An Efficient Implementation of Braid Groups

Jae Choon Cha[1], Ki Hyoung Ko[1], Sang Jin Lee[1], Jae Woo Han[2], and
Jung Hee Cheon[3]

[1] Department of Mathematics
Korea Advanced Institute of Science and Technology, Taejon, 305–701, Korea.
{jccha,knot,sjlee}@knot.kaist.ac.kr
[2] National Security Research Institute, Taejon, 305–335, Korea.
jwhan@etri.re.kr
[3] International Research center for Information Security
Information and Communications University, Taejon, 305–732, Korea.
jhcheon@icu.ac.kr

Abstract. We implement various computations in the braid groups
via practically efficient and theoretically optimized algorithms whose
pseudo-codes are provided. The performance of an actual implementa-
tion under various choices of parameters is listed.

1 Introduction

A new cryptosystem using the braid groups was proposed in [5] at Crypto 2000.
Since then, there has been no serious attempt to analyze the system besides
one given by inventors [7]. We think that this is because the braid group is
not familiar to most of cryptographers and cryptanalysts. The primary purpose
to announce our implementation is to encourage people to attack the braid
cryptosystem. In [7], a necessary condition for the instances of the mathematical
problem which the braid cryptosystem is based on is found so that it makes the
mathematical problem intractable. This means that a key selection is crucial
to maintain the theoretical security of the braid cryptosystem. Thus the key
generation is one of the areas where much research is required and we think that
the search for strong keys should be eventually aided by computers. This is the
secondary purpose of our implementation.

In this paper we discuss implementation issues of the braid group given by
either the Artin presentation [2] or the band-generator presentation [1]. Due to
the analogy between the two presentations, our implementations on the two pre-
sentations are basically identical, except the low-level layer consisting of data
structures and algorithms for canonical factors, which play the role of the build-
ing blocks for braids. Even though the algorithms of the present implementation
in the braid groups are our initial work, they are theoretically optimized so that
all of single operations can be executed at most in $\mathcal{O}(n \log n)$ where n is the
braid index n that is the security parameter corresponding to the block sizes in
other cryptosystems. This excellent speed is achieved because the canonical fac-
tors are expressed as permutations that can be efficiently and naturally handled
by computers. The efficiency of the implementation shows that the braid group

is a good source of cryptographic primitives [5,6]. It is hard to think of any other non-commutative groups that can be digitized as efficiently as the braid group. Matrix groups are typical examples of non-commutative groups and in fact any group can be considered as a matrix group via representations. But the group multiplication in the braid group of index n is faster than the multiplication of $(n \times n)$ matrices.

This paper is organized as follows. Section 2 is a quick review of the minimal necessary background on braid groups. In Section 3 and 4, we develop data structures and algorithms for canonical factors and braids, respectively. In Section 5, we show how to generate random braids. In Section 6, we discuss the performance of our implementation, through the braid cryptosystems in [7]. Section 7 is our conclusion.

2   A Quick Review of the Braid Groups

A *braid* is obtained by laying down a number of parallel strands and intertwining them so that they run in the same direction. In our convention, this direction is horizontally toward the right. The number of strands is called the *braid index*. The set B_n of isotopy classes of braids of index n has a group structure, called the *n-braid group*, where the product of two braids x and y is nothing more than laying down the two braids in a row and then matching the end of x to the beginning of y.

Any braid can be decomposed as a product of simple braids. One type of simple braids is the *Artin generators* σ_i that have a single crossing between i-th and $(i+1)$-st strand as in Figure 1 (a), and the other type is the *band-generators* a_{ts} that have a single half-twist band between t-th and s-th strand running over all intermediate strands as in Figure 1 (b).

The n-braid group B_n is presented by the Artin generators $\sigma_1, \ldots, \sigma_{n-1}$ and relations $\sigma_i \sigma_j = \sigma_j \sigma_i$ for $|i - j| > 1$ and $\sigma_i \sigma_j \sigma_i = \sigma_j \sigma_i \sigma_j$ for $|i - j| = 1$. On the other hand, B_n is also presented by the band-generators a_{ts} for $n \geq t > s \geq 1$ and relations $a_{ts} a_{rq} = a_{rq} a_{ts}$ for $(t - r)(t - q)(s - r)(s - q) > 0$ and $a_{ts} a_{sr} = a_{tr} a_{ts} = a_{sr} a_{tr}$ for $n \geq t > s > r \geq 1$.

These will be called the *Artin presentation* and the *band-generator presentation*, respectively. There are theoretically similar solutions to the word and conjugacy problems in B_n for both presentations [1,2,3]. The band-generator presentation has a computational advantage over the Artin as far as the word problem is concerned. Since almost all the machineries are identical in the two theories, it will be convenient to introduce unified notation so that we may review both theories at the same time.

