The Plymouth Student Scientist

Volume 13 | Issue 1 Article 14

2020-10-10

Assessing the security of cryptographic primitives for infinite groups

Mathew J. Walter
University of Plymouth

Recommended Citation

Walter, M.J. (2020) 'Assessing the security of cryptographic primitives for infinite groups', The Plymouth Student Scientist, 13(1), p. 432-475.

Assessing the security of cryptographic primitives for infinite groups

Walter, M.J.

*The Plymouth Student Scientist, 2020, **13**, (1), 432-475*

Assessing the security of cryptographic primitives for infinite groups

Mathew J. Walter

Project Advisor: Dr. Matthew Craven, School of Engineering, Computing, and Mathematics, University of Plymouth, Drake Circus, Plymouth, PL4 8AA

Abstract

This paper considers the application of group theory to cryptography using a non-abelian infinite group (the braid group). The practical application of cryptographic protocols are determined by their security and feasibility. Both research papers and experiments will be used to measure feasibility and security of the protocol, with the intention of ultimately deeming the protocol either effective or ineffective. Having secure cryptography is vital to providing anonymity, confidentiality and integrity to data and as the quantum threat creeps towards us, the ever greater importance of new secure cryptography is becoming clear.

Introduction

The focus of this paper is to assess the security of possible future cryptographic protocols which implement infinite non-abelian groups.

The most common cryptographic systems such as RSA [1] and Diffie-Hellman [2] are applied in all manners of ones' life, from securing online bank transfers to communication networks. Whilst these methods have so far kept us secure, the power of computing machinery over the last few years has made it theoretically possible for susceptible attacks. Mathematicians have considered these methods becoming increasingly less secure. Particularly with the ongoing development of the quantum computer and quantum algorithms, such as Shors algorithm [3] which can break RSA under increasingly larger parameters. This is due to the complexity of these number theory based protocols, for example RSA has a brute force complexity strength that is exponential [4] and it may be the case that this complexity level is not sufficient for security in the quantum world.

In response, there is ongoing research into developing cryptographic algorithms using non-commutative platforms. This is because many of these protocols have greater than exponential (such as factorial) levels of security complexity making them far more difficult to attack. We have seen many interesting proposed cryptosystems involving different non-abelian groups and different 'hard problems'.

This paper will be examining the most popular non-abelian public exchange protocols (Ko-Lee's 2000 paper [5] and AAG's 1999 paper [6]). Interestingly, most of the research is very modern and many of these ideas are still in their early years. These protocols will be examined from a multitude of different angles including: using modern research papers that have attempted to attack these protocols, looking back into the history of group theory Mathematics (1969) at the possible mathematical methods for attacking the hard problems from Garside (which could prove effective in the modern world) and where there is a gap in knowledge using our own experimental data to estimate values such as the complexity and effect of parameter sizes on the cryptosystems. To do so, one will produce code to implement AAG and build attacking experiments in the programming language Sage.

Finally, when satisfactory evidence has been acquired we shall be making an assessment of the protocols considering various attacks, suggesting modification where appropriate and examining whether these protocols are secure relative to modern information and research. One will be able to consider whether these protocols may be suitable for the future cryptography.

1 Groups

A group is a non-empty set of elements with a binary operation which can be denoted $*$, forming an algebraic structure which must satisfy four axioms:

1. Closure - elements of a set which are acted upon by the group operator shall always produce a member of the same set. For example, the positive integers $\{1, 2, 3, \ldots, \}$ are closed under addition of positive numbers but not under subtraction (addition of negative numbers).
2. Associativity - in the case of elements which are grouped with parentheses, the order of which you perform the binary operations is not of significance i.e. for all $a, b, c \in G, (a * b) * c = a * (b * c)$.
3. Inverse - each element must have an inverse element, if 'a' is an element then a^{-1} is the inverse. $\exists e \in G : aa^{-1} = e,$ for all $a \in G$.
4. Identity - by combining an element with its inverse the resultant is an identity element. The identity element is denoted e or 1. The identity element alone is the trivial subgroup of any group. For example if a is an element $e{*}a = a$. If the binary operator is addition the identity element is 0. If the operator is multiplication the identity element is 1.

Groups must also contain at least one element. A group with just one unique element is named the trivial group. Groups can also hold other properties such as commutativity, these are called abelian groups.

1.1 Subgroups

It may also be useful to define a subgroup. A subset of a group is also a subgroup if and only if:

1. The subset is closed under the group operation.
2. The identity of the subset is the identity of the group.
3. Each element of the subset has an inverse that is also in the subset.

If a subset H of G is a subgroup of G, this is usually denoted $H \leq G$.

1.2 Presentation of a Group

A mathematical way to represent a group is through defining a group by a presentation. In order to do so, one must specify a set S of generators and a set R of relations. In its simplest form, a presentation is written $\{S|R\}$. For an example of a presentation of a group see (1) below.

$$\langle A\rangle = \{a_{i_1}^{\varepsilon j}, \dots, a_{i_n}^{\varepsilon j} | a_{i_j} \in A, \varepsilon_j \in \{1,-1\}, n \in \mathbb{N}\} \tag{1}$$

It is important for one to understand that when the binary operator is addition $3^0 = 0$ and it is **not** the case that $3^0 = 1$, which would only be correct if the group operator was multiplication. For example,

$$< a >=< 3 >= \{3^0, 3^1, 3^2, 3^3\} = \{0,\ 3,\ 3+3,\ 3+3+3\} = \{0,3,6,9\}.$$

But what is a generator? In a group $\{G,*\}$, a generator is a subset of elements or a single element, which under the group operator * can produce all elements of the group G.

Example 1. Find all generators of the group $\mathbb{Z}_8 = \{0,1,2,3,4,5,6,7\}$ under addition.

N	Elements generated by N	Order	Is N a generator of $\mathbb{Z}_8$?
0	0	1	No
1	0,1,2,3,4,5,6,7	8	Yes
2	2,4,6,0	4	No
3	3,6,1,4,7,2,5,0	8	Yes

Table 1: Testing various elements of $\mathbb{Z}_8$ to check if they are generators.

It is clear that for an element to be a generator it needs to have an order of 8. Rather than writing out and checking all possible generators one may observe a pattern; if the generator is relatively prime to 8 then it generates $\mathbb{Z}_8$. Therefore $1,3,5,7$ are generators of $\mathbb{Z}_8$.

1.3 Examples of Groups and Subgroups

Cyclic Groups: Let G be a group with operation $*$. Pick a single element $a \in G$. If the group is generated solely by a then it is called a cyclic group ($G =< a >$). Cyclic subgroups, $H < G$, are those generated by a generator of G, and must obey the axioms for a subgroup as detailed in Section 1. The group generated by a under the operation multiplication is written as

$$< a >= \{\dots, a^{-3},\ a^{-2},\ a^{-1},\ 1,\ a,\ a^2,\ a^3, \dots\} = \{a^n | n \in \mathbb{Z}\},$$

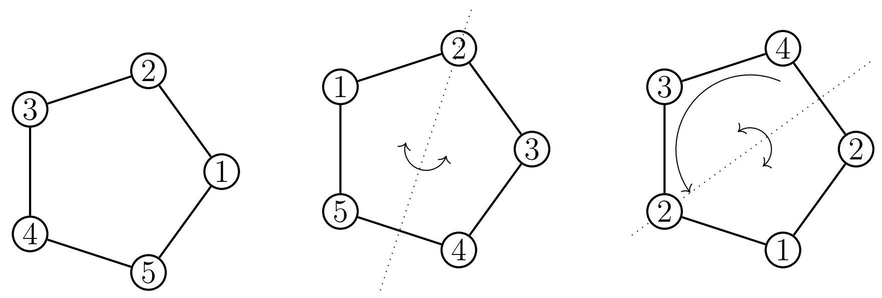
whereas the group generated by a under the operation addition is

$$< a >= \{\dots, -3a, -2a, -a, 0, a, 2a, 3a, \dots\} = \{na | n \in \mathbb{Z}\}.$$

Example 2. An example of an infinite cyclic group is the group $\mathbb{Z}$ under addition with a single generator of 1. Then, $< 1 >= \{\dots, -2, -1, 0, 1, 2, \dots\}$.

Example 3. An example of a finite cyclic group is the group $\mathbb{Z}_n$ under addition for $n \neq \infty$. The group is cyclic and can be generated by 1; i.e., $G =< 1 >= \{0, 1, 2, \ldots, n-1\}$. The group generated by 1 will cycle through the numbers 0 to $n - 1$.

Example 4. By observing the symmetries (reflection, rotation) of a regular polygon one can create a group, the Dihedral group. An anti-clockwise rotation of the shape by $2\pi/n$ radians (where n is the number of vertices) is a symmetry (denoted 'r'). In general a sequence of 'a' rotations is denoted r^a. The second symmetry is reflection, denoted f. For an n-sided polygon there are n distinct reflections. Generators (transformations) such as a reflection followed by a rotation, can be combined to find the remaining symmetries. A reflection and 4 rotations would be fr^4. To obtain all the remaining symmetries one would have to reflect the shape then perform $n - 1$ rotations (fr^{n-1}) (Fig.1).



Initial position (e) Single reflection (f) $(r^2 f)$

Figure 1: Symmetries of the dihedral group.

An interesting property of the dihedral group is that it is non-abelian when $n \geq 3$, i.e., rf does not equal fr. The notation for Dihedral groups is D_n where n is the number of sides of the polygon; alternatively, it may be denoted as D_{2n} where $2n$ is the number of symmetries in the group. The Dihedral group is then given by

$$D_n = \{e, r, r^2, \ldots, r^{n-1}, f, rf, r^2 f, \ldots, r^{n-1} f\}. \tag{2}$$

One possible presentation of a finite dihedral group is

$$D_n = \{rf | r^n = e, f^2 = e, rfr = f\}. \tag{3}$$

One possible presentation of an infinite dihedral group is

$$D_\infty = \{rf | f^2 = e, rfr = f\}. \tag{4}$$

In Table 2, the Cayley table clearly shows and compares combinations of generators in a Dihedral group. To compute the elements, take any two generators and multiply them together as a product of generators i.e. to compute the product of e and r^2 simply multiply $e * r^2$ to obtain r^2.

2 Free Groups

2.1 Notation

In order to examine groups more formally one must be comfortable with some of the key notation and definitions used in group theory. Some useful definitions are as follows: let X be an arbitrary subset from the group G. A *word* in the set X is a sequence

	e	r	r^2	r^3	r^4	f	rf	r^2f	r^3f	r^4f
e	e	r	r^2	r^3	r^4	f	rf	r^2f	r^3f	r^4f
r	r	r^2	r^3	r^4	e	rf	r^2f	r^3f	r^4f	f
r^2	r^2	r^3	r^4	e	r	r^2f	r^3f	r^4f	f	rf
r^3	r^3	r^4	e	r	r^2	r^3f	r^4f	f	rf	r^2f
r^4	r^4	e	r	r^2	r^3	r^4f	f	rf	r^2f	r^3f
f	f	rf	r^2f	r^3f	r^4f	e	r	r^2	r^3	r^4
rf	rf	r^2f	r^3f	r^4f	f	r	r^2	r^3	r^4	e
r^2f	r^2f	r^3f	r^4f	f	rf	r^2	r^3	r^4	e	r
r^3f	r^3f	r^4f	f	rf	r^2f	r^3	r^4	e	r	r^2
r^4f	r^4f	f	rf	r^2f	r^3f	r^4	e	r	r^2	r^3

Table 2: Cayley table to show all the permutations of D_5.

of elements from the subset X. We can denote this $w = y_1, \cdots, y_n, y_i \in X$ and see that y_i are the elements in X. Note y_i's do not have to be distinct. The *length* of the word is the number of elements in the word denoted $|w|$ i.e. for the word $w = \sigma_3\sigma_4$ there is two elements thus $|w| = 2$. Later L, L_1 and L_2, will also be used to define a words length, L denoting the length of the word, L_1 denoting the lowest possible bound for the length of a word and L_2 denoting the greatest possible upper bound length of a word. A word can also be empty denoted ϵ i.e. it has no letters.

For an arbitrary set X closed under inversion, consider all the inverses of X, $X^{-1} = \{x^{-1}|x \in X\}$. Then x and x^{-1} can be defined as *literals* in X. The intersection of the two literals is the identity element and the union of the two literals of X, $X^{\pm 1} = X \cup X^{-1}$ contains the set of all literals in X.

A word is said to be *reduced* if it contains no subword of the type xx^{-1} or $x^{-1}x$, where $x \in X \cup X^{-1}$. To describe this reduction process one needs to obtain a reduced word from an arbitrary word, to do so one needs to delete all subwords. For example if all the subword elements commute, and the word in question is $w = aba^{-1}c$ then the word can be reduced by deleting the sub-word aa^{-1} which yields the reduced word $w = bc$.

2.2 Free Group Definition

Starting with the formal definition extracted from Myasnikov, Shpilrain and Ushakov's group-based cryptography book [7]:

Definition 2.1. *A group G is called a free group if there exists a generating set X of G such that every non-empty reduced group word in X defines a non-trivial element of G.*

A free group G, commonly denoted $F(S)$, must have a generating set S. The generators are not related, other than the relation between a generator and its inverse. This set S is defined as the *basis*. The group elements of $F(S)$ are made up of finite sequences of symbols from the basis S; i.e., $\sigma_2\sigma_5 = s$ where s is an element (word) in G with the operation of concatenation. All group elements must be reduced. In the case of free groups the identity element is the empty sequence, i.e., the element of s which contains no symbols and has length zero.

3 Braid Groups

In this chapter we shall be examining braid groups from an algebraic and geometric perspective, along with the presentation of the braid group, the braid monoid, the word problem, pure braids and some of Garside's work on the fundamental braid and left canonical form [8].

3.1 Algebraic and Geometric Perception

Braid groups are used in many applications of science. They are non-abelian groups (for $n \geq 3$ as B_2 is an infinite cyclic group, it is isomorphic to $\mathbb{Z}$ and thus non-abelian) and thus make an interesting platform for new cryptosystems. To describe braids from a geometric perspective, consider taking n equal length and equally spaced parallel lines (strands) in $\mathbb{R}^3$ all with initial positions on axes perpendicular to the strands i.e. they could have the initial positions $(0,0,0)$, $(0,1,0)$, $(0,2,0)$, $\ldots$, $(0,n,0)$. If we then extended these strands across a plane (in our example, the XY plane) where strings can cross but no strands can intersect one another or turn back, we would have a simple example of a braid. Where these strands cross each other is of great interest and by studying these permutations of crossings one can devise a braid group. Figure 2 shows a braid in the braid group aligned horizontally.

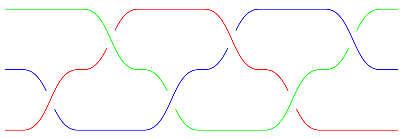


Figure 2: A geometric example of a braid.

Braid groups can have an infinite number of strands and as braid groups are groups they must of course follow the axioms of a group. One such axiom is that they possess an operator called the *composition*: this is equivalent to attaching one end of braid A to one end of braid B. Other properties include being associative, having inverses and having a trivial braid or identity braid in which no strands cross, this is illustrated in Figure 3 below.

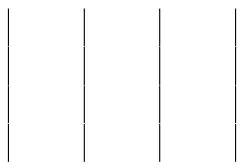


Figure 3: A vertically aligned trivial braid.

For describing braids, we use the notation σ to show a crossing in the strands. σ_1 is therefore the crossing of the first and second braid in a clockwise (left to right) direction, to cross in the opposite direction/anticlockwise using the same strings we would denote this σ_1^{-1} and this is the inverse of σ_1. To cross the second and third string in a clockwise direction we would denote this σ_2. The generators for a braid group are therefore $\sigma_1, \sigma_2, \sigma_3, \cdots, \sigma_{n-1}$ assuming n strings. The elements of a braid group are words made up of letters $\sigma_i^{\pm 1}$ where $|i| \leq n-1$. Figure 4 gives an example of a braid with positive and negative generators.

