Algorithms in Braid Groups

Matthew J. Campagna
matthew.campagna@pb.com
Secure Systems
Pitney Bowes, Inc.

Abstract

Braid Groups have recently been considered for use in Public-Key Cryptographic Systems. The most notable of these system has been the Birman-Ko-Lee system presented at Crypto 2000. This article gives a brief introduction into braid groups and the hard problems on which public key systems have been defined. It suggests a canonical form *max run form* using the Artin generators and supplies some supporting algorithms necessary for cryptographic operations.

1 Introduction

Braid groups were introduced into the literature by Artin in 1947 [1]. A classical problem that arose from this work is known in the literature as the word conjugacy problem. The difficulty of this problem and other similar problems, conjugacy search problem, have been incorporated into various braid group cryptosystems [2]. The primary purpose of this paper is to introduce algorithms for performing braid group operations necessary in a cryptosystem, so that any such cryptosystem can be better understood and researched.

In this paper we review the braid group using the Artin representation. All the algorithms developed in subsequent sections will be based on the Artin generators for the braid group. A similar suite of algorithms for performing braid group operations on the band-generator presentation is given in [3].

Section 2 provides a review and some notational conventions used throughout the paper. Section 3 introduces the max-run form and provides some motivation for its use, and its usage with in various cryptosystems. Section 4 details algorithms for performing cryptographic operations.

2 Braid Groups

A *braid* can be understood as the intertwining or crossing over of parallel strands or strings. We use the common convention that the leftward moving strand crosses over the rightward moving strand represents a *positive twist*. The number of strands is called the *braid index*, and the set of all braids on n strands is denoted B_n. In an effort to align the algorithm presentation with source code implementation we will use a zero-based indexing convention.

DEFINITION: Let $n > 0$ be an integer, then we define the $n-$*braid group*, $(B_n, \cdot, e)$, as the group generated by $\sigma_0, \sigma_1, \ldots \sigma_{n-2}$ subject to the following relations

$$B_n = \left\langle \sigma_0, \ldots, \sigma_{n-1} \mid \begin{array}{cc} \sigma_i\sigma_j\sigma_i = \sigma_j\sigma_i\sigma_j & |i-j| = 1 \\ \sigma_i\sigma_j = \sigma_j\sigma_i & otherwise \end{array} \right\rangle \tag{1}$$

We refer to σ_i, $0 \le i < n-1$ as an *elementary braid on n-strands* and interpret that as the braid that interchanges strand i and $i+1$ by passing $i+1$ over i.

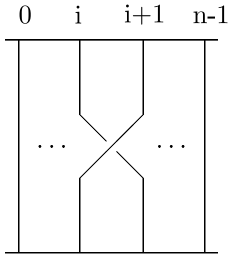

Figure 1: Elementary Braid σ_i

We multiply two elementary braids together σ_i and σ_j, $\sigma_i \cdot \sigma_j$ by performing the i^{th} elementary operation and then the j^{th} operation.

The inverse of an elementary braid element, σ_i is an exchange of the strands i and $i+1$ where the right moving strand i crosses over the left moving strand $i+1$. A *braid word* $\alpha \in B_n$ is any sequence of elementary braids, or their inverses, alternately just a *braid* or *word*. A braid word consisting of no inverses is called a *positive braid*, and we denote the set forming a monoid under the group operation as B_n^+. The length of a braid α, $|\alpha|$, is the number of elementary braids and inverses in the presentation of α. Two braids are equal when they have identical representation. Two positive braids are equal when either they have identical representation or they can be transformed into an identical representation through a sequence of positive word transformation using the group relation.

Definition: A braid $\alpha \in B_n$ is called a *positive permutation braid* if it a positive braid and it can be drawn without any two strings crossing each other more than once. We denote the set of positive permutation braids as $\tilde{\Sigma}_n$.

There is a natural embedding of the symmetric group on n-letters Σ_n into $\tilde{\Sigma}_n$, induced by the mapping $\psi(i, i+1) = \sigma_i$, where $(i, i+1)$ represents the two cycle transposition that takes i to $i+1$, and $i+1$ to i.

Let $\pi \in \Sigma_n$ be a permutation, the since $\pi : \{0, \dots n-1\} \longrightarrow \{0, \dots n-1\}$, we simply express $\pi = (\pi(0), \pi(1), \dots \pi(n-1))$. Under this presentation permutation $\pi = (2, 0, 1, 3)$ can be represented as the product of two cycles $(0, 1)(1, 2)$ which is mapped to $\sigma_0 \cdot \sigma_1$. It is often desirable to take a permutation and place it into a positive permutation braid, or permutation braid.

Definition: Let $a \in G$ where $< G, \cdot, 1 >$ is a group. An element of the form $b^{-1} \cdot a \cdot b$, for some $b \in G$ is called *a conjugate of a*.

We are now in a position to state a few classical problems in braid group theory.