1. Let B_n^+ be the monoid defined by the same generators and relations in a given presentation. Elements in B_n^+ are called *positive braids* or *positive words*. The relations in the Artin and band-generator presentations preserve word-length of positive braids and so the word-length is easy to compute for positive braids. The natural map $B_n^+ \rightarrow B_n$ is injective. [1,4]. There are no known presentations of B_n except these two that enjoy this injection property needed for a fast solution to the word problem.
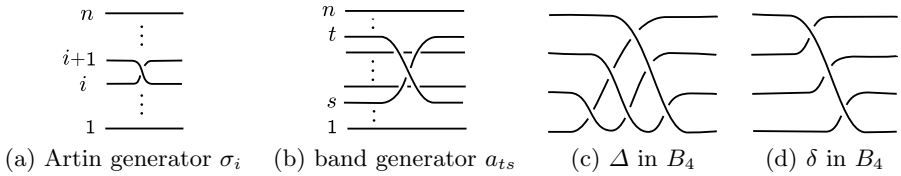
(a) Artin generator σ_i (b) band generator a_{ts} (c) Δ in B_4 (d) δ in B_4

Fig. 1. Generators and fundamental braids

2. There is a *fundamental braid* **D**. It is $\Delta = (\sigma_1 \cdots \sigma_{n-1})(\sigma_1 \cdots \sigma_{n-2}) \cdots \sigma_1$ in the Artin presentation and $\delta = a_{n(n-1)}a_{(n-1)(n-2)} \cdots a_{21}$ in the band-generator presentation as shown in Figure 1 (c), (d). The fundamental braid **D** can be written in many distinct ways as a positive word in both presentations. Due to this flexibility, it has two important properties:
   (i) For each generator a, $\mathbf{D} = aA = Ba$ for some $A, B \in B_n^+$.
   (ii) For each generator a, $a\mathbf{D} = \mathbf{D}\tau(a)$ and $\mathbf{D}a = \tau^{-1}(a)\mathbf{D}$ where τ is the automorphism of B_n defined by $\tau(\sigma_i) = \sigma_{n-i}$ for the Artin presentation and $\tau(a_{ts}) = a_{(t+1)(s+1)}$ for the band-generator presentation.
3. There are partial orders '$\leq$', '$\leq_L$' and '$\leq_R$' in B_n. For two words V and W in B_n, we say that $V \geq W$ (resp. $V \geq_L W$, $V \geq_R W$) if $V = PWQ$ (resp. $V = WP$, $V = PW$) for some $P, Q \in B_n^+$. If a word is compared against either the empty word e or a power of **D**, all three orders are equivalent due to the property (ii) above. Note that the partial orders depend on a presentation of B_n and W is a positive word if and only if $W \geq e$.
4. For two elements V and W in a partial order set, the *meet* $V \wedge W$ (resp. *join* $V \vee W$) denotes the largest (resp. smallest) element among all elements smaller (resp. larger) than V and W. If both the meet and join always exist for any pair of elements in a partially order set, the set is said to have a *combinatorial lattice structure*. The braid group B_n has a combinatorial lattice structure for '$\leq_L$' and '$\leq'_R$' in any of both presentations [3,1]. When we want to distinguish the meet and join for left and right versions, we will use '$\wedge_L$', '$\wedge_R$', '$\vee_R$' and '$\vee_L$'.
5. A braid satisfying $e \leq A \leq \mathbf{D}$ is called a *canonical factor* and $[0,1]_n$ denotes the set of all canonical factors in B_n. The cardinality of $[0,1]_n$ is $n!$ for the Artin presentation, and the nth Catalan number $C_n = \dfrac{(2n)!}{n!(n+1)!}$ for the band-generator presentation. Note that C_n is much smaller than $n!$ and this is one of main reasons why it is sometimes computationally easier to work with the band-generator presentation than the Artin presentation.
6. For a positive braid P, a decomposition $P = A_0 P_0$ is *left-weighted* if $A_0 \in [0,1]_n$, $P_0 \geq e$, and A_0 has the maximal length (or maximal in '$\leq_L$') among all such decompositions. A left-weighted decomposition $P = A_0 P_0$ is unique. A_0 is called the *maximal head* of P. The notion 'right-weighted' can be also defined similarly.
7. Any braid W given as a word can be decomposed uniquely into

$$W = \mathbf{D}^u A_1 A_2 \cdots A_k, \qquad e < A_i < \mathbf{D}, \ u \in \mathbb{Z}, \tag{1}$$

where the decomposition $A_i A_{i+1}$ is left-weighted for each $1 \leq i \leq k - 1$. This decomposition, called the *left canonical form* of W, is unique and so it solves the word problem. The integer u (resp. $u + k$) is called the *infimum* (resp. *supremum*) of W and denoted by $\inf(W)$ (resp. $\sup(W)$). The infimum (resp. supremum) of W is the smallest (resp. largest) integer m such that $\mathbf{D}^{-m}W \geq e$ (resp. $\leq e$). The *canonical length* of W, denoted by $\text{len}(W)$, is given by $k = \sup(W) - \inf(W)$ and will be used as an important parameter together with the braid index n. The right canonical form of W can be also defined similarly.

