MODELING AND ANALYSIS OF INFORMATION SYSTEMS, VOL. 30, NO. 4, 2023
journal homepage: www.mais-journal.ru

THEORY OF DATA

# Fast computation of cyclic convolutions and their applications in code-based asymmetric encryption schemes

A. N. Sushko[1], B. Y. Steinberg[1], K. V. Vedenev[1], A. A. Glukhikh[1], Y. V. Kosolapov[1]

DOI: 10.18255/1818-1015-2023-4-354-365

[1]Southern Federal University, 105/42 Bolshaya Sadovaya str., Rostov-on-Don, 344006, Russia.

MSC2020: 68P30; 68W99
Research article
Full text in English

Received November 6, 2023
After revision November 22, 2023
Accepted November 29, 2023

The development of fast algorithms for key generation, encryption and decryption not only increases the efficiency of related operations. Such fast algorithms, for example, for asymmetric cryptosystems on quasi-cyclic codes, make it possible to experimentally study the dependence of decoding failure rate on code parameters for small security levels and to extrapolate these results to large values of security levels. In this article, we explore efficient cyclic convolution algorithms, specifically designed, among other things, for use in encoding and decoding algorithms for quasi-cyclic LDPC and MDPC codes. Corresponding convolutions operate on binary vectors, which can be either sparse or dense. The proposed algorithms achieve high speed by compactly storing sparse vectors, using hardware-supported XOR instructions, and replacing modulo operations with specialized loop transformations. These fast algorithms have potential applications not only in cryptography, but also in other areas where convolutions are used.

**Keywords:** cyclic convolutions; fast algorithms; encryption schemes

INFORMATION ABOUT THE AUTHORS

| | |
|---|---|
| Andrey N. Sushko | orcid.org/0009-0009-7528-8532. E-mail: andrew-sush@mail.ru<br>Undergraduate Student. |
| Boris Y. Steinberg | orcid.org/0000-0001-8146-0479. E-mail: borsteinb@mail.ru<br>Head of the Chair. |
| Kirill V. Vedenev | orcid.org/0000-0002-7893-655X. E-mail: vedenevk@gmail.com<br>PhD Student. |
| Anton A. Glukhikh | orcid.org/0009-0005-0160-3609. E-mail: Antong.21072003@gmail.com<br>Undergraduate Student. |
| Yury V. Kosolapov<br>corresponding author | orcid.org/0000-0002-1491-524X. E-mail: itaim@mail.ru<br>Associate professor, Ph.D. |

**For citation**: A. N. Sushko, B. Y. Steinberg, K. V. Vedenev, A. A. Glukhikh, and Y. V. Kosolapov, "Fast computation of cyclic convolutions and their applications in code-based asymmetric encryption schemes", *Modeling and analysis of information systems*, vol. 30, no. 4, pp. 354-365, 2023.

МОДЕЛИРОВАНИЕ И АНАЛИЗ ИНФОРМАЦИОННЫХ СИСТЕМ, ТОМ 30, № 4, 2023
сайт журнала: www.mais-journal.ru

THEORY OF DATA

# Быстрое вычисление циклических сверток и их приложения в кодовых схемах асимметричного шифрования

А. Н. Сушко[1], Б. Я. Штейнберг[1], К. В. Веденев[1], А. А. Глухих[1], Ю. В. Косолапов [1]

DOI: 10.18255/1818-1015-2023-4-354-365

[1]Южный федеральный университет, ул. Большая Садовая, 105/42, Ростов-на-Дону, 344006, Россия.

УДК 004.421.4+004.051
Научная статья
Полный текст на английском языке

Получена 6 ноября 2023 г.
После доработки 22 ноября 2023 г.
Принята к публикации 29 ноября 2023 г.

Разработка быстрых алгоритмов генерации ключей, шифрования и дешифрования не только повышает эффективность соответствующих операций. Такие быстрые алгоритмы, например, для асимметричных криптосистем на квазициклических кодах, позволяют экспериментально исследовать зависимость вероятности ошибочного расшифрования от параметров кода для малых параметров безопасности и экстраполировать эти результаты на большие значения параметров безопасности. В этой статье мы исследуем эффективные алгоритмы циклической свертки, специально разработанные, в том числе, для использования в алгоритмах кодирования и декодирования квазициклических LDPC и MDPC кодов. Соответствующие свертки работают с двоичными векторами, которые могут быть как разреженными, так и плотными. Предлагаемые алгоритмы достигают высокой скорости за счет компактного хранения разреженных векторов, использования аппаратно поддерживаемых инструкций XOR и замены операций по модулю специализированными преобразованиями цикла. Эти быстрые алгоритмы имеют потенциальное применение не только в криптографии, но и в других областях, где используются свертки.

