos] = j+1 or handle[pos] = -(j+1) then
               handle := Concatenation(
                         handle{[1..pos-1]},
                         [-e*(j+1), (handle[pos]/(j+1))*j, e*(j+1)],
                         handle{[pos+1..Size(handle)]}
                    );
               pos := pos + 2;
          fi;
          pos := pos + 1;
     od;
     Print(handle, "\n");
     return handle;
end;
```

```
reducedWord := function(wordx, n)
     local word, pos, gen, genIndex, prevGen, hash, genInvPos, genInvIndex, jMinusOneIndex, j

     ##convert string to list of integers
     word := List(wordx, i -> INT_CHAR(i));
     for pos in [1..Size(word)] do
          if word[pos]<97 then
               word[pos] := -(word[pos] - 64); #capital letters are inverses
          else
               word[pos] := word[pos] - 96;
          fi;
     od;
     ##

     hash := [1..(2*n+1)];
     hash := List(hash, i -> i*[]);
     pos := 1;
     prevGen := 0;

     while(pos<=Size(word)) do

          gen := word[pos];

          ##Free reductions
          if gen = -prevGen then
               Print("Free reduction at ", pos, "\n");
               Remove(word, pos);
               Remove(word, pos-1);
               pos := pos - 1;
               if pos>1 then
                    prevGen := word[pos - 1];
               else
                    prevGen := 0;
               fi;
               continue;
          fi;
          ##

          prevGen := gen;

          genIndex := gen + (n+1);
          genInvIndex := -gen + (n+1);
          jMinusOneIndex := AbsoluteValue(gen)-1 + (n+1);
          jMinusOneInvIndex := 1-AbsoluteValue(gen) + (n+1);

          hash[genIndex] := Concatenation([pos], hash[genIndex]);
```

```
##before reading from the hash, clean out information that is no longer accurate
###clean up gen^-1
hash[genInvIndex] := Filtered(
     hash[genInvIndex],
     i -> i<pos and word[i] = -gen
);
###
###clean up j-1
hash[jMinusOneIndex] := Filtered(
     hash[jMinusOneIndex],
     i -> i<pos and word[i] = AbsoluteValue(gen)-1
);
###
###clean up -(j-1)
hash[jMinusOneInvIndex] := Filtered(
     hash[jMinusOneInvIndex],
     i -> i<pos and word[i] = 1-AbsoluteValue(gen)
);
###


genInvPos := First(hash[genInvIndex], i -> true);

if (
     not genInvPos = fail and
     (
          AbsoluteValue(gen) = 1 or
           (
               not (
                     First(hash[jMinusOneIndex], i->true) > genInvPos and
                     First(hash[jMinusOneIndex], i->true) < pos
               ) and
               not (
                    First(hash[jMinusOneInvIndex], i->true) > genInvPos and
                    First(hash[jMinusOneInvIndex], i->true) < pos
               )
          )
     )
) then
     word := Concatenation(
               word{[1..genInvPos-1]},
               reducedHandle(word{[genInvPos..pos]}),
                word{[pos+1..Size(word)]}
          );
     pos := genInvPos-1;
     if pos>1 then
          prevGen := word[pos];
     else
```

```
                    prevGen := 0;
               fi;
          fi;

          pos := pos + 1;
     od;