3   Canonical Factors

3.1   Data Structures

A canonical factor in the Artin presentation of B_n can be identified with the associated n-permutation, which is obtained by replacing the i-th Artin generator σ_i by the transposition of i and $i+1$. We represent an n-permutation as an array A of n integers, where $A[i]$ is equal to the image of i under the permutation. A is called a *permutation table*.

A canonical factor in the band-generator presentation is also uniquely determined by the associated permutation. Thus a canonical factor can be represented by a permutation table as before, but a permutation is associated to a canonical factor in the band-generator presentation only if it is a product of "disjoint parallel descending cycles" [1]. Two descending cycles $(s_i s_{i-1} \cdots s_1)$ and $(t_j t_{j-1} \cdots t_1)$, where $s_i > \cdots > s_1$ and $t_j > \cdots > t_1$, are called *parallel* if s_a and s_b do not separate t_c and t_d (i.e. $(s_a - t_c)(s_a - t_d)(s_b - t_c)(s_b - t_d)$ is positive) for all $1 \leq a < b \leq i$ and $1 \leq c < d \leq j$. Thus a canonical factor can also be represented by an array X of n integers where $X[i]$ is the maximum in the descending cycle containing i. X is called a *descending cycle decomposition table*. The permutation table is useful for products and inverses, and the descending cycle decomposition table is useful for the meet operation discussed later. The two tables can be converted in $\mathcal{O}(n)$ time. Thus any one of them can be chosen to implement the braid groups without affecting the complexities of algorithms. We describe concrete algorithms in Algorithm 1 and 2.

Algorithm 1 *Convert a permutation table to a descending cycle decomposition table.*

> **Input:** *permutation table A of length n.*
> **Output:** *descending cycle decomposition table X.*

> for $i \leftarrow 1$ to n do $X[i] \leftarrow 0$;
> for $i \leftarrow n$ to 1 step -1 do begin
>   if $(X[i] = 0)$ then $X[i] \leftarrow i$;
>   if $(A[i] < i)$ then $X[A[i]] \leftarrow X[i]$;
> end

Algorithm 2 *Convert a descending cycle decomposition table to a permutation table.*

> **Input:** *Descending cycle decomposition table X.*
> **Output:** *Permutation table A.*
> *(We need an array Z of size n.)*

> for $i \leftarrow 1$ to n do $Z[i] = 0$;
> for $i \leftarrow 1$ to n do begin
>   if $(Z[X[i]] = 0)$ then $A[i] \leftarrow X[i]$ else $A[i] \leftarrow Z[X[i]]$;
> $Z[X[i]] \leftarrow i$;
> end

3.2   Operations

Comparison. Two given canonical factors are identical if and only if their representations given by either permutation tables or descending cycle decomposition tables are identical. Thus the comparison is an $\mathcal{O}(n)$ operation.

Product and Inverse. The product and inverse operations in permutation groups are done in $\mathcal{O}(n)$. If the product of two canonical factors is again a canonical factor, the composition of associated permutations is the permutation associated to the product in both presentations. Hence in this case the product of canonical factors is computed in $\mathcal{O}(n)$.

The Automorphism τ^u. The automorphism τ defined by $\tau(a) = \mathbf{D}^{-1}a\mathbf{D}$ sends canonical factors to canonical factors. An arbitrary power $\tau^u(a)$ for a canonical factor a can also be computed in $\mathcal{O}(n)$, independent of u, since the permutation table of $\mathbf{D}^u$ can be obtained immediately from the parity (resp. the modulo n residue class) of u in the Artin (resp. band-generator) presentation.