**Ключевые слова:** циклические свертки; быстрые алгоритмы; схемы шифрования

ИНФОРМАЦИЯ ОБ АВТОРАХ

| | |
|---|---|
| Андрей Николаевич Сушко | orcid.org/0009-0009-7528-8532. E-mail: andrew-sush@mail.ru<br>студент. |
| Борис Яковлевич Штейнберг | orcid.org/0000-0001-8146-0479. E-mail: borsteinb@mail.ru<br>заведующий кафедрой. |
| Кирилл Владимирович Веденев | orcid.org/0000-0002-7893-655X. E-mail: vedenevk@gmail.com<br>аспирант. |
| Антон Анатольевич Глухих | orcid.org/0009-0005-0160-3609. E-mail: Antong.21072003@gmail.com<br>студент. |
| Юрий Владимирович Косолапов<br>автор для корреспонденции | orcid.org/0000-0002-1491-524X. E-mail: itaim@mail.ru<br>доцент, канд. техн. наук. |

**Для цитирования**: A. N. Sushko, B. Y. Steinberg, K. V. Vedenev, A. A. Glukhikh, and Y. V. Kosolapov, “Fast computation of cyclic convolutions and their applications in code-based asymmetric encryption schemes”, *Modeling and analysis of information systems*, vol. 30, no. 4, pp. 354-365, 2023.

## Introduction

Convolution computation finds wide applications in digital signal processing [1], image processing [2], steganography [3], deep neural network training and inference [4], and other areas of applied mathematics. The Fourier transform can be utilized to compute convolutions: classical Fast Fourier Transform (FFT) algorithms such as Cooley-Tukey, Good-Thomas, Rader, and Winograd have a computational complexity of $O(n\log(n))$, where $n$ is a length of vectors. However, these algorithms are limited to specific lengths $n$, such as powers of 2 or products of coprime numbers. In addition, these algorithms use complex numbers arithmetic that may require more memory than the original vectors. Moreover, when using the Fourier transform for convolution computation, the inverse Fourier transform must also be used. As memory access is the performance bottleneck for modern computers [5], the use of Fourier transform could only be beneficial for large-scale instances. For small vector lengths, and especially if the vector coordinates are Boolean, optimized direct convolution computations can be faster. Note that Boolean convolutions over cyclic groups find widespread applications across various computer science domains. One prominent area is error-correction codes, where many practical codes exhibit cyclic or quasi-cyclic properties. By leveraging convolutions over Galois fields, message encoding can be efficiently implemented for codes with cyclic or quasi-cyclic properties. Additionally, for specific codes such as Quasi-Cyclic Low-Density Parity-Check (QC-LDPC) and Quasi-Cyclic Moderate Density Parity Check Codes (QC-MDPC), convolutions play a crucial role in implementation of decoding algorithms. Moreover, convolutions are extensively utilized in recently proposed cryptographic protocols for public-key encryption and digital signatures. One notable example is the Bit-flipping Key Encapsulation (BIKE) [6], a post-quantum public-key encryption algorithm that utilizes random QC-MDPC codes. The convolutions involved in BIKE operate on both sparse and dense vectors over both two-element Galois field $\mathrm{GF}(2)$ and ring of integers $\mathbb{Z}$.

In the proposed paper, we delve into the optimization techniques for software implementations of convolutions used in BIKE and in the decoding of QC-MDPC codes. Specifically, we consider dense-by-sparse convolutions over $\mathrm{GF}(2)$ and $\mathbb{Z}$. Our proposed optimizations allow achieving high-speed performance through the compact storage of sparse vectors, leveraging hardware-supported vector-executed XOR (exclusive OR) instructions, and replacing the modulo operation with specialized loop transformations. These techniques aim to enhance the performance of the convolutions involved in software implementations of BIKE and decoding of QC-MDPC codes.

