Resistance Against Timing Attacks for Non-commutative Cryptography

Collin Berman

Abstract— **Constant-time programs are investigated in the domain of non-commutative cryptography. We present a constant-time implementation of an algorithm to compute normal forms in the braid group. The notion of constant-time is applied in the generic case to demonstrate potential cryptographic use for Dehornoy's handle reduction algorithm despite its unknown worst-case complexity.**

I. Introduction

Cryptography, the study of secret writing, has been around for thousands of years. One of the main problems in cryptography is to encrypt a message so that it can only be decrypted using a secret key. Classically, the communicating parties had to meet privately to arrange a secret key before they could encrypt messages. This changed when Diffie, Hellman [16], and Merkle [24] published protocols for two parties to securely generate a shared secret key by publicly exchanging unsecured messages.

The security of the Diffie-Hellman protocol and other widely-deployed cryptosystems rely on the intractability of computational problems on integers. For example, products of large primes are thought to be difficult to factor. However, it is unknown whether these problems are actually intractable, or whether it is simply the case that no one has discovered an efficient algorithm yet.

In order to achieve higher levels of confidence in cryptography, Wagner and Magyarik [29] proposed a cryptosystem based on a provably intractable problem. Instead of using integers as their platform, they present a system based on a non-commutative structure. Such a structure differs from the integers in that multiplication depends on the order of the factors: matrix multiplication, for example, is not commutative.

Although non-commutative cryptography has developed into a research field in its own right, interest remains for the most part purely academic. Large-scale cryptographic deployments, such as transport layer security [15], still use traditional commutative cryptosystems. Software implementations of non-commutative structures, such as the CRyptography And Groups (CRAG) Library [26], are not held to the same standard of security as other common cryptographic libraries.

Consider the Networking and Cryptography library (NaCl), which is used by networked systems that handle billions of queries a day from millions of computers [5]. After carefully selecting secure cryptographic protocols, the developers of NaCl made sure that their implementations are resistant to timing attacks, that is, attacks that extract secret keys by timing how long it takes for a system to respond to queries.

Resistance to timing attacks comes at a cost: algorithms must be implemented to run in the same amount of time, regardless of the secret inputs (for fixed input size). Such an implementation is called constant-time. In general, constant-time implementations of an algorithm will be less efficient on some inputs than a typical implementation of the same algorithm. This requirement has a drastic effect on the algorithms that may be used in a cryptographic protocol. Algorithms that are efficient for most–but not all–inputs should be passed over in favor of algorithms with better worst-case complexity, even if there is a hit to the generic-case complexity.

Our contributions: We present a constant-time implementation of Garside's normal form algorithm, an important algorithm in non-commutative cryptography. Such an implementation is required in order to achieve security of non-commutative cryptographic libraries, even if the protocols used are provably secure.

We chose Garside's algorithm for constant-time implementation due to its excellent worst-case complexity. Even though other algorithms may run faster in practice [14], their worst-case complexities make them unsuitable for constant-time implementation. However, we show that it is sufficient for an implementation to be constant-time only on generic, or most, inputs, rather than the entire input space. By focusing on typical inputs, it could be possible to obtain much faster algorithms, while still maintaining resistance to timing attacks.

Organization: In Section II we provide a basic mathematical background for the reader who is unfamiliar with cryptography or complexity theory. In Section III we provide further background on non-commutative cryptography. In Section IV we discuss our constant-time implementation of Garside's normal form algorithm. In Section V we investigate the notion of programs that are constant-time for most inputs. We conclude in Section VI.

II. Background

A. Cryptography

A classic problem in cryptography is to encrypt a message so that only authorized parties may decrypt and recover the plaintext. This can be achieved by using secret keys that are shared amongst the communicating parties. Historically, the parties had to agree on the secret key before they could securely exchange messages. For example, during World War II, the German Luftwaffe's Enigma machines used secret-key cryptography, switching to a new key every day. Because all communicating parties needed the keys to encrypt and decrypt messages, the Luftwaffe distributed monthly lists of secret keys. But if an enemy gained access to just a single copy of a key list, the security of the operation was compromised for the entire month.

The nature of encryption fundamentally changed in 1976, when researchers Diffie and Hellman gave a way for two parties to securely generate a secret key by exchanging messages over an insecure channel [16]. Now instead of distributing lists of keys in advance, parties wishing to communicate securely could generate secret keys on the fly. This secret key could then be used to encrypt secret messages, in such a way that a malicious adversary would be unable to decrypt the messages.

When Alice and Bob want to use the Diffie–Hellman key exchange protocol, they first choose a prime p. All successive computations will then be carried out modulo p. We denote the set of integers modulo p by $\mathbb{Z}_p$. The prime p is made public, as is a small number $g \in \mathbb{Z}_p$. As seen in the diagram below, Alice then randomly samples an element a from $\mathbb{Z}_p$, and sends Bob g^a.

$$\begin{array}{lcl} \textbf{Alice} & & \textbf{Bob} \\ \hline a \leftarrow_{\$} \mathbb{Z}_p & & \\ & \xrightarrow{g^a} & \\ & & b \leftarrow_{\$} \mathbb{Z}_p \\ & \xleftarrow{g^b} & \\ \mathsf{k} \leftarrow (g^b)^a = g^{(ba)} & & \mathsf{k} \leftarrow (g^a)^b = g^{(ab)} \end{array}$$

Similarly, Bob randomly samples b from $\mathbb{Z}_p$ and sends g^b to Alice. Now Alice and Bob are both able to compute the same secret key $\mathsf{k} = g^{ab} = g^{ba}$.