Meet. In the Artin presentation, an algorithm computing the meet of two canonical factors with $\mathcal{O}(n \log n)$ running time and $\mathcal{O}(n)$ space is known [3, Chapter 9]. We explain the idea of the algorithm briefly. Suppose that A and B are canonical factors and $C = A \wedge_L B$ be the left meet. We view A, B and C as permutation tables. The algorithm sorts the integers $1, \ldots, n$ according to the order "$\prec$" defined by $x \prec y$ if and only $C[x] < C[y]$. The final result is the permutation table of the inverse of C, and by inverting it the permutation table of C is obtained. Using the standard divide-conquer trick, we divide the sequence to be sorted into two parts, to say X and Y, sort each of X and Y recursively, and merge them according to $\prec$. In the merging step, we need to compare integers $x \in X$ and $y \in Y$ according to $\prec$. The essential point is that $y \prec x$ if and only if the infimum of $A[i]$ over all $i \in X$ lying in the right-hand side of x is greater than the supremum of $A[j]$ over all $j \in Y$ lying in the left-hand side of y, and the analogous condition holds for B. This can be checked in constant time using tables of infimums and supremums, which can be constructed before the merge

step in linear time proportional to the sum of the sizes of X and Y. Hence the total timing is equal to that of standard divide-conquer sorting, $\mathcal{O}(n \log n)$. We describe the left meet algorithm explicitly in Algorithm 3.

Algorithm 3 *Compute the meet of two canonical factors in the Artin presentation.*

> **Input:** *Permutation tables A, B*
> **Output:** *The permutation table C of the meet $A \wedge_L B$.*
> *(We need arrays U, V, W of size n.)*
>
> Initialize C as the identity permutation;
> Sort $C[1] \cdots C[n]$ according to A and B (see the subalgorithm below);
> $C \leftarrow$ inverse permutation of C;
>
> **Subalgorithm:** *Sort $C[s] \cdots C[t]$ according to A and B.*
>
> if $t \le s$ then return;
> $m \leftarrow \lfloor (s+t)/2 \rfloor$;
> Sort $C[s] \cdots C[m]$ according to A and B;
> Sort $C[m+1] \cdots C[t]$ according to A and B;
> $U[m] \leftarrow A[C[m]]$;
> $V[m] \leftarrow B[C[m]]$;
> if $s < m$ then
>   for $i \leftarrow m-1$ to s step -1 do begin
> $U[i] \leftarrow \min(A[C[i]], U[i+1])$;
> $V[i] \leftarrow \min(B[C[i]], V[i+1])$;
>   end
> $U[m+1] \leftarrow A[C[m+1]]$;
> $V[m+1] \leftarrow B[C[m+1]]$;
> if $t > m+1$ then
>   for $i \leftarrow m+2$ to t do begin
> $U[i] \leftarrow \max(A[C[i]], U[i-1])$;
> $V[i] \leftarrow \max(B[C[i]], V[i-1])$;
>   end
> $l \leftarrow s$;
> $r \leftarrow m+1$;
> for $i \leftarrow s$ to t do begin
>   if $(l > m) \vee ((r \le t) \wedge (U[l] > U[r]) \wedge (V[l] > V[r]))$
>   then $W[i] \leftarrow C[r]$; $r \leftarrow r+1$;
>   else $W[i] \leftarrow C[l]$; $l \leftarrow l+1$;
> end
> for $i \leftarrow s$ to t do $C[i] \leftarrow W[i]$;

The right meet is computed in a similar way, or alternatively by the identity $A \wedge_R B = (A^{-1} \wedge_L B^{-1})^{-1}$, where the inverse notations denote the inverses in the permutation group.

In the band-generator presentation, it is known that the meet of two canonical factors can be computed in $\mathcal{O}(n)$ time [1]. Basically, the meet is obtained by

computing the refinement of the two partitions of $\{1, \ldots, n\}$ that corresponds to the parallel descending cycle decompositions. We describe below an algorithm to compute the meet, which is an improved version of one in [1]. We remark that the left meet and the right meet are the same in the band-generator presentation.

Algorithm 4 *Compute the meet of two canonical factors in band-generator presentation.*

> **Input:** *Descending cycle decomposition tables A and B.*
> **Output:** *The descending cycle decomposition table C of the meet $A \wedge B$.*
>
> for $i \leftarrow 1$ to n do $U[i] \leftarrow n - i + 1$;
> Sort $U[1] \cdots U[n]$ such that
> $(A[U[i]], B[U[i]], U[i])$ is descending in the dictionary order;
> $j \leftarrow U[n]$; $C[j] \leftarrow j$;
> for $i \leftarrow n - 1$ to 1 step -1 do begin
>   if $(A[j] \neq A[U[i]]) \vee (B[j] \neq B[U[i]])$ then $j \leftarrow U[i]$;
> $C[U[i]] \leftarrow j$;
> end

The complexity is determined by the sorting step since all the other parts are done in linear time. In braid cryptosystems, it is expected that n is not so large (perhaps less than 500) and hence it is practically reasonable to apply the bucket sort algorithm. The bucket sort algorithm can be applied twice to sort pairs $(A[U[i]], B[U[i]], U[i])$ lexicographically. (Recall that the original order is preserved as much as possible by the bucket sort.) Since we have at most n possibilities for the values of $A[U[i]]$ and $B[U[i]]$, both space and execution time are linear in n. In some situations, the following trade-off of space and execution time is useful. We may sort the pairs $(A[U[i]], B[U[i]])$ using the bucket sort algorithm once, where $\mathcal{O}(n^2)$-space is required but the practical execution speed is improved. To save space (e.g. on small platforms), usual sorting algorithms by comparisons (e.g. divide-conquer sort) can be applied to get an $\mathcal{O}(n \log n)$ algorithm that requires no additional space.