     ##convert new list of integers back to string
     for pos in [1..Size(word)] do
          if word[pos]<0 then
               word[pos] := -word[pos] + 64;
          else
               word[pos] := word[pos] + 96;
          fi;
     od;
     word := List(word, i -> CHAR_INT(i));
     ##
     return word;
end;
```

Part 4. Appendix

12. Appendix A

Proposition 12.1. $GL(2, \mathbf{Z}_n) \cong GL(2, \mathbf{Z}_p) \times GL(2, \mathbf{Z}_q)$

Notation:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}_m := \begin{bmatrix} a \mod m & b \mod m \\ c \mod m & d \mod m \end{bmatrix}$$

To compute the order of the group, we will first show that $GL(2, \mathbf{Z}_n) \cong GL(2, \mathbf{Z}_p) \times GL(2, \mathbf{Z}_q)$ by defining a map:

$$\varphi : GL(2, \mathbf{Z}_n) \to GL(2, \mathbf{Z}_p) \times GL(2, \mathbf{Z}_q)$$
$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}_n \mapsto \left(\begin{bmatrix} a & b \\ c & d \end{bmatrix}_p, \begin{bmatrix} a & b \\ c & d \end{bmatrix}_q \right)$$

We first show that the image of this map is in fact a subset of $GL(2, \mathbf{Z}_p) \times GL(2, \mathbf{Z}_q)$ and that it is well-defined.

Let $M = \begin{bmatrix} a & b \\ c & d \end{bmatrix}_n$ be an arbitrary element of $GL(2, \mathbf{Z}_n)$. Since M is invertible, $det(M) = ad - bc \in \mathbf{Z}_n^*$. Then, $(ad - bc, n) = 1$. As $n = pq$ for primes p and q, we conclude that $(ad - bc, p) = (ad - bc, q) = 1$, so that $ad - bc \in \mathbf{Z}_p^*$ and $ad - bc \in \mathbf{Z}_q^*$. Thus, $det(M) \in \mathbf{Z}_p^*$ and $det(M) \in \mathbf{Z}_q^*$. Consequently, $Im\ \varphi \subseteq GL(2, \mathbf{Z}_p) \times GL(2, \mathbf{Z}_q)$.

To show that φ is well-defined, suppose $M' = \begin{bmatrix} s & t \\ u & v \end{bmatrix}_n$ is another arbitrary element of $GL(2, \mathbf{Z}_n)$ such that $M_{ij} \equiv M'_{ij} \pmod{n}$ for all $i, j \in \{1, 2\}$. Then n divides $M_{ij} - M'_{ij}$. Since both p and q divide n, we have $M_{ij} \equiv M'_{ij} \pmod{p}$ and $M_{ij} \equiv M'_{ij} \pmod{q}$ for all $i, j \in \{1, 2\}$. Thus $\varphi(M) = \varphi(M')$, which proves that φ is well-defined.

The following shows that φ is a homomorphism:

Let $A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}_n, B = \begin{bmatrix} e & f \\ g & h \end{bmatrix}_n \in GL(2, \mathbb{Z}_{pq})$. Using modular arithmetic, we have the following:

$$\varphi(AB) = \varphi(\begin{bmatrix} ae+bg & af+bh \\ ce+dg & cf+dh \end{bmatrix}_n) = (\begin{bmatrix} ae+bg & af+bh \\ ce+dg & cf+dh \end{bmatrix}_p, \begin{bmatrix} ae+bg & af+bh \\ ce+dg & cf+dh \end{bmatrix}_q)\varphi(A)\varphi(B) = (\begin{bmatrix} a & b \\ c & d \end{bmatrix}_p, \begin{bmatrix} a & b$$

To show that $GL(2, \mathbf{Z}_n)$ and $GL(2, \mathbf{Z}_p) \times GL(2, \mathbf{Z}_q)$ are isomorphic, we must show that φ is bijective. To do this, we first show that φ has a trivial kernel, which guarantees its being injective.

Since

$$ker\,(\varphi) = \{\begin{bmatrix} a & b \\ c & d \end{bmatrix}_n \in GL(2, \mathbf{Z}_n) | \varphi\left(\begin{bmatrix} a & b \\ c & d \end{bmatrix}_n\right) = \left(\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}_p, \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}_q\right)\}$$

all matrices in the kernel, must satisfy the following congruences:

$$a \equiv 1(\mod p)$$
$$a \equiv 1(\mod q)$$

$$b \equiv 0(\mod p)$$
$$b \equiv 0(\mod q)$$

$$c \equiv 0(\mod p)$$
$$c \equiv 0(\mod q)$$

$$d \equiv 1(\mod p)$$
$$d \equiv 1(\mod q)$$

, where $a, b, c, d \in \mathbf{Z}$.

Since the p and q are coprime, we can apply the Chinese Remainder Theorem to each pair of congruences. The theorem not only guarantees that solutions exist for each pair but also that all the solutions of a given pair are congruent modulo n–that is, each pair of congruences yields a unique solution in $\mathbf{Z}_n$. The four unique solutions a, b, c, and d in turn provide a unique matrix $\begin{bmatrix} a & b \\ c & d \end{bmatrix}_n$, which becomes the only matrix in the kernel. In particular, only the entries of the identity matrix in $GL(2, \mathbf{Z}_n)$ satisfy these congruences. φ is therefore injective.

