A Partial Cryptanalysis of BIKE's Key Encapsulation Mechanism

Katarina Spasojevic
kspas036@uottawa.ca
Mentor: Monica Nevins
University of Ottawa

August 5, 2019

Abstract

One contender in the search for a secure post-quantum cryptosystem under consideration by the National Institute of Standards and Technology (NIST) is the Bit-Flipping Key Encapsulation (BIKE) system. This new cryptosystem is based on quasi-cyclic codes. This paper will present an analysis of the algebraic structure of BIKE's encryption and decryption schemes. The security of the BIKE will be proven by examining the tests for IND-CPA and IND-CCA security, which determine the advantage of a potential adversary. We will also present a guide for the installation of the BIKE software and the Number Theory Library (NTL) on Windows.

1 Introduction

A new concern in the field of cryptography is the power of quantum computing and its potential to compromise current cryptosystems. Quantum computers allow for algorithms that could quickly solve the hard problems forming the basis of encryption schemes used today. Consequently, research into post-quantum cryptosystems, which are based on problems resistant to the known attacks of quantum computers, is of interest to mathematicians today. For example, the RSA encryption scheme in use today would not be secure in a future with quantum computers. It is based on the difficulty of factoring very large numbers, however, quantum computers could perform factorizing algorithms that factor much faster than classical computers, allowing for algorithms that "break" RSA, as shown in [10]. One contender in the search for a secure post-quantum cryptosystem presented to the National Institute of Standards and Technology (NIST) is the Bit-Flipping Key Encapsulation (BIKE) system. This new cryptosystem is based on quasi-cyclic codes.

1.1 Asymmetric Ciphers

First, it is important to note that a symmetric cipher is based on a secret key two parties share, and only they know. Thus, when messages are exchanged, the key used can be completely random, resulting in a practically random ciphertext. This renders symmetric ciphers very secure and efficient. However, what if two parties that do not share a secret key wish to communicate securely? For example, a bank in Toronto may wish to communicate with a bank in London, and they can only use public channels that can be intercepted, such as the Internet. Here, we would have to employ a key encapsulation mechanism, where a short shared secret can be securely communicated to the other party, creating a private channel. We *encapsulate* the private key so we can employ the efficient channel of communication. This secure communication over a public channel, in an effort to establish a shared secret key, describes the notion of asymmetric ciphers, and mathematicians are very interested in this field!

To illustrate the idea of public key encryption, as shown in [5], imagine Bob and Alice wish to transmit a message. Bob, the receiver, runs an key generating algorithm G that outputs a pair of keys:

$$(pk, sk) \xleftarrow{\mathrm{R}} G().$$

The key pk is the *public key* and sk is the *private key*. Suppose Alice wishes to send Bob an encrypted message m. Suppose she has the public key pk (as it can be publicly shared). To send Bob the encrypted message, she computes a ciphertext,

$$c \xleftarrow{\mathrm{R}} E(pk, m).$$

She then sends c to Bob using a communication channel (like email). Bob receives the ciphertext and decrypts it using his secret key in the following function,

$$m \leftarrow D(sk, c).$$

Encryption and decryption are functions that satisfy,

$$D(sk, E(pk, m)) = m.$$

1.2 Computational Hardness

The idea of the difficulty, or hardness, of a decryption scheme is important. The decryption function must be difficult to solve without knowing the private key. Error correction in codes, as seen in BIKE, or the computation of a shortest vector in a lattice, as seen in NTRU, are hard problems in algebra that allow for decryption to be executable only by an honest party. Without such mathematical problems, which are shown to require a monstrous amount of time to solve by brute-force, public key cryptography would crumble. In fact, as mentioned previously, some cryptosystems employ decryption functions that become insecure with the computational power of quantum computers. We will now discuss BIKE, a cryptosystem based on quasi-cyclic codes.

2 BIKE Cryptosystem

My research this summer is concerned with mathematically describing a new post-quantum cryptosystem, BIKE. It is a contender for approval by the National Institute of Standards and Technology (NIST) in the "process of soliciting, evaluating, and standardizing one or more quantum-resistant public-key cryptographic algorithms" [6].

2.1 Preliminaries: The Basics of Coding Theory

2.1.1 Error correcting codes

To begin discussing codes, we will give a few examples of codes that are not quite error correcting, as seen in [12]. ASCII, for example, is not error correcting, as any character can be represented by a number, however if this number were to be modified while the code is transmitted, there would be no way to identify the error, as a different character would be received. Morse code is partially error detecting, as an error in transmission could produce a code that does not correspond to any letter. Alternatively, a slightly different error could produce a code that corresponds to a different letter. So it can produce an invalid ciphertext (which is the error detecting property), or change the message.

Continuing this idea, we can define a linear (n, k) code, of length n and rank k. A *linear code* over the finite field F is a subspace C of F^n. Each element of $c \in C$ is called a *codeword*. If $\dim(C) = k$, where $k \leq n$, then it is called an (n, k)-code over F. If $|F| = q$, then C is a q-ary linear code [12].