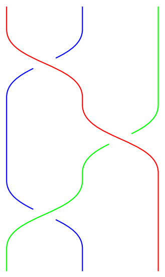


Figure 4: The braid word $\sigma_1\sigma_2\sigma_1^{-1}$.

3.1.1 Presentation of the Braid Group

Braids are difficult to examine in their natural state and therefore mathematicians sometime re-write these as matrices. For example, one such linear algebra representation of a braid group is the Burau representation [9] which adopts the use of matrices to describe braid groups. Another example is the Lawrence-Krammer presentation [10].

There are many ways to present a braid group. A common representation is using an Artin group (or generalised braid group) suggested by Emil Artin [11]. He denotes braid groups as B_n where n is the number of strands. Artin also first suggested the generators be denoted σ_i. The Artin presentation of a braid group on n strands is:

$$B_n = \left\langle \sigma_1, \ldots, \sigma_{n-1} : \begin{array}{c} \sigma_i\sigma_j = \sigma_j\sigma_i \text{ for } |i-j| \geq 2 \\ \sigma_i\sigma_{i+1}\sigma_i = \sigma_{i+1}\sigma_i\sigma_{i+1} \text{ for } 1 \leq i \leq n-2 \end{array} \right\rangle \tag{5}$$

From the relation $\sigma_i\sigma_j = \sigma_j\sigma_i$ for $|i-j| \geq 2$ if two strand crossings do not share strands then they commute. With regards to the second relation, Figure 5 shows $\sigma_i\sigma_{i+1}\sigma_i = \sigma_{i+1}\sigma_i\sigma_{i+1}$ for $1 \leq i \leq n-2$. Artin stated these two relations were sufficient to describe a braid group, but it is yet to be proven that these two relations are the only relations required.

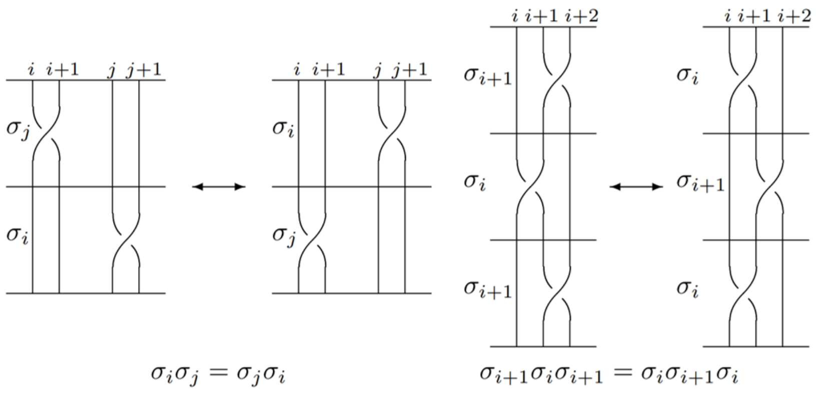


Figure 5: The braid group relations (from [12]).

3.1.2 Pure Braids

Regardless of how many crossings are formed with other braids and the position of those crossings, if the start of the strands and end of the same strands are in a position which is perfectly adjacent to each other, one can say the braid is *pure*. For an example, consider in $\mathbb{R}^3$ two parallel braid strands L_1 and L_2 with length 1 unit. If the start of the braid strands are positioned on the x-axis where L_1's initial position is $(1, 0, 0)$ and L_2 is at $(2, 0, 0)$ and extended across the positive XY plane (regardless of crossings), for a braid to be pure, L_1 must end at position $(1, 1, 0)$ and L_2 must end at position $(2, 1, 0)$. See Figure 6 for a further example: note the start and finish braid position.

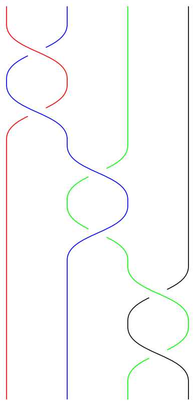


Figure 6: An example of a pure braid.

3.1.3 The Positive Braid Monoid

When computing braids we may only require positive braid elements. In which case one could use the braid monoid. A monoid holds similar properties to a group except a monoid does not contain inverse elements. It is made up of the positive powers of Artin generators. We can denote the braid monoid B_n^+ and present it as:

$$B_n^+ = \left\langle \sigma_1, \ldots, \sigma_{n-1} : \begin{array}{c} \sigma_i\sigma_j =^+ \sigma_j\sigma_i \text{ for } |i-j| \geq 2 \\ \sigma_i\sigma_{i+1}\sigma_i =^+ \sigma_{i+1}\sigma_i\sigma_{i+1} \text{ for } 1 \leq i \leq n-2 \end{array} \right\rangle^+ \tag{6}$$

The braid monoid is homogeneous. That is, if two positive braids are equivalent the length (order) of both words is the same.

3.1.4 The Word Problem

Braid groups can also have multiple representations of the same word in which braid strands can cross at different positions and a different number of times, making them look different from a geometric and non-normalised algebraic perspective but still the two words can be equivalent. If this is the case we can use the symbol $\equiv$ to mean two words are equal in the braid group but not necessarily equal in the sense that the two words are not made from the same generators in the exact same positions of the word. However very often both = and $\equiv$ symbols are used interchangeably. From this one can form an argument; are two braid words equivalent? This 'decision problem' is commonly known as the *word problem*. The word problem (along with the conjugacy problem) was first posed by Max Dehn in 1911 [13]. Dehn proposed given an arbitrary word in group G, decide if the word is equal to the identity element of G. Fortunately, on the n strand braid group containing elements (words) of size L, the word problem for this group is solvable in polynomial time with a known algorithm having a run time of $O(L^2n)$ [6]. This polynomial-time algorithm computes the left canonical form of a braid word and hence provides a feasible solution to the braid group word problem.

3.2 The Fundamental Braid

Suggested by Garside [8], a relevant example of a positive braid from the braid monoid B_n^+ is the fundamental braid which is denoted Δ_n. The fundamental braid can be observed geometrically as a half twist of the trivial braid where any two strands cross positively exactly once (see Figure 7). One can define the fundamental braid by:

$$\Delta_n = (\sigma_1 \dots \sigma_{n-1})(\sigma_1 \dots \sigma_{n-2}) \dots \sigma_1 \tag{7}$$

Note $\Delta_1 = \sigma_1$. From the above, any example of the fundamental braid may be computed by simply inputting values of n into the form. For example $\Delta_5 = \sigma_1\sigma_2\sigma_3\sigma_4\sigma_1\sigma_2\sigma_3\sigma_1\sigma_2\sigma_1$. The fundamental braid can also be written

$$\Delta_n = \Delta_{n-1}\sigma_{n-1}\sigma_{n-2} \dots \sigma_1. \tag{8}$$

For example:

$$\Delta_2 = \Delta_1\sigma_1 = \sigma_1\,, \quad \Delta_3 = \Delta_2\sigma_2\sigma_1 = \sigma_1\sigma_2\sigma_1$$
$$\Delta_4 = \Delta_3\sigma_3\sigma_2\sigma_1 = \sigma_1\sigma_2\sigma_1\sigma_3\sigma_2\sigma_1$$
$$\Delta_5 = \Delta_4\sigma_4\sigma_3\sigma_2\sigma_1 = \sigma_1\sigma_2\sigma_1\sigma_3\sigma_2\sigma_1\sigma_4\sigma_3\sigma_2\sigma_1.$$

It should now be clear that both equations, (7) and (8) generating Δ_5 give the same fundamental word although expressed differently algebraically. It is possible to trivially convert between the two using the braid relations to see (7) and (8) are equal in the braid group. A positive braid is called a *simple braid* if it is a leftover factor from the divisor of the fundamental braid. It is also referred to as a *permutation braid*. From a geometric perspective a braid which only has positive crossings and every pair of strands crosses at most once is a permutation braid.

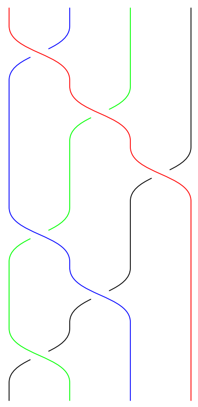


Figure 7: The fundamental, or half-twist, braid Δ_4.

Lemma 3.1 ([14]). In B_n^+ as well as in B_n we have $\Delta\sigma_i = \sigma_{n-i}\Delta$.

Proof. For B_n this is obvious from the half twist (Δ) in Figure 7.

3.3 Garside Normal form or Left Normal Form

Consider a horizontally aligned braid. If no crossings can be moved from the right half of the braid to the left half without changing the word then one can say the braid is left-weighted. As it is possible to write a word in many different ways using different generators in different positions, there needs to exist a standard so that every braid admits a unique representation (a normal word). From the fundamental braid and permutation braids one can express words in the Garside normal form [8], also known as the greedy normal form (greedy referring to the maximum number of elements from the word being written as the fundamental braid). If a braid is expressed in its normal form, with an additional condition where the permutation braids on the left are as large as possible then this braid is in left normal form, which makes this representation unique. It is important to be able to express elements of a group in their normal form in order to obtain a unique presentation of all the elements. Hence solving the word problem which allows one to compare braids.

Theorem 4.3.1 [8] For every braid $w \in B_n$, there is a unique presentation given by:

$$w = \Delta_n^r P_1 P_2 \dots P_k \tag{9}$$

where $r \in \mathbb{Z}$ is the infimum of w or $\inf(w)$. Note the infimum is the greatest lower bound of a set and the supremum is the least upper bound of a set (the number of permutation braids i.e. $\sup(w) = k$). P_i are permutation braids where $P_k \neq \varepsilon$ and $P_1 P_2 \dots P_k$ is a left-weighted decomposition.

Example 5. Using Sage, this example computes the left-normal form of a braid $w = \sigma_1 \sigma_3^{-1} \sigma_2 \in B_4$.

First, replace all negative power generators σ_i^{-1} by $\Delta_n^{-1} P_i$. Therefore, as w admits a decomposition with a negative letter, replace σ_3^{-1} by $\Delta_4^{-1}\sigma_2\sigma_1\sigma_3\sigma_2\sigma_1$ to create:

$$w = \sigma_1 \cdot \Delta_4^{-1}\sigma_2\sigma_1\sigma_3\sigma_2\sigma_1 \cdot \sigma_2$$

First check:

$$\sigma_3^{-1} = \Delta_4^{-1}\sigma_2\sigma_1\sigma_3\sigma_2\sigma_1 = \sigma_1^{-1}\sigma_2^{-1}\sigma_3^{-1}\sigma_1^{-1}\sigma_2^{-1}\sigma_1^{-1}\sigma_2\sigma_1\sigma_3\sigma_2\sigma_1$$

Using the two braid group relations rearrange:

$$\sigma_1^{-1}\sigma_2^{-1}\sigma_3^{-1}\sigma_2^{-1}\sigma_1^{-1}\sigma_2^{-1}\sigma_2\sigma_1\sigma_3\sigma_2\sigma_1$$

We can first cancel $\sigma_2^{-1}\sigma_2$ and then cancel $\sigma_1^{-1}\sigma_1$:

$$\sigma_1^{-1}\sigma_2^{-1}\sigma_3^{-1}\sigma_2^{-1}\sigma_3\sigma_2\sigma_1$$

We can then exchange $\sigma_2^{-1}\sigma_3^{-1}\sigma_2^{-1}$ with $\sigma_3^{-1}\sigma_2^{-1}\sigma_3^{-1}$:

$$\sigma_1^{-1}\sigma_3^{-1}\sigma_2^{-1}\sigma_3^{-1}\sigma_3\sigma_2\sigma_1$$

Cancel $\sigma_3^{-1}\sigma_3$ and then cancel $\sigma_2^{-1}\sigma_2$:

$$\sigma_1^{-1}\sigma_3^{-1}\sigma_1$$

Finally rearrange, cancel $\sigma_1^{-1}\sigma_1$ and hence:

$$\sigma_3^{-1} = \sigma_3^{-1} = \Delta_4^{-1}\sigma_2\sigma_1\sigma_3\sigma_2\sigma_1.$$

After replacing all the negative power generators σ_i^{-1} by $\Delta_n^{-1}P_i$. Use **Lemma 3.1** to move all appearances of Δ_n to the left and hence move Δ_4^{-1} to the start of the word. As $n = 4$ and $i = 1$, this yields:

$$w = \Delta_4^{-1}\sigma_3\sigma_2\sigma_1\sigma_3\sigma_2\sigma_1\sigma_2$$

After decomposing the positive part into a left-weighted decomposition (left-weighted due to the order of the permutation braids):

$$w = \Delta_4^{-1} \cdot \sigma_2\sigma_1\sigma_3\sigma_2\sigma_1 \cdot \sigma_1\sigma_2$$

Check using Sage:

```
Input: B = BraidGroup(4)
       W = B([1,-3,2])
       W.left_normal_form()
Output:(s0^-1*s1^-1*s2^-1*s0^-1*s1^-1*s0^-1, s1*s0*s2*s1*s0,
         s0*s1)
```

The complexity, also known as the canonical length of w is given by $\mathrm{len}(w) = \sup(w) - \inf(w)$. The canonical length of w is denoted $len(w)$. Therefore in Example 5 the canonical length is $2 - (-1) = 3$. This notation is used heavily throughout the length based attacks (Section 5.4).

4 Braid Group Cryptosystems

When Ko-Lee et al.[5] were deciding on a non-abelian platform group to operate their cryptographic protocol, they needed to choose a platform group which met certain criteria. Commonly accepted criteria are given by Myasnikov, Shpilrain and Ushakov [7, p. 38–39].

1. Perhaps the most obvious criterion for a group to be cryptographically secure would be suggesting the ‘hard problem’ be sufficiently ‘difficult’ for an adversary to solve. For braid groups we can use the properties of non-commutativity to our advantage to create a hard problem. This problem is the group *conjugacy search problem*. The conjugacy search problem is a decision problem in which given two words a and b in group G, determine whether or not they represent conjugate elements in G. The CSP is a very practical and well known hard problem for application in cryptosystems. It is clear the conjugacy search problem needs to be ‘difficult’. It also seems necessary to mathematically define the term ‘difficult’. For this, the conjugacy search problem should not be solvable with modern computers in sub-exponential time.

2. A natural avenue may now be to question: how would we know if the conjugacy search problem for a given group is ‘difficult’? Bringing us on to the second criterion. The group must be well studied and known so that mathematicians have had a chance to evaluate whether the conjugacy search problem has a subexponential-time solution by a deterministic algorithm. For the case of braid

groups there are many hard problems. These include but are not limited to the conjugacy decomposition problem [13], the Markov problem [5] and the conjugacy search problem [13]. These problems can be exploited to design 'trap door' one-way functions. Mathematicians have spent many years trying to develop algorithms that can solve the conjugacy search and decision problem but have been unsuccessful in solving it in polynomial time $O(n^c)$ with a deterministic algorithm. Further, proving a group to have conjugacy search problem unsolvable in subexponential-time is very difficult (hence the importance of using a well-studied group).

3. Next, the word problem in G (deciding whether two words in the generators represent the same element in G) should have a fast solve time, i.e., either linear ($O(n)$) or quadratic ($O(n^2)$) time. This means there should be an efficient computable normal form for elements of G. For braid groups the word problem is solved quickly by an algorithm which computes the canonical form. The canonical form of a braid is described in an ordered list called a tuple $(r; P_1, P_2, \cdots, P_k)$ where r is an integer and P_i's are permutations. Here $(r; P_1, P_2, \cdots, P_k) = \Delta_n^r P_1 P_2 \dots P_k$. By converting to canonical form the computer can handle the data more efficiently and it removes any difficulty in using the words in the description of groups. The complexity of transforming a word into a canonical form is $O(|W|^2 n)$, where $|W|$ is the word length in B_n [15].