## 1. Cryptographic research motivation

Modern algorithms for asymmetric encryption and electronic digital signature are based on assumptions about the computational complexity of solving the discrete logarithm problem in a finite group of large order and the problem of factoring a large composite number into a product of two large prime numbers. However, it turned out that these assumptions are confirmed so far only for the computation model on a Turing machine, while for the quantum computation model, P. Shor's efficient algorithm for solving these problems is known. In 2016, the US National Institute of Standards and Technology (NIST) announced a competition to develop asymmetric encryption and digital signature algorithms that would be resistant to attacks carried out using quantum computing. One of the finalists in the NIST competition for an asymmetric encryption scheme is the BIKE cipher [6], whose security is based on the difficulty of decoding a random binary QC-MDPC code of length $2r$, $r \in \mathbb{N}$. Note, that some computational operations during encryption and decryption in BIKE can be implemented using cyclic convolutions. Recall, that for ring $\mathcal{R}$ and vectors $a, b \in \mathcal{R}^r$ the cyclic convolution $a \star b$ of $a = (a_0, \ldots, a_{r-1})$ and $b = (b_0, \ldots, b_{r-1})$ is defined as follows

$$c = a \star b = (c_0, \ldots, c_{r-1}),\ c_i = \sum_{j=0}^{r-1} a_{(i-j) \bmod r} \cdot b_j,\ i = 0, \ldots, r-1. \tag{1}$$

The cryptosystem BIKE uses convolutions over rings $\mathbb{Z}$ (vectors of integers) and GF(2) (binary vectors), so the notation $\star_i$ and $\star_b$ are used for the corresponding convolutions. Note, that for GF(2), the sum operator in (1) is $\oplus$, and in the case $\mathbb{Z}$ sum is common operation with integers. Multiplication is common operation with integers in both cases. To show which operations in BIKE can be calculated using convolutions, we briefly describe this cryptosystem.

Let $\mathrm{GF}(2) = \{0, 1\}$ be a Galois field with additive operation $\oplus$: $0 \oplus 0 = 1 \oplus 1 = 0$, $1 \oplus 0 = 0 \oplus 1 = 1$. Denote by $R_n = \mathrm{GF}(2)[x]/(x^r - 1)$ the cyclic factoring ring and let us consider the mapping $L : \mathrm{GF}(2)^r \to R_n$ such that for $a = (a_0, \ldots, a_{r-1}) \in \mathrm{GF}(2)^r$, $L(a) = \sum_{i=0}^{r-1} a_i x^i$. (Here and below, vector coordinates are numbered starting from zero.) The number of non-zero elements in vector $a \in \mathrm{GF}(2)^r$ we denote as $\mathrm{wt}(a)$. The secret key in the BIKE system is a pair of polynomials $(h_0, h_1) \in R_n^2$, where each polynomial is chosen randomly and equiprobably such that $\mathrm{wt}(L^{-1}(h_0)) = \mathrm{wt}(L^{-1}(h_1)) = w/2 \approx \sqrt{2r}/2$ and $h_0$ must be invertible in $R_n$. The public key is the polynomial $h = h_1 \cdot h_0^{-1}$, where multiplication is taken in $R_n$. The plain text is represented as a pair $(e_0, e_1) \in \mathrm{GF}(2)^r \times \mathrm{GF}(2)^r$, where $\mathrm{wt}(e_0) + \mathrm{wt}(e_1) = t \approx \sqrt{2r}$. The corresponding ciphertext $s \in \mathrm{GF}(2)^r$ is obtained by the rule

$$s = e_0 \oplus L^{-1}(L(e_1) \cdot h), \tag{2}$$

where for the binary vectors $a = (a_0, \ldots, a_{r-1})$ and $b = (b_0, \ldots, b_{r-1})$ the result of operation $a \oplus b$ is the binary vector $(a_0 \oplus b_0, \ldots, a_{r-1} \oplus b_{r-1})$. When decrypting, the appropriate decoder for QC-MDPC codes is used, which takes the vector $s$, the secret key $(h_0, h_1)$ and returns the vector $(e_0, e_1)$ or $\perp$ if decoding failure occurs.

The product $L(e_1) \cdot h$ in encryption rule (2) is a multiplication of polynomials in the $R_n$, which can be realized as a cyclic convolution of a sparse vector $e_1$ and a dense vector $L^{-1}(h)$. So the rule (2) can be rewritten as $s = e_0 \oplus (e_1 \star_b L^{-1}(h))$. Some operations in known decoders of QC-MDPC codes can also be implemented using cyclic convolutions. Such operations may include calculating the current value of the syndrome and unsatisfied parity check (UPC) counters (see pseudo code of some known decoding algorithms for QC-MDPC-codes for example in [7]). For example, in Algorithm 1 the pseudo code of *BitFlip* decoding algorithm is shown, where cyclic convolutions are calculated at each iteration: two convolutions over GF(2) (calculating the current value of the syndrome $s'$) and two over $\mathbb{Z}$ (calculating the UPCs $upc_0$ and $upc_1$).