First, to create a linear code, allow for a generator matrix (in standard form) $G = [I_k|P]^\top$, where I_k is a $k \times k$ identity matrix and P is a $k \times (k - n)$ matrix. Then, a parity-check matrix $H = [-P^\top|I_{n-k}^\top]$ can be generated, where $P^\top$ is the transpose of P. In general, encoding allows for the generation of a ciphertext c from the message m, as shown in the following equation,

$$Gm = c,$$

and decoding verifies that

$$HGm = 0,$$

which indicates no error occurred, and $Gm = c$ can be solved by row reduction. For an *error-correcting linear code*, we assume a small error occurred, changing some entries of c; we represent this sum by,

$$c = Gm + e,$$

where e is a vector with very few nonzero entries. Decoding will return

$$Hc = HGm + He = He,$$

where e is a vector with very few non-zero entries and can be extracted from the product He using algorithms, such as the Bit-Flipping Algorithm, if the encryption scheme employs *low-density parity-check (LDCP) codes*. An LDPC code has a low percentage of 1's in the parity-check matrix, i.e., the parity-check matrix is sparse.

In general, a generator matrix G is any matrix whose **columns are a set of basis vectors of the linear subspace C**. If C is an (n, k)-code, then G will be a $n \times k$ matrix. As well, the code C is the column space of G. In addition, a dual code $C^{\perp}$ is the orthogonal complement of C. $C^{\perp}$ is a subspace and thus another linear code called the dual code of C. If C is an (n, k)-code then $C^{\perp}$ is an $(n, n-k)$ code. Thus, the generator matrix for $C^{\perp}$ is called a parity check matrix H for the linear code C. If C is an (n, k)-code then a parity check matrix for C will be an $n-k \times n$ matrix. The generator and parity-check matrices have the property that $HG = 0$. The descriptions of generator and parity-check matrices are adapted from [7].

2.1.2 Bit-Flipping Algorithm

In order for decryption to work, BIKE employs Quasi-Cyclic Moderate Density Parity Check (QC-MDPC) codes, which can be efficiently decoded through bit flipping decoding techniques. The Bit-Flipping Algorithm, used in BIKE's key encapsulation, is relatively simple, using syndrome decoding, with the description below adapted from [2]. Syndrome decoding pre-computes the syndrome corresponding to each error, and attempts to locate the smallest e satisfying $s = He$.

Let the syndrome $s = Hc$, where H is the private key and c is the ciphertext. Let $c = Fm + e$, where F is the public key, m is a random vector in $\mathbb{F}_2^k$ and e is the error, or in BIKE's case, the message. To figure out the relationship between the syndrome and the transmitted error e, allow

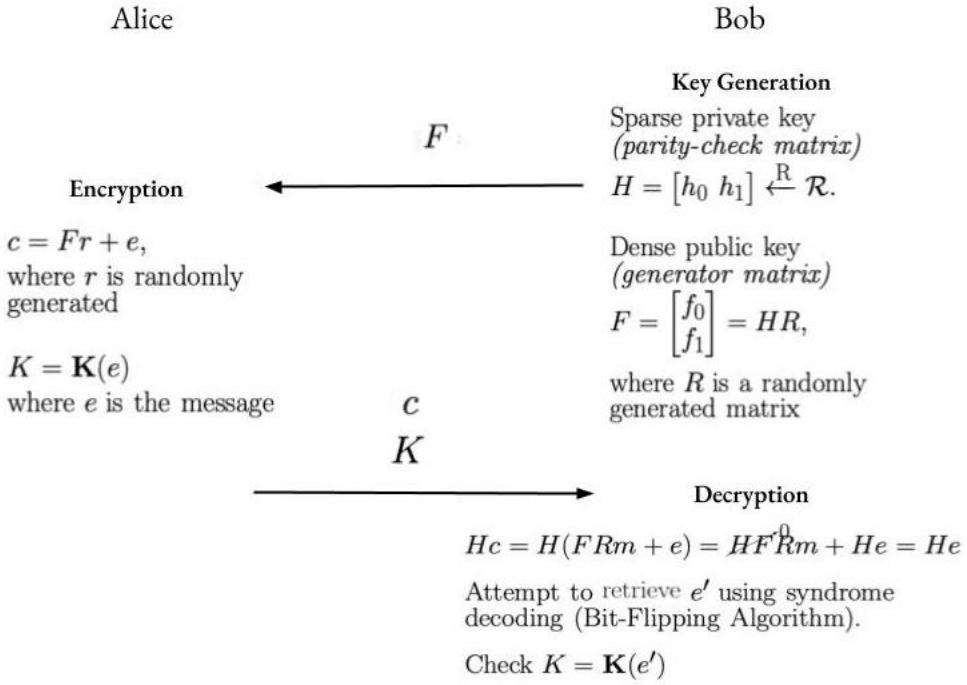
$$s = H \cdot (F + e)$$

$$s = HF + He$$

$$s = He$$

If $e = 100...0$, then the syndrome He is the result when the first data bit, is in error. In general, if element i of e is 1 and the other elements are 0, the resulting syndrome He corresponds to the case when bit i in the codeword is wrong. Under the assumption that there is at most one bit error, we store the syndromes when one of the first k elements of e is 1.