4. The next criterion is that G should be a group of super-polynomial growth. This means that as the number of elements of length n increases, the size of G should grow faster than any polynomial in n. This is to prevent brute force attacks by key space exhaustion. Braid group cryptography is thought to become stronger as n increases. Fortunately, the braid groups B_n are of exponential growth if $n \geq 3$ [5].

5. Finally, it should be impossible to recover through inspection the conjugator x from $x^{-1}wx$ or else the problem would become trivial. We could disguise elements by writing in normal form. i.e. we have seen from Section 3.3, $\sigma_1^{-1}\sigma_2^{-1}\sigma_1^{-1}\sigma_3^{-1}$ may be expressed as $\Delta_4^{-1}\sigma_1\sigma_2$. Braids can be converted to normal form, thus concealing elements.

Subject to this, assessment, braid groups appear a suitable candidate for cryptography.

4.1 Diffie-Hellman

Before we examine some braid group cryptosystems we will take a brief detour to cover the idea of Diffie-Hellman [2]. Diffie-Hellman is a key establishment protocol (KEP) that conventionally uses number theory as elements of the protocol. A key establishment protocol is a multi-party algorithm where a secret key can be shared between authorised parties for encryption and decryption. The beauty of Diffe-Hellman is that both authorised parties can agree on a secure key for encryption over the public domain, meaning even with a hostile party intercepting the messages, without the unshared secure private keys the hostile party is unable to decrypt the messages without solving a 'hard' problem. This problem is very similar to the discrete logarithm problem [16]. The original Diffie-Hellman uses the multiplicative group of integers modulo N. The algorithm works as follows:

1. Alice and Bob must first agree on two things, a modulus N, thus agreeing on a finite cyclic group G. Then they agree on a generator $g \in G$. This is all done in the public domain. Often these are pre-agreed to save time and it is best to use a large prime number for N.
2. Alice agrees on a private key $a \in \mathbb{N}$ that only she will know. This is kept in the private domain, and she sends g^a to Bob.
3. Bob agrees on a private key $b \in \mathbb{N}$ that only he knows. This is kept in the private domain and he sends g^b to Alice.
4. Alice computes $(g^b)^a = g^{ba}$.
5. Bob computes $(g^a)^b = g^{ab}$.

As integers are commutative under multiplication, the powers (private keys) are identical, that is $ab = ba$, thus Alice and Bob have agreed on a secure shared key to encrypt future communications g^{ab}.

Any eavesdropper would have to solve either g^a for the power a, or g^b for b (while only knowing g, G, g^a and g^b). This is considered to be a hard problem when using a good choice of G and g. This hard problem (although this is not yet proven and may never be!) is called the *Diffie-Hellman problem* and can be considered the same for practical purposes.

4.2 AAG and Ko-Lee Et Al. Cryptosystem

Section 4 concluded that the use of the braid group as a platform group could be a successful option for creating infinite group based cryptographic protocols. Throughout the rest of Section 4 we shall examine and refer to the work of Ko-Lee et al. [5] (2000) and Anshel-Anshel-Goldfeld [17] (revised in 2003).

4.2.1 Anshel-Anshel-Goldfeld Key Exchange Method

The remarkable advantage for this key establishment protocol is that providing the two groups meet the criteria detailed in Section 4 neither the subgroups nor the platform group need to be abelian, unlike Ko-Lee et al. [5] (where we will see that the subgroups must commute). The key exchange is as follows. Let G be a group (in our case a braid group) $G = \langle g_1, \ldots, g_n | R \rangle$. Alice and Bob are two parties that need to securely communicate with each other and so need to agree on a shared secret key in the public domain. Alice and Bob agree on parameters:

- N = the number of elements in Alice's and Bob's tuples.
- L = the length of the private keys.
- L_1, L_2 = the minimum and maximum bounds respectively for the length of a word in a tuple.

Alice and Bob then perform the following actions, referring to [18].

1. Alice chooses a number (N) of words in generators of G to form an N-tuple $(a_i)_{i=1}^{N} = (a_1, \ldots, a_N)$ each word in (a_i) must have a length between L_1 and L_2. She then publishes $(a_i)_{i=1}^{N}$ in the public domain;

2. Bob chooses a number (N) of words in generators of G to form an N-tuple $(b_i)_{i=1}^{N} = (b_1, \ldots, b_N)$ each word in (b_i) must have a length between L_1 and L_2. He then publishes $(b_i)_{i=1}^{N}$ in the public domain;

3. Alice chooses a private key $A = a_{\mu_1}^{\epsilon_1} a_{\mu_2}^{\epsilon_2} \ldots a_{\mu_L}^{\epsilon_L}$ where $\mu_i \in \{1, \ldots, N\}$ (this is to record the position of the word in the tuple $(a_i)_{i=1}^{N}$ to be used later) and $\epsilon_i \in \pm 1$ for all $i = 1, \ldots, L$ (this adds more complexity, essentially doubling tuple size with minimal computational effort);

4. Bob chooses a private key in a similar way, $B = b_{\nu_1}^{\delta_1} b_{\nu_2}^{\delta_2} \ldots b_{\nu_L}^{\delta_L}$ where $\nu_i \in \{1, \ldots, N\}$ and $\delta_i \in \pm 1$ for all $i = 1, \ldots, L$;

5. Alice then computes the tuple of conjugates

$$A^{-1}(b_i)_{i=1}^{N} A = \{A^{-1} b_1 A, \ldots, A^{-1} b_N A\}$$

   and transmits to Bob;

6. Bob then computes the tuple of conjugates

$$B^{-1}(a_i)_{i=1}^{N} B = \{B^{-1} a_1 B, \ldots, B^{-1} a_N B\}$$

   and transmits to Alice;

7. Alice then computes the common private key K by multiplying by her private key:

$$\begin{aligned} &A^{-1} \cdot ((B^{-1} a_1 B), \ldots, (B^{-1} a_n B)) \\ &= A^{-1}(B^{-1} a_{\mu_1}^{\epsilon_1} B \ldots B^{-1} a_{\mu_L}^{\epsilon_L} B) \\ &= A^{-1}(B^{-1} a_{\mu_1}^{\epsilon_1} \ldots a_{\mu_L}^{\epsilon_L} B) = A^{-1} B^{-1} A B = K \end{aligned}$$

8. Similarly, Bob then computes the common private key K:

$$\begin{aligned} &B^{-1} \cdot ((A^{-1} b_1 A), \ldots, (A^{-1} b_n A)) \\ &= B^{-1}(A^{-1} b_{\nu_1}^{\delta_1} A \ldots A^{-1} b_{\nu_L}^{\delta_L} B) \\ &= B^{-1}(A^{-1} b_{\nu_1}^{\delta_1} \ldots b_{\nu_L}^{\delta_L} A) = B^{-1} A^{-1} B A = K^{-1} \end{aligned}$$

9. Bob then computes the inverse of K^{-1}:

$$(B^{-1} A^{-1} B A)^{-1} = A^{-1} B^{-1} A B = K.$$

K is the established shared key for encryption and $A^{-1}B^{-1}AB$ is called the *commutator*. Once K is established we can construct a public key cryptosystem (Subsection 4.2.3). Original parameters were suggested by Anshel in 2001 [19]. These parameters are $n = 80$, $N_1 = N_2 = 20$, $L_1 = 5$, $L_2 = 8$, $L = 100$. Note that the AAG relies on a variation of the simultaneous conjugacy search problem, called the subgroup-restricted conjugacy search problem (SR-SCSP) and one can say the security of the AAG protocol is partially based, but not equivalent, on the assumption that SR-SCSP is hard.

Example 6. Using Sage and a self-engineered AAG protocol application (see appendices). This example indicates the sizes of braids under small parameters using parameters $n = 10$, $N_1 = N_2 = 5$, $L_1 = 3$, $L_2 = 5$, $L = 8$.

1. Alice and Bob generate tuples and publish them:

```
Input: At
Output: [s2^-1*s0^-1*s1*s0*s7, s7*s5^-1*s8^-1,
         s7^-2*s3^-1*s2*s1^-1, s2*s3*s6*s7, s6^-1*s3^-1*s2]

Input: Bt
Output: [s8, s8^-1*s2, s1^-1*s3^-1*s2*s1*s3,
         s5^-1*s4*s0^-1, s2*s5^-1*s6*s0^-1*s7^-1]
```

2. Alice chooses her private key and computes the inverse:

```
Input: A_priv
Output: s7^-2*s3^-1*s2*s1^-1*s2^-1*s3*s6*s1*s2^-1*s3*s7^2
*s1*s2^-1*s3*s7*s0^-1*s1^-1*s0*s2*s1*s2^-1*s3*s7^2*s2
*s3*s6*s7*s2^-1*s0^-1*s1*s0*s7

Input: A_privinv
Output: s7^-1*s0^-1*s1^-1*s0*s2*s7^-1*s6^-1*s3^-1*s2^-1
*s7^-2*s3^-1*s2*s1^-1*s2^-1*s0^-1*s1*s0*s7^-1*s3^-1*s2*s1^-1
*s7^-2*s3^-1*s2*s1^-1*s6^-1*s3^-1*s2*s1*s2^-1*s3*s7^2
```

3. Bob also chooses a private key and computes the inverse:

```
Input: B_priv
Output: s1^-1*s3^-1*s2*s1*s3*s8*s0*s4^-1*s5*s2*s3^-1
*s1^-1*s2^-1*s3*s1*s2^-1*s8*s1^-1*s3^-1*s2*s1*s3

Input: B_privinv
Output:s3^-1*s1^-1*s2^-1*s3*s1*s8^-1*s2*s1^-1*s3^-1*s2
*s1*s3*s2^-1*s5^-1*s4*s0^-1*s8^-1*s3^-1*s1^-1*s2^-1*s3*s1
```

4. Alice and Bob generate their tuple of conjugates:

```
Input: A_conj
Output:[s7^-1*s0^-1*s1^-1*s0*s2*s7^-1*s6^-1*s3^-1*s2^-1
*s7^-2*s3^-1*s2*s1^-1*s2^-1*s0^-1*s1*s0*s7^-1*s3^-1*s2*s1^-1
*s7^-2*s3^-1*s2*s1^-1*s6^-1*s3^-1*s2*s1*s2^-1*s3*s7^2*s8
*s7^-2*s3^-1*s2*s1^-1*s2^-1*s3*s6*s1*s2^-1*s3*s7^2*s1*s2^-1
*s3*s7*s0^-1*s1^-1*s0*s2*s1*s2^-1*s3*s7^2*s2*s3*s6*s7*s2^-1
*s0^-1*s1*s0*s7, s7^-1*s0^-1*s1^-1*s0*s2*s7^-1*s6^-1*s3^-1
*s2^-1*s7^-2*s3^-1*s2*s1^-1*s2^-1*s0^-1*s1*s0*s7^-1*s3^-1*s2
*s1^-1*s7^-2*s3^-1*s2*s1^-1*s6^-1*s3^-1*s2*s1*s2^-1*s3*s7^2
*s8^-1*s2*s7^-2*s3^-1*s2*s1^-1*s2^-1*s3*s6*s1*s2^-1*s3*s7^2
*s1*s2^-1*s3*s7*s0^-1*s1^-1*s0*s2*s1*s2^-1*s3*s7^2*s2*s3*s6
*s7*s2^-1*s0^-1*s1*s0*s7, s7^-1*s0^-1*s1^-1*s0*s2*s7^-1
*s6^-1*s3^-1*s2^-1*s7^-2*s3^-1*s2*s1^-1*s2^-1*s0^-1*s1*s0
*s7^-1*s3^-1*s2*s1^-1*s7^-2*s3^-1*s2*s1^-1*s6^-1*s3^-1*s2*s1
*s2^-1*s3*s7^2*s1^-1*s3^-1*s2*s1*s3*s7^-2*s3^-1*s2*s1^-1
*s2^-1*s3*s6*s1*s2^-1*s3*s7^2*s1*s2^-1*s3*s7*s0^-1*s1^-1*s0
*s2*s1*s2^-1*s3*s7^2*s2*s3*s6*s7*s2^-1*s0^-1*s1*s0*s7,s7^-1
*s0^-1*s1^-1*s0*s2*s7^-1*s6^-1*s3^-1*s2^-1*s7^-2*s3^-1*s2*
s1^-1*s2^-1*s0^-1*s1*s0*s7^-1*s3^-1*s2*s1^-1*s7^-2*s3^-1*s2
*s1^-1*s6^-1*s3^-1*s2*s1*s2^-1*s3*s7^2*s5^-1*s4*s0^-1*s7^-2
*s3^-1*s2*s1^-1*s2^-1*s3*s6*s1*s2^-1*s3*s7^2*s1*s2^-1*s3*s7
*s0^-1*s1^-1*s0*s2*s1*s2^-1*s3*s7^2*s2*s3*s6*s7*s2^-1*s0^-1
*s1*s0*s7,s7^-1*s0^-1*s1^-1*s0*s2*s7^-1*s6^-1*s3^-1*s2^-1
*s7^-2*s3^-1*s2*s1^-1*s2^-1*s0^-1*s1*s0*s7^-1*s3^-1*s2*s1^-1
*s7^-2*s3^-1*s2*s1^-1*s6^-1*s3^-1*s2*s1*s2^-1*s3*s7^2*s2
*s5^-1*s6*s0^-1* s7^-3*s3^-1*s2*s1^-1*s2^-1*s3*s6*s1*s2^-1
*s3*s7^2*s1*s2^-1*s3*s7*s0^-1*s1^-1*s0*s2*s1*s2^-1*s3*s7^2
*s2*s3*s6*s7*s2^-1*s0^-1*s1*s0*s7]
```

```
Input: B_conj

Output:[s3^-1*s1^-1*s2^-1*s3*s1*s8^-1*s2*s1^-1*s3^-1*s2
*s1*s3*s2^-1*s5^-1*s4*s0^-1*s8^-1*s3^-1*s1^-1*s2^-1*s3
*s1*s2^-1*s0^-1*s1*s0*s7*s1^-1*s3^-1*s2*s1*s3*s8*s0*s4^-1
*s5*s2*s3^-1*s1^-1*s2^-1*s3*s1*s2^-1*s8*s1^-1*s3^-1*s2*s1
*s3,s3^-1*s1^-1*s2^-1*s3*s1*s8^-1*s2*s1^-1*s3^-1*s2*s1*s3
*s2^-1*s5^-1*s4*s0^-1*s8^-1*s3^-1*s1^-1*s2^-1*s3*s1*s7
*s5^-1*s8^-1*s1^-1*s3^-1*s2*s1*s3*s8*s0*s4^-1*s5*s2*s3^-1
*s1^-1*s2^-1*s3*s1*s2^-1*s8*s1^-1*s3^-1*s2*s1*s3,s3^-1
*s1^-1*s2^-1*s3*s1*s8^-1*s2*s1^-1*s3^-1*s2*s1*s3*s2^-1
*s5^-1*s4*s0^-1*s8^-1*s3^-1*s1^-1*s2^-1*s3*s1*s7^-2*s3^-1
*s2*s1^-2*s3^-1*s2*s1*s3*s8*s0*s4^-1*s5*s2*s3^-1*s1^-1
*s2^-1*s3*s1*s2^-1*s8*s1^-1*s3^-1*s2*s1*s3,s3^-1*s1^-1
*s2^-1*s3*s1*s8^-1*s2*s1^-1*s3^-1*s2*s1*s3*s2^-1*s5^-1*s4
*s0^-1*s8^-1*s3^-1*s1^-1*s2^-1*s3*s1*s2*s3*s6*s7*s1^-1
*s3^-1*s2*s1*s3*s8*s0*s4^-1*s5*s2*s3^-1*s1^-1*s2^-1*s3*s1
*s2^-1*s8*s1^-1*s3^-1*s2*s1*s3,s3^-1*s1^-1*s2^-1*s3*s1
*s8^-1*s2*s1^-1*s3^-1*s2*s1*s3*s2^-1*s5^-1*s4*s0^-1*s8^-1
*s3^-1*s1^-1*s2^-1*s3*s1*s6^-1*s3^-1*s2*s1^-1*s3^-1*s2*s1
*s3*s8*s0*s4^-1*s5*s2*s3^-1*s1^-1*s2^-1*s3*s1*s2^-1*s8
*s1^-1*s3^-1*s2*s1*s3]
```