4   Braids

4.1   Data Structures

Writing a given braid as $\beta = \mathbf{D}^q A_1 A_2 \cdots A_\ell$, where q is an integer and each A_i is a canonical factor, we represent the braid as a pair $\beta = (q, (A_i))$ of an integer q and a list of ℓ canonical factors (A_i) in both presentations. We note that this representation is not necessarily the left canonical form of β, and hence ℓ may be greater than the canonical length of β.

A braid given as a word in generators is easily converted into the above form, in both presentations, by rewriting each negative power σ^{-1} of generators as a product of $\mathbf{D}^{-1}$ and a canonical factor $\mathbf{D}\sigma^{-1}$ and collecting every power of $\mathbf{D}$ at the left end using the fact $(\prod A_i)\mathbf{D}^{\pm 1} = \mathbf{D}^{\pm 1}(\prod \tau^{\pm 1}(A_i))$ for any sequence of canonical factors A_i. This is done in $\mathcal{O}(n\ell)$, where n is the braid index and ℓ is the length of the given word.

4.2   Operations

Group Operations. Basic group operations are easily implemented. From the identity

$$(\mathbf{D}^p A_1 \cdots A_\ell)(\mathbf{D}^q B_1 \cdots B_{\ell'}) = \mathbf{D}^{p+q} \tau^q(A_1) \cdots \tau^q(A_\ell) B_1 \cdots B_{\ell'} \qquad (2)$$

the multiplication of two braids is just the juxtaposition of two lists of permutation and applying τ. The inverse of a braid can be computed using the formula

$$(\mathbf{D}^q A_1 \cdots A_\ell)^{-1} = \mathbf{D}^{-q-\ell} \tau^{-q-\ell}(B_\ell) \cdots \tau^{-q-1}(B_1) \qquad (3)$$

where $B_i = A_i^{-1}\mathbf{D}$, viewing A_i and $\mathbf{D}$ as permutations. Since a power of τ is computed in linear time in n, braid multiplication and inversion have complexity $\mathcal{O}(\ell n)$. A conjugation consists of two multiplications and one inversion, and hence also has the complexity $\mathcal{O}(\ell n)$.

Left Canonical Form. A representation of a braid can be converted into the left canonical form by the algorithms in [3, Chapter 9] and [1]. Given a positive braid $P = A_1 \cdots A_\ell$, where A_i is a canonical factor, the algorithm computes the maximal heads of $A_{\ell-1}A_\ell$, $A_{\ell-2}A_{\ell-1}A_\ell$, ..., $A_1 \cdots A_\ell = P$ sequentially using the following facts [3, Chaper 9] [2] [1].

1. For any positive braid A and P, the maximal head of AP is the maximal head of the product of A and the maximal head of P.
2. For two canonical factors A and B, the maximal head of AB is $A((\mathbf{D}A^{-1})\wedge_L B)$, where the inverse is taken in the permutation group.

From these facts, the i-th maximal head is the maximal head of the product of $A_{\ell-i}$ and the $(i-1)$-st maximal head, and it can be computed using meet operation once. At the last step, we obtain the left weighted decomposition $P = B_1 P_1$. Doing it again for P_1, we obtain the left weighted decomposition $P_1 = B_2 P_2$, and repeating this, finally we obtain the left canonical form of P. Note that this process is very similar to the bubble sort, where the maximum (or minimum) of given elements is found at the first stage, and repeat it for the remaining elements. The complexity of left canonical form algorithm is the same as that of the bubble sort: complexities are $\mathcal{O}(\ell^2 n \log n)$ and $\mathcal{O}(\ell^2 n)$ in the Artin presentation and the band-generator presentations, respectively. The difference comes from the complexity of the meet operation. We describe the left canonical form algorithm in a concrete form.

Algorithm 5 *Convert a braid into the left canonical form.*

   Input: *A braid representation $\beta = (p, (A_i))$.*
   Output: *The left canonical form of β.*

   $\ell \leftarrow \ell(\beta)$;
   $i \leftarrow 1$;

```
while (i < ℓ) do begin
        t ← ℓ;
        for j ← ℓ − 1 to i step − 1 do begin
            B ← (DA_j^{-1}) ∧_L A_{j+1};
            if (B is nontrivial) then begin
                t ← j; A_j ← A_jB; A_{j+1} ← B^{-1}A_{j+1};
            end
        end
        i ← t + 1;
end
while (ℓ > 0) ∧ (A_1 = D) do begin
        Remove A_1 from β; ℓ ← ℓ − 1; p ← p + 1;
end
while (ℓ > 0) ∧ (A_ℓ is trivial) do begin
        Remove A_ℓ from β; ℓ ← ℓ − 1;
end
```

The multiplications and inversions in lines 6 and 8 are performed viewing $\mathbf{D}$, B and A_k as permutations.