2.2 Algorithm Specification for BIKE-1 IND-CPA



Key encapsulation mechanism for BIKE-1

2.2.1 Key Generation

BIKE uses ephemeral keys, meaning a new pair of keys is generated at each key exchange.
Note that $\mathcal{R}$ is the cyclic polynomial ring $\mathbb{F}_2[X]/\langle X^r - 1\rangle$.

Selection of Parameters

Let r be a prime such that $(X^r - 1)/(X - 1) \in \mathbb{F}_2[X]$ is irreducible. Let w be the row weight of the parity-check matrix. The parity-check row weight w and the error weight t will be chosen so that $wt = O(n)$. This ensures, for a code-length n, the decryption failure rate is expected to decay exponentially.

Input: λ the target quantum security level.

1. Select parameters r, w as described above.
2. Generate uniformly at random $h_0, h_1 \in \mathcal{R}$ both of odd weight $|h_0| = |h_1| = w/2$.
3. Generate uniformly at random $g \in \mathcal{R}$ of odd weight (so $|g| \approx r/2$).
4. Compute $(f_0, f_1) = (gh_1, gh_0)$.

Output: the sparse private key (h_0, h_1) and the dense public key (f_0, f_1).

Notice that if we think of (f_0, f_1) as the column matrix F with these entries, and $H = (h_0, h_1)$ as the row matrix, then $HF = (gh_1h_0 + gh_0h_1) = 0$. Thus, F really is a generator matrix and H a parity check matrix, and F is hidden by multiplying by the random vector.

2.2.2 Encryption

Input: the dense public key (f_0, f_1).

1. Select $(e_0, e_1) \in \mathcal{R}^2$ such that $|e_0| + |e_1| = t$. The vector e is the message being transmitted.

2. Generate uniformly at random $m \in \mathcal{R}$.

3. Compute ciphertext $(c_0, c_1) = (mf_0 + e_0, mf_1 + e_1)$.

4. Compute $K \leftarrow \mathbf{K}(e_0, e_1)$.

Output: the encapsulated key K and ciphertext c.

Notice the generation of the ciphertext (c_0, c_1) with the products mf_0 and mf_1 is unclear. The representation of polynomials as cyclic matrices can be used to ensure the products mf_0 and mf_1 produce a vector in the ring $\mathcal{R}$.

2.2.3 Interpreting Encryption

Let $\mathcal{R} = F[X]/\langle X^r - 1\rangle$; this is the ring of polynomials such that $X^r = 1$. Identify each $f = a_0 + \ldots + a_{r-1}x^{r-1}$ in $\mathcal{R}$ with a vector $\vec{f} = (a_0, a_1, \ldots, a_{n-1})$ and alternatively with the corresponding circulant matrix F, of the following form,

$$F = \begin{pmatrix} a_0 & a_1 & a_2 & \ldots & a_{n-1} \\ a_{n-1} & a_0 & a_1 & a_2 & \ldots \\ a_{n-2} & a_{n-1} & a_0 & \ldots & \\ \ddots & \ddots & \ddots & \ddots & \ddots \\ a_1 & a_2 & \ldots & a_{n-1} & a_0 \end{pmatrix}.$$

Theorem 1. *For any two $f, m \in \mathcal{R}$, $\vec{fm} = F^\top \vec{m}$.*

Proof. Let $f, m \in \mathcal{R}$ be two polynomials of degree r. The product,

$$fm = \left(\sum_{i=0}^{r-1} a_i x^i\right)\left(\sum_{j=0}^{r-1} m_j x^j\right) = \sum_{i,j=0}^{r-1} a_i m_j x^{i+j}$$

. As $x^r = 1$, we can rewrite the summation as,

$$\sum_{i,j=0}^{r-1} a_i m_j x^{i+j} = a_0 m_0 + \ldots + a_0 m_{r-1}x^{r-1} + a_1 m_0 x + \ldots + a_1 m_{r-1}x + \ldots + a_{r-1}m_0 x^{r-1} + \ldots + a_{r-1}m_{r-1}$$

Now, consider the matrix product of a circular convolution and a vector. Let $F^\top$ denote the transpose of the circulant matrix of the polynomial f and $\vec{m}$ denote the vector identified by the polynomial m.

$$y = F^\top \vec{m} = \begin{pmatrix} a_0 & a_1 & a_2 & \ldots & a_{n-1} \\ a_{n-1} & a_0 & a_1 & a_2 & \ldots \\ a_{n-2} & a_{n-1} & a_0 & \ldots & \\ \ddots & \ddots & \ddots & \ddots & \ddots \\ a_1 & a_2 & \ldots & a_{n-1} & a_0 \end{pmatrix} \begin{pmatrix} m_0 \\ m_1 \\ \vdots \\ m_{n-1} \end{pmatrix}$$

The formula for the entries of $y = F^\top \vec{m}$ can be written with the following formula,
$y_0 = a_0 m_0 + a_1 m_1 + a_2 m_2 + \cdots y_1 = a_{n-1} m_0 + a_0 m_1 + a_1 m_2 + \cdots y_3 = a_{n-2} m_0 + a_{n-1} m_1 + a_0 m_2$. The entries of y can be represented by the following summation,

$$y = \sum_{i,j=0}^{r-1} a_{r-j+i} m_j x^i,$$

where if $i - (r-1) < 0$, we define $c_{i \pm r-1} = c_i$, indicating the subscripts are modulo r.