The security of this protocol relies be reduced to the difficulty of the *discrete logarithm problem*. That is, it should be hard for an attacker to recover a from g^a. If this were possible, the attacker could compute the key k just as Alice does. Fortunately, the best known algorithms for attacking this problem are not very fast. Unfortunately, they are also not slow enough to defend against a sufficiently advanced adversary [1].

B. Complexity Theory

Computational problems: In this section we formalize the notion of a computational problem, following the presentation given in [25]. Consider the problem of deciding whether a given natural number is even. The problem consists of the set of possible *instances* for the problem, namely all natural numbers, and the subset of *positive* instances, here the set of even natural numbers. In order to concretely specify these sets, a representation for numbers must be chosen. One possible choice is to write a number in binary as a string of 0's and 1's. We write $\{0,1\}^n$ for the set of all strings of 0's and 1's of length n, and we denote the set of strings of 0's and 1's of arbitrary length (including the empty string ϵ of length 0) by $\{0,1\}^*$.

We can now define the problem of deciding whether a natural number is even as $\mathcal{E} = (E, \{0,1\}^*)$, where $E \subseteq \{0,1\}^*$ is the subset of binary strings representing those natural numbers that are divisible by 2. Since a natural number is even if and only if the final, least-significant digit in its binary expansion is 0, we have the explicit description

$$E = \{0,1\}^* 0 \cup \{\epsilon\} \tag{1}$$

(we choose to interpret the empty string ϵ as the natural number 0). That is, a string of 0's and 1's is in E whenever it ends in a 0 or is empty.

The explicit description of the positive instances E in Equation (1) suggests a simple algorithm to solve the decision problem $\mathcal{D}$. Define an algorithm

$$\mathcal{B}(w) = \begin{cases} \textit{Yes} & \text{if } w = \epsilon \\ \textit{Yes} & \text{if } w = u0 \text{ for some } u \in \{0,1\}^* \\ \textit{No} & \text{otherwise} \end{cases},$$

which takes a string $w \in \{0,1\}^*$ as input and returns *Yes* or *No* according to the final binary digit in w. By (1), we have that for any string $w \in \{0,1\}^*$,

$$\mathcal{B}(w) = \textit{Yes} \text{ if and only if } w \in E,$$

and we say that $\mathcal{B}$ *decides* the problem $\mathcal{E} = (E, \{0,1\}^*)$.

In general, a decision problem $\mathcal{D} = (L, I)$ is given by its instances I and subset of positive instances L. An algorithm $\mathcal{A}$ decides $\mathcal{D}$ if, for all $w \in I$,

$$\mathcal{A}(w) = \textit{Yes} \text{ if and only if } w \in L.$$

In contrast to the description for E given in Equation (1), in general the positive instances L need not be explicitly described. In fact, L may not even admit such an explicit description.

Worst-case complexity: How efficient is the algorithm for deciding evenness described above? No matter how long a string of 0's and 1's it is given, the algorithm only ever inspects the final digit. Thus the time the algorithm takes to run does not depend on the size of its input, and we say the *runtime* is *constant*.

Consider now the harder problem of deciding whether a given natural number is prime. A naive algorithm for this problem may check whether the given number n is divisible by any smaller number m with $2 \leq m < n$. Such a divisor exists if and only if n is composite, so this algorithm decides the problem. This algorithm, which we denote $\mathcal{A}_1$, takes longer to run when n is bigger.

We may construct a more efficient algorithm by noticing that if n has a factor $d > \sqrt{n}$, then it also has a smaller factor $\frac{n}{d} < \sqrt{n}$. Thus instead of testing divisibility up to $n-1$, we only need to check for divisors in the range $2 \leq m \leq \sqrt{n}$. Denote this algorithm by $\mathcal{A}_2$.

If we were to implement these algorithms in some programming language, what would their runtime be for a given input n? A precise answer to this question depends on how many steps it takes to test divisibility and whether an algorithm requires some overhead for setup. Furthermore, different choices of programming languages or hardware platforms may lead to different answers.

To eliminate these dependencies on particular implementation choices, we choose to ignore constants in the runtime by using big-O notation. Even though the algorithm $\mathcal{A}_1$ may take $2(n-2)$ or $100(n-2)$ steps depending on how divisibility is tested, we say that $\mathcal{A}_1$ has runtime $O(n)$. Similarly, algorithm $\mathcal{A}_2$ has runtime $O(\sqrt{n})$. A formalization of this notation may be found in [12].

Note that if algorithm $\mathcal{A}_2$ finds that 2 divides n, it may immediately return, declaring that n is composite, without needing to check for other factors. Yet the algorithm cannot be sure n is prime unless it checks all the way up to $\sqrt{n}$. Even though $\mathcal{A}_2$ sometimes returns after the first check, its *worst-case* runtime is $O(\sqrt{n})$.

We now move to the general case. Let $\mathcal{D} = (L, I)$ be a decision problem and $\mathcal{A}$ an algorithm for $\mathcal{D}$. In order to characterize the runtime of $\mathcal{A}$ in terms of the size of its inputs, we require a size function $s : I \to \mathbb{N}$. Then a function $f : \mathbb{N} \to \mathbb{N}$ is an *upper bound* for $\mathcal{A}$ if evaluating $\mathcal{A}$ on $w \in I$ takes no more than $f(s(w))$ steps. If this holds, the runtime of $\mathcal{A}$ is $O(f(n))$, and we say that $\mathcal{D}$ is *decidable within time* $f(n)$.

III. Non-commutative Cryptography

In their paper outlining their key exchange protocol [16], Diffie and Hellman express interest in finding a cryptographic scheme whose security rests on the difficulty of problems provably harder than the discrete logarithm problem. We present here the Diffie–Hellman-like protocol from [21].