5. Alice and Bob both respectively compute their private keys K1 and K3:

```
Input: K1 = A_privinv * B_conj

Output: s7^-1*s0^-1*s1^-1*s0*s2*s7^-1*s6^-1*s3^-1*s2^-1
*s7^-2*s3^-1*s2*s1^-1*s2^-1*s0^-1*s1*s0*s7^-1*s3^-1*s2
*s1^-1*s7^-2*s3^-1*s2*s1^-1*s6^-1*s3^-1*s2*s1*s2^-1*s3
*s7^2*s3^-1*s1^-1*s2^-1*s3*s1*s8^-1*s2*s1^-1*s3^-1*s2*s1
*s3*s2^-1*s5^-1*s4*s0^-1*s8^-1*s3^-1*s1^-1*s2^-1*s3*s1
*s7^-2*s3^-1*s2*s1^-1*s2^-1*s3*s6*s1*s2^-1*s3*s7^2*s1
*s2^-1*s3*s7*s0^-1*s1^-1*s0*s2*s1*s2^-1*s3*s7^2*s2*s3*s6
*s7*s2^-1*s0^-1*s1*s0*s7*s1^-1*s3^-1*s2*s1*s3*s8*s0*s4^-1
*s5*s2*s3^-1*s1^-1*s2^-1*s3*s1*s2^-1*s8*s1^-1*s3^-1*s2*s1*s3
```

```
Input: K3 = (B_privinv * A_conj)^{-1}

Output: s7^-1*s0^-1*s1^-1*s0*s2*s7^-1*s6^-1*s3^-1*s2^-1
*s7^-2*s3^-1*s2*s1^-1*s2^-1*s0^-1*s1*s0*s7^-1*s3^-1*s2
*s1^-1*s7^-2*s3^-1*s2*s1^-1*s6^-1*s3^-1*s2*s1*s2^-1*s3
*s7^2*s3^-1*s1^-1*s2^-1*s3*s1*s8^-1*s2*s1^-1*s3^-1*s2*s1
*s3*s2^-1*s5^-1*s4*s0^-1*s8^-1*s3^-1*s1^-1*s2^-1*s3*s1
*s7^-2*s3^-1*s2*s1^-1*s2^-1*s3*s6*s1*s2^-1*s3*s7^2*s1
*s2^-1*s3*s7*s0^-1*s1^-1*s0*s2*s1*s2^-1*s3*s7^2*s2*s3*s6
*s7*s2^-1*s0^-1*s1*s0*s7*s1^-1*s3^-1*s2*s1*s3*s8*s0*s4^-1
*s5*s2*s3^-1*s1^-1*s2^-1*s3*s1*s2^-1*s8*s1^-1*s3^-1*s2*s1*s3
```

4.2.2 The Ko-Lee Et Al. Key Agreement Method

This system is a braid group version of Diffie-Hellman: consider the non-commutative braid group B_{l+r} and two naturally commuting subgroups LB_l and RB_r, where $LB_l, RB_r \in B_{l+r}$. The subgroup LB_l is generated by $\{\sigma, \dots, \sigma_{l-1}\}$ and RB_r by $\{\sigma_{l+1}, \dots, \sigma_{l+r-1}\}$. The subgroups LB_l and RB_r commute with each other because the elements have positions which are separated by at least an element (i.e. braids from RB_r and LB_l do not share the same strands). As the two subgroups do not share the same strands one can apply the first braid relation in Figure 5 to allow commutativity between the subgroups. Both subgroups and the braid $w \in B_{l+r}$ are made public.

Note that the two subgroups LB_l and RB_r are commuting element-wise and thus $ab = ba$ for all $a \in LB_l$, $b \in RB_r$. However $a_i a_{i+1} \neq a_{i+1} a_i$. Finally, note, that all braids are exchanged in left normal form, so it is not immediately obvious how to determine a and b when given the braid ab. Alice and Bob are two parties that need to securely communicate with each other:

1. Alice chooses an element (Alice's private key) $a \in LB_l$ and trivially computes the inverse $a^{-1} \in LB_l$. She then computes $Y_1 = awa^{-1}$ before sending Y_1 to Bob (where w is the public tuple);
2. Bob chooses an element (Bob's private key) $b \in RB_r$ and computes $b^{-1} \in RB_r$. He then computes $Y_2 = bwb^{-1}$ before sending Y_2 to Alice;
3. Alice receives Y_2 and computes the shared key
$$a(Y_2)a^{-1} = a(bwb^{-1})a^{-1} = K;$$
4. Bob receives Y_1 and computes the shared key
$$b(Y_1)b^{-1} = b(awa^{-1})b^{-1} = a(bwb^{-1})a^{-1} = K.$$

Since $a \in LB_l$ and $b \in RB_r$, $ab = ba$. Thus

$$a(Y_2)a^{-1} = a(bwb^{-1})a^{-1} = b(awb^{-1})b^{-1} = b(Y_1)b^{-1} = K.$$

An adversary, Eve, who captures the public elements $Y_1 = awa^{-1}$ and $Y_2 = bwb^{-1}$ can compute combinations of Y_1Y_2. For example,

$$Y_1Y_2 = (awa^{-1})(bwb^{-1}) = (awb)(a^{-1}wb^{-1})$$

but none of these combinations are equal to $K = a(bwb^{-1})a^{-1}$ due to non-commutativity in the subgroups.

Just like the discrete log problem is considered near equal to the Diffie-Hellman problem, this problem can be considered near equal to the *generalised search problem* or more accurately the *subgroup-restricted conjugacy search problem* due to it being very similar in complexity [5], this will be examine later. Furthermore, just as an attacker Eve would have to find x and y from g^x and g^y in Diffie-Hellman, in the case above Eve would have to find a^{-1} or a from $y = awa^{-1}$, and b or b^{-1} from $Y_2 = bwb^{-1}$.

4.2.3 The Ko-Lee Et Al. Cryptosystem Method

Once the key agreement has been performed it is possible to construct a public key cryptosystem using an ideal hash function (a one-to-one function) from the braid group to the message space $H : B_{l+r} \to \{0, 1\}^k$.

1. To encrypt Bobs's message $m \in \{0, 1\}^k$ with the public key (x, y) where $Y = awa^{-1}$ obtained from Alice, Bob must randomly choose a braid (his private key) $b \in LB_r$. Then generate ciphertext (c,d) where $c = bwb^{-1}$ and $d = H(bYb^{-1}) \oplus m$. Note $\oplus$ is the 'exclusive or' function commonly denoted 'XOR'.

2. For Alice to decrypt the ciphertext, she computes $m = H(aca^{-1}) \oplus d$.

As a and b commute:

$$aca^{-1} = abwb^{-1}a^{-1} = bawa^{-1}b^{-1} = byb^{-1}$$

Hence

$$H(aca^{-1}) \oplus d = H(byb^{-1}) \oplus H(byb^{-1}) \oplus m = m.$$

Example 7. In this interpretation of the key agreement system from Ko-Lee ([5]), Bob and Alice intend to use a binary message encryption protocol. Bob wishes to transmit the message "Hello Alice" to Alice. In binary "Hello Alice!" translates to

```
01001000 01100101 01101100 01101100 01101111 00100000 01000001
01101100 01101001 01100011 01100101 00100001.
```

Bob has both his own public key $Y_2 = bwb^{-1} = \sigma_1\sigma_2\sigma_1^{-1}$ and Alice's public key $Y_1 = awa^{-1} = \sigma_3\sigma_2\sigma_3^{-1}$ available to him in the public domain. In this example we have chosen Y_1 and Y_2 at random using the properties discussed in Section 4.2.2. Bob then creates a ciphertext (c, d) where c is his public key $Y_2 = \sigma_1\sigma_2\sigma_1^{-1}$ and d is computed as:

$$d = H(b(Y)b^{-1}) \oplus m = H(b(Y_1)b^{-1}) \oplus m = H(\sigma_1(\sigma_3\sigma_2\sigma_3^{-1})\sigma_1^{-1}) \oplus m$$

We first translate the shared symmetric key into a tuple with integer representation, $b(Y_1)b^{-1}$ = $[1, 3, 2, -3, -1]$ and then convert from ASCII text to binary:

```
00110001 00100000 00110011 00100000 00110010 00100000 00101101
00110011 00100000 00101101 00110001
```

After computing:

$$d = H(b(Y_1)b^{-1}) \oplus m$$

$$= \texttt{0100100001010100010011000101111101001111000100100110000101}$$
$$\texttt{00000101011010010000110100100000010000}$$

Bob can now transmit the cipher text (Y_2, d) to Alice.

Alice has received (c, d) = (Y_2, d) = $(\sigma_1\sigma_2\sigma_1^{-1}, H(abwa^{-1}b^{-1}) \oplus m)$ and now needs to decrypt m. She proceeds thus:

$$\begin{aligned} m &= H(a(Y_2)a^{-1}) \oplus d = H(a(bwb^{-1})a^{-1}) \oplus d \\ &= H(\sigma_3(\sigma_1\sigma_2\sigma_1^{-1})\sigma_3^{-1}) \oplus d = H(\sigma_1(\sigma_3\sigma_2\sigma_3^{-1})\sigma_1^{-1}) \oplus d. \end{aligned}$$

Computing $(\sigma_1(\sigma_3\sigma_2\sigma_3^{-1})\sigma_1^{-1})$ into a tuple and then into binary gives:

```
00110001 00100000 00110011 00100000 00110010 00100000 00101101
00110011 00100000 00101101 00110001
```

Therefore, computing $H(a(Y_2)a^{-1}) \oplus d$ gives:

```
    00110001 00100000 00110011 00100000 00110010 00100000
    00101101 00110011 00100000 00101101 00110001 ⊕ 010010000101
    0100010011000101111101001111000100100110000101000001010101101
    00100001101001000000010000
=   0100100001100101011011000110110001101111001000000100000101 1
    0110001101001011000110110010100100001
=   ‘‘Hello Alice!’’
```

5 Attacks on the Group Cryptosystems

5.1 Summit Set Attacks

Ko-Lee et al. and AAG's protocol both implement the basis of the Conjugacy Search Problem as a trapdoor. Although it seems appropriate to note that neither of these protocols have been proven to use the CSP directly [5]. It is therefore tangible to suggest, the security of these cryptographic protocols rely on the confidence that the conjugacy search problem is not yet solvable in sub-exponential time. However if an algorithm was formulated with the capability to solve the CSP in less than $2^{O(n)}$ time, it is possible these braid group crypto-protocols could be deemed insecure. In fact experimental data has suggested the problem is feasibly solved in polynomial time, although as of yet no algorithm exists! [20]

The basis for one such promising algorithm, endeavouring to solve the conjugacy search problem in $2^{O(n)}$ time, was from work based on Garside's finite subsets in 1969 [8]. Garside formulated an algorithm that could compute a set of conjugates called the *Summit Set*. Garside's raw algorithm was inefficient and the sets created contained far too many elements for solving the conjugacy search problem. However it was a good basis in which to develop smaller sets within the summit set.

In 1994, two mathematicians El-Rifai and Morton [21] published a paper containing an advancement to solving the CSP made by computing a smaller subset of the summit set called the *Super Summit Set (SSS)*. Simply put, the idea suggested associating a finite set of conjugates (with conditions) with every braid then performing the exhaustive search in that level. This significantly weakened the conjugacy search problem and hence weakened the security of using braid group cryptosystems relying on CSP. However the super summit set was still exponential in size with respect to n, hence the time complexity to compute the set was $O(n!)$ [22] thus still not feasible for large n. After some refinement by Franco and González-Meneses in 2003 [23], the subset was further reduced to in size to the *Ultra Summit Set (USS)* - the computation of this set has unknown complexity but is efficient in practice (by [9]). This deems the CSP even more vulnerable, since the subset size is reduced.

Examining in chronological order. Garside's method was to construct two finite subsets $I_x, I_y \in B_n$. For which when $I_x = I_y$, x braid is conjugate to y braid (recall that if x and y are two elements of group G, they are conjugate if there exists an element a such that $axa^{-1} = y$). For notation this can be written $x \sim y$ meaning "x is conjugate to y". I_x is also denoted SS(x) and is called the summit set of x. To construct such sets, we will initially consider constructing the set I_x. First let x be an arbitrary element from the braid group B_n (in the context of attacks, x is an element obtained from the public tuple). For this element x, the objective is to create the subset I_x of the conjugacy class of x. For every $x \in B_n$, the subset I_x must be non-empty, finite and only depends on the conjugacy class of x. There must also be an efficient algorithm to compute a representative $\overline{x} \in I_x$ and a conjugating element a such that $axa^{-1} = \overline{x}$. Noting for the application of crypto-attacks $\overline{x}$ and x are both in the public domain. Finally there must be an algorithm which can construct the whole set I_x from any representative $\overline{x} \in I_x$. More formally $I_x = SS(x)$, the summit set of x is the set of conjugates of x having maximal infimum.

To solve the CDP and CSP for two elements of the braid B_n one can use different algorithms based on this approach:

1. Find the representatives $\overline{x} \in I_x$ and $\overline{y} \in I_y$.

2. As described above, use an algorithm which constructs the whole set for I_x from any representative $\overline{x} \in I_x$ to compute further elements of I_x. While keeping note of the conjugating elements. Similarly, compute the whole set of I_y with the same method also keeping note of these conjugating elements.

3. At this point one should have both complete sets I_x and I_y along with the respected conjugating elements. We can now compare the sets, comparing the elements $\overline{x}$ with the elements in set I_y. There are two possibilities during this comparison, either $\overline{x} \in I_y$ proving $x \sim y$ or, $\overline{x} \notin I_y$ and x and y are therefore not conjugate. Not only are we now able to determine the conjugates (solving the CDP) but because we kept note of the conjugators we also have the respected conjugators (solving the CSP), and in crypto applications this would reveal the private keys.

5.2 Super Summit Set

The super summit set SSS(x), is defined to be the set of all braids $\overline{x} = axa^{-1}$ for every a such that length($\overline{x}$) is minimal. With a computable set, the conjugacy problem is solved as:

$$SSS(x) = SSS(\overline{x}) \iff \overline{x} = axa^{-1}$$

The Super Summit Set is a smaller set than the Summit Set, it has a much greater infimum and smaller supremum. Creating bounds which reduce the size of the Summit Set. Therefore the conjugates of x have minimal canonical length $\operatorname{len}(x)$.

To create super summit sets, we follow a similar procedure as summit sets described in Section 5.1, denoting our $I_x = SSS(x)$. We first require an element $\overline{x} \in SSS(x)$, one then follows a series of *cyclings* and *decyclings* until another element in the SSS(x) is found. Repeated cyclings will achieve the maximum infimum of the set whilst decylings will achieve the minimum supremum, as one requires a set with the greatest infimum and smallest supremum both cycling and decycling can be performed simultaneously. For example, if an element $x \in B_n$ such that $\inf(x)$ is not equal to the maximum infimum in the conjugacy class of x, then performing repeated cyclings will increase the infimum. By cycling we can conjugate x to another element of maximal infimum. Once an element of maximal infimum has been obtained, if the supremum in not minimal in the conjugacy class one can use decycling to reduce its supremum. Simply put, to find SSS(x):

1. Find a conjugate $\overline{x}$ of x ($\overline{x} \in SSS(x)$) and find a conjugate of $\overline{y}$ of y, $\overline{y} \in SSS(y)$ using cycling and decycling. Remembering prior to each iteration of cycling and decycling the Garside normal form needs to be calculated.

2. Starting with the conjugate $\overline{x}$, we compute each further element of SSS(x) using conjugation. If $\overline{x}$ is equal to $\overline{y}$, x and y conjugates have been found.

Definition 5.1. [22] Let $= \mathit{\Delta}^P x_1 \dots x_r \in B_n$ be given in Garside normal form and assume $r > 0$.