Thus, we see $\vec{fm} = F\vec{m}$ yield the same summation. □

2.2.4 Decryption

Input: the sparse private key (h_0, h_1) and the ciphertext c.

1. Compute the syndrome $s \leftarrow c_0 h_0 + c_1 h_1$.
2. Use Bit-Flipping Algorithm to decode syndrome s and recover error vector (e'_0, e'_1)
3. If $|(e'_0, e'_1)| \neq t$ or decoding fails, halt.
4. Compute $K' \leftarrow \mathbf{K}(e'_0, e'_1)$.
5. If $K' = K$, decryption was successful.

Output: the decapsulated key K or failure.

2.3 Algorithm Specification for BIKE-2 IND-CPA

2.3.1 Key Generation

The selection of parameters follows from BIKE-1 algorithm specification.

Input: λ the target quantum security level.

1. Select parameters r, w as described above.
2. Generate uniformly at random $h_0, h_1 \in \mathcal{R}$ both of odd weight $|h_0| = |h_1| = w/2$.
3. Compute $f = h_1 h_0^{-1}$.

Output: the sparse private key (h_0, h_1) and the dense public key f.

2.3.2 Encryption

Input: the dense public key f.

1. Select $(e_0, e_1) \in \mathcal{R}^2$ such that $|e_0| + |e_1| = t$. The vector e is the message being transmitted.
2. Compute ciphertext $c = e_0 + e_1 h$.
3. Compute $K \leftarrow \mathbf{K}(e_0, e_1)$.

Output: the encapsulated key K and ciphertext c.

2.3.3 Decryption

Input: the sparse private key (h_0, h_1) and the ciphertext c, and hash K.

1. Compute the syndrome $s \leftarrow c h_0$.
2. Use Bit-Flipping Algorithm to decode syndrome s and recover error vector (e'_0, e'_1)
3. If $|(e'_0, e'_1)| \neq t$ or decoding fails, halt.
4. Compute $K' \leftarrow \mathbf{K}(e'_0, e'_1)$.
5. If $K' = K$, decryption was successful.

Output: the decapsulated key K or failure.

2.4 Algorithm Specification for BIKE-3 IND-CPA

2.4.1 Key Generation

The selection of parameters follows from BIKE-1 algorithm specification.

Input: λ the target quantum security level.

1. Select parameters r, w as described above.
2. Generate uniformly at random $h_0, h_1 \in \mathcal{R}$ both of odd weight $|h_0| = |h_1| = w/2$.
3. Generate uniformly at random $g \in \mathcal{R}$ of odd weight (so $|g| \approx r/2$).
4. Compute $(f_0, f_1) = (h_1 + gh_0, g)$.

Output: the sparse private key (h_0, h_1) and the dense public key (f_0, f_1).
The public key requires $f_0 = h_1 + gh_0$ as...

2.4.2 Encryption

Input: the dense public key (f_0, f_1).

1. Select $(e, e_0, e_1) \in \mathcal{R}^3$ such that $|e| = t/2$ and $|e_0| + |e_1| = t$. The vector e is the message being transmitted.
2. Compute ciphertext $c = (c_0, c_1) = (e + e_1 f_0, e_0 + e_1 f_1)$.
3. Compute $K \leftarrow \mathbf{K}(e, e_0, e_1)$.

Output: the encapsulated key K and ciphertext c.

2.4.3 Decryption

Input: the sparse private key (h_0, h_1) and the ciphertext c.

1. Compute the syndrome $s \leftarrow c_0 + c_1 h_0$.
2. Use Bit-Flipping Algorithm to decode syndrome s and recover error vector (e'_0, e'_1)
3. If $|(e'_0, e'_1)| \neq t$ or decoding fails, halt.
4. Compute $K' \leftarrow \mathbf{K}(e'_0, e'_1)$.
5. If $K' = K$, decryption was successful.

Output: the decapsulated key K or failure.