Word Problem The simplest problem is given two braids $\alpha_1, \alpha_2 \in B_n$ determine if $\alpha_1 = \alpha_2$.

Conjugacy Decision Problem Given two braids $\alpha, \beta \in B_n$ can it be determined if α is a conjugate of β.

Conjugacy Search Problem Given two braids $\alpha, \beta \in B_n$ β a conjugate of α can a γ be found such that $\gamma^{-1}\alpha\gamma = \beta$.

Generalized Conjugacy Search Problem Given two braids $\alpha, \beta \in B_n$,

Most of the solutions for solving the word problem are related to putting a braid into some kind of standardized form. We will present one such form and define algorithms to place an arbitrary braid into this standardized form.

Despite its relatively simple description the Braid Group proves to be a fairly complex group. This section gives some of the basic mathematical properties of the braid group. Artin, explored the relationship between the braid group on n strands B_n to the symmetric group on n elements Σ_n, defined by the natural map $\phi : B_n \longrightarrow \Sigma_n$, that takes generators σ_i to transpositions $(i, i+1)$. There is a natural mapping from the symmetric group into the braid group. All elements of the symmetric group, $\pi \in \Sigma_n$ can be represented by a sequence of transpositions of distance 1, there is an obvious embedding of the Σ_n in B_n, denoted $\tilde{\Sigma}_n$, by $\psi : (i, i+1) \mapsto \sigma_i$, we refer this set of braids as the permutation braids. Artin's solution to the word problem was improved by Garside. Garside also introduced the notion of the fundamental braid.

Definition: Let B_n be a braid group. The *fundamental braid* Δ is defined as the braid in which every strand crosses every other strand exactly once, it can be represented as:

$$\Delta = (\sigma_0\sigma_1 \cdots \sigma_{n-2})(\sigma_0\sigma_1 \cdots \sigma_{n-3}) \cdots (\sigma_0\sigma_1)(\sigma_0).$$

Garside demonstrated that all braids $\alpha \in B_n$ can be put into a canonical form $\alpha = \Delta^r\beta$ where $r \in \mathbb{Z}$, and $\beta \in B_n^+$, with some additional constraints on the representation of β. Let $A \in B_n$ then $A(i)$ is the permutation result of applying the permutation induced by the braid on the i^{th} stand.

Proposition 1. *For all permutation braids $\alpha \in \tilde{\Sigma}_n$ such that $|\alpha| = |\Delta|$ then $\alpha = \Delta$.*

Proof: For a given n, $|\Delta| = n \cdot (n-1)/2$, which is n choose 2. Now each element in α introduces a twist between two strands, since there are at most n choose 2 choices for strand twists, and their are n choose 2 twists in α a permutation braid, it is clear that no two strands cross twice and so every two strand crosses exactly once. Hence α is the fundamental braid.

Some results based found in the literature follows.

1. $\Delta = \sigma_i A_i = B_i\sigma_i$ for some permutation braids $A_i, B_i \in \tilde{\sum}_n$, and for $-1 < i < n-1$.
2. $\sigma_i\Delta = \Delta\sigma_{n-2-i}$, for $-1 < i < n-1$.
3. $\sigma_i^{-1} = \Delta^{-1}B_i$, where $\Delta = B_i\sigma_i$, for a permutation braid $B_i \in \tilde{\Sigma}_n$.
4. For all $W \in B_n$, $W = \Delta^r V$ where $r \in \mathbb{Z}$, and $V \in B_n^+$.

In order to effectively work in a braid group, we need algorithms for computing inverses and commuting elements to the left or to the right. What follows is an algorithm for commuting an element to the left. All elements here are assumed positive.

3 Max Run Form

In an effort to develop some efficient algorithms designed for a computer a new normalized form was defined and used. We begin this section with a description of this new normalized form. We then go on to prove how it uniquely defines all elements in a braid group, and provide some algorithms for basic braid group operations.

Definition A braid $\beta \in B_n$ is in *maximal run form* when it is expressed as

$$\beta = \Delta^n \alpha_0 \alpha_1 \cdot \ldots \cdot \alpha_{k-1}$$

where $n \in \mathbb{Z}$ and α_0 is the longest possible braid after moving all powers of the fundamental braid to the left, which can be formed by series of sequences of increasing generators from $\sigma_0\sigma_1 \ldots \sigma_{n-2}$ then $\sigma_0 \ldots \sigma_{n-3}$ then $\sigma_0 \ldots \sigma_{n-4}$ etc.., skipping any value in the *run* that cannot be obtained in the re-sequencing.