The cycling of x, denoted by $\mathbf{c}(x)$ is:

$$\mathbf{c}(x) = \mathit{\Delta}^P x_2 \dots x_r \tau^{-P}(x_1),$$

where τ is the involution which maps σ_i to σ_{n-i}, for all $1 \leq i \leq n$.

The decycling of x, denoted by $\mathbf{d}(x)$ is:

$$\mathbf{d}(x) = x_r \mathit{\Delta}^P x_1 x_2 \dots x_{r-1} = \mathit{\Delta}^P \tau^P(x_r) x_1 x_2 \dots x_{r-1}.$$

If $r = 0$, we have $\mathbf{c}(x) = \mathbf{d}(x) = x$.

Properties: $\mathbf{c}(x) = (\tau^{-P}(x_1))^{-1} x (\tau^{-P}(x_1)),\ \mathbf{d}(x) = x_r x x_r^{-1}$

$$\inf(x) \leq \inf(\mathbf{c}(x)),\ \sup(x) \geq \sup(\mathbf{d}(x))$$

Simply put, these properties mean to cycle an element of positive canonical length one needs to move the first permutation braid x_1 to the end of the word. To decycle an element of positive canonical length one only needs to move the final permutation braid x_r to the start of the word.

5.2.1 Cycling

Cycling of a braid x is usually denoted $\partial_+(x)$. Let $x = \Delta^p x_1 \dots x_r \in B_n$ be given in Garside's normal form, r being the final element assuming $r > 0$. To compute the cycling of x we start by moving the first permutation braid of x to the end of the word, in this case moving x_1 to the end of the word (after x_r) and applying the involution mapping (sometimes called the shift map), σ_i to σ_{n-i} . Therefore the cycling of x is:

$$x = \Delta^p x_2 \dots x_r \tau^{-p}(x_1)$$

Example 8. This example shows a cycling of the element (an element of positive canonical length) $P = \sigma_1\sigma_2^2\sigma_3\sigma_1\sigma_2^2$ with the intention of increasing the infimum. Then the answer is confirmed with a Sage example.

First convert P to left canonical form as detailed in Section 3.3, more simply for this example re-write P element by element i.e. re-write $\sigma_2^2 = \sigma_2\sigma_2$.

$$P = (\sigma_1\sigma_2)(\sigma_2\sigma_3\sigma_1\sigma_2)(\sigma_2);$$

Then $\inf(P) = 0$ and $\sup(P) = 3$. Performing the first cycling, moving x_1 (in our example $\sigma_1\sigma_2$) to the end of the word:

$$\mathbf{c}(P) = (\sigma_2\sigma_3\sigma_1\sigma_2)(\sigma_2)(\sigma_1\sigma_2) = (\sigma_2\sigma_3\sigma_1\sigma_2)(\sigma_2\sigma_1\sigma_2)$$

Next apply the second braid relation swapping $\sigma_2\sigma_1\sigma_2$ with $\sigma_1\sigma_2\sigma_1$, then put into left canonical form:

$$\mathbf{c}(P) = (\sigma_2\sigma_3\sigma_1\sigma_2)(\sigma_1\sigma_2\sigma_1) = (\sigma_2\sigma_3\sigma_1\sigma_2\sigma_1)(\sigma_2\sigma_1)$$

Then the $\inf(\mathbf{c}(P)) = 0$ and $\sup(\mathbf{c}(P)) = 2$. After one further cycling: check the braid is in left canonical form, then as before moving x_1 (in our example $\sigma_2\sigma_3\sigma_1\sigma_2\sigma_1$) to the end of the word:

$$\mathbf{c^2}(P) = (\sigma_2\sigma_1)(\sigma_2\sigma_3\sigma_1\sigma_2\sigma_1)$$

Then, for this example, re-write into left canonical form by completing the following actions. Re-arrange the permutation braids:

$$(\sigma_2\sigma_1\sigma_2)(\sigma_3\sigma_1\sigma_2\sigma_1)$$

Apply the second braid relation swapping $\sigma_2\sigma_1\sigma_2$ with $\sigma_1\sigma_2\sigma_1$:

$$(\sigma_1\sigma_2\sigma_1)(\sigma_3\sigma_1\sigma_2\sigma_1)$$

Swap σ_3 and σ_1 using the first braid relation to give $(\sigma_1\sigma_2\sigma_3)(\sigma_1\sigma_1\sigma_2\sigma_1)$. Finally, perform the second braid relation swapping $\sigma_1\sigma_2\sigma_1$ with $\sigma_2\sigma_1\sigma_2$:

$$(\sigma_1\sigma_2\sigma_3\sigma_1\sigma_2\sigma_1)(\sigma_2) = \Delta_4\sigma_2$$

Now in left canonical form one can see $\inf(\mathbf{c^2}(P)) = 1$ and $\sup(\mathbf{c^2}(P)) = 2$.

```
Input: p = BraidGroup(4)
Input: P = p([1, 2, 2, 3, 1, 2, 2])
Input: P.left_normal_form()
Output: [1, s0*s1, s1*s0*s2*s1, s1]

Input: P.super_summit_set()
Output: [s0*s1*s0*s2*s1*s0*s1]
```

Note that if the infimum is not maximal in the conjugacy class (as was the case in Example 8), we can cycle elements, in which there exists a positive integer k_1 such that $\inf(\mathbf{c}^{\mathbf{k_1}}(x)) > \inf(x)$. If the supremum is not minimal in the conjugacy class, we can decycle elements, in which there exists a positive integer k_2 such that $\sup(\mathbf{d}^{\mathbf{k_2}}(x)) > \sup(x)$. Applying this to Example 8, note $\inf(\mathbf{c}^{\mathbf{2}}(P)) > \inf(P)$ our intention.

Proposition 5.1. [22] A sequence of at most rm cyclings and decyclings applied to x produces a representative $\overline{x} \in SSS(x)$. Where m is the length of $\varDelta$ in Artin generators and r is the canonical length of x.

5.2.2 Decycling

Decycling of a braid x is usually denoted $\partial_{-}(x)$. Let $x = \varDelta^p x_1 \dots x_r \in B_n$ be given in Garside's normal form, where $r > 0$. To compute the decycling of x start by moving the last permutation braid of x to the front of the word, in this case moving x_r to the start of the word (before $\varDelta^p x_1$). Therefore the decycling of x is:

$$x = x_r \varDelta^p x_1 \dots x_{r-1}$$

Alternatively, one may want to return the product of a decycling in left normal form. For which case move x_r in front of the fundamental braid and then applying the involution τ to x_r. Therefore the decycling of x can also be obtained as:

$$x = \varDelta^P \tau^P(x_r) x_1 x_2 \dots x_{r-1}$$

5.2.3 The Cycling and Decycling Combination

To summarise creating the super summit set: Given x find an element in the SSS(x); then compare the infimum and supremum of $x \in B_n$ with the maximum infimum in the conjugacy class of x and the minimum supremum in the conjugacy class of x. If the $\inf(x)$ is not equal to the infimum of x in the conjugacy class then perform a cycling or multiple cyclings to increase the $\inf(x)$. By performing these cyclings one can cycle to another element $\hat{x}$ of maximal infimum. After a finite number of cyclings we obtain an element in the SSS(x).

If the $\sup(x)$ of this element is not minimal in the conjugacy class of x perform decyclings to a similar effect as cycling, however instead of increasing $\inf(x)$ we shall be decreasing $\sup(x)$. Apply both cycling and decycling for elements that do not have maximum infimum in the conjugacy class of x and do not have the minimum supremum in the conjugacy class of x, to obtain an element that does meet these two conditions (an element in the SSS(x)). Note the decompositions of cycling and decycling are not, in general, Garside normal forms and hence the left normal form should be computed after every iteration.

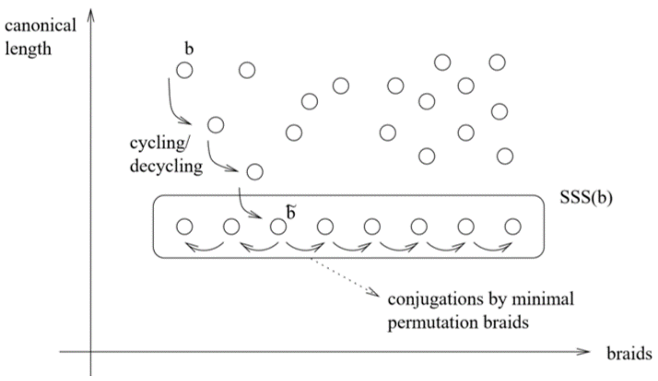


Figure 8: Geometric interpretation of cycling and decycling, from [22].

From Figure 8, starting with a braid b we can perform cycling/decycling to reduce its canonical length and hence its complexity. If b is not a braid that belongs to SSS(b) then it will not have minimal complexity/length and cycling/decycling can be performed. Eventually after a finite sequence of cyclings/decyclings we end up with a minimal complexity/length braid $\bar{b}$ which lies in SSS(b).

Once braid $\bar{b}$ is found, one needs to find the rest of the set SSS(b) by an exhaustive search of b under conjugation by simple braids a. This is done by considering all conjugates aba^{-1}; when a braid that has the same complexity as b is found we keep it and add it to SSS(b).

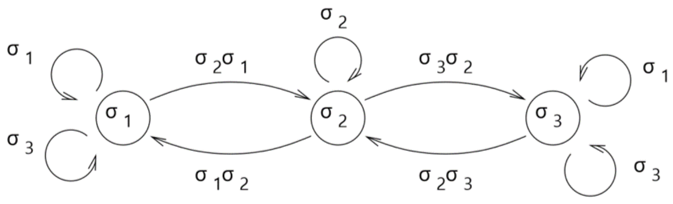


Figure 9: Geometric interpretation of cycling and decycling, the graph of SSS(σ_1) in B_4, from [22].

Example 9. This example shows a comparison in size of the two sets SS(x) and SSS(x)for the element $x = \Delta_4\sigma_1\sigma_1 \in B_4$. The summit set of x is:

$$SS(x) = \{\Delta_4 \cdot \sigma_1\sigma_3, \Delta_4 \cdot \sigma_1 \cdot \sigma_1, \Delta_4 \cdot \sigma_3\sigma_3\}.$$

However, the super summit set of x is:

$$SSS(x) = \{\Delta_4 \cdot \sigma_1\sigma_3\}.$$

5.3 Ultra Summit Set

Although SSS(x)) is a considerably smaller subset of the SS(x), there exists a subset that contains fewer elements than the SSS(x)). This set is called the Ultra Summit Set (USS(x)) this is the set where $\sup(SSS(x) \geq \sup(USS(x))$ and $\inf(SSS(x)) \leq \inf(USS(x))$. The USS($x$) consists of the conjugates of x in the SSS(x), which satisfy $\mathbf{c}^{k_1}(y) = y$ for some integer $k_1 > 0$. The USS(x) consists of a finite set of disjoint orbits, closed under cycling, decycling and the operator τ. For clarification when a group G

acts on a set x it permutes the element x so that x moves around in a fixed circular path called an orbit. For example, from the permutation group $G =< (123) >$ we can obtain the orbit $\{$(123),(132),e$\}$. Any further permutations would return a previously obtained element. Formally in set theory one can define an orbit:

Definition 5.2. Let G be a group acting on a set X. The orbit of an element $x \in X$ is defined as

$$\mathrm{Orb}(x) := \{y \in X : \exists g \in G : y = g * x\},$$

where * denotes the group action. That is, $\mathrm{Orb}(x) = G{*}x$. Thus the orbit of an element is all its possible destinations under the group action.

To find an element in the USS(x)we use an analogous algorithm to the super summit set algorithm. However to compute the USS rather than the SSS take the element $\overline{x} \in USS(x)$ and perform cyclings. When cycling the element, two integers k_1 and k_2 can be obtained where $k_1 < k_2$, which satisfy

$$\mathbf{c}^{k_1}(\overline{x}) = \mathbf{c}^{k_2}(\overline{x}).$$

Then $\hat{x} = \mathbf{c}^{k_1}(\overline{x}) \in USS(x)$, since $\mathbf{c}^{k_2-k_1}(\hat{x}) = \hat{x}$. The element $\overline{x}$ must be equal when k_1 cyclings have been performed to $\overline{x}$ as to when k_2 cyclings have been performed to $\overline{x} \in SSS(x)$. When this property has been satisfied we have an element of the USS, $\hat{x} \in USS(x)$. Each element is an orbit under cycling. It is therefore clear, each member of the USS are the elements of the SSS that are enclosed in orbit under cycling. The reason iterated cycling of any representative of SSS(x) and USS(x) must eventually become periodic is because the set SSS(x) and USS(x) is finite. Concerning the complexity of this algorithm, the number of times one needs to cycle (K_2) until obtaining an element in the USS is not yet known.

Definition 5.3. [22] Given $x \in B_n$, $y \in USS(x)$. A permutation braid $s \neq 1$ is *minimal* for y with respect to USS(x) if $s^{-1}ys \in USS(x)$, and no proper prefix of s satisfies this property.

Algorithm 4.71. is from Gebhardt (2005) [24] of an algorithm computing the Ultra summit set U_x of x.

Algorithm 4.71. *Given an element x of a Garside group.*

```
Compute $\overline{x} \in U_x$, set $U = T_x$ and $U_0 = U$.
If $\overline{x} = \delta^k$ for some K then
return $\{\delta^k\}$
end if
While $U \neq U_0$ do
Let $y_1, \ldots, y_m \in U$ such that $U = U_0 \cup T_{y1} \cup \ldots \cup T_{ym}$. Set $U_0 = \emptyset$
for $y \in \{y_1, \ldots, y_m\}$ do
Compute $C_y$ and set $U = U \cup U_{c \in c_y} T_{y^c}$
end for
end while
return U
```

As discussed in section 5.1, if $U_x = U_y$ the two elements x and y of G are conjugate in G. Or if $U_x \cap U_y \neq \emptyset$.

Example 10. This example taken from Gebhardt [24] demonstrates a comparison of size between the SSS and USS of a braid. We will then confirm this with Sage. Let $x = \sigma_1\sigma_3\sigma_2\sigma_1 \cdot \sigma_1\sigma_2 \cdot \sigma_2\sigma_1\sigma_3 \in B_4$. Then $|USS(x)| = 6$ while $|SSS(x)| = 22$.

```
Input: X = BraidGroup(4)
Input: x = X([1, 3, 2, 1, 1, 2, 2, 1, 3])
Input: x.super_summit_set()
Output: [s0*s2*s1*s0^2*s1^2*s0*s2, s1*s2*s1*s0^2*s1*s2^2*s1,
 (s0*s1)^2*s0*s2^2*s1*s0, s0*s1*s0^2*s1*s2^2*s1*s0,
 s1*s0*s2*s0*(s1*s2)^2*s1, s0*s1*(s0*s2)^2*s1*s0^2,
 s0*s1^2*(s0*s2)^2*s1*s0, s0*(s1*s2*s1)^2*s1*s0,
 s0*s1*s2*s1^2*s2^2*s1*s0, s0*s2*(s1*s0)^2*s0*s1*s2,
 s0*s1*s0*s2^2*s1*s0^2*s1, (s1*s2*s1)^2*s0^2*s1,
 s1*(s0*s2)^2*s1*s0^2*s1, s0*(s1*s2)^2*s1^2*s0*s2,
 s0*s1*(s0*s2)^2*s1^2*s0, s0*s2*s1*s0*(s0*s1)^2*s2,
 s1*s2*s1*s0^2*s2*s1^2*s2, (s0*s1*s0)^2*s2^2*s1,
 (s1*s2)^2*s1*s0^2*s1*s2, s1*s2*s1^2*s0*s2*s0*s1*s2,
 s2*s1^2*s0*s2*s0*s1*s2*s1, s1*s2*s1*s0^2*(s1*s2)^2]

Input: len(x.super_summit_set())
Output: 22

Input: x.ultra_summit_set()
Output: [[s0*s2*s1*s0^2*s1^2*s0*s2, s0*s1^2*(s0*s2)^2*s1*s0,
s1*(s0*s2)^2*s1*s0^2*s1], [s0*(s1*s2)^2*s1^2*s0*s2,
s2*s1^2*s0*s2*s0*s1*s2*s1, s1*s0*s2*s0*(s1*s2)^2*s1]]

Input: len(x.ultra_summit_set())
Output: 6
```

Example 11. This example considers a property of the USS(x) Again let $x = \sigma_1\sigma_3\sigma_2\sigma_1 \cdot \sigma_1\sigma_2 \cdot \sigma_2\sigma_1\sigma_3 \in B_4$.