We remark that Algorithm 5 can be modified for parallel processing. For convenience, we denote the job of lines 6–9 for (i, j) by $S(i, j)$. Then $S(i, j)$ can be processed after $S(i − 1, j − 1)$ is finished. Thus the jobs $S(1, k), S(2, k + 2), \ldots, S(\ell − 1, k + 2(\ell − 2))$ can be processed simultaneously for $k = \ell − 1, \ell − 2, \ldots, 1, 0, −1, \ldots, −\ell + 3$. ($S(i, j)$ for invalid (i, j) is ignored here.) This method offers algorithms with $\mathcal{O}(\ell n \log n)$ and $\mathcal{O}(\ell n)$ execution time in the Artin and the band-generator presentation, using $\mathcal{O}(\ell)$ processors.

Comparison. In order to compare two braids β_1 and β_2 with ℓ_1 and ℓ_2 canonical factors, we need to convert them into their canonical forms since the same braid can be represented in different forms. Assuming β_1 and β_2 are in left canonical form, the comparison is done by comparing the exponents of $\mathbf{D}$ and the lists of canonical factors, and so has complexity $\mathcal{O}(\min\{\ell_1, \ell_2\} \cdot n)$. Without the assumption, the total complexity of comparison is equal to that of the conversion into left canonical form, $\mathcal{O}(\min\{\ell_1, \ell_2\} \cdot n \log n)$ and $\mathcal{O}(\min\{\ell_1, \ell_2\} \cdot n)$ for the Artin presentation and band-generator presentation, respectively. (Note that for comparison, Algorithm 5 can be executed simultaneously for β_1 and β_2 to extract the canonical factors in the left canonical forms, and stopped if either different canonical factors are found or nothing is left for any one of β_1 and β_2.)

5   Random Braids

Random braids play an important role in braid cryptosystems [5,7]. Since the braid group B_n is discrete and infinite, a probability distribution on B_n makes no sense. But there are finitely many positive n-braids with ℓ canonical factors, we may consider randomness for these braids. Since such a braid can be generated by concatenating ℓ random canonical factors, the problem is reduced to how to choose a random canonical factors in both presentations.

5.1   Artin Presentation

In the Artin presentation of B_n, a canonical factor can be chosen randomly by generating a random n-permutation. It is well known that this is done by using a random number oracle $(n-1)$ times; we start with the identity permutation table A, and for $i = 1, 2, \ldots, n-1$, pick a random number j between i and n and swap $A[i]$ and $A[j]$.

5.2   Band-Generator Presentation

In the band-generator presentation, we need more complicated arguments. Parallel descending cycle decompositions can be identified with non-crossing partitions of the set $\{1, \ldots, n\}$. It is known that they are again naturally bijective to the set BS_n of *ballot sequences* $s_1 s_2 \cdots s_{2n}$ of length $2n$, which are defined to be sequences satisfying $s_1 + \cdots + s_k \geq 0$ for all k and $s_1 + \cdots + s_{2n} = 0$ (e.g. see [8]). Of course, $|BS_n|$ is equal to the n-th Catalan number C_n. The recurrence relation

$$C_n = C_0 C_{n-1} + C_1 C_{n-2} + \cdots + C_{n-1} C_0 \tag{4}$$

can be naturally interpreted by means of ballot sequences as follows. For a given ballot sequence $s_1 \cdots s_{2n}$, choose the minimal i such that $s_1 + \cdots + s_i = 0$. Then $s_1 = 1$, $s_i = -1$ and the subsequences $s_2 \cdots s_{i-1}$ and $s_{i+1} \cdots s_{2n}$ are again ballot sequences of length $2(i-1)$ and $2(n-i)$, respectively. This establishes a bijection between BS_n and the disjoint union $\bigcup_{i=1}^{n-1} BS_{i-1} \times BS_{n-i}$. We inductively define a linear order on BS_n via the bijection, by the following rules: elements in $BS_{i-1} \times BS_{n-i}$ are smaller than elements in $BS_{j-1} \times BS_{n-j}$ if and only if $i < j$, and elements in $BS_{i-1} \times BS_{n-i}$ are lexicographically ordered. Then a random ballot sequence can be generated as follows. Choose a random number k between 1 and C_n, and take the k-th ballot sequence. Algorithm 6 does the second step, by tracing the above bijection recursively. By an induction, it can be shown that the running time of Algorithms 6 is $\mathcal{O}(n \log n)$.