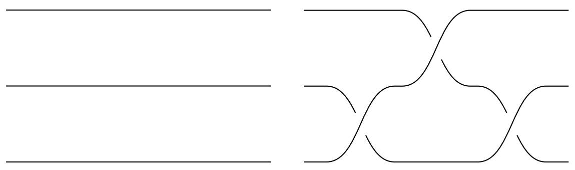
The correctness of the Diffie–Hellman key exchange protocol comes from the fact that $ab = ba$ when a and b are integers. This property of integer multiplication is known as *commutativity*, and is required for the Diffie–Hellman protocol to work. On the other hand, the element g is never required to commute with either a or b.

This suggests generalizing the Diffie-Hellman key exchange protocol to use a *non-commutative* structure in place of $\mathbb{Z}_p$. Non-commutativity means that the result of a multiplication depends on the order of the products, as in matrix computation. For instance,

$$\begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix},$$

whereas

$$\begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}.$$


(a) The identity braid (b) A nontrivial braid

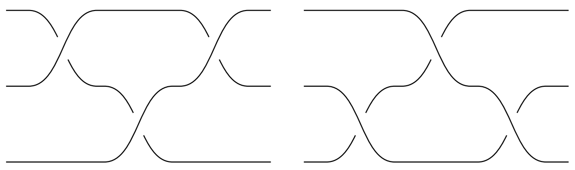

(c) Braid (b) again (d) The inverse of (b)

Fig. 1. Example braids on 3 strands.

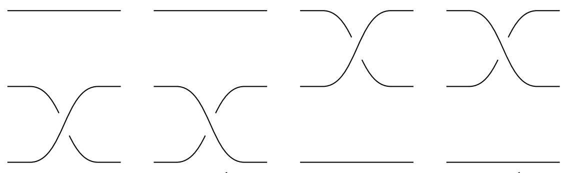
A. The Braid Group

In [21], Ko, Lee et al. generalize Diffie–Hellman key exchange by replacing the set $\mathbb{Z}_p$ of integers modulo p with a particular non-commutative structure called the *braid group*. Braids admit numerous rigorous definitions via different mathematical constructions [6], but can also be intuitively described geometrically: for some natural number n (called the *width*), the braid group B_n consists of sets of n intertwined, non-intersecting strings called *braids*. Two braids are considered equivalent if one can be transformed into the other by deforming the strands without passing one through another.

Two braids in B_n can be combined to form a new braid by tying the bottoms of the strands of one braid to tops of another. This operation gives B_n the structure of a *group*, that is, it has an identity and each element has an inverse. In the group of integers, the identity element is 0, since $0 + n = n$ for any integer n, and inverses are given by negation, since $n + (-n) = 0$ for any n.

Figure 1 shows the identity braid and an example of inverse braids in B_3 (the reader is invited to draw the braids in (b) and (d) next to each other and verify that the strands can be shifted to yield the identity braid). Braid (c) in the figure represents the same braid as in (b), but some of the strands have been moved.

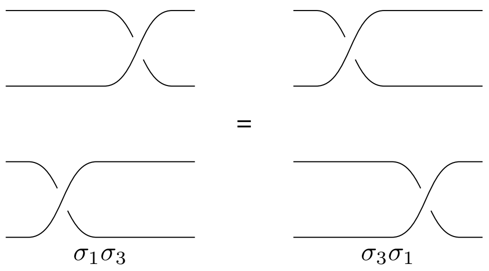
An algebraic presentation: Although the geometric definition of braids given above provides good intuition for the braid group, it is unclear how a braid can be represented by a computer. Instead of drawing a picture, a braid can be described in terms of its crossings. For a braid in B_3, there are 4 possible crossings:


σ_1 σ_1^{-1} σ_2 σ_2^{-1}

Using this notation, the braid in Figure 1 (b) can be written

$\sigma_1\sigma_2\sigma_1$, whereas the braid shown in (c), which is actually the same braid, is written $\sigma_2\sigma_1\sigma_2$. Thus we see that while this notation gives a simple way to write down a braid, the same braid may be written in multiple ways.

For general n, a braid in B_n can be written as a product of the generators $\sigma_1, \dots, \sigma_{n-1}$ and their inverses, where σ_i represents crossing strand i over strand $i+1$. Such a product is called a *braid word*, and the set of all words in these generators is denoted W_n. The equality $\sigma_1\sigma_2\sigma_1 = \sigma_2\sigma_1\sigma_2$ of 3-strand braids carries over to B_n for $n > 3$. In general, $\sigma_i\sigma_{i+1}\sigma_i = \sigma_{i+1}\sigma_i\sigma_{i+1}$. Yet another relation can occur when $n > 3$. Since two crossings can be taken in any order if they deal with different strands, we have $\sigma_i\sigma_j = \sigma_j\sigma_i$ whenever $|i-j| \geq 2$.



In [8], Bohnenblust shows that the two relations described above are essentially the only relations (besides $\sigma_i\sigma_i^{-1} = e$, the identity braid) one needs in order to move between two different words representing the same braid. This property can be written succinctly as

$$B_n = \left\langle \sigma_1, \dots, \sigma_{n-1} \;\middle|\; \begin{array}{l} \sigma_i\sigma_{i+1}\sigma_i = \sigma_{i+1}\sigma_i\sigma_{i+1} \\ \sigma_i\sigma_j = \sigma_j\sigma_i, \quad |i-j| \geq 2 \end{array} \right\rangle . \tag{2}$$

The *word problem* for the braid group asks for an algorithm to decide when to words represent the same braid (or equivalently, whether a word represents the identity braid). The existence of the presentation in (2) does not give a solution to the word problem, since it does not specify in what order the relations should be applied. In general, a finitely presented group need not admit a solution to the word problem [27], but many algorithms exist for solving the word problem in the braid group [6].