Note that USS(x) consists of two closed orbits under cycling ($USS(x) = O_1 \cup O_2$). Each orbit contains 3 rigid elements, rigid meaning that the left normal form changes only in the obvious way under cycling and decycling.

$$O_1 = \{\sigma_1\sigma_3\sigma_2\sigma_1 \cdot \sigma_1\sigma_2 \cdot \sigma_2\sigma_1\sigma_3, \sigma_1\sigma_2 \cdot \sigma_2\sigma_1\sigma_3 \cdot \sigma_1\sigma_3\sigma_2\sigma_1, \sigma_2\sigma_1\sigma_3 \cdot \sigma_1\sigma_3\sigma_2\sigma_1 \cdot \sigma_1\sigma_2\},$$

$$O_2 = \{\sigma_3\sigma_1\sigma_2\sigma_3 \cdot \sigma_3\sigma_2 \cdot \sigma_2\sigma_3\sigma_1, \sigma_3\sigma_2 \cdot \sigma_2\sigma_3\sigma_1 \cdot \sigma_3\sigma_1\sigma_2\sigma_3, \sigma_2\sigma_3\sigma_1 \cdot \sigma_3\sigma_1\sigma_2\sigma_3 \cdot \sigma_3\sigma_2\}.$$

Note the property that $O_2 = \tau(O_1)$. For example if we take the first element of O_1, $\sigma_1\sigma_3\sigma_2\sigma_1$ and apply the involution (τ) which maps σ_i to σ_{n-i}, for all $1 \leq i \leq n$. In this example $n = 4$. The first element of O_1 becomes $\sigma_3\sigma_1\sigma_2\sigma_3$, i.e., the first element of O_2.

Example 12. This final example brings together the USS content and performs a USS attack on the AAG system. A considerable difference from this attack to one which would be implemented in a real world scenario is that the braid group size in this example is very small (B_5) making it easier to find the SSS and USS.

Assume Alice and Bob intend on sending secure messages using AAG's protocol and have performed the necessary algorithm for an AAG protocol (simulated by the Sage code given in the appendix).

First the attacker (Eve), captures the public tuple of conjugates from Alice (or Bob) denoted A_conj and she captures Alice's public tuple, Bt. This process is trivial as Bt and A_conj are in the public domain. Eve selects a single conjugate from A_conj which we denote A_conjx. Eve starts by computing the USS of A_conjx and the USS of Bt. It is important to note that each element of Bt is scrambled for the additional level of security and thus it is not obvious from inspection the element Bt[i] for any $0 \leq i \leq N-1$. After computing the USS of Bt and A_conjx as per the algorithm:

```
Input = I_x = A_conjx.ultra_summit_set()
        print('The USS of A_conjx below')
        print(I_x)
        I_y = Bt[N-1].ultra_summit_set()
        print('The USS of Bt below')
        print(I_y)
Output = The USS of A_conjx below:
         [[s0^-1*s1^-1*s2^-1*s3^-1*s0^-1*s1^-1*s2^-1* s0^-1
         *s1^-1*s0^-1*s2*s1*s0*s3*s2*s1*s0, s0^-1*s1^-1
         *s2^-1*s3^-1*s0^-1*s3*s2],[s0^-1*s1^-1*s2^-1*s3^-1
         *s0^-1*s1^-1*s2^-1*s0^-1*s2*s3*s2*s1*s0]]
         The USS of Bt below:
         [[s0^-1*s1^-1*s2^-1*s3^-1*s0^-1*s3*s2, s0^-1*s1^-1
         *s2^-1*s3^-1*s0^-1*s1^-1*s2^-1*s0^-1*s1^-1*s0^-1*s2
         *s1*s0*s3*s2*s1*s0],[s0^-1*s1^-1*s2^-1*s3^-1*s0^-1
         *s1^-1*s2^-1*s0^-1*s2*s3*s2*s1*s0]]
```

After some period of time (which depends on the complexity of the braid), Eve should have the two ultra summit sets, I_x and I_y. Eve then applies a function that returns the intersection of the two sets. Eve should then detect one element $(\overline{x})$ in each set being identical; i.e., $\overline{x} \in I_x$ and I_y.

```
Input: it = [x for x in E if tuple(x) in set(map(tuple, F))]
       print(it)
Output: [[s0^-1*s1^-1*s2^-1*s3^-1*s0^-1*s1^-1*s2^-1*s0^-1*s2
        *s3*s2*s1*s0]]
```

As the intersection of the two USS's has been computed to be non-empty, we have located an element in Bt. To find the conjugators we use a similar algorithm to the one above. However, this time instead of discarding the conjugators we remember and return them as A_priv. Then compute the inverse, A_privinv.

```
Input: A_priv = A_conjx.conjugating_braid(Bt)
       A_priv
Output: s0*s1*s2*s1*s3*s2*s1*s0^2*s2*s1*s0*s3*s2*s1*s0^2*
        (s1*s0*s2*s1*s3)^2*s2*s1*s0*(s0*s1*s2*s1*s0*s3*s2)^2
        *s1*s0^2*s1*s0*s2*s1*s3*s2*s1*s0^2*s1*s2*s1*s0*s3*s2
        *s1*s0^2*s1*s0*s2*s1*s3*s2*s1*(s1*s0*s2*s3*s2*s1*s0
        ^2)^3*s1*s0*s2*s1*s3*s1*s0*s2*s3*s2*s1*s0*s1*s2*s3
Input: A_privinv = (A_priv)^-1
Output: s3^-1*s2^-1*s1^-1*s0^-1*s1^-1*s2^-1*s3^-1*s2^-1*s0
        ^-1*s1^-1*s3^-1*s1^-1*(s2^-1*s0^-1*s1^-1*s0^-2*s1^-1
        *s2^-1*s3^-1)^3*s2^-1*s0^-1*s1^-2*s2^-1*s3^-1*s1^-1
        *s2^-1*s0^-1*s1^-1*s0^-2*s1^-1*s2^-1*s3^-1*s0^-1*s1
        ^-1*s2^-1*s1^-1*s0^-2*s1^-1*s2^-1*s3^-1*s1^-1*s2^-1
        *s0^-1*s1^-1*s0^-2*s1^-1*(s2^-1*s3^-1*s0^-1*s1^-1*s2
        ^-1*s1^-1*s0^-1)^2*s0^-1*s1^-1*s2^-1*(s3^-1*s1^-1*s2
        ^-1*s0^-1*s1^-1)^2*s0^-2*s1^-1*s2^-1*s3^-1*s0^-1*s1
        ^-1*s2^-1*s0^-2*s1^-1*s2^-1*s3^-1*s1^-1*s2^-1*s1^-1
        *s0^-1
```

Finally we shall check we have obtained the valid private keys:

```
Input: A_privinv * Bt * A_priv == A_conjx
Output: True
```

We have obtained Alice's private keys and therefore successfully attacked and broken the AAG cryptosystem with small parameters using the USS attack. For the SSS attack the same procedure applies. However, we choose not to demonstrate a SSS attack as there are a large number of elements in this set, making it impractical to list the SSS. See Section 6.1.1 for details of general sizes of super summit sets under various parameters.

5.4 The Length Based Attack

In this section the length based attack (LBA) on the AAG protocol is considered. As discussed in Subsection 4.2.1, the AAG protocol uses the subgroup-restricted simultaneous conjugacy problem (SR-SCSP), a variation of the simultaneous conjugacy search problem. AAG therefore relies on the assumption that the SR-SCSP is hard, although this is yet to be proven.

Interestingly, although in previous subsections attacks have been discussed, the only attack that breaks the SR-SCSP directly is the length based attack. LBA is considered an heuristic computer science attack for SR-SCSP, whilst computing the sets SSS(x) or USS(x) is a mathematical attack aimed at breaking SCSP [25]. This attack was first introduced by Hughes and Tannenbaum in [26]. After several modifications it has been shown to break AAG protocol (with the original parameters) with a high success rate.

In this section evidence is also presented for a more vigilant approach in selecting private keys, this is necessary for AAG to be unsusceptible to the length based attack, thus showing keys generated uniformly randomly are insecure.

The LBA is a heuristic procedure for finding Alice and Bob's private keys. In this subsection the notation of Subsection 4.2.1 is applied to keep some level of consistency throughout this paper. Therefore recall Alice and Bob's public N-tuple $(a_i)_{i=1}^N = (a_1, \ldots, a_N)$ and $(b_i)_{i=1}^N = (b_1, \ldots, b_N)$. Alice and Bob also choose a private key for example $A = a_{\mu_1}^{\epsilon_1} a_{\mu_2}^{\epsilon_2} \ldots a_{\mu_L}^{\epsilon_L}$ and $B = b_{\mu_1}^{\epsilon_1} b_{\mu_2}^{\epsilon_2} \ldots b_{\mu_L}^{\epsilon_L}$. Finally recall Alice then computes the tuple of conjugates $A^{-1}(b_i)_{i=1}^N A = \{A^{-1}b_1A, \ldots, A^{-1}b_NA\} = \{b'_1, \cdots b'_{N2}\} = \overline{b'}$ and transmits this publicly to Bob. Bob also computes the tuple of conjugates $B^{-1}(a_i)_{i=1}^N B = \{B^{-1}a_1B, \ldots, B^{-1}a_NB\} = \{a'_1, \cdots a'_{N2}\} = \overline{a'}$ and transmits to Alice. Note each $\overline{b'}$ is a sequence of conjugations of b_i by the conjugator (private key) A:

$$\begin{array}{c} b_i \\ \downarrow \\ a_{\mu_1}^{-\epsilon_1} \;\; b_i \;\; a_{\mu_1}^{\epsilon_1} \\ \downarrow \\ a_{\mu_2}^{-\epsilon_2} a_{\mu_1}^{-\epsilon_1} \;\; b_i \;\; a_{\mu_1}^{\epsilon_1} a_{\mu_2}^{\epsilon_2} \\ \downarrow \\ \cdots \\ \downarrow \\ b'_i = a_{\mu_L}^{-\epsilon_L} \ldots a_{\mu_2}^{-\epsilon_2} a_{\mu_1}^{-\epsilon_1} \;\; b_i \;\; a_{\mu_1}^{\epsilon_1} a_{\mu_2}^{\epsilon_2} \ldots a_{\mu_L}^{\epsilon_L} \end{array} \tag{10}$$

The conjugating sequence is the same for each b_i - in fact, the only element that differs is b_i. Clearly, if we were given the bottom row of (10) and had an algorithm which could reverse the sequence back to the top by peeling away generator after generator we could recover b_i and the conjugator A (private key). This is the intention of the length based attack. More formally:

$$\text{for elements } a, b \in B_n \;\; l(a^{-1}ba) > l(b). \tag{11}$$

Here l represents a length function which will allow one to set the conjugating element to a minimisation problem and solve using a heuristic optimisation method. There are multiple length functions available, at present the most suitable seems to be the geodesic length (the length of the shortest path in the corresponding Cayley graph) function denoted by $|\cdot|$. This seems to be the best candidate because although there is no known efficient algorithm for computing $|\cdot|$ for practical purposes one can approximate $|\cdot|$ using a method proposed in [27]. It is clear that as the length of a and b grows then $2|a| = |b| - |a^{-1}ba|$ which is smaller than $2|a|$ this also means that $|a^{-1}ba| > |b|$ and the difference is large.

The other choice that needs to be made is the heuristic approach/algorithm used. There are several to choose from each with their own advantages and disadvantage. One of these algorithms is defined below [25].

Here $\overline{c} = (c_1, \ldots, c_k)$ is an arbitrary tuple of braids and $|\overline{c}|$ is its total length i.e. $\Sigma_{i=1}^k |c_i|$. The algorithm below enumerates all possible sequences of conjugations decreasing the length of a tuple. We maintain set S which contains tuples in the algorithm.

Algorithm 6.4. *LBA with backtracking [25]*

1) Initialise a set $S = \{(\overline{b'}, e)\}$, where e is the identity of B_n.

2) **If** $S = \emptyset$ **then output** FAIL.

3) **Choose** a pair $(\bar{c}, x) \in S$ with minimal $|\bar{c}|$. **Remove** $(\bar{c}, x)$ from S.

4) **For** each $i = 1, \ldots, N_1$, $\varepsilon = \pm 1$ compute $\delta_{i,\epsilon} = |\bar{c}| - |\bar{c}^{a_a^\epsilon}|$.

5) **If** $\delta_{i,\epsilon} > 0$ **then** add $(\bar{c}^{a_a^\epsilon}, xa_i^\epsilon)$ into S.

6) **If** $\bar{c}^{a_a^\epsilon} = \bar{a}$ **then** output xa_i^ϵ.

7) Otherwise goto step 2.

5.4.1 Peaks

Unfortunately condition (11) is not the only condition needed to satisfy the success of all LBA's. Below is an example where this condition alone is not satisfactory, taken from [25].

Example 13. Consider B_{80} and two braids $a_1 = \sigma_{39}^{-1}\sigma_{12}\sigma_7\sigma_3^{-1}\sigma_1^{-1}\sigma_{70}\sigma_{25}\sigma_{24}^{-1}$ and $a_2 = \sigma_{42}\sigma_{56}^{-1}\sigma_8\sigma_{18}^{-1}\sigma_{19}\sigma_{73}\sigma_{33}^{-1}\sigma_{22}^{-1}$, which we think of as elements from Alice's public set. It is easy to check that

$$a_1^{-1}a_2^{-1}a_1 = \sigma_7^{-1} \cdot a_2^{-1} \cdot \sigma_7 = \sigma_7^{-1} \cdot \sigma_{22}\sigma_{33}\sigma_{73}^{-1}\sigma_{19}^{-1}\sigma_{18}\sigma_{56}\sigma_{42}^{-1} \cdot \sigma_7$$

and

$$a_1^{-1}a_2^{-1}a_1a_2 = \sigma_7\sigma_8^{-1}.$$

Hence $|a_1| = 8$, $|a_1^{-1}a_2^{-1}| = 16$, $|a_1^{-1}a_2^{-1}a_1| = 10$ and $|a_1^{-1}a_2^{-1}a_1a_2| = 2$. Now let $\bar{b} = (b_1, \ldots, b_N)$ be a random tuple of braids thought of as Bob's public set. As seen before, for the majority of the braids conjugation increases the length by almost twice the length of the conjugator. Hence, for generic tuple $\bar{b}$ the following length growth would be expected:

$$\begin{array}{c}
\bar{b} \\
\downarrow \\
|a_{s_1}^{-\epsilon_1} \;\; \bar{b} \;\; a_{s_1}^{\epsilon_1}| \approx |\bar{b}| + 8N \\
\downarrow \\
|a_{s_2}^{-\epsilon_2}a_{s_1}^{-\epsilon_1} \;\; \bar{b} \;\; a_{s_1}^{\epsilon_1}a_{s_2}^{\epsilon_2}| \approx |\bar{b}| + 16N \\
\downarrow \\
|a_{s_3}^{-\epsilon_2}a_{s_2}^{-\epsilon_2}a_{s_1}^{-\epsilon_1} \;\; \bar{b} \;\; a_{s_1}^{\epsilon_1}a_{s_2}^{\epsilon_2}a_{s_2}^{\epsilon_3}| \approx |\bar{b}| + 10N \\
\downarrow \\
|a_{s_4}^{-\epsilon_4}a_{s_3}^{-\epsilon_3}a_{s_2}^{-\epsilon_2}a_{s_1}^{-\epsilon_1} \;\; \bar{b} \;\; a_{s_1}^{\epsilon_1}a_{s_2}^{\epsilon_2}a_{s_2}^{\epsilon_3}a_{s_4}^{\epsilon_4}| \approx |\bar{b}| + 2N
\end{array} \tag{12}$$

Clearly, the length based attack fails for such an element A because to guess the first correct conjugator it is required to increase tuple length substantially (from $|\bar{b}|+2N$ to $|\bar{b}| + 10N$). The reason for the attack failure in the previous example is that Alice's private key (a_1, a_2) forms a peak. The formal definition of a peak is as follows:

Definition 5.4. [25] (Peak) Let $G = \langle X; R\rangle, l_G$ a length function on G, and $H = \langle w_1, \ldots, w_k\rangle$. We say that a word $w = w_{i1} \ldots w_{in}$ is an n-peak in H relative to l_G if there is no $1 \le j \le n-1$ such that

$$l_G(w_{i1} \ldots w_{in}) \le l_G(w_{i1} \ldots w_{ij}).$$

We say that $w = w_{i1} \dots w_{in}$ is m-hard if there exist $s \in \{1, \dots, n\}$ such that for each $j = 1, \dots, k$,

$$l_G(w_{i1} \dots w_{is+k1}) \leq l_G(w_{i1} \dots w_{is+kj})$$

and m is the maximal number with such property.