Algorithm 6 *Construct the k-th ballot sequence of length $2n$.*

> **Input:** *An integer k between 1 and C_n.*
> **Output:** *The k-th ballot sequence $s_1 \cdots s_{2n}$.*
>
> if $k \leq C_0 C_{n-1}$ then $i \leftarrow 1$;
> elseif $k > C_n - C_{n-1} C_0$ then begin $i \leftarrow n$; $k \leftarrow k - C_n + C_{n-1} C_0$; end
> else for $i \leftarrow 1$ to n do
>   if $(k \leq C_{i-1} C_{n-i})$ then break;
>   else $k \leftarrow k - C_{i-1} C_{n-i}$;
> $x \leftarrow \lfloor k/C_{n-i} \rfloor$; $y \leftarrow k - x C_{n-i}$;
> $s_1 \leftarrow 1$; $s_{2i-1} \leftarrow -1$;
> if $i > 1$ then $s_2 \cdots s_{2i-2} \leftarrow$ the $(x+1)$-st ballot sequence of length $2(i-1)$;
> if $i < n$ then $s_{2i} \cdots s_{2n} \leftarrow$ the $(y+1)$-st ballot sequence of length $2(n-i)$;

A ballot sequence can be transformed to a permutation table associated to a canonical factor in the band generator presentation, via the correspondence between ballot sequences and non-crossing partitions of $\{1, \ldots, n\}$ [8]. We describe an $\mathcal{O}(n)$ algorithm.

Algorithm 7 *Convert a ballot sequence to a disjoint cycle decomposition table.*

> **Input:** *A ballot sequence* $s_1 \cdots s_{2n}$.
> **Output:** *A permutation table* A.
> *(We need a stack S of maximal size n.)*

> for $i \leftarrow 1$ to $2n$ do begin
>   if $s_i = 1$ then push i into S;
>   else begin
>   Pop j from S;
>   if i is odd then $A[(i + 1)/2] = j/2$
>   else $A[j/2] = (i + 1)/2$;
>   end
> end

In the above discussion, we assume that the Catalan numbers C_n is known. It is not a severe problem, since a table of C_n can be computed very quickly using the recurrence relation $C_{n+1} = (4n + 2)C_n/(n + 2)$. If you want to avoid division of big integers, the recurence relation (4) is useful.

We finish this section with a remark on the distribution generated by out algorithm. Since the same braid can be represented in different ways in our implementation, the distribution is not uniform on the set of positive n-braids of canonical length ℓ. However, the distribution has a property that more complex braids, which can be represented in more different ways, are generated with higher probability. It seems to be a nice property for braid cryptosystems.

6   Performance

In this section we consider the braid cryptosystem proposed in [7], which is a revised version of one in [5]. Let LB_n and UB_n be the subgroups of B_n generated by the Artin generators $\sigma_1, \ldots, \sigma_{\lfloor n/2 \rfloor - 1}$ and $\sigma_{\lfloor n/2 \rfloor}, \ldots, \sigma_n$, respectively. A secret key is given as a pair (a_1, a_2), where a_1 and a_2 are in LB_n, and the associated public key is a pair (x, y) such that $y = a_1 x a_2$. The encryption and decryption scheme is as follows.

Encryption Given a message $m \in \{0, 1\}^M$,
    1. Choose $b_1, b_2 \in UB_n$.
    2. Ciphertext is $(c_1, c_2) = (b_1 x b_2, H(b_1 y b_2) \oplus m)$.
Decryption Given a ciphertext (c_1, c_2), $m = H(a_1 c_1 a_2) \oplus c_2$.

In the above scheme, $H: B_n \rightarrow \{0, 1\}^M$ is a collision-free hash function. H can be obtained by composing a collision free hash function of bitstrings into $\{0, 1\}^M$ with a conversion function of braids into bitstrings. A braid given as its

left canonical form $\mathbf{D}^u A_1 \cdots A_\ell$ can be converted into a bitstring by dumping the integer u and the permutation tables of A_i as binary digits for $i = 1, \ldots, \ell$ sequentially. Since different braids are converted into different bitstrings, this conversion can be used as a part of the hash H.

We remark that if the secret key is of the form (a, a^{-1}) and b_1^{-1} is taken as b_2 in the above encryption procedure, the cryptosystem in [5] is obtained. Hence in performance issues, there is no difference between the cryptosystems in [7] and [5].