It is worth noting that fast convolution algorithms can speed up the key generation time of a system BIKE when using combinations of iterative and majority logic decoding algorithms, as considered in [8, 9]. Indeed, to ensure a low decoding failure rate (DFR) in such systems, it is necessary to select keys with a large majority margin. Therefore, the task of quickly calculating this boundary for randomly generated keys is relevant. It is known that such a margin for each key can be calculated using cyclic convolution over $\mathbb{Z}$.

Therefore, optimizing convolution algorithms can improve the speed of key generation, encryption and decryption operations in BIKE. Note that, the similar techniques can also be employed in other modern cryptographic primitives like HQC, NTRU, LWE, etc. As the length of vectors is not a power of 2 and is not a product of coprime numbers in many encryption schemes, the use of FFT algorithms for convolution calculations is impractical and new optimization algorithms are needed.

## 2. Fast cyclic convolution algorithms

This section presents algorithms for optimizing convolution computation. The algorithms are implemented in C language. It is assumed that the convolution is calculated for vectors of length $n$. The direct implementation *GF2_noonpt* of convolution over GF(2) is presented in Listing 1. The direct implementation *Z_noopt* of convolution over $\mathbb{Z}$ looks similar without taking modulo 2.

### 2.1. Generic optimizations for GF(2) and $\mathbb{Z}$

The direct implementation of convolution can be improved. Indeed, all convolution algorithms used in the decoder 1 involve at least one sparse vector. To store a sparse binary vector, we will use an integer

**Algorithm 1.** *BitFlip*: iterative decoder for QC-MDPC codes

**Input:** The vector $s \in \mathrm{GF}(2)^r$, the pair $(h_0, h_1) \in R_n \times R_n$ and the number of iterations $I$.

**Output:** $(e_0, e_1) \in \mathrm{GF}(2)^r \times \mathrm{GF}(2)^r$ or fail message $\perp$.

$u \leftarrow \mathbf{0}, v \leftarrow \mathbf{0}$; /* Initialize vectors $u$ and $v$ by zero value $\mathbf{0} \in \mathrm{GF}(2)^r$ */
$s' \leftarrow s$;
**for** $i = 1, \ldots, I$ **do**
  compute threshold $T$ by $s'$;
  $upc_0 \leftarrow s' \star_i L^{-1}(h_0)$, $upc_1 \leftarrow s' \star_i L^{-1}(h_1)$; /* Two cyclic convolutions over $\mathbb{Z}$ */
  **for** $j = 0, \ldots, r-1$ **do**
    **if** $upc_{0,j} \geqslant T$ **then**
      $u_j \leftarrow u_j \oplus 1$; /* Flipping $j$th bit in $u$ */
    **if** $upc_{1,j} \geqslant T$ **then**
      $v_j \leftarrow v_j \oplus 1$; /* Flipping $j$th bit in $v$ */
  $s' \leftarrow s \oplus (u \star_b L^{-1}(h_0)) \oplus (v \star_b L^{-1}(h_1))$; /* Two cyclic convolutions over GF(2) */
  **if** $s' = \mathbf{0}$ **then**
    **return** $(u, v)$
**return** $\perp$;

**Listing 1.** Direct implementation of convolution over GF(2) according to (1)

```
void GF2_noopt(unsigned short n,
               unsigned short *a,
               unsigned short *b,
               unsigned short *c) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            c[j] = (c[j] + b[(j - i + n) % n] * a[j]) % 2;
        }
    }
}
```

vector called *inda*, where each element will contain the position of a non-zero element (i.e., 1) in the original vector, in ascending order. Let's denote the length of the resulting vector as *arrSize*, which is smaller than $n$. For example, if the vector $a$ looked like this $a = (0, 1, 0.0, 0, 1, 0, 1)$, then the vector *inda* would be $(1, 5, 7)$. Therefore, to accelerate the computations, the sparse vector $a$ has been replaced with the *inda* vector of size *arrSize* (see listing for *GF2_opt1* in Listing 2 for convolutions over GF(2)). Using this new vector, we can reduce the number of iterations and eliminate the multiplication operation. The code *Z_opt1* for computing convolution over $\mathbb{Z}$, excluding the modulo 2, looks similar to *GF2_opt1*.