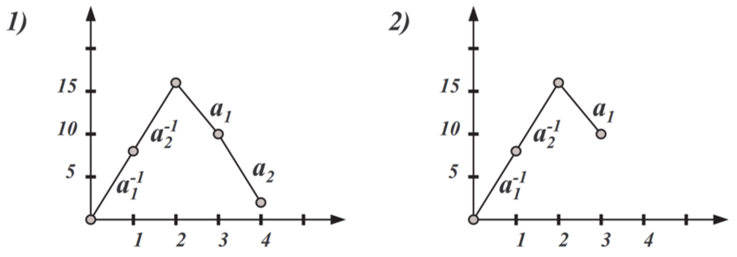


Figure 10: 1) Commutator-type 4-peaks $[a_1, a_2]$ from Example 13
2) Conjugator-type 2-peak as in Example 13 for $a_1^{-1} a_2^{-1} a_1$.

6 Analysis of AAG and Ko-Lee

This section examines the security of two protocols (AAG and Ko-Lee) considering the super summit, ultra summit and length based attacks. With regards to research papers and experiments, from self-engineered code in Sage, the complexity and conditions required to provide secure protocols can be examined. Finally this section makes suggestions as to how the protocols can be securely modified with current information.

6.1 Assessment of Ko-Lee and AAG Against Attacks

As discussed in Section 4, the word problem in a braid group can be solved by transforming an arbitrary word into its canonical form. The complexity required to transform a word into a canonical form is $O(|W|^2 n)$, where $|W|$ is the length of the word in B_n [15]. This quadratic complexity results in the calculation of the canonical form of a braid being practical for computers. Further, a left weighted canonical form allows the computer to handle the data more efficiently and protects the conjugates from being recovered by inspection. The braid group is also a group with super-polynomial growth which prevents against brute force attacks. It therefore seems braid groups could be practically used for future cryptography. Finally the last criterion a secure cryptosystem should yield is it contains a hard trap door problem. A weak trap door problem with low complexity would result in an ineffective cryptographic protocol.

AAG is based on the Subgroup-Restricted Simultaneous Conjugacy Search Problem (SR-SCSP), a variation of the Simultaneous Conjugacy Search Problem (SCSP), and hence the security of AAG is partially based (but not equivalent to) on the assumption that SR-SCSP is hard. The work [17] does not contain any information on theoretical operating characteristics of the PKC although implementation code of the AAG protocol, allows one to estimate its security based on various attacks.

s	p	Braid index n	Expected message length k	Ellapsed time t (sec)	Kbits/(sec)	Hardness of brute force attack
11	5	50	1071	0.0112	93.4	251
11	5	70	1662	0.0210	77.3	399
11	5	90	2294	0.0344	65.2	559
17	7	50	1499	0.0173	84.6	418
17	7	70	2327	0.0325	69.9	665
17	7	90	3212	0.0532	59.0	931
32	12	50	2570	0.0326	77.0	837
32	12	70	3989	0.0611	63.8	1329
32	12	90	5507	0.1037	51.9	1863

Table 3: Performance of the left canonical form conversion algorithm [5].

With regards to Ko-Lee's protocol the hard problem is based on (although not equivalent) to the Generalised Conjugacy Search Problem but with current information one can assume that the hard problem has the security of the GCSP and that the GCSP is equivalent in security to the CSP. With regard to the complexity of this protocol, there are two parameters of interest: p (canonical length), n (braid index). The message length is $pn\log(n)$ and the encryption and decryption are computed in $O(p^2 n\log(n))$ operations. Finally the security level against brute force attacks was determined to be $O((n!)^p) = O(\exp(pn\log(n)))$ from which the parameters n and p rapidly increase the security level (although the speed is quadratic in p and linear in $n\log n$, and so increasing n rather than p increases the security level without sacrificing computational speed [5]). Refer to Table 3 for further information on brute force attack hardness with respect to parameter sizes.

Plaintext block	$pn\log n$ bits
Ciphertext block	$4pn\log n$ bits
Encryption speed	$\mathcal{O}(p^2 n\log n)$ operation
Decryption speed	$\mathcal{O}(p^2 n\log n)$ operation
Message expansion	4-1
Private key length	$\frac{1}{2}pn\log n$ bits
Public key length	$3pn\log n$ bits
Hardness of brute force attack	$(\frac{n}{2}!)^p \sim \exp(\frac{1}{2}pn\log n)$

Table 4: The operating characteristics of the Ko-Lee PKC [5].

Concerning standards by which to compare the two protocols, we will compare AAG and Ko-Lee with current encryption methods such as RSA. The encryption and decryption algorithms of RSA are polynomial $O(b^3)$, the word problem for RSA cryptography is not required and the RSA brute force attack hardness is exponential $O(2^b)$ (where b is the number of bits in n). The best published asymptotic running time algorithm for solving the integer factorisation problem upon which RSA is based is the general number field sieve (GNFS) algorithm, which, for a b-bit number n, is:

$$O\left(exp\sqrt[3]{\frac{64}{9}b(logb)^2}\right) \tag{13}$$

Providing a sub-exponential solution to the integer factorisation problem [4]. With that considered it seems appropriate so far to use AAG and Ko-Lee protocols for encryption as the encrypt and decrypt times are both manageable sizes for a computer just as RSA however AAG and Ko-Lee protocols have greater than exponential (factorial) brute force attack sizes. This then suggests why AAG and Ko-Lee may be more secure than RSA through a brute force complexity perspective and as computer performance advances more secure encryption methods like AAG and Ko-Lee may be required to replace lower brute force complexity methods. However this high level of security complexity only stands effective if the trap door hard problems for these protocol's are equally secure.

To conclude brute force attacks are not practicable against AAG and Ko-Lee which leaves one other option to successfully attack AAG and Ko-Lee and deem both these protocols insecure; that is to attack the hard problem successfully. The following subsections contain attacks which intend to do just that, they attack the various trap door hard problems embedded within the cryptosystem.

6.1.1 Super Summit Attack on Ko-Lee and AAG

First it is important to consider that the multiple conjugacy problem that AAG relies on may be easier than the original conjugacy search problem [22].

The SSS uses an algorithm to compute the sets by cycling and decyclying elements. The number of cyclings and decyclings required by the SSS algorithm to solve the conjugacy problem is proportional to the canonical length of the braid, therefore the number of steps in this algorithm is linear in the complexity of the braid. Although the number of simple braids of length n is $n!$, so the cost to enumerate all simple braids grows faster than exponential, hence creating an exponentially large set. This results in a non-feasible algorithm for large values of n, which we confirm from Figure 11. The graph shows the means of the super summit set size and time to compute the SSS for different values of n from the experiment.

For this paper many experiments were performed using the Plymouth University high-performance computer (HPC cluster). After implementing the AAG protocol in Sage computer programming language, experimentation's by changing the parameters and calculating the time and size of the ultra summit and super summit set were conducted. As expected the experiments were held back by the slow time it took to calculate the SSS and due to this we were not able to gather huge amounts of data for the SSS in the limited time available. However it can be confidently concluded that the SSS is not suitable for attacking AAG under a large n parameter in almost all cases.

Interestingly there was a single case of AAG under the original parameters that the SSS managed to successfully attack although it should be stressed that this was an isolated case likely due to a randomly generated trivial braid choice (this brings into question the importance of randomly generated braids and setting conditions on braids generated in this way). It is therefore important to note that in rare cases the SSS may be able to break AAG but from a probabilistic perspective SSS is very unlikely to break AAG under its original parameters. By Figure 11 the complexity of SSS can be estimated to be exponential or greater. Given more time one may be able to improve this experiment by running the SSS for a greater period of time or/and utilise the parallel computation capability on the HPC cluster.

Due to the poor efficiency of computing the SSS (the set being exponentially large) for large n it is appropriate to rule out the SSS as a feasible AAG attack. With regards to a SSS attack against Ko-Lee's protocol, we refer to an extract from Ko-Lee's work: "The adversary may try to use a mathematical solution to the conjugacy problem by Garside, Thurston, Elrifai-Morton and Birman-Ko-Lee. But the known algorithms find an element $a \in B_{l+r}$, not in LB_l. Hence the attack using the super summit set will not be successful." [5]

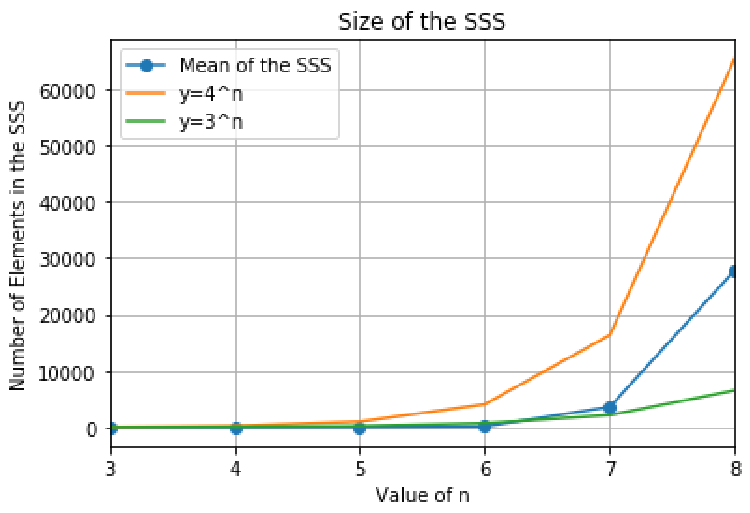


Figure 11: SSS estimated complexity.

Subject to this assessment we can assume AAG and Ko-Lee are both secure against SSS attacks (for large values of n when applying AAG).

6.1.2 Ultra Summit Attack on Ko-Lee and AAG

Figure 11 demonstrates the SSS having exponential complexity and is therefore unsuitable for attacks however the ultra summit set is much smaller than the super summit set in practice (although this has not yet been mathematically proven) [22] and may lead to a suitable solution to the CSP. Furthermore, Figure 12 from an experiment comparing the size of the SSS and USS also shows further evidence of the USS being a much smaller set than the ultra summit set. Unfortunately due to a slow SSS compute time we have limited data up to $n = 8$. This experiment could be improved by increasing the time spent performing the experiments or making use of more than one core on the HPC cluster.

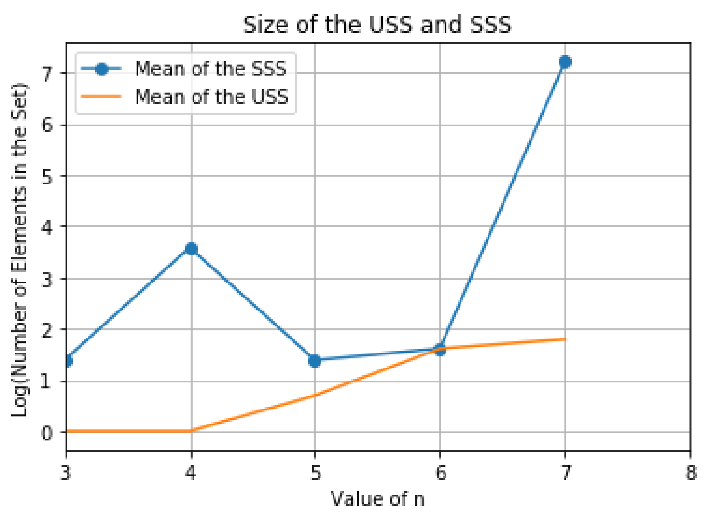


Figure 12: A comparison of the SSS and USS.

From the experimental data and suggestions from Garber [22] it seems that the average complexity using USS may be polynomial (Fig. 13). However the conjugacy search problem yet remains secure since the size of the summit sets may be exponential in general and it is difficult to obtain good bounds on the cardinality. The number of times one needs to apply cycling/decycling (the value of K_2 in Section 5.3 for finding an element in USS(x)) is unknown in general. If one could find the mathematical bounds and prove the USS is polynomial in size, it is possible to state with confidence that the CSP is unsecure for use in cryptography. Further, note that even if the USS is polynomial this does not guarantee the USS attack is feasible against AAG protocol operating under a large n, as the USS complexity could be n^2 polynomial resulting in a feasible USS size for many values of n or n^{1000} polynomial resulting in a feasible USS size for only the very small values of n and so the effectiveness is dependent upon on the power n is raised to.

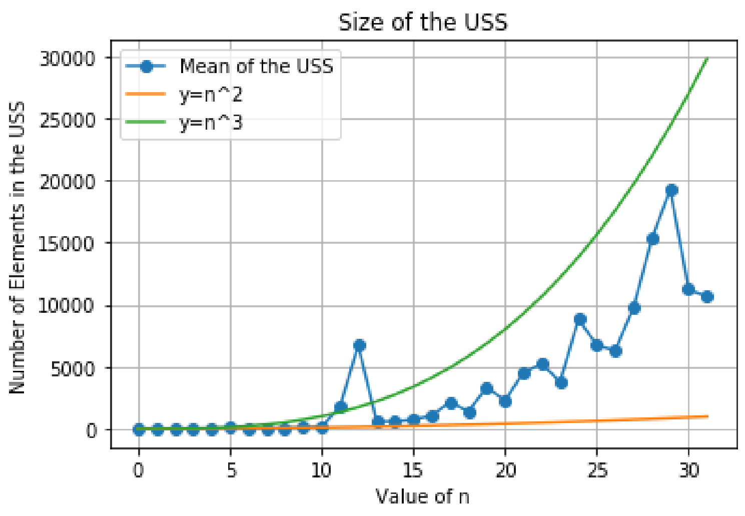


Figure 13: Estimating the complexity of the USS from experimental data.

6.2 Further Sage Experiments

In further experiments with the ultra summit set, analysis was performed on the other parameters under increasing n to determine if these increases effected the size of the USS and time taken to compute the set. Unfortunately subject to computing restraints we were not able to collect sufficient quantities of data to make confident predictions. However we did observe some patterns which needed to be backed up with more data. First, checking $L = 6, 8, 10$ for which, as L is increased the size and time to compute the USS increases. This is intuitive as increasing the length of the conjugating element (the private key) increases the braid length, hence more possible conjugating elements and therefore a bigger ultra summit set. Furthermore as L and n increase the size and time to compute the USS also appears to be more volatile and the deviation from the mean increases. There was also an anomaly whereby the parameters $L = 6$ and $n = 14$ generated a significantly large USS of 40830 elements and further investigation should be given to why some braids generate much smaller and larger USS sets than expected. Performing the same experiment thirty times (for the USS) and taking a mean assisted in correcting the anomalies.