B. Ko–Lee Key Exchange

Now that the braid group B_n has been defined, we can present the Ko–Lee key exchange protocol from [21]. As in Diffie–Hellman key exchange, the protocol has public parameters. When Alice and Bob want to use the Ko–Lee protocol, they choose some natural number n, a base element $x \in B_n$, and two *commuting* subgroups $A, B \leq B_n$. Such commuting subgroups can be constructed using the second relation of the presentation 2 by

$$A = \langle \sigma_1, \dots, \sigma_{\lfloor n/2 \rfloor - 1} \rangle,$$

$$B = \langle \sigma_{\lceil n/2 \rceil + 1}, \dots, \sigma_{n-1} \rangle.$$

To carry out the key exchange, Alice selects a random secret braid $a \in A$, and sends Bob the *conjugate* axa^{-1}. Similarly, Bob sends Alice bxb^{-1}, where $b \in B$ is Bob's secret braid. Now Alice and Bob can both compute the same secret key $(ab)x(ab)^{-1}$, since A and B commute.

Alice		**Bob**
$a \leftarrow_{\$} A$		
	$\xrightarrow{axa^{-1}}$	
		$b \leftarrow_{\$} B$
	$\xleftarrow{bxb^{-1}}$	
$\mathsf{k} \leftarrow a(bxb^{-1})a^{-1}$		$\mathsf{k} \leftarrow b(axa^{-1})b^{-1}$

IV. A Constant-time Program for Computing Normal Forms in the Braid Group

When two parties Alice and Bob use the Ko–Lee key exchange protocol in practice, they will choose their secret braids a and b as words in the generators $\sigma_1, \dots, \sigma_{n-1}$, rather than as abstract braids in B_n. Similarly, the base element x will be specified as a word in W_n. Note that the inverse of a word can be computed easily using the fact $(uv)^{-1} = v^{-1}u^{-1}$ for $u, v \in B_n$.

Now when Alice sends Bob axa^{-1}, she cannot simply concatenate the respective words and send the result over the insecure channel. Since $x \in W_n$ is public, an adversary would be able to recover a by finding the substring of axa^{-1} matching x. Using this knowledge, they would be able to compute the secret key using Alice's formula in terms of bxb^{-1}, breaking the security of the protocol.

Thus Alice needs a way to hide the structure of her conjugate axa^{-1}. This can be accomplished by computing a *normal form* of axa^{-1}. A normal form is a canonical way of writing a braid $y \in B_n$ as a word $w \in W_n$. Any two words representing the same braid have the same normal form (so normal forms provide a solution to the word problem).

There are multiple algorithms for computing normal forms in the braid group [7], [14], but not all are suited for cryptographic use. In this section, we discuss how a normal form algorithm can be implemented in a cryptographically secure way.

A. Constant-time Programs

Timing attacks: When researchers design new cryptography, it is common practice for them to accompany their construction with a proof of its security. While this can increase confidence that the new scheme will be unbreakable, the mathematical model describing the scheme often fails to completely match reality. As such, it can be possible to defeat the security of a cryptographic scheme, even when the scheme has been proven secure.

As an example, consider Krawczyk's 2001 proof [23] that the order of encryption and authentication in SSL is secure. Using formal mathematical models for secure encryption and authentication primitives, Krawczyk proved that authenticating first and applying encryption afterwards is quantitatively secure. However, Krawczyk's models failed to account for how SSL servers handle errors. When an adversary sends the server invalid ciphertext, the server responds with an error message instead of an acknowledgement. Access to such extraneous information can allow the adversary to decrypt sensitive network communications [28].

Simply hiding the error message is not enough to restore security. In the event that the ciphertext was correctly generated, the web server still needs to verify that the message is properly authenticated. Because this check only happens for a valid ciphertext, the time it takes for the server to respond will vary based on ciphertext validity. By measuring how long the server takes to respond to various queries, an adversary can intercept passwords that should be protected by SSL [10].

Constant-time programming: Such timing attacks are a widespread problem faced by those who implement cryptography. One effective countermeasure against timing attacks is to ensure that the time a program takes to run does not depend on sensitive inputs, such as plaintext or key material. This approach, known as constant-time programming, clearly prevents timing attacks, since an adversary gains no information by measuring differences in execution times for different inputs. As such, constant-time programming has become widely deployed, and new cryptographic implementations are often designed to run in constant-time [5], [9], [13].

Despite its success in preventing timing attacks, the constant-time programming discipline presents its own set of challenges. Writing a constant-time implementation requires deviating from standard programming practices, resulting in code that can be harder to read [3]. For example, consider a program that assigns a variable *x* one of two values based on a secret value *b*. If *v1* and *v2* are functions that compute the two possible values for *x*, then a developer may implement the program as

```
if b then x := v1() else x := v2().    (3)
```

However, if the function *v1* takes longer to run than *v2*, an attacker who has access to execution times can determine the secret value *b*. This data-dependency can be eliminated by always executing both functions, which can be implemented as

```
x := b*v1() + (1-b)*v2(),    (4)
```

assuming *b* only takes on values 0 or 1.

This example shows that constant-time implementations will generally run slower than an optimized non-constant-time implementation. This property, known as the worst-case principle [2], forces an algorithm's best-case run time to always be as slow as in the worst-case. This principle has a strong effect on the algorithmic choices a programmer makes. Associative arrays, like hash tables, are commonly used for searching data due to their average complexity of $O(1)$, despite their worst-case performance of $O(n)$ when every input hashes to the same value. Even though a well-chosen hash function can ensure the latter case never happens in practice, a constant-time implementation of a hash table will always need to search through the entire table, rendering the table completely useless.