2.5 Formal Security

2.5.1 Notions of Encryption Scheme Security

Cryptosystems can be proven secure if they are extremely difficult to break even when the adversary has access to the public key or ciphertexts. You can think of the names of the notions of security as indicating attainment certain goals and resistance to a possible attack model. The following description of notions of security are adapted from [3]. We will consider two different goals: *indistinguishably of encryption* and *non-malleability*. Indistinguishably (IND) formalizes an adversary's inability to learn any information about the plaintext message x underlying a ciphertext y. This is a very strong notion of security; no information about messages is revealed through ciphertexts. Non-malleability (NM) formalizes an adversary's inability, given a ciphertext y, to output a different ciphertext y' such that the plaintexts x, x' underlying the ciphertexts

are meaningfully related. For example, modifying a ciphertext y by one character, forming a new ciphertext y' does not result in an underlying message x' that differs by one character from the original message x.

The possible attacks from an adversary must also be considered. The three different attacks are: *chosen-plaintext attack* (CPA), *non-adaptive chosen-ciphertext attack* (CCA1) and *adaptive chosen-ciphertext attack* (CCA2). If a cryptosystem satisfies CPA security, the adversary can generate ciphertexts of any plaintexts, or equivalently, they have access to the public key, and is unable to discern any correlation between ciphertexts and plaintexts. One can imagine an attack where the adversary has access to the encryption function, and can create any number of ciphertexts from plaintexts. CPA security would imply the adversary cannot use ciphertexts they generated to decrypt a given ciphertext from a key exchange. CCA1 security implies an adversary is unable to compromise the cryptosystem with access to the public key as well as an oracle for the decryption function. With CCA1 security, the adversary may only use ciphertexts selected without knowing the challenge ciphertext y, thus they would most likely not resemble y. CCA2 differs from CCA1 only with the additional freedom of the adversary to use the decryption function on ciphertexts after knowing the challenge y, with the only restriction being the adversary cannot ask for the decryption of y itself.

In the original paper, the authors present a theorem proving CPA security. This proof introduced a series of games that follow the same steps as the test for IND-CPA security, but replacing s step of the game with a randomly generated variable. The first game is the test for IND-CPA security, the second game replaces the calculation of the public key with a randomly generated public key and the third game replaces the calculation of the ciphertext with a randomly generated ciphertext. A stronger assertion can be made to prove IND-CPA security, using a proof by contradiction.

2.5.2 Underlying Hard Problems for Decoding of Linear Codes

The problem of syndrome decoding was shown to be NP-complete in [4] for a (n, k) binary linear code, C, used on a binary symmetric channel. If y is the received message, then the syndrome is $s = yH$ where H is $n \times (n - k)$ parity check matrix. The receiver's best estimate of the transmitted codeword is $x = y + z_0$ where z_0 is minimum weight solution to the equation $s = zH$. The best general algorithm for finding a minimum weight solution to $s = zH$ has a running time which is an exponential function of the number of inputs. Thus, for the (2,1) and (3,1)- QCCF (replace) codes, syndrome decoding is hard on average.

2.5.3 Proof of IND-CPA Security

First, we will define the IND-CPA game that must hold for secure cryptosystems, based on the description in [1], followed by our expanded proof of security. The IND-CPA game has two honest parties exchanging a message using BIKE's key encapsulation mechanism. A third party, the referee, generates the hash of a random string. The referee randomly selects between the hash of the message and the hash of the random string. Then, an adversary is given access to the public key, ciphertext and the selected hash. A cryptosystem is secure if the adversary does not have an advantage in discerning whether the hash is of a random string or the honest message.

Let $\mathcal{K}$ denote the space of all possible outputs of a SHA-2 or SHA-3 hash function.

Definition 2.1. *A key-encapsulation mechanism is IND-CPA (passively) secure if, for any polynomial-time adversary $\mathcal{A}$, the advantage of $\mathcal{A}$ in the following game is negligible.*

Game IND-CPA

1. $(pk, sk) \leftarrow \text{GEN}(\lambda)$
2. $(c, K_0) \leftarrow \text{ENCRYPT}(pk)$
3. K_1 generated uniformly at random from $\mathcal{K}$
4. Assign output c of ENCRYPT() to c^*
5. $b \in \{0, 1\}$ generated uniformly at random
6. $K^* \leftarrow K_b$
7. $b' \leftarrow \mathcal{A}(pk, c^*, K^*)$
8. The cryptosystem is secure if the adversary's advantage $Adv^{IND-CPA}(\mathcal{A}) = Pr[b' = b] = 1/2$ is negligible

In [1], security is proved in a series of games, each of which progressively adds another element of randomness the adversary must be able to detect and differentiate from an honest input. The paper shows that in the variations of the IND-CPA game, the adversary has a lesser advantage than in the Random Oracle Model under the (2, 1)-Quasi-Cyclic Codeword Finding (QCCF) and (2, 1)-Quasi-Cyclic Syndrome Decoding (QCSD) assumptions or in the Random Oracle Model under the (3, 1)-QCSD and (2, 1)-QCSD assumptions, for BIKE-1,2 or BIKE-3, respectively. However, by using the Random Oracle Model, the authors are not allowing the consideration of any vulnerability of the hash function **K**; the message (e_0, e_1) could potentially be found as a preimage of the hash. The proof presented in [1] will be strengthened, proving BIKE's IND-CPA security by contradiction.