This form has a flavor similar to the descending cycles form given in [3], but there is no direct correlation. An example probably best illustrates the form. Consider

$$\begin{aligned}
\beta &= \Delta^2 \sigma_0\sigma_1\sigma_0\sigma_2\sigma_1\sigma_1\sigma_3\sigma_1\sigma_0\sigma_2 \\
&= \Delta^2 \sigma_0\sigma_1\sigma_2\sigma_0\sigma_1\sigma_1\sigma_3\sigma_1\sigma_0\sigma_2 \\
&= \Delta^2 \sigma_0\sigma_1\sigma_2\sigma_3\sigma_0\sigma_1\sigma_1\sigma_1\sigma_0\sigma_2 \\
&= \Delta^2 \sigma_0\sigma_1\sigma_2\sigma_3\sigma_0\sigma_1\sigma_1\sigma_1\sigma_2\sigma_0
\end{aligned}$$

The result is the factorization $\beta = \Delta^2\alpha_0\alpha_1\alpha_2\alpha_3$ where $\alpha_0 = \sigma_0\sigma_1\sigma_2\sigma_3\sigma_0\sigma_1$, $\alpha_1 = \sigma_1$, $\alpha_2 = \sigma_1\sigma_2$, and $\alpha_3 = \sigma_0$. The standardization algorithm is dependent on some other more basic algorithms that on the surface are less complex then in practice. Two relatively symmetric algorithms are the *commute left* and *commute right* algorithms.

A second convenient form is to express braids $P \in B_n$ in the form $P = \Delta^r A_0 \cdots A_{k-1}$ where $A_i \in \tilde{\Sigma}_n$, where the decomposition into factors are such that $A_i A_{i+1}$ is left-weighted [2]. The decomposition into permutation braids induced by the max run form does not directly correlate into left weighted factorization. It has not been investigated how the two decompositions relate, but there does seem to be a tendency to induce left-weighted results, but not always.

Our permutation-based expression is formed from the max-run form. It is a decomposition of a braid $\beta = \Delta^r\alpha_0 \cdot \alpha_{m-1}$ in max run form, to an expression of the form $\beta = \Delta^r A_0 \cdots A_{k-1}$ where the A_is represent partitioning of the elementary braids, without re-ordering, in such a way that A_i is of maximum length.

4 Algorithms

A variety of algorithms are required to perform group operations using the maximal run form. We will standardize on zero based indexing, that is all arrays and consequently strands of a braid will start at the label 0. We will use the notation $b[m]$ to signify an array of length m, and b[i] to signify the i^{th} element in the array. Hopefully the distinction between an element in the array, and the description of the array will be clear from the context

4.1 Permutation to Positive Braid Algorithm

We use the following convention when describing a permutation. We define a permutation $\pi = (2, 0, 1, 3)$ to be a permutation on 4 objects, zero indexed to indicate the following mapping:

element at location 0 $\mapsto$ location 2
element at location 1 $\mapsto$ location 0
element at location 2 $\mapsto$ location 1
element at location 3 $\mapsto$ location 3

or $\pi(0, 1, 2, 3) = (1,\ 2,\ 0,\ 3)$, or more symbolically $\pi(a, b, c, d) = (b, c, a, d)$

Algorithm 1. ***Permutation to Permutation Braid***

Input: *A permutation* $A = (a_0, a_1, ..., a_{n-1})$*, represented by an array* $a[n]$.
Output: *A braid* $P = \sigma_{b[0]} ... \sigma_{b[m-1]}$*, represented by the array* $b[m]$

1. *Set* $i \leftarrow 0$*, set* $m \leftarrow 0$*,* $b \leftarrow \emptyset$*, and set* $l \leftarrow 0$.
2. *While* $i < n$
   (a) *If* $a[i] = l$*, then*
      i. *Set* $j \leftarrow i - 1$
      ii. *While* $j > l - 1$

A. Set $b[m] \leftarrow j$*,* $m \leftarrow m+1$*,* $a[j+1] \leftarrow a[j]$*,* $j \leftarrow j-1$

iii. Set $l \leftarrow l+1$*,* $i \leftarrow l$*.*

(b) Otherwise $i \leftarrow i+1$*.*

3. Output m *and* $b[m]$

The algorithm consists of a double loop, an outer loop where l ranges from 0 to $n-1$ and an inner loop where i ranges from l to $n-1$. For each l we eventual find a value i such that $a[i] = l$, and so advance l, and restart the i loop from $i = l$ to $n-1$. Eventually $l = n$ and the inside loop fails to find an i such that $a[i] = n$, and the algorithm terminates. Runtime is $\mathcal{O}(n^2)$.