B. Constant-time Normal Forms

The worst-case principle associated with constant-time programs influences what algorithms should be used in a cryptographic protocol. We consider two different algorithms for computing normal forms in the braid group in this context.

In [19], Garside gives the first solution to the conjugacy problem in B_n, which asks to decide, for two braids $a, b \in B_n$, whether there exists another braid $c \in B_n$ such that $b = cac^{-1}$. His solution to the conjugacy problem also gives an algorithm for finding a normal form known today as the Garside normal form. Although Garside was not concerned with the complexity of his algorithm, Thurston showed in [18] that a modified version of the algorithm runs in $O(\ell^2 n \log n)$, where ℓ is the length of the input word. Further improvements in [7] bring the runtime down to $O(\ell^2 n)$.

A fundamentally different normal form, called the Dehornoy handle free form, is given is [14]. Any braid can be converted to its handle free form by applying a sequence of handle reductions. Although the algorithm runs extremely fast in practice, the best upper bound for the number of handle reductions needed to converge to the normal form is $2^{n^4 \ell}$.

Even though Dehornoy's algorithm runs quickly on certain braids, a constant-time implementation would need to apply $2^{n^4 \ell}$ handle reductions on every input braid, making the program unusable in practice. On the other hand, the modified Garside algorithm from [7] yields a constant-time program with worst-case runtime $O(\ell^2 n)$, making this algorithm a better choice for cryptographic use.

We implemented the normal form algorithm given in [7] using the techniques described in [11]. The latter paper provides efficient data structures for computing on braids, but is vulnerable to timing attacks, making it unsuitable for cryptographic use. Our implementation is carefully written to avoid branching on secret data, as described in Equation (4), ensuring that the program executes the same steps in the same order for every braid word of a given width and length.

Our code is available from `https://github.com/cberman/braid-crypto`. We tested the program by using normal forms to verify relations given in [7].

V. Generic Constant-time Programs

As we saw in the previous section, the Garside normal form for braids has a much better worst-case runtime of $O(\ell^2 n)$ [7] compared to the exponential $O(2^{n^4 \ell})$ upper bound for the Dehornoy handle free form [14]. However, the theoretical worst-case runtimes of these two algorithms do not give the complete picture. In Table I, we reproduce measurements of the runtime of handle reduction on random braids from [14]. The table shows that handle reduction is much faster in practice than the greedy normal form algorithm derived from the automatic structure of B_n, despite

	4 strands	16 strands	64 strands
64 crossings	0.20 vs. 5.36	0.03 vs. 8.65	0.016 vs. 23.1
256 crossings	2.71 vs. 77.4	0.45 vs. 105	0.14 vs. 194
1,024 crossings	54.5 vs. 1,526	10.2 vs. 1,378	1.56 vs. 1,899
4,096 crossings	1,560 vs. 29,900	1,635 vs. 21,990	33 vs. 23,640

TABLE I
HANDLE REDUCTION VS. GREEDY NORMAL FORM: COMPARISON OF AVERAGE CPU TIMES IN MILLISECONDS [14]

the latter's $O(\ell^2)$ runtime [18]. In this section we discuss the possibility of preserving this high performance in a constant-time implementation.

A. Background on Generic-case Complexity

In [20], Kapovich, Myasnikov, Schupp, and Shpilrain note that the worst-case complexity (and even the average-case complexity) of an algorithm does not adequately describe its effectiveness at breaking cryptography. For example, an algorithm for breaking encryption that takes exponential time for some small number of inputs, yet runs in polynomial time on almost all inputs, should be considered an effective cryptographic break despite the worst-case exponential runtime.

To characterize such behavior, the authors of [20] formalize the notion of "typical" inputs. We follow their exposition here, but in less generality. Rather than working with arbitrary pseudo-measures on infinite sets, we restrict our attention to the asymptotic density induced by a size function.

Let I be a set with a size function $s : I \to \mathbb{N}$. We denote by B_n the elements of I of size n, that is

$$I_n = \{w \in I \mid s(w) = n\}.$$

For a subset $R \subseteq I$, the probability that a randomly chosen element of I_n lies in R is given by the density

$$\mu_n(R) = \frac{|R \cap I_n|}{|I_n|}$$

of R in I_n. Now the *asymptotic density* of R in I is given by the limit

$$\rho(R) = \lim_{n\to\infty} \mu_n(R). \tag{5}$$

We say that the set R is *generic* in I if $\rho(R) = 1$. As a simple example, the set R of nonzero binary strings is generic in $\{0,1\}^*$, since $\mu_n(R) = 1 - 2^{-n}$ approaches 1 as n goes to infinity.

Thus far, every decision algorithm we have considered has been complete, meaning that it gives the correct answer on every instance $w \in I$. In contrast, a *partial* decision algorithm is allowed to give up on inputs, returning "?" instead of *Yes* or *No*. The subset of instances on which a partial decision algorithm $\mathcal{A}$ produces an answer

$$H_{\mathcal{A}} = \{w \in I \mid \mathcal{A}(w) \neq ?\}$$

is called the *halting set* of $\mathcal{A}$.

Now let $\mathcal{D} = (L, I)$ be a decision problem and $\mathcal{A}$ a partial decision algorithm. If $H_{\mathcal{A}}$ is generic in I, and $\mathcal{A}$ gives the correct answer for each $w \in H_{\mathcal{A}}$, we say that $\mathcal{A}$ *generically decides* $\mathcal{D}$, or that $\mathcal{A}$ is a *generic algorithm* for $\mathcal{D}$. Moreover, a function $f : \mathbb{N} \to \mathbb{N}$ is a *generic upper bound* for $\mathcal{A}$ if evaluating $\mathcal{A}$ on $w \in H_{\mathcal{A}}$ takes no more than $f(s(w))$ steps, where $s : I \to \mathbb{N}$ is a size function.