### 2.2. Optimizations for convolutions over GF(2)

In C language there are bitwise operators ~, &, |, and ^, which correspond to bitwise negation, bitwise AND, bitwise OR, and bitwise XOR operations, respectively. For example, the XOR operation (operator ^) can be used to replace addition of two numbers and taking the modulo 2 (see the listing *GF2_opt2* in Listing 3). In the listings *GF2_noopt*, *GF2_opt1* and *GF2_opt2* the bit vector $b$ is represented as an array of type *unsigned short*, where each element occupies 16 bits but stores only one bit of useful information. Let us

**Listing 2.** Optimization 1 over GF(2): using sparsity of one vector

```
void GF2_opt1(unsigned short n,
              int arrSize,
              const unsigned short *inda,
              unsigned short *b,
              unsigned short *c) {
    for (int i = 0; i < arrSize; i++) {
        for (int j = 0; j < n; j++) {
            c[j] = (c[j] + b[(j - inda[i] + n) % n]) % 2;
        }
    }
}
```

**Listing 3.** Optimization 2 over GF(2): using XOR in *GF2_opt1* instead of sum modulo 2

```
void GF2_opt2(unsigned short n,
              int arrSize,
              const unsigned short *inda,
              unsigned short *b,
              unsigned short *c) {
    for (int i = 0; i < arrSize; i++) {
        for (int j = 0; j < n; j++) {
            c[j] = (c[j] ^ b[(j - inda[i] + n) % n]);
        }
    }
}
```

transform vector $b$ in such a way that each bit in the vector corresponds to the corresponding element of the original vector. The resulting vector is denoted by $b2$ and has a length of $m = \lceil n/16 \rceil$. Thus, the first element of $b2$ contains the first 16 elements of vector $b$. Note that 16 is the length of *unsigned short* type in bits. However, since the data type can be changed to any other type of unsigned number, the code uses the variable $elementSize = 16$. Let's replace the dense vector $b$ with $b2$. Since the convolutions used involve adding a cyclically shifted vector, we need the ability to shift the vector by $n$ bits to the right. The output vector *res* is also compact, just like $b2$. Due to the changes in the input and output vectors, we need to be able to assemble new elements of the output vector. For this purpose, we will introduce two arrays of masks: *arrSize* and *negMasks*. Each element in the *arrSize* array represents a number whose binary representation contains *elementSize* ones, shifted to the left by the element's index. Therefore, $masks[0]$ is a number with *elementSize* ones in its binary representation, $masks[1]$ has $elementSize - 1$ ones, and so on. Similarly, *negMasks* is an array where each element can be obtained by bitwise negating the corresponding element in the *arrSize* array. Variables *mod*, *modNeg* and *it* are service variables for joining vector concatenation. In particular, *mod* represents the remainder of dividing $inda[i]$ by *elementSize*, which is necessary to select the appropriate mask and correctly shift the element since the shift may not always be a multiple of *elementSize*. This change allows for the computation of 16 elements in a single iteration of the outer loop, rather than just 1 element, resulting in significant acceleration. An optimized version of the convolution calculation is shown in Listing 4.

**Listing 4.** Optimization 3 over GF(2): use compact representation of vectors in *GF2_opt2*

```
void GF2_opt3(unsigned short n,
              int arrSize, int m,
              const unsigned short *inda,
              unsigned short *b2,
              unsigned short *res) {
    unsigned short modG=n % elementSize;
    unsigned short modGNeg=(elementSize - modG) % elementSize;
    unsigned short lastElementShifted =
            (b2[m - 1] >> modGNeg) +
            ((b2[m - 2] & negMasks[modGNeg]) << modG);
    for (int i = 0; i < arrSize; i++) {
        unsigned short mod = inda[i] % elementSize;
        unsigned short modNeg = (elementSize - mod) % elementSize;
        int start = inda[i] / elementSize;
        res[start] = res[start] ^ ((b2[0] >> mod) +
                ((lastElementShifted & negMasks[mod]) << modNeg));
        int j = 1;
        for (int it = start + 1; it < m; it++, j++) {
            res[it] = res[it] ^ (((b2[j]) >> mod)+
                                 ((b2[j-1] & negMasks[mod]) << modNeg));
        }
        int targetElement = (start - 1) * elementSize + n - inda[i] + 15;
        int modBack = targetElement % elementSize;
        int backStartTarget = targetElement / elementSize;
        int newMod = elementSize - modBack - 1;
        int newModNeg = (elementSize - newMod) % elementSize;
        j = backStartTarget;
        for (int it = start - 1; it >= 0; it--, j--) {
            res[it] = res[it] ^ (b2[j] >> newMod) ^
                    ((b2[j - 1] & negMasks[newMod]) << newModNeg);
        }
    }
    res[m - 1] = res[m - 1] & masks[modGNeg];
}
```