The effect of each iteration over l is to move the element destined for location l by means of elementary transpositions $j- > j+1$, and $j+1- > j$, 2-cycles $(j, j+1)$. This is then represented by the elementary braid σ_j. Starting at $l = 0$, an i_0 is found such that $a[i_0] = 0$, the i_0^{th} element is then moved to 0^{th} position, by elementary transpositions, induced by elementary braids $\sigma_{i_0-1} \cdot \sigma_0$. We denote the l^{th} intermediate braid α_l, and the updated permutation represented by the array $a[n]$ as π_l. Now when $n = 2$ this algorithm as a mapping $\psi\Sigma_n \longrightarrow B_n^+$ takes $\psi((0,1)) = \sigma_0$, and $\psi(e) = e$, where $\sigma_0 \in \tilde{\Sigma}_2$. Suppose now that this is true for all values $k < n$, and consider the mapping $\psi : \Sigma_n \longrightarrow B_n^+$. Now for some i $a_i = 0$, and so $\alpha_0 = \sigma_{i-1} \cdots \sigma_0$, and $\pi_i = (0, a_0, \ldots a_{i-1}, a_{i+1}, \ldots a_{n-1}$. π_i is a permutation on $n-1$ letters, and the algorithm (with a simple index translation) produces a permutation braid β such that $\alpha = \alpha_0\beta$. It suffices to show then then α is a permutation braid. Since α_0 moves the i^{th} strand to the 0^{th} position, and it does not induce any crosses between the remaining $n-1$ strands; and the permutation braid β does not introduce more than one cross between any of the other $n-1$ strands, and does not include σ_0 moving the resting place of the i^{th} strand, no double crosses appear in the product. Hence the resulting braid is a permutation braid. It is a simple exercise to show the permutation induced by the resulting braid P is in fact the permutation A.

The permutation $p = (2, 0, 1, 3)$ results in the positive braid $\alpha = \sigma_0\sigma_1$

The following results were obtained running un-optimized compiled C code on an Intel Pentium 4, 2.0 Ghz CPU. The table lists the number of permutations of a given braid index that are added to a empty braid. The timings, and all subsequent timings are based on 1000 trial calculations using the performance counter.

Braid Index	Permutations	Time ms
100	1	0.002091
100	5	0.087765
150	1	0.003532
150	5	0.081514
200	1	0.009485
200	5	0.144352
250	1	0.163379
250	5	0.899496

Table 1: Permutation to Braid

4.2 Commute Algorithms

We represent a positive braid $P = \sigma_{a[0]}\sigma_{a[1]}, ..., \sigma_{a[m-1]}$, as a zero-indexed array $a[m]$, where $a[i]$ is the i^{th} index of the elementary braid element, $\sigma_{a[i]}$. In this way, the braid, $P = \sigma_0\sigma_1$, would be represented by the array $a[2]$, where $a[0] = 0$, and $a[1] = 1$. The following algorithm takes a source location s and a destination location d, $s < d$, and attempts to commute the elementary element at location s to location d using the braid group binary operation rules. An analogous commute left algorithm is also given.

Algorithm 2. ***Commute Right***

Input: *A positive braid* $P = \sigma_{a[0]}\sigma_{a[1]}, ..., \sigma_{a[m-1]}$, *given in an zero-indexed array* $a[m]$ *and source and destination indexes,* $-1 < s < d < m$
Output: *A positive braid* $P = \sigma_{b[0]}\sigma_{b[1]}, ..., \sigma_{b[m-1]}$, *given in a zero-indexed array* $b[m]$, *and a resulting index* i *of the current location of the input generator at location* s

1. *While* $s < d$
   (a) *If* $a[s] = a[s+1]$ *Set* $i \leftarrow s$, *output the array* $b[m] = a[m]$ *and STOP*
   (b) *If* $|a[s] - a[s+1]| > 1$ *then*
      i. *Set* $tmp \leftarrow a[s+1]$
      ii. *Set* $a[s+1] \leftarrow a[s]$
      iii. *Set* $a[s] \leftarrow tmp$
      iv. *Set* $s \leftarrow s+1$
      v. *Goto (1)*
   (c) *Otherwise*
      i. $j \leftarrow s-1$
      ii. *While* $(j > -1)$

A. *If* $a[j] = a[s+1]$ *then*
Apply this algorithm to destination $s-1$*, and source* j*.*
Set $a[k]$ *for* $k = 0, ..m-1$ *to the output*
If the resulting location is not $s-1$*,*
set $i \leftarrow s$*, Output the array* $b[m] = a[m]$ *and STOP.*
Else
$a[s-1] \leftarrow a[s]$
$a[s] \leftarrow a[s+1]$
$a[s+1] \leftarrow a[s-1]$

B. $s \leftarrow s+1$

C. *Goto (1)*

iii. *Else Set* $j \leftarrow j-1$

2. *Set* $i \leftarrow s$*, Output* $b[m] = a[m]$*, and STOP*

This algorithm loops over the distance $d-s$, at each iteration it attempts to move the element at location s to location $s+1$. The worse runtime case occurs when the difference between the $a[s]$ and $a[s+1]$ is 1. In this case the algorithm must locate another occurrence of the $a[s+1]$ value to the left of $a[s]$ (searching a list of length $s-1$) and move it to the $s-1$ position (performing the same algorithm on a shorter braid). In order to permute $a[s]$ with $a[s+1]$ in this case we use the binary operation rule $\sigma_i\sigma_j\sigma_i = \sigma_j\sigma_i\sigma_j$ when $|i-j| = 1$. So this algorithm runs in $\mathcal{O}(m^2)$, so polynomial in the length m. Many braids will have twists that prevent the commuting algorithm to succeed, this is expected. It is also clear that since only applications of the relation defined by the group operations are used the resulting braid is equal.