B. Hardness of generic restrictions

The impressive performance of handle reduction on random braids leads Myasnikov, Shpilrain, and Ushakov [25] to conjecture that Dehornoy's algorithm has linear *generic-case complexity*. The authors note that while other algorithms for the word problem in braid groups have known linear-time generic-case complexity [20], these algorithms do not compute normal forms, and only give fast answers for nontrivial braids. Such algorithms are therefore unsuitable for use in cryptographic protocols, which require efficiently-computable normal forms for nontrivial braids.

Suppose the conjecture holds, and Dehornoy's algorithm computes normal forms in linear time for a generic subset D_n of the set of braid words W_n. Then upon restricting our attention to only the words in D_n, we have an algorithm to compute normal forms with linear-time *worst-case* complexity. Such an algorithm will admit a constant-time implementation with linear-time complexity, outperforming a constant-time implementation for Garside's normal form while still achieving protection against timing attacks.

However, the linear-time algorithm described above would only be useful in preventing timing attacks in the Ko–Lee key exchange protocol if communicating parties only use braids in D_n. Even if an adversary is unable to break the protocol when the full braid group B_n is used, it may appear that restricting to D_n could open up new avenues of attack. In Theorem 3 we will prove in a quantitative sense that, to the contrary, restricting the instances of a hard decision problem to a generic subset does not make the problem easier.

We first formalize the notion of restricting the instances of a decision problem. Let $\mathcal{D} = (L, I)$ be a decision problem. For a subset $I' \subseteq I$, we define the *restricted* decision problem $\mathcal{D}' = (L \cap I', I')$. The size function $s' : I' \to \mathbb{N}$ for the restricted problem is obtained by restricting the domain of the size function $s : I \to \mathbb{N}$ for $\mathcal{D}$. The following proposition shows that the restricted restricted problem $\mathcal{D}'$ is never any harder than the original problem $\mathcal{D}$.

Proposition 1: Let $\mathcal{D} = (L, I)$ be a decision problem, and set $\mathcal{D}' = (L \cap I', I')$ for some $I' \subseteq I$. If $\mathcal{D}$ is decidable within time $f(n)$ for some function $f : \mathbb{N} \to \mathbb{N}$, then so is the restricted problem $\mathcal{D}'$.

Proof: Suppose $\mathcal{A}$ is an algorithm for $\mathcal{D}$ with upper bound $f : \mathbb{N} \to \mathbb{N}$. Since $\mathcal{A}$ gives the correct answer on every instance $w \in I$, it also gives the correct answer for all $w \in I' \subseteq I$, that is, $\mathcal{A}$ decides the restricted problem $\mathcal{D}'$.

Now if $s : I \to \mathbb{N}$ is the size function for $\mathcal{D}$, evaluating $\mathcal{A}$ on any instance $w \in I$ takes no more than $f(s(w))$ steps. Restricting to instances $w \in I'$, we see that evaluating $\mathcal{A}$ takes no more than $f(s'(w))$ steps. Therefore $\mathcal{A}$ decides $\mathcal{D}'$ within time $f(n)$. ■

Restricting the instances of a decision problem never results in a harder problem, but the converse to Proposition 1 may not hold. If $\mathcal{A}'$ is an algorithm for a restriction $\mathcal{D}' = (L \cap I', I')$ of the decision problem $\mathcal{D} = (L, I)$, the algorithm may not even be defined for an instance $w \in I \setminus I'$. Even if $\mathcal{A}'$ produces the correct answer for such an instance outside I', there is no guarantee that evaluation will take no more than $f(n)$ steps when $f : \mathbb{N} \to \mathbb{N}$ is an upper bound for $\mathcal{A}'$.

In Theorem 3 we will give a partial converse to Proposition 1 when I' is generic in I. In this case, we call $\mathcal{D}' = (L \cap I', I')$ a *generic restriction* of $\mathcal{D} = (L, I)$.

We will need the following technical lemma.

Lemma 2: Let Ω be a countably infinite set with size function $s : \Omega \to \mathbb{N}$ and generic subset $R \subseteq \Omega$. If a subset $S \subseteq R$ is generic in R, then it is also generic in Ω.

Proof: Since R is generic in Ω,

$$\lim_{n\to\infty} \frac{|R \cap \Omega_n|}{|\Omega_n|} = 1, \tag{6}$$

where $\Omega_n = \{\omega \in \Omega \mid s(\omega) = n\}$. Similarly, since S is generic in R,

$$\lim_{n\to\infty} \frac{|S \cap R_n|}{|R_n|} = 1, \tag{7}$$

where $R_n = \{\omega \in R \mid s(\omega) = n\} = R \cap \Omega_n$.

Now we compute

$$\begin{aligned} \lim_{n\to\infty} \frac{|S \cap \Omega_n|}{|\Omega_n|} &= \lim_{n\to\infty} \frac{|S \cap \Omega_n|}{|R \cap \Omega_n|} \cdot \frac{|R \cap \Omega_n|}{|\Omega_n|} \qquad (8)\\ &= \left(\lim_{n\to\infty} \frac{|S \cap \Omega_n|}{|R \cap \Omega_n|}\right)\left(\lim_{n\to\infty} \frac{|R \cap \Omega_n|}{|\Omega_n|}\right). \end{aligned}$$

The denominator of the first limit is equal to $|R_n|$ by definition, and the numerator can be rewritten

$$|S \cap \Omega_n| = |(S \cap R) \cap \Omega_n| = |S \cap (R \cap \Omega_n)| = |S \cap R_n|$$

since $S \subseteq R$. Thus the first limit is given by Equation (7), while the second limit is given by Equation (6). Plugging into Equation (8), we find

$$\lim_{n\to\infty} \frac{|S \cap \Omega_n|}{|\Omega_n|} = 1.$$

Therefore S is generic in Ω. ∎

We can now state and prove our main theorem.