Theorem 2. *Let $\mathcal{A}$ represent an algorithm that breaks the IND-CPA game. Suppose there exists an algorithm $\mathcal{X}$ that could, in polynomial time, find a small preimage of a SHA-2 or SHA-3 hash function. Then,*

$$Adv^{IND-CPA}(\mathcal{A}) \geq Adv(\mathcal{X}).$$

Proof. Suppose there exists an algorithm $\mathcal{X}$ that could, in polynomial time, find a small preimage of a SHA-2 or SHA-3 hash function. We could use $\mathcal{X}$ to create $\mathcal{A}$, as in the IND-CPA game, the hash K is the only element that can accept either an honest input or a random input, dependent on the value of b. This implies the advantage of the algorithm $\mathcal{A}$ would be greater than or equal to the advantage of the algorithm $\mathcal{X}$.

However, a non-negligible advantage of $\mathcal{X}$ would contradict several previous results [11], [9] that prove SHA-2 and SHA-3 hash functions are "perfectly preimage resistant" [11], even when considering the computational complexity achieved by quantum computers. □

The converse of this theorem may not be true, as we do not assume the construction of $\mathcal{A}$. The converse states if the algorithm $\mathcal{A}$ were to succeed by finding a preimage of K then we'd have $AdV(\mathcal{A}) \leq Ad(\mathcal{X}).This would allow for a considerable advantage, as (e_0, e_1)$ transmit the message directly, and if the adversary were able to read (e_0, e_1), they would be able to discern if the strings contain a meaningful message or a random string of bits. In addition, the ciphertext c is dependent on (e_0, e_1), the public key (f_0, f_1) and in BIKE-1, m, generated randomly. If you were to select a hash function that is not sufficiently secure, then BIKE is not secure, and hence SHA-2 and SHA-3 are good choices.

In all, this theorem, combined with the proof of the ciphertext's indistinguishability in [1], would indicate BIKE is a strong contender in the search for a post-quantum standard.

2.6 A Potential Brute-Force Search Attack

In BIKE's encryption function, we noticed the transmitted message, conveyed in (e_0, e_1), has a defined size, as $|e_0| + |e_1| = t$, in all variations of BIKE. An adversary could perform a search of all possible values for

Bounds	BIKE-1 and BIKE-2	BIKE-3
Lower Bound	2^{581}	2^{1000}
Upper Bound	2^{1301}	2^{1402}

Figure 1: Search spaces for brute-force attack on (e_0, e_1)

(e_0, e_1), dependant on their size. The error bounds below are given from the suggested parameter listed for each of NIST's levels of security in [1].

Thus, a brute-force attack would not be effective.

3 Downloading BIKE and Number Theory Library on Windows

This reference will show you how to download and run the BIKE tests and algorithms published on `bikesuite.org`

1. Install Code:Blocks IDE from `http://www.codeblocks.org/downloads/26`. This link will take you to the binary release.

2. Select the file *codeblocks-17.12mingw-setup.exe* on either `FossHUB` or `Sourceforge.net`. This installs the GNU GCC compiler as well as the IDE on your computer.

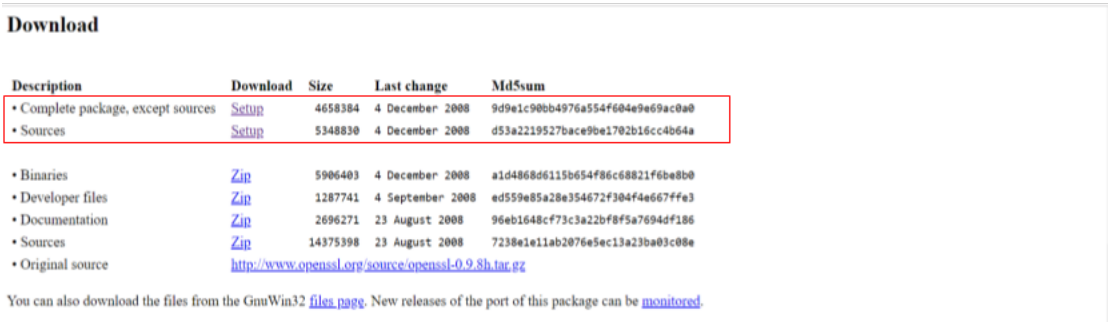
3. From `https://bikesuite.org/#implementation`, select the Reference Implementation of the BIKE software. When you press here, you will get a .zip file containing BIKE's source code.

   - Tip: Indicate on folder the unmodified source code

4. The BIKE software uses 2 libraries, OpenSSL and NTL. These libraries must be modified for our version of the software to be able to access them.

5. Download the OpenSSL library from `http://gnuwin32.sourceforge.net/packages/openssl.htm`. Select Setup link for both *Complete package, except sources* and *Sources*. The source files will be useful, as sub-algorithms BIKE uses can be accessed and read.

   - Tip: Save it in the same sub-directory as the BIKE project.