With further regards to anomalies, we also encountered a trivial example in which the original AAG parameters were broken with a USS attack in 2.14 seconds! This again further emphasises the importance of more research to determine why some braids generate much smaller as well as larger USS sets than expected and further modifications should include ways to generate non trivial (easily computed USS) braids. This instability is a serious threat to the practical use of braid group cryptography because unless one can find a way to constrain the generation of the braids to be non-trivial, or be easily computed by the USS, there shall always exist instances where braids can be trivially broken. These trivial braids may only occur in rare cases, perhaps in 1/100 braids or even 1/1000, but the reality is a frequency of 1/100 transmissions being successfully attacked is not acceptable.

Then consideration was made for increasing all the parameter N and $L_1 = 5$ to $L_1 = 10$ and $L_2 = 8$ to $L_2 = 13$ and determining through intuition that after increasing the value of all parameters the size and time to compute the USS increases, although we did not have sufficient data to show this. Furthermore under small parameters ($n \leq 11$) the size of the USS and SSS does not differ significantly. It is only when n holds larger values is there a significant increase in the size and time taken to compute the USS. This demonstrates that by increasing these parameters it increases the difficulty of attacking AAG with the USS attack but consider it may significantly increase the AAG protocol compute time and hence render the AAG protocol impractical.

6.3 USS Analysis Summary and Modifications

With the experimental data one could now make the assumption that USS has polynomial complexity, and is therefore insecure for the parameters initially suggested by AAG. This would result in the USS attack posing a credible threat to AAG. Further to the experiments Mathematicians have also suggested the USS may be polynomial [22]. That said, we are still unable to put solid bounds on the complexity of the USS, it could be a very high powered polynomial, in which case it is possible AAG is secure just by increasing the parameter n to increase the time required for a successful attack. This conclusion therefore depends on the choice of assumptions, if considering only that USS complexity has not formally been mathematically bounded one could naively assume the CSP and its variations are still secure against the USS attack. However if considering the experimental data collected we can approximate the USS has polynomial complexity (at least for all the randomly generated braids in this experiment) in which case one could deem AAG insecure against ultra summit set attacks, particular for low parameters such as AAG's original parameters.

Due to similar conditions described in Ko-Lee's paper with regards to why the super summit attack would be unsuccessful, the same conditions apply to the USS, therefore the attack using the ultra summit set will also not be successful in its original form. Although, outside the scope of this paper, it is worth noting, Hughes was able to utilise the left super-summit-set, an invariant under conjugation, to attack this PKC [28].

Fortunately AAG and Ko-Lee protocols rely on the assurance that CSP is hard, and so if braid groups eventually are determined not to be relevant in cryptography, it may still be possible to implement these group based cryptography protocols with another platform group and/or another hard problem [22].

A modification to the USS suggested by Gebhardt and Gonzalez-Meneses [29] to find a polynomial-time solution to the CSP, was to replace the cycling and decycling by an operation denoted **cyclic sliding** ($SC(x)$). The sets of $SC(x)$ and their elements naturally satisfy all the properties that were shown for the USS but often with better properties such as it yielding a simpler algorithm to solve the CSP (hence the algorithm is also easier to implement). Further, for elements of canonical length 1, cycling and decycling are trivial operations, but SC is not. Generally the set $SC(x)$ is smaller than the USS(x) although it still may be exponential in the length of x.

6.4 Nielsen-Thurston Trichotomy

One possible cause for the unexpected variations in Ultra Summit Set sizes may be the geometrical properties of the braid. One can classify braids into reducible or irreducible, for irreducible braids we can further classify the braid into periodic or pseudo-Anosov. This is the Nielsen-Thurston trichotomy (reducible, periodic and pseudo-Anosov). Determining which braid falls into which classification is one of the main algorithmic decision problems regarding braids. Fortunately there is a quadratic time algorithm (w.r.t. braid length) that can solve this decision problem for braids [30]. As the problem ultimately determines whether or not a given non-periodic braid is reducible (or pseudo-Anosov), the problem is also often referred to as the reducibility problem.

Determining whether the braid a is periodic is rather trivial, to do so, a is periodic if and only if its nth power or its $(n-1)$st power is a power of the half-twist $\varDelta$. There exists a polynomial-time algorithm for solving the CSP for the case of periodic braids suggested by Birman, Gebhardt and Gonzalez-Meneses [31]. To determine the classification of the other two types of braid requires a more complex algorithm outside the scope of this paper. It is possible that the class of the braid may be a cause for the extreme unexpected values of some braids, as for the case of reducible braids one has to make an unknown number of cyclings and decyclings. For the case of pseudo-Anosov braid there exists a small power of a pseudo-Anosov braid which is conjugate to a rigid braid and an algorithm solving the CSP for rigid braids in polynomial time does not yet exist. For example, when we broke AAG under the original parameters with the USS in 2.14 seconds it is possible this braid may have had a reducible or pseudo-Anosov classification. Below is an example of three different types of braids using Sage to return the Thurston classification of a braid.

```
Input: B = BraidGroup(3)
       b = B([1, 2, -1])
       b.thurston_type()
Output:'reducible'

Input: a = B([2, 2, -1, -1, 2, 2])
       a.thurston_type()
Output:'pseudo-anosov'

Input: c = B([2, 1, 2, 1])
       c.thurston_type()
Output:'periodic'
```

6.5 Length Based Attack on Ko-Lee and AAG

From Hughes and Tannenbaum [26], one needs not to efficiently solve the conjugacy problem in order to break a braid-based cryptosystem. Furthermore using heuristic alogrithms to solve the conjugacy problem can be very effective under certain parameters, and the attack (based on the observation that representatives of conjugate braids in the super summit set are likely to be conjugate by a permutation braid (a particularly simple braid)) demonstrated weaknesses of both Ko-Lee and the AAG protocol for random instances. We have therefore realised the importance of both parameter sizes and for the case of length based attacks consideration should be made when choosing random private keys. This has led to further research into generating secure keys. It is also important to consider that length based attacks will not solve every instant of conjugates but provides a probabilistic way of solving the CSP in certain cases. The probabilistic attack depends on the specific length function employed. For braid groups, there are a number of suitable length functions that allow this attack to be mounted. We comment that length-based attacks need to be modified in practice, to ensure (for example) that we do not get stuck in short loops.

Another reason for carefully choosing random braid is not to choose a braid where the normal form of a product ab yields too much information about a and b. Further problems arise due to many of the random braid sequences already being left weighted. Possible modifications/solutions have been suggested (i.e. insert small permutations into the middle of the word) for choosing random keys, however there is also a general worry that seemingly contradictory security requirements arise from different attacks. For example, the length attack implies that the generators in the Commutator protocol should have a small length but that makes the Conjugacy problem easier. As explained before, the Conjugacy problem can be too easy if random sequences of simple words are chosen, but it is difficult to guarantee a large length otherwise. A good optimisation is required to prevent against both length based attacks as well as USS attacks. Length Based Attacks are a credible threat to AAG and Ko-Lee under large L_1 and L_2 parameters, however certain braids can form peaks causing a failure in the attack.

The length based attack gives a probabilistic solution to the conjugacy problem but can also be used to solve other hard problems such as the group membership problem and the decomposition problem. Moreover length based attacks are applicable in any group that has a reasonable length function, e.g., Thompson group [32].

A modification to the length based attacks which increased the success rate in experimentation is to use a length based attack with memory [22]. In the original length based attack we only remember the best conjugator in each iteration however the problem is that sometimes a prefix of the correct conjugator is not the best congujagtor at some iteration and so it is thrown out. Hence this algorithm then fails. In the memory length based approach we remember a given number (which is the size of the memory) of possible conjugators of this length. Then in the next step we add one more conjugate to the memory and we again choose the best conjugator among all possibilities. This results in the correct conjugator usually being found in the first place of the memory if the attack has been successful. This modified length based attack is a threat to AAG and Ko-Lee as the success rate of length based attacks with memory increase as the memory increases and this has been seen from experimentation [22].

LBA experiments were performed by Myasnikov and Ushakov [25] where the following parameters were chosen $n = 80$, $N_1 = N_2 = 20$, $L = 50$ and the parameters L_1 and L_2 were varied to demonstrate the success rate with longer subgroup generators. Different algorithms were used for performing the length based attack. There were 100 problems generated for each set of parameters. Table 5 shows their results.

L_1, L_2	10,13	20,23	30,33	40,43
Success rate	00%	51%	97%	96%

Table 5: Success rate of the length based attack (%) [25].

From Table 5 there is a high success rate for large lengths of words (L_1 and L_2 values) in the public tuple. One can also see how the LBA on small values of L_1 and L_2 is ineffective and the success rate increases as L_1 and L_2 values increase. Furthermore success rates of over 50% were also observed when cutting peaks contained in uniformly random generated keys. More information on how to cut peaks can be found in [25]. Note also the authors make a reasonable suggestion that the success rate could be improved with greater computing power. From this data one can assume Ko-Lee and AAG protocol is not secure against LBA's for large L_1, L_2 parameters.

7 Conclusion and Further Work

This paper has examined AAG and Ko-Lee from a number of different perspectives, including implementation and the experimentation of AAG as a practical application to cryptography.

It has been observed that L and n are the driving parameters to increasing complexity. With n being the most effective way to increase complexity without increasing the time taken to generate the braids. With regards to the attacks, this paper has seen how some attacks have proved effective at attacking AAG and Ko-Lee protocols under certain conditions. Further observing how the SSS is not effective at breaking AAG due to the size and time taken to compute the set using Garside's methods.

However through experimentation and research papers we have seen how the considerably smaller USS may be polynomial in complexity and therefore can be effective at attacking AAG for the original AAG parameters. It has been shown how Ko-Lee and AAG can both be attacked with a high probability of success rate by a computer science length based attack for high parameter values of L_1 and L_2. Then considering instances where this attack may not be so successful (peaks). Furthermore one has also seen the security contradiction where by increasing the canonical length of the generators in the AAG protocol will decrease the probability of a successful attack using USS, however this will in turn increase the probability of a successful attack using the length based attacks and vice-versa.

Furthermore, this paper has suggested modifications to reduce the probability of a successful attack, and how modifications and increasing parameters may still hold some effectiveness against current attacks. Although there will be a limit to just increasing parameters as this will eventually become impractical. Therefore we expect to continue seeing research papers being published with modifications increasing the security of the protocols along with further papers attempting to attack these modified systems.

Concluding, AAG and Ko-Lee as originally suggested cannot be considered secure. Further investigation should be given to why some braids generate much smaller and larger USS sets than expected and why it was possible to have been able to break AAG under the original parameters in some instances. Further modifications should include ways to generate non trivial (easily computed USS) braids.

Although the search for the future of cryptosystems in braid groups under the conjugacy search problem may now seem a vain inefficacious attempt (and that very well could be the case), there has still been huge developments in the group based mathematical community with regards to group based cryptography leading one to believe perhaps in another platform group, that satisfies the Myasnikov, Shpilrain and Ushakov criteria (as we laid out in Section 4) or/and, with another hard problem may lie the future of cryptography.

The code used in this paper may be found at the following link: `http://math-sciences.org/?page_id=1357`

References

[1] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[2] Whitfield Diffie and Martin Hellman. New directions in cryptography. *IEEE transactions on Information Theory*, 22(6):644–654, 1976.

[3] Peter W Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM review*, 41(2):303–332, 1999.

[4] Wikipedia. Integer factorization. *https://en.wikipedia.org/wiki/Integer_factorization*, 2019. [Online; accessed 31/03/19].

[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 *Annual International Cryptology Conference*, pages 166–183. Springer, 2000.

[6] Iris Anshel, Michael Anshel, and Dorian Goldfeld. An algebraic method for public-key cryptography. *Mathematical Research Letters*, 6:287–292, 1999.

[7] Alexei Myasnikov, Vladimir Shpilrain, and Alexander Ushakov. *Group-based cryptography*. Springer Science & Business Media, 2008.

[8] Frank A Garside. The braid group and other groups. *The Quarterly Journal of Mathematics*, 20(1):235–254, 1969.

[9] Joan S Birman and Tara E Brendle. Braids: a survey. In *Handbook of knot theory*, pages 19–103. Elsevier, 2005.

[10] Stephen Bigelow. The lawrence-krammer representation. *arXiv preprint math/0204057*, 2002.

[11] Emil Artin. Theory of braids. *Annals of Mathematics*, pages 101–126, 1947.

[12] Dr Nicholas Jackson. Notes on braid groups, 2004. [Online; accessed 10-November-2018].

[13] Max Dehn. Über unendliche diskontinuierliche gruppen. *Mathematische Annalen*, 71(1):116–144, 1911.

[14] Daan Krammer. Lecture notes for MA4F2 braid groups (University of Warwick). *https://homepages.warwick.ac.uk/~masbal/MA4F2Braids/braids.pdf*, 2005.

[15] J Birman, KH Ko, and SJ Lee. A new approach to the word problem in the braid groups. *Advances in Math*, 139:2, 1998.

[16] Ueli M Maurer. Towards the equivalence of breaking the diffie-hellman protocol and computing discrete logarithms. In *Annual International Cryptology Conference*, pages 271–281. Springer, 1994.

[17] Iris Anshel, Michael Anshel, and Dorian Goldfeld. Non-abelian key agreement protocols. *Discrete Applied Mathematics*, 130(1):3–12, 2003.

[18] Matthew J Craven and Daniel Robertz. A parallel evolutionary approach to solving systems of equations in polycyclic groups. *Groups Complexity Cryptology*, 8(2):109–125, 2016.

[19] Iris Anshel, Michael Anshel, Benji Fisher, and Dorian Goldfeld. New key agreement protocols in braid group cryptography. In *Cryptographers Track at the RSA Conference*, pages 13–27. Springer, 2001.

[20] Karl Mahlburg. An overview of braid group cryptography. *http://www.math.wisc.edu/~boston/mahlburg.pdf*, 2004.

[21] Elsayed A Elrifai and Hugh R Morton. Algorithms for positive braids. *Quarterly Journal of Mathematics*, 45(180):479–498, 1994.

[22] David Garber. Braid group cryptography. In *Braids: Introductory lectures on braids, configurations and their applications*, pages 329–403. World Scientific, 2010.

[23] Nuno Franco and Juan González-Meneses. Conjugacy problem for braid groups and garside groups. *Journal of Algebra*, 266(1):112–132, 2003.

[24] Volker Gebhardt. A new approach to the conjugacy problem in garside groups. *Journal of Algebra*, 292(1):282–302, 2005.

[25] Alex D Myasnikov and Alexander Ushakov. Length based attack and braid groups: cryptanalysis of anshel-anshel-goldfeld key exchange protocol. In *International Workshop on Public Key Cryptography*, pages 76–88. Springer, 2007.

[26] James Hughes and Allen Tannenbaum. Length-based attacks for certain group based encryption rewriting systems. *arXiv preprint cs/0306032*, 2003.

[27] Alexei Myasnikov, Vladimir Shpilrain, and Alexander Ushakov. Random subgroups of braid groups: an approach to cryptanalysis of a braid group based cryptographic protocol. In *International Workshop on Public Key Cryptography*, pages 302–314. Springer, 2006.

[28] Jim Hughes. The left sss attack on ko-lee-cheon-han-kang-park key agreement scheme in b45. *Rump session CRYPTO*, 2000, 2000.

[29] Volker Gebhardt and Juan González-Meneses. The cyclic sliding operation in garside groups. *Mathematische Zeitschrift*, 265(1):85–114, 2010.

[30] Matthieu Calvez. Fast nielsen–thurston classification of braids. *Algebraic & Geometric Topology*, 14(3):1745–1758, 2014.

[31] Joan S Birman, Volker Gebhardt, and Juan González-Meneses. Conjugacy in garside groups i: Cyclings, powers, and rigidity. *arXiv preprint math/0605230*, 2006.

[32] Vladimir Shpilrain and Alexander Ushakov. Thompsons group and public key cryptography. In *International Conference on Applied Cryptography and Network Security*, pages 151–163. Springer, 2005.