Theorem 3: Let $\mathcal{D} = (L, I)$ be a decision problem, with generic restriction $\mathcal{D}' = (L \cap I', I')$ such that membership in I' can be computed in polynomial time. If $\mathcal{D}$ is not decidable generically in polynomial time, then neither is $\mathcal{D}'$.

Proof: We prove the contrapositive, that is, starting from a polynomial generic algorithm for the restricted problem $\mathcal{D}'$, we construct a polynomial generic algorithm for the original problem $\mathcal{D}$. Suppose $\mathcal{A}'$ is a generic decision algorithm for $\mathcal{D}'$ with a generic polynomial upper bound $p(n)$.

We extend $\mathcal{A}'$ to a partial decision algorithm $\mathcal{A}$ for $\mathcal{D}$ by defining

$$\mathcal{A}(w) = \begin{cases} \mathcal{A}'(w) & \text{if } w \in I' \\ ? & \text{otherwise} \end{cases} \tag{9}$$

for all $w \in I$. Clearly $H_{\mathcal{A}} = H_{\mathcal{A}'}$, which is generic in I' since $\mathcal{A}'$ generically solves $\mathcal{D}'$. Then by Lemma 2, $H_{\mathcal{A}}$ is also generic in I, since I' is a generic subset of I. Thus $\mathcal{A}$ generically solves the original decision problem $\mathcal{D}$.

It remains to show that $\mathcal{A}$ has a generic polynomial upper bound. But since membership in I' can be computed in polynomial time, say in $q(n)$ steps, the polynomial sum $p(n) + q(n)$ (plus some constant overhead for combining the membership test and evaluation of $\mathcal{A}'$) serves as a generic upper bound for $\mathcal{A}$. ∎

Note the requirement in Theorem 3 of a polynomial-time algorithm for deciding membership in the set I' of restricted instances. This assumption is necessary due to the undefined behavior of $\mathcal{A}'$ on words $w \in I \setminus I'$. Running $\mathcal{A}'$ on such an instance may either take too long to run or produce an incorrect answer. Such a membership test may be derived from some special structure enjoyed by elements of I', but other methods are possible.

In the case of handle reduction, Dehornoy [14] proved that every sequence of handle reductions from a braid word w of length ℓ and width n converges in at most $2^{n^4 \ell}$ steps. One way to show that the handle reduction algorithm has linear generic-case complexity would be to show that the set of braid words for which handle reduction sequences converge in at most $k\ell$ steps is generic in W_n, for some constant k. Denote this subset of instances by D_n. Now given a word w of length ℓ, membership in D_n can be decided simply by applying handle reduction $k\ell+1$ times, producing a sequence of words

$$w = w_0 \to w_1 \to \cdots \to w_{k\ell} \to w_{k\ell+1}.$$

If $w_{k\ell} = w_{k\ell+1}$ then the algorithm has already converged, so $w \in D_n$ and $w_{k\ell}$ is the handle free normal form for w.

If D_n is generic in W_n, this approach yields a generic algorithm for the word problem for braids with linear-time generic-case complexity. But for this algorithm to be useful in the Ko–Lee key exchange protocol, D_n (or whatever generic subset of W_n a generic solution to the word problem halts on) needs to carry some additional structure. It must be possible to choose the public base element $x \in B_n$ and commuting subgroups $A, B \leq B_n$ (or rather, sets of words representing commuting subgroups) so that axa^{-1}, bxb^{-1}, and $(ab)x(ab)^{-1}$ lie in D_n for all $a \in A$ and $b \in B$.

It is also worth noting that Ko–Lee key exchange is an unauthenticated protocol, and so should be combined with some form of authentication. Otherwise, an adversary who can impersonate Bob to Alice is provided an oracle distinguishing whether $aya^{-1} \in D_n$ for $y \in W_n$ of the adversary's choosing.

VI. Conclusion

Timing attacks successfully break cryptography as it is applied in practice, even when protocols have been proven secure, but constant-time programming provides an effective defense. Non-commutative cryptography promises various cryptographic primitives, but the lack of constant-time implementations precludes their deployment. Our implementation of an algorithm for computing normal forms in the braid group is an important first step towards preparing non-commutative cryptography for real-world use, but much work remains to be done.

Although our implementation is secure against timing-attacks by an adversary who watches the control flow of the program, it is not protected against more powerful adversaries. In particular, our program contains no defenses against cache-based timing attacks, which exploit variance in the time it takes to fetch data from the cache [4]. Algorithms for computing normal forms and other non-commutative cryptographic constructions still need to be secured against these and other side channel attacks.

Generic-case complexity is an important tool in non-commutative cryptography, but has seen little application towards other cryptographic systems. Future research should use generic-case analysis to identify the hard instances of computational problems. These results could either be used to speed up constant-time implementations as in Section V, or to bolster the security of a cryptographic scheme by restricting to hard instances.

References

[1] Adrian, D., Bhargavan, K., Durumeric, Z., Gaudry, P., Green, M., Halderman, J. A., ... Zimmermann, P. (2015). Imperfect forward secrecy: How Diffie-Hellman fails in practice. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (pp. 5-17). New York, NY: ACM.