Algorithm 3. ***Commute Left***

Input: *A positive braid* $P = \sigma_{a[0]}\sigma_{a[1]}, ..., \sigma_{a[m-1]}$*, given in a zero-based indexed array* $a[m]$ *and source and destination indexes,* $-1 < s < d < m$
Output: *A positive braid* $P = \sigma_{b[0]}\sigma_{b[1]}, ..., \sigma_{b[m-1]}$*, given in a zero-based indexed array* $b[m]$*, and a resulting index* i *of the current location of the input generator at location* s

1. *While* $s > d$

   (a) *If* $a[s] = a[s+1]$ *Set* $i \leftarrow s$*, output the array* $b[m] = a[m]$ *and STOP*

   (b) *If* $|a[s] - a[s+1]| > 1$ *then*

      i. *Set* $tmp \leftarrow a[s-1]$

      ii. *Set* $a[s-1] \leftarrow a[s]$

      iii. *Set* $a[s] \leftarrow tmp$

      iv. *Set* $s \leftarrow s-1$

v. *Goto (1)*

(c) *Otherwise*

i. $j \leftarrow s + 1$

ii. *While* $(j < m)$

A. *If* $a[j] = a[s-1]$ *then*
Apply this algorithm to destination $s + 1$, *and source* j.
Set $a[k]$ *for* $k = 0, ..m - 1$ *to the output*
If the resulting location is not $s - 1$,
set $i \leftarrow s$, *Output the array* $b[m] = a[m]$ *and STOP.*
Else
$a[s-1] \leftarrow a[s]$
$a[s] \leftarrow a[s+1]$
$a[s+1] \leftarrow a[s-1]$

B. $s \leftarrow s - 1$

C. *Goto (1)*

iii. *Else Set* $j \leftarrow j + 1$

2. *Set* $i \leftarrow s$, *Output* $b[m] = a[m]$, *and STOP*

Table 2 lists the times required to commute an element a specified distance on various length braids. The times are recorded in millisecons. No indication is given whether the braid successfully commuted the entire distance.

Braid Index	**Length**	**Distance**	**Time ms**
100	100	14	0.001197
100	500	284	0.001356
150	150	21	0.001251
150	750	428	0.001585
200	200	28	0.001249
200	1000	568	0.001468
250	250	35	0.001280
250	1250	712	0.002476

Table 2: Commute Left

4.3 Permutation Factorization Algorithm

This algorithm takes a positive braid and expresses it as a permutation factorization.

Algorithm 4. ***Permutation Factorization***

Input: *An array $a[m]$ which represents a positive braid in max run form $P = \sigma_{a[0]}\sigma_{a[1]} \cdots \sigma_{a[m-1]}$ in B_n*
Output: *A factorization into permutation braids $P = A_0 A_1 \cdots A_{p-1}$ and permutations $\pi_0, \pi_1, \ldots, \pi_{p-1}$.*

1. *Allocate and zeroize arrays $I[n]$, $B[m]$, $s[n][n]$, set, $p \leftarrow 1$, $k \leftarrow 0$*
2. *Set $I[i] \leftarrow i$ for $i = 0, \ldots, m-1$*
3. *While $k < m$*
   (a) *If $s[I[a[k]]][I[a[k]+1] = 1$ Then*
      i. *Set $s[i][j] \leftarrow 0$, for $-1 < i, j < n$*
      ii. *Set $B[p] \leftarrow k$, $p \leftarrow p+1$*
   (b) *Otherwise*
      i. *Set $s[I[a[k]]][I[a[k]+1]] \leftarrow 1$*
      ii. *Set $s[I[a[k]+1]][I[a[k]]] \leftarrow 1$*
      iii. *Set $t \leftarrow I[a[k]]$*
      iv. *Set $I[a[k]] \leftarrow I[a[k]+1]$*
      v. *Set $I[a[k]+1] \leftarrow t$*
   (c) *Set $k \leftarrow k+1$*
4. *Set $A_i = \sigma_{a[B[j]]} \cdots \sigma_{a[B[j+1]-1]}$ for $-1 < i < p$.*
5. *Set π_i as the permutation induced by the permutation braid, A_j, for$-1 < i < p$.*

Table 3 contains timings in milliseconds for factoring a positive braid into permutations. The resulting permutation braids are not left weighted, nor is the starting positive braid considered to be in any specific form.

4.4 Positive Braid to Max Run Form Algorithm

This algorithm takes a positive braid and puts in to Max Run Form.

Note: Algorithm 1.3 uses an analogous algorithm to algorithm 1.2 which commutes an elementary braid left (denoted commute_left in the body of the algorithm 1.4).

Note: This algorithm has been modified to put a positive braid into a max run form, called the max run form.