The above scheme is easily implemented based on our works. In the encryption, two random braid generations, four multiplications and two left canonical form operations are involved. In the decryption, two multiplications and one left canonical form operation are involved. Thus both operations have running time $\mathcal{O}(\ell^2 n \log n)$ and $\mathcal{O}(\ell^2 n)$ in the Artin and the band-generator presentation, respectively. In Table 1, we show the performance of an implementation of the cryptosystem using the Artin presentation, at various security parameters suggested in [5]. The security levels are estimated using the results of [7]. In order to focus on the performance of braid operations, the execution time of the hash function is ignored. This experiment is performed on a computer with a Pentium III 866MHz processor.

Table 1. Performance of the braid cryptosystem at various parameters

n	ℓ	Block Size (Kbyte)	Encryption Speed (Block/sec)	(Kbyte/sec)	Decryption Speed (Block/sec)	(Kbyte/sec)	Security Level
100	15	1.97	74.46	146.53	95.60	188.13	2^{85}
150	20	4.36	37.44	163.40	47.42	206.94	2^{125}
200	30	9.34	17.21	160.71	22.30	208.26	2^{199}
250	40	16.36	10.61	173.66	13.62	222.78	2^{280}

7   Conclusion

Table 2 summaries braid algorithms discussed and their complexities. In Input and Output columns, PT, DT, AB and BB mean a permutation table, a descending cycle decomposition table, a braid given by the Artin presentation and a braid given by the band-generator presentations, respectively. As usual n is the braid index and ℓ the maximum of canonical lengths (or numbers of canonical factors) of input braids, except for the comparison algorithm, where ℓ denotes the minimum of canonical lengths of two given braids. The complexities of the algorithms are measured by the number of steps required. The space complexities of the algorithms are easily seen to be either constant or linear.

Table 2. Complexities of braid algorithms

Operation	Input	Output	Complexity	Reference
PT → DT	PT	DT	$\mathcal{O}(n)$	Alg. 1
DT → PT	DT	PT	$\mathcal{O}(n)$	Alg. 2
Product	PT	PT	$\mathcal{O}(n)$	3.2
Inverse	PT	PT	$\mathcal{O}(n)$	3.2
τ^k	PT	PT	$\mathcal{O}(n)$	3.2
Meet (Artin)	PT	PT	$\mathcal{O}(n \log n)$	Alg. 3
Meet (Band)	DT	DT	$\mathcal{O}(n)$	Alg. 4
Comparison	PT (or DT)	True/False	$\mathcal{O}(n)$	3.2
Random (Artin)		PT	$\mathcal{O}(n)$	5.1
Random (Band)		PT	$\mathcal{O}(n \log n)$	5.2, Alg. 6, 7
Product	AB (or BB)	AB (or BB)	$\mathcal{O}(\ell n)$	4.2
Inverse	AB (or BB)	AB (or BB)	$\mathcal{O}(\ell n)$	4.2
Left Canonical	AB	AB	$\mathcal{O}(\ell^2 n \log n)$	Alg. 5
Form	BB	BB	$\mathcal{O}(\ell^2 n)$	Alg. 5
Comparison	AB	True/False	$\mathcal{O}(\ell^2 n \log n)$	4.2
	BB	True/False	$\mathcal{O}(\ell^2 n)$	4.2
Random		AB	$\mathcal{O}(\ell n)$	5
		BB	$\mathcal{O}(\ell n \log n)$	5

References

1. J. S. Birman, K. H. Ko and S. J. Lee, *A new approach to the word and conjugacy problem in the braid groups*, Advances in Mathematics 139 (1998), 322-353.
2. E. A. Elrifai and H. R. Morton, *Algorithms for positive braids*, Quart. J. Math. Oxford 45 (1994), 479–497.
3. D. Epstein, J. Cannon, D. Holt, S. Levy, M. Paterson and W. Thurston, *Word processing in groups*, Jones & Bartlett, 1992.
4. F. A. Garside, *The braid group and other groups*, Quart. J. Math. Oxford 20 (1969), no. 78, 235–254.
5. K. H. Ko, S. J. Lee, J. H. Cheon, J. H. Han, J. S. Kang and C. Park, *New public key cryptosystem using braid groups*, Advances in Cryptology, Proceedings of Crypto 2000, Lecture Notes in Computer Science 1880, ed. M. Bellare, Springer-Verlag (2000), 166–183.
6. E. Lee, S. J. Lee and S. G. Hahn, *Pseudorandomness from braid groups*, Advances in Cryptology, Proceedings of Crypto 2001, Lecture notes in Computer Science 2139, ed. J. Kilian, Springer-Verlag (2001), 486–502.
7. K. H. Ko, et al., *Mathematical security analysis of braid cryptosystems*, preprint.
8. R. P. Stanley, *Enumerative combinatorics*, Wadsworth and Brooks/Cole, 1986.