[2] Agat, J. & Sands, D. (2001). On Confidentiality and algorithms. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy* (pp. 64-). Washington, DC: IEEE Computer Society.

[3] Almeida, J. B., Barbosa, M., Barthe, G., Dupressoir, F., & Emmi, M. (2016). Verifying constant-time implementations. *Proceedings of the 25th USENIX Security Symposium.*

[4] Berstein, D. J. (2005). Cache-timing attacks on AES. Preprint available from `https://cr.yp.to/antiforgery/cachetiming-20050414.pdf`

[5] Bernstein, D. J., Lange, T., & Schwabe, P. (2012). The security impact of a new cryptographic library. In A. Hevia and G. Neven (Eds.), *Progress in Cryptology LATINCRYPT 2012* (pp. 159-176). Berlin: Springer-Verlag.

[6] Birman, J. & Brendle, T. E. (2005). Braids: A survey. In W. Menasco, M. Thistlethwaite (Eds.), *Handbook of Knot Theory* (pp. 19-104). Amsterdam, NL: Elsevier.

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

[8] Bohnenblust, F. (1947). The algebraical braid group. *Annals of Mathematics*, *48*(1), 127-136.

[9] Bos, J. W., Costello, C., Naehrig, M., & Steblia, D. (2015) Post-quantum key exchange for the TLS protocol from the ring learning with errors problem. *2015 IEEE Symposium on Security and Privacy*, 553-570.

[10] Canvel, B., Hiltgen, A. P., Vaudenay, S., & Vuagnoux, M. (2003) Password interception in a SSL/TLS channel. In D. Boneh (Ed.), *Advances in Cryptology – CRYPTO 2003* (pp. 583-599). Berlin: Springer.

[11] Cha, J. C., Ko, K. H., Lee, S. J., Han, J. W., & Cheon, J. H. (2001). An efficient implementation of braid groups. In C. Boyd (Ed.) *Advances in Cryptology ASIACRYPT 2001* (pp. 144-156). Berlin: Springer.

[12] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to algorithms* (2nd ed.). Cambridge, MA: MIT Press.

[13] Costello, C., Longa, P., & Naehrig, M. (2016). Efficient algorithms for supersingular isogeny Diffie-Hellman. In M. Robshaw & J. Katz (Eds.), *Advances in Cryptology – CRYPTO 2016* (pp. 572-601). Berlin: Springer.

[14] Dehornoy, P., Dynnikov, I., Rolfsen, D. & Wiest, B. (2002). *Why are braids orderable?* Societe Mathematique De France.

[15] Dierks, T., & Rescorla, E. (2008, August). The transport layer security (TLS) protocol version 1.2 [RFC 5246].

[16] Diffie, W. & Hellman, M. (1976). New directions in cryptography. *IEEE Transactions on Information Theory*, *22*(6), 644-654.

[17] Doychev, G., Köpf, B., Mauborgne, L., & Reineke, J. (2015). CacheAudit: A tool for the static analysis of cache side channels. *ACM Transactions on Information and System Security*, *18*(1), 1-32.

[18] Epstein, D. B. A., Paterson, M. S., Cannon, J. W., Holt, D. F., Levy, S., V., & Thurston, W. P. (1992). *Word processing in groups*. Boston, MA: Jones and Bartlett Publishers.

[19] Garside, F. A. (1969). The braid group and other groups. *The Quarterly Journal of Mathematics*, *20*(1), 235-254.

[20] Kapovich, I., Myasnikov, A., Schupp, P., & Shpilrain, V. (2003). Generic-case complexity, decision problems in group theory, and random walks. *Journal of Algebra*, *264*(2), 665-694.

[21] Ko, K. H., Lee, S., Cheon, J. H., Han, J. W., Ju-Sung Kang, & Park, C. (2000). New public-key cryptosystem using braid groups. In M. Bellare (Ed.), *Proceedings of the 20th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO '00)* (pp. 166-183), London, UK: Springer-Verlag.

[22] Kocher, P. C. (1996) Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In N. Koblitz (Ed.), *Advances in Cryptology – CRYPTO 96* (pp. 104-113). Berlin: Springer.

[23] Krawczyk, H. (2001). The order of encryption and authentication for protecting communications (or: how secure is SSL?). In J. Kilian (Ed.), *Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology* (pp. 310-331). London, UK: Springer-Verlag.

[24] Merkle, R. C. (1978). Secure communications over insecure channels. *Communications of the ACM*, *21*(4), 294-299.

[25] Myasnikov, A., Shpilrain, V., & Ushakov, A. (2011). *Non-commutative cryptography and complexity of group-theoretic problems*. Providence, RI: American Mathematical Society.

[26] Myasnikov, A., & Ushakov, A. (2004). CRyptography And Groups (CRAG) [Computer software]. Hoboken, NJ: The Algebraic Cryptography Center at Stevens. Available from `http://web.stevens.edu/algebraic/downloads.php`

[27] Novikov, P. S. (1955). On the algorithmic unsolvability of the word problem in group theory. *Proceedings of the Steklov Institute of Mathematics*, *44*, 3-143.

[28] Vaudenay, S. (2002). Security flaws induced by CBC padding - Applications to SSL, IPSEC, WTLS.... In L. R. Knudsen (Ed.), *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques: Advances in Cryptology* (pp. 534-546). London, UK: Springer-Verlag.

[29] Wagner, N. R. & Magyarik, M. R. (1985). A public key cryptosystem based on the word problem. In G. R. Blakley & D. Chaum (Eds.), *Proceedings of CRYPTO 84 on Advances in cryptology* (pp. 19-36). New York, NY: Springer-Verlag.