Braid Index	Length	Time ms
100	200	0.019665
100	500	0.045279
150	300	0.043827
150	750	0.093849
200	400	0.066952
200	1000	0.126926
250	500	0.245058
250	1250	0.366513

Table 3: Permutation Factorization

Algorithm 5. ***Positive Braid to Max Run Form***

Input: *An array* $a[m]$ *representing a positive braid* $P = \sigma_{a[0]}\sigma_{a[1]}\cdots\sigma_{a[m-1]}$ *in* B_n
Output: *An array* $b[s]$ *and integer* r *such that the* P *is factorized into powers of the fundamental braid and a max run form positive braid,* $P = \Delta^r \sigma_{b[0]}\sigma_{b[1]}\cdots\sigma_{b[s-1]}$

1. *Set* $b[k] \leftarrow a[k]$ *for* $k = 0, \ldots, m-1$, *set* $k \leftarrow 0$, *set* $t \leftarrow 0$, $l \leftarrow 0$, $max \leftarrow n \cdot \lceil n/2 \rceil$, $e \leftarrow n-2$, $level \leftarrow 0$, $r \leftarrow 0$, $s \leftarrow m$
2. *while* $(k < m)$
   (a) *Set* $i \leftarrow k$
   (b) *While* $(i < m)$
      i. *If* $(a[i] = t)$
         A. *Apply the commute_left algorithm to* $b[m]$ *with destination* k *and starting position* i, *and set* $b[m]$ *to the returning array* j *the returning index valule.*
         B. *If* $(j = k)$
            1. $l \leftarrow l + 1$
            2. *if* $(l = max)$ *Set* $r \leftarrow r+1$, $e \leftarrow n-2$, $t \leftarrow -1$, $l \leftarrow 0$, $s \leftarrow s - max$, $k \leftarrow -1$, $b[j] \leftarrow b[max + j]$, *for* $j = 0, \ldots, s$.
            3. *Set* $k \leftarrow k + 1$
            4. *If*$(t = e)$, *set* $e \leftarrow e - 1$, $t \leftarrow -1$

         C. *else, Set* $l \leftarrow 0$

         D. *Set* $t \leftarrow t + 1$ *GOTO 2.c.*

      ii. *set* $i \leftarrow i + 1$
   (c) *If* $(i = m)$, *set* $l \leftarrow 0$, $t \leftarrow t + 1$

(d) *If* $(t > e)$*, set* $t \leftarrow 0$*,* $e \leftarrow e - 1$

(e) *If* $(e < 0)$

   i. *Set* $l \leftarrow 0$*,* $e \leftarrow n - 2$

   ii. *Set* $level \leftarrow level + 1$

   iii. *If* $(level > 1)$*, set* $k \leftarrow k + 1$

3. *Output* $b[s]$ *and* r *where* $P = \Delta^r \sigma_{b[0]}\sigma_{b[1]} \cdots \sigma_{b[s-1]}$*.*

Table 4 contains the times in milliseconds for taking a positive braid and expressing it in max run form. Various braid lengths and braid indexes are used.

Braid Index	**Length**	**Time ms**
100	200	0.356913
100	500	2.050400
150	300	0.735445
150	750	4.116140
200	400	1.974683
200	1000	7.581301
250	500	2.441234
250	1250	10.044801

Table 4: Positive Braid to Max Run Form

4.5 Arbitrary Braid to Max Run Form Algorithm

Note: This is a newly expressed algorithm which takes a random braid and converts it into a max run form and outputs both a positive braid form and a permutation form.

Note: Some basic rules about braid groups are used here first is the notion of the commutability of the fundamental braid Δ . $\Delta\sigma_i = \sigma_{n-2-i}$ for $i = 0, \ldots, n-2$, similarly for negative powers of the fundamental braid $\Delta^{-1}\sigma_i = \sigma_{n-2-i}\Delta^{-1}$ for $i = 0, \ldots, n-2$. Consequently for any integer r, we have the rule:

$$\Delta^r \sigma_i = \begin{cases} \sigma_i \Delta^r & \text{if } r \pmod{n} = 0, \\ \sigma_{n-2-i}\Delta^r & \text{otherwise.} \end{cases}$$

Note: Converting inverses to a power of the fundamental braid. This is a simple application that the fundamental braid Δ can be written as a positive braid $H_i\sigma_i$ for any $i = 0, \ldots, n-2$. Therefore $\sigma_i^{-1} = \Delta^{-1}\Delta\sigma_i^{-1} = \Delta^{-1}H_i\sigma_i\sigma_i^{-1} = \Delta^{-1}H_i$. This is accomplished using Algorithm 6. It is simply substituted here.

Note: H_i is of length $max-1$, where $max = n \cdot \lceil n/2 \rceil$, i.e. $H_i = c[i][0], c[i][1] \ldots, c[i][max-1]$, for some $c[n-1][max-1]$ matrix of integers, which can be computed using Algorithm 6.