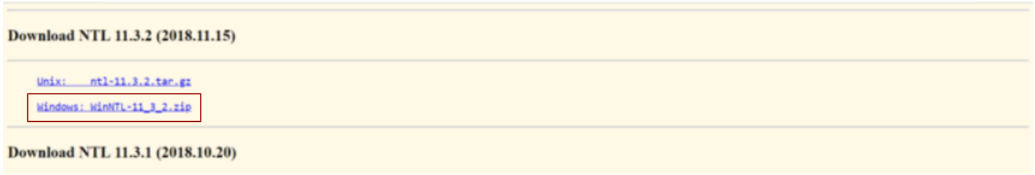


Download

Description	Download	Size	Last change	Md5sum
• Complete package, except sources	Setup	4658384	4 December 2008	9d9e1c90bb4976a554f604e9e69ac0a0
• Sources	Setup	5348830	4 December 2008	d53a2219527bace9be1702b16cc4b64a
• Binaries	Zip	5906403	4 December 2008	a1d4868d6115b654f86c68821f6be8b0
• Developer files	Zip	1287741	4 September 2008	ed559e85a28e354672f304f4e667ffe3
• Documentation	Zip	2696271	23 August 2008	96eb1648cf73c3a22bf8f5a7694df186
• Sources	Zip	14375398	23 August 2008	7238e1e11ab2076e5ec13a23ba03c08e
• Original source	http://www.openssl.org/source/openssl-0.9.8h.tar.gz			

You can also download the files from the GnuWin32 files page. New releases of the port of this package can be monitored.

6. Download the NTL library from `https://www.shoup.net/ntl/download.html`. NTL is described as a Library for doing Number Theory. It "is a high-performance, portable C++ library providing data structures and algorithms for manipulating signed, arbitrary length integers, and for vectors, matrices, and polynomials over the integers and over finite fields []".



Download NTL 11.3.2 (2018.11.15)

Unix: ntl-11.3.2.tar.gz

Windows: WinNTL-11_3_2.zip

Download NTL 11.3.1 (2018.10.20)

7. What you have downloaded cannot be used by the software quite yet. First, we must make a static library. Unlike with OpenSSL, it has not been pre-compiled for us.

8. Open your Code:Blocks window and select File → New → Project → Static library. Save this new project in the same sub-directory as all the files so far and select Open → *NTL Source Code*

9. Now you can make the project that will run BIKE. Open a new empty project in Code:Blocks. Add all of the files from the source code folder to the project.

10. In the folder FromNIST, in the file PQCgenKAT_kem.c, comment out the main() function. First we need to verify that the main() function in the test.c file runs properly.

11. After running the program, you should see the statement "Success! decapsulated key is the same as encapsulated key!" printed on the screen. To see more detail about each of the steps in the key generation, encryption and decryption schemes, change the VERBOSE macro in the defs.h file to 1, 2 or 3.

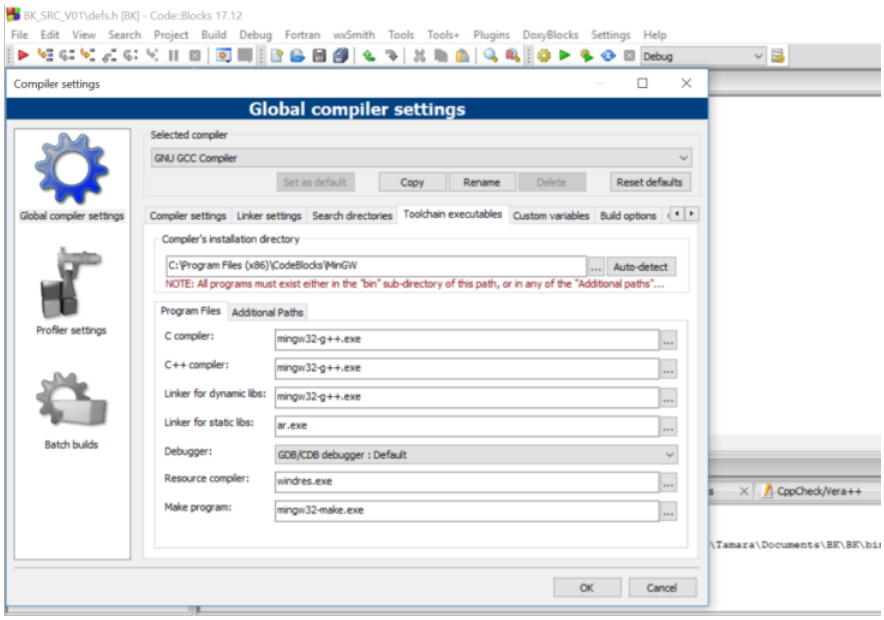
12. Once the KEM is functioning correctly, comment out the main() function in test.c and uncomment the main() function in PQCgenKAT_kem.c. When you run the program, two text files will be outputted, in the same directory as the project itself. These files will show the private key, public key, ciphertext and syndrome computed.