To show that φ is onto, we must find an element $\begin{bmatrix} a & b \\ c & d \end{bmatrix}_n \in GL(2, \mathbf{Z}_n)$ for any element in the image of the form $\left(\begin{bmatrix} s & t \\ u & v \end{bmatrix}_p, \begin{bmatrix} w & x \\ y & z \end{bmatrix}_q\right)$, where $\begin{bmatrix} s & t \\ u & v \end{bmatrix}_p \in GL(2, \mathbf{Z}_p)$ and $\begin{bmatrix} w & x \\ y & z \end{bmatrix}_q \in GL(2, \mathbf{Z}_q)$.

We note that $\varphi\left(\begin{bmatrix} a & b \\ c & d \end{bmatrix}_n\right) \mapsto \left(\begin{bmatrix} s & t \\ u & v \end{bmatrix}_p, \begin{bmatrix} w & x \\ y & z \end{bmatrix}_q\right)$ iff the following pairs of congruences hold:

$$\begin{aligned} a &\equiv s \mod p \\ a &\equiv w \mod q \end{aligned}$$

$$\begin{aligned} b &\equiv t \mod p \\ b &\equiv x \mod q \end{aligned}$$

$$\begin{aligned} c &\equiv u \mod p \\ c &\equiv y \mod q \end{aligned}$$

$$\begin{aligned} d &\equiv v \mod p \\ d &\equiv z \mod q. \end{aligned}$$

The Chinese Remainder theorem guarantees that each pair of these congruences has a solution. This means that for any element $\left(\begin{bmatrix} s & t \\ u & v \end{bmatrix}_p, \begin{bmatrix} w & x \\ y & z \end{bmatrix}_q\right)$ in the image, we can find an element $\begin{bmatrix} a & b \\ c & d \end{bmatrix}_n$ in the preimage such that $\varphi\left(\begin{bmatrix} a & b \\ c & d \end{bmatrix}_n\right) = \left(\begin{bmatrix} s & t \\ u & v \end{bmatrix}_p, \begin{bmatrix} w & x \\ y & z \end{bmatrix}_q\right)$. Therefore φ is surjective.
□

13. Appendix B

Proposition 13.1. *Let $n = pq$ where p and q are distinct primes. Then,*

$$\mathbb{Z}_n^* \cong \mathbb{Z}_p^* \times \mathbb{Z}_q^*$$

Proof. Consider the map $\phi\colon \mathbb{Z}_n^* \to \mathbb{Z}_p^* \times \mathbb{Z}_q^*$ given by $a \mapsto (a \mod p, a \mod q)$.

Well-Defined:
Let $a, b \in Z_n^*$ such that $a \equiv b \pmod{n}$. Then $n|(a-b)$. Since p and q both divide n, it follows that $p|(a-b)$ and $q|(a-b)$. Consequently, $a \equiv b \pmod{p}$ and $a \equiv b \pmod{q}$. Then ϕ is well-defined.

Operation-Preserving:
Let $a, b \in \mathbb{Z}_n^*$. Then $\phi(ab) = (ab \mod p, ab \mod q) = ((a \mod p)(b \mod p) \mod p, (a \mod q)(b \mod q) \mod q)$.
Alternatively $phi(a)\phi(b) = (a \mod p, a \mod q)(b \mod p, b \mod q) = ((a \mod p)(b \mod p) \mod p, (a \mod q)(b \mod q) \mod q) = \phi(ab)$.

Onto:
Let $(x, y) \in \mathbb{Z}_p^* \times \mathbb{Z}_q^*$. By the Chinese Remainder Theorem, there exists a solution a such that $0 < a < n$ and $a \equiv x \mod p, a \equiv y \mod q$. Now, $x \neq 0$ and $y \neq 0$, so $p \nmid a$ and $q \nmid a$. Therefore, $pq \nmid a$. So, $gcd(a, n) = 1$, and we have $a \in \mathbb{Z}_n^*$ and

$a \mapsto (x, y)$.

One-to-One:
Let $a, b \in \mathbb{Z}_n^*$ such that $(a \mod p, a \mod q) = (b \mod p, b \mod q)$. So we have $a \mod p = b \mod p$ and $a \mod q = b \mod q$. This implies that $p|(a-b)$ and $q|(a-b)$. So we have $a - b = pk_1, a - b = qk_2$(for some$k_1, k_2 \in \mathbb{Z}$). From this, we have $pk_1 = qk_2$. This implies that $p|qk_2$. Now $p \nmid q$, so $p|k_2$ and we have $k_2 = pk_3$. From this, we see that $a - b = qpk_3 = nk_3$. Therefore, $n|(a-b)$. Hence, $a \equiv b \mod n$. □

References

[1] Iris Anshel, Michael Anshel, and Dorian Goldfeld. An algebraic method for public-key cryptography. *Math. Res. Lett.*, 6(3-4):287–291, 1999.

[2] Patrick Dehornoy. A fast method for comparing braids. *Adv. Math.*, 125(2):200–235, 1997.

[3] Elsayed A. El-Rifai and H. R. Morton. Algorithms for positive braids. *Quart. J. Math. Oxford Ser. (2)*, 45(180):479–497, 1994.

[4] Sarah Flannery and David Flannery. *In code*. Workman Publishing, New York, 2001. A mathematical journey, Reprint of the 2000 original.

[5] Ki Hyoung Ko, Sang Jin Lee, Jung Hee Cheon, Jae Woo Han, Ju-sung Kang, and Choonsik Park. New public-key cryptosystem using braid groups. In *Advances in cryptology—CRYPTO 2000 (Santa Barbara, CA)*, volume 1880 of *Lecture Notes in Comput. Sci.*, pages 166–183. Springer, Berlin, 2000.