Algorithm 6. *Arbitrary Braid to Max Run Form*

Input: *An arbitrary braid Q in B_n - the braid group on n-strings. $Q = \sigma_{a[0]}^{p[0]} \sigma_{a[1]}^{p[1]} \cdots \sigma_{a[m-1]}^{p[m-1]}$. Where $a[m]$ is an m-long array of integers in $[0, n-1]$ and $p[i] = 1$, or -1, and the inversion matrix $c[n-1][max-1]$*
Output: *An array $b[k]$ such that factorization of $Q = \Delta^r \sigma_{b[0]} \cdots \sigma_{b[k-1]}$ in max run form.*

1. *Set $b[i] \leftarrow a[i]$ for $i = 0, \ldots, m-1$, set $q \leftarrow m$, $t \leftarrow 0$, $r \leftarrow 0$, $max \leftarrow n \cdot \lceil n/2 \rceil$*
2. *While $(t < q)$*
   (a) *If $(p[t] = -1)$*
      i. *Set, $r \leftarrow r - 1$, $i \leftarrow 0$, $q \leftarrow q + max - 1$.*
      ii. *While$(i < t)$*
         A. *Set $b[i] \leftarrow n - 2 - b[i]$, $i \leftarrow i + 1$*
      iii. *Set, $i \leftarrow t + max - 1$*
      iv. *While $(i < q)$*
         A. *Set $b[i] \leftarrow b[i - max + 2]$, $i \leftarrow i + 1$*
      v. *Set $i \leftarrow 0$*
      vi. *While $(i < max - 1)$*
         A. *Set $b[t + i] \leftarrow c[d][i]$ for $i = 0, \ldots, max - 2$*
      vii. *$t \leftarrow t + max - 1$*
   (b) *else, Set $t \leftarrow t + 1$*
3. *Apply Algorithm 1.4 to the positive braid $\sigma_{b[0]} \cdots \sigma_{b[q-1]}$ this will result in a braid in the max run form: $\Delta^h \sigma_{b[0]} \cdots \sigma_{b[k-1]}$.*
4. *Set $r \leftarrow r + h$*
5. *Output the array $b[k]$, and integer r where $Q = \Delta^r \sigma_{b[0]} \cdots \sigma_{b[k-1]}$ as the max run form*

4.6 Elementary Inverse Algorithm

Note: This algorithm computes an expression for the inverse of an elementary braid in the form of an inverse of the fundamental braid and a positive permutation braid.

Algorithm 7. ***Elementary Inverse***

Input: *An integer i in $[0, n-2]$ and a braid group B_n.*
Output: *An array $c[max-1]$, where the inverse of σ_i is $\sigma_i^{-1} = \Delta^{-1}\sigma_{c[0]} \cdots \sigma_{c[max-1]}$*

1. *Set $max \leftarrow n \cdot \lceil n/2 \rceil$, allocate $b[max]$, $j \leftarrow 0$, $k \leftarrow 0$, $t \leftarrow 0$*
2. *While $(t < n-1)$*
   (a) *Set $j \leftarrow 0$*
   (b) *While $(j < n-1-t)$*
      i. *Set $b[k] \leftarrow j$, $k \leftarrow k+1$*
3. *Set $j \leftarrow max - 1$*
4. *While $(j > -1)$*
   (a) *If $(b[j] = i)$*
      i. *Set $s \leftarrow j$, $d \leftarrow max - 1$*
      ii. *Apply Algorithm 1.2 to $b[max]$ with s and d.*
      iii. *Set $c[k] \leftarrow b[k]$ for $k = 0, \ldots, max-2$*
      iv. *Set $j \leftarrow -1$*
   (b) *Else, Set $j \leftarrow j - 1$*
5. *Output $c[max-1]$*

4.7 Permutation Factorization to Positive Braid Algorithm

Note: This algorithm converts a permutation-based expression for a braid $P = \Delta^r A_0 A_1 \cdots A_{p-1}$ to a positive braid based expression

Algorithm 8. ***Permutation Factorization to Positive Braid***

Input: *A permutation-based braid $P = \Delta^r A_0 A_1 \cdots A_{p-1}$*
Output: *An array $b[m]$ such that $P = \Delta^r \sigma_{b[0]} \cdots \sigma_{b[m-1]}$*

1. *Set $k \leftarrow 0$, $i \leftarrow 0$, $b \leftarrow \emptyset$*
2. *While $(i < p)$*
   (a) *Apply Algorithm 1 to A_i and get output array $a[t]$ an array of length t.*
   (b) *Set $b[k+j] \leftarrow a[j]$ for $j = 0, \ldots, t-1$.*
   (c) *Set $k \leftarrow j + t$, $i \leftarrow i + 1$*

3. *Set* $m \leftarrow k$

4. *Output* $b[m]$ *where* $P = \Delta^r \sigma_{b[0]} \cdots \sigma_{b[m-1]}$.