3.1 Environment Setup and Troubleshooting

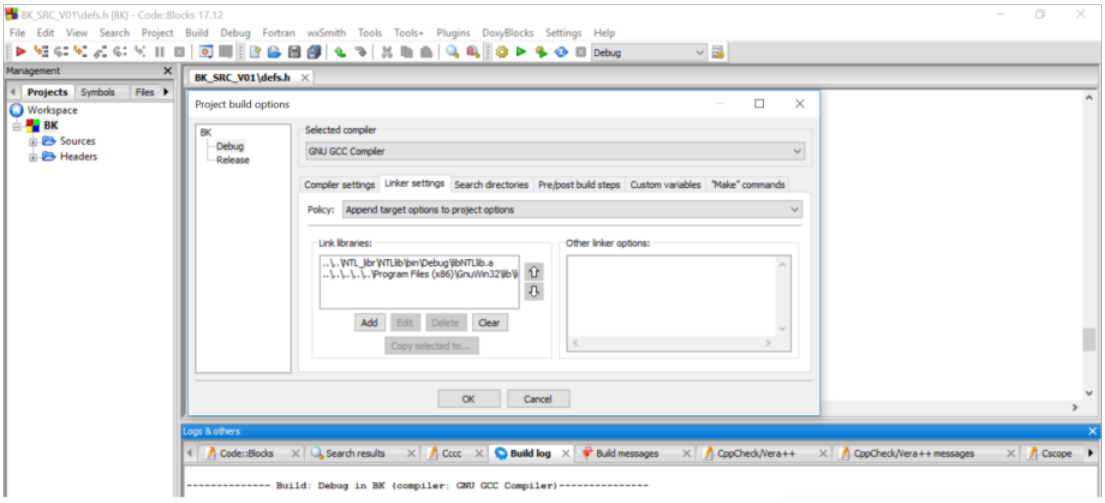
The engineers that designed BIKE used Unix, and did not use Code:Blocks, so some adjustments must be made. First I will discuss the environment, and what settings you should have for your compiler and debugger, for both the project and the whole environment.

The first problem you may encounter if you were to simply run the imported project is an error with the linker. The compiler may report that it is unable find functions in files with a .cpp file type. For example, kem.c may not be able to see functions in ntl.cpp file. This is due to the fact BIKE uses both C and C++ files. Note that if you search for the functions that are not compiling in their respective object files (double check), there may be additional characters added after the function name, as C++ performs name mangling while C does not. Therefore, a file compiled by a C++ compiler will contain function names a C compiler cannot read.

This is a problem in the global compiler settings. To resolve this, go to Settings in the toolbar, then Compiler Toolchain executables. Verify that both the C compiler and C++ compiler are using the file mingw32-g++.exe.



You will also need to set up the project build options. Enter the project build options by right-clicking on the project name and then Build options. Add the path to your compiled NTL library (.a file) and the path to the compiled dynamic link library from OpenSSL. This file, libeay32.dll.a is in the lib folder. BIKE only uses this dll.



References

[1] N. Aragon et al., *BIKE: Bit Flipping Key Encapsulation Round 2 Submission.*, NIST, 2019

[2] H. Balakrishnan, G. Verghese, 2012: Introduction to EECS II Digital Communication Systems Lecture Notes

[3] M. Bellare, A. Desai, D. Pointcheval, and P. Rogaway, Advances in Cryptology, 2001: *Relations Among Notions of Security for Public-Key Encryption Schemes*

[4] E. Berlekamp, R. J. McEliece, and H. van Tilborg. *On the inherent intractability of certain coding problems (corresp.)*, Information Theory, IEEE Transactions on, 24(3):384-386, May 1978. DOI: 10.1109/TIT.1978.1055873

[5] D. Boneh, and V. Shoup, *A Graduate Course in Applied Cryptography*, 2018

[6] CSRC.NIST.gov, "Post-Quantum Cryptography Standardization," January 3, 2017, `https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Post-Quantum-Cryptography-Standardization`

[7] I. Djordjevic, *Quantum Information Processing and Quantum Error Correction*, Academic Press, 2012

[8] J. Hoffstein, J. Pipher, and J. H. Silverman, *An Introduction to Mathematical Cryptography*, Springer, New York, 2010

[9] P. Kim, D. Han, and K. C. Jeong *Time–space complexity of quantum search algorithms in symmetric cryptanalysis: applying to AES and SHA-2*, Quantum Information Processing, **17** (2018), 339. DOI 10.1007/s11128-018-2107-3

[10] V. Mavroeidis, K. Vishi, M. Zych, and A. Jøsang, *The Impact of Quantum Computing on Present Cryptography*, International Journal of Advanced Computer Science and Applications, `https://arxiv.org/pdf/1804.00200.pdf`, 2018

[11] A. Neish, T. Walter, and P. Enge, *Quantum-resistant authentication algorithms for satellite-based augmentation systems*, NAVIGATION. **66** (2019), 199-209. DOI 10.1002/navi.287

[12] M. Nevins, 2010: Abstract Algebra Lecture Notes