4.8 Max Run Form Inverse Algorithm

This algorithm performs an inversion. It takes an arbitrary braid in maximum run form, inverts it and puts the output in maximal run form.

Algorithm 9. ***Max Run Form Inverse***

Input: *An integer* r*, and an array* $a[m]$ *representing a braid in maximal run form* $P = \Delta^r \sigma_{a[0]} \sigma_{a[1]} \cdots \sigma_{a[m-1]}$*, and the inversion matrix* $c[n-1][max-1]$*, where* $max = n \cdot \lceil n/2 \rceil$
Output: *An integer* t *and an array* $b[k]$ *representing the braid* P^{-1} *in maximal run form* $P^{-1} = \Delta^t \sigma_{b[0]} \sigma_{b[1]} \cdots \sigma_{b[k-1]}$.

1. *Set* $b \leftarrow \emptyset$, $i \leftarrow m-1$, $k \leftarrow 0$
2. *While* $(i > -1)$
   (a) *Set* $j \leftarrow k-1$
   (b) *While* $(j > -1)$
      i. *If* $(b[j] = a[i])$
         A. *Set* $s \leftarrow j$*, and* $d \leftarrow m-1$.
         B. *Apply Algorithm 1.2 to* $b[k]$*,* s*, and* d*, and get return values* $c[k]$*, and* l.
         C. *Set* $b[v] \leftarrow c[v]$ *for* $v = 0, \ldots, k-1$
         D. *If* $(l = d)$*, Set* $k \leftarrow k-1$*,* $i \leftarrow i-1$ *GOTO 2*
      ii. *Set* $j \leftarrow j-1$.
   (c) *Set* $b[k+j] \leftarrow c[a[i]][j]$ *for* $j = 0, \ldots max-1$.
   (d) *Set* $k \leftarrow k + max - 1$*,* $t \leftarrow t-1$*,* $i \leftarrow i-1$
3. *Set* $t \leftarrow t + r$
4. *If* $(r mod 2) == 1)$
5. *Set* $i = 0$
6. *While* $(i < k)$
   (a) *Set* $b[i] \leftarrow n - 2 - b[i]$*,* $i \leftarrow i+1$
7. *Output* $b[k]$*, and* t.

4.9 Other Computations

While the algorithms presented here are fast they do not combine to create an effective braid group cryptosystem. This is primarily due to the lack of a fast braid inversion algorithm for the representation used in this paper. The algorithm to invert a particular elementary braid is quite trivial. But when this algorithm is used element wise on a complex braid the resulting braid is unwieldily and results in a massive braid for standardization. This section contains timings for a number of basic operations required to perform cryptographic computations. All computations were done on a Intel Pentium 4, 2.0Ghz CPU.

Table 5 contains times in milliseconds for a Standardization and Multiplication Standardization operations.

Operation	**Braid Index**	**Braid Length**	**Time ms**
Standardize	100	200	0.356913
Standardize	100	500	2.050400
Standardize	150	300	0.735445
Standardize	150	750	4.116140
Standardize	200	400	1.974683
Standardize	200	1000	7.581301
Standardize	250	500	2.441234
Standardize	250	1250	10.044801
Multiply and Standardize	100	100	0.434160
Multiply and Standardize	100	300	3.265176
Multiply and Standardize	150	150	0.739434
Multiply and Standardize	150	450	7.084434
Multiply and Standardize	200	200	1.104407
Multiply and Standardize	200	600	6.399480
Multiply and Standardize	250	250	3.634891
Multiply and Standardize	250	750	15.689327

Table 5: Various Computations

It seems plausible to use some basic properties of the fundamental braid to develop a fast inversion and standardization technique for this representation. In particular, the observation that for $\sigma_i^{-1} = \Delta^{-1}B_i$, B_i has the property that $B_i = \sigma_j C_{i,j}$ for all $j! = n-1-i$. This combined with the property that $\Delta\sigma_j = \sigma_{n-1-j}\Delta$, will result in a reduction in running time by a half of on the naive approach.

References

[1] E. Artin, 'Theory of Braids,' *Ann. Math.* **48**, (1947), 101-126.

[2] Birman, J.S., Ko, K.H., and Lee, S.J. , 'A new approach to the word and conjugacy problems in the braid groups,' *Adv. Math.* 139 (1998), 322-353.

[3] Cha, J. C., Ko, K. H., Lee S. J., Han, J. W., Cheon, J. H. 'An Efficient Implementation of Braid Groups,' *AsiaCrypt 2001, LNCS 2248*, pp. 148-156. Springer Verlag, Berlin, 2001.

[4] E.A. Elrifai, H.R. Morton, 'Algorithms for Positive Braids,' *Quart. J. Math. Oxford* (2) 45, (1994) 479-497. LNCS 2248 (2001), 144-156.

[5] F.A. Garside, 'The braid group and other groups,' *Quart. J. Math. Oxford* (2) 78, (1969), 235-254