### 2.3. Optimizations for convolutions over $\mathbb{Z}$

To speed up the calculation of convolution over $\mathbb{Z}$, we use a partition of the iteration space into two triangular ones to get rid of the operation of taking the modulus (see Listing 5).

## 3. Experimental results

Testing for the correctness of the decoder 1 implementation was performed using known answer tests for two sets of parameters $(r, w, t)$ of BIKE system: $(r = 12323, w = 142, t = 134)$ and $(r = 24659, w = 206, t = 199)$. Note that these parameters sets correspond to the cases BIKE Level-1 and BIKE Level-3. Performance testing was performed with the same initial states of the pseudorandom number generators. Testing was carried out both as a separate evaluation of the convolution operation and as part of the decoder 1 for

**Listing 5.** Optimization 2 over $\mathbb{Z}$: rearrange loops in *Z_opt1*

```
void Z_opt2(unsigned short n,
            int arrSize,
            const unsigned short *inda,
            const unsigned short *b,
            unsigned short *c) {
    unsigned short *pointer_c, *end = inda + arrSize;
    int j, new_j;
    for (unsigned short *i = inda; i != end; i++) {
        pointer_c = c;
        j = 0;
        new_j = n - *i;
        for (; j < *i; j++) {
            *pointer_c = (*pointer_c + b[new_j]);
            new_j++;
            pointer_c++;
        }
        new_j = j - *i;
        for (; j < n; j++) {
            *pointer_c = (*pointer_c + b[new_j]);
            new_j++;
            pointer_c++;
        }
    }
}
```

**Table 1.** CPU computer specification

| | |
|---|---|
| Manufacturer | AMD |
| Number of cores | 6 |
| Number of threads | 12 |
| Frequency | 3.60 Ghz / 3600 Mhz |
| Turbo Core | 4.20 Ghz / 4200 Mhz |
| L1 cache | 384Kb (6 x 32Kb + 6 x 32Kb) |
| L2 cache | 3Mb (6 x 512Kb) |
| L3 cache | 32Mb |
| Core (architecture) | Matisse (Zen2, x86-64) |
| Process | 7 nm |
| PCIe controller | PCI Express 4.0 (16 lines) |

number of iteration $I = 100$; total number of tests is 1 000. In the experiments, the threshold $T$, used in the decoder 1, was not dynamically calculated, but was specified by an appropriate constant. All measurements were performed using the CPU with the parameters, presented in the Table 1.

The program code was compiled using *clang* compiler for C/C++. For each set of parameters, the running time of the algorithms *GF2_noopt*, *GF2_opt1*, *GF2_opt2*, *GF2_opt3*, *Z_noopt*, *Z_opt1* and *Z_opt2* was estimated without compiler optimization, as well as using optimization options *O3* and *Ofast*. Operating speed is given

**Table 2.** Average performance for $(r = 12323, w = 142, t = 134)$

| Operation | *O0* | | *O3* | | *Ofast* | |
|---|---|---|---|---|---|---|
| | seconds | cycles | seconds | cycles | seconds | cycles |
| *GF2_noopt* | 0.228056 | 228056 | 0.224785 | 224784 | 0.220196 | 220196 |
| *GF2_opt1* | 0.001029 | 1029 | 0.001094 | 1094 | 0.001035 | 1035 |
| *GF2_opt2* | 0.000742 | 742 | 0.000748 | 748 | 0.000792 | 792 |
| *GF2_opt3* | 0.000017 | 17 | 0.000017 | 17 | 0.000017 | 17 |
| *Z_noopt* | 0.237824 | 237824 | 0.239727 | 239727 | 0.246001 | 246001 |
| *Z_opt1* | 0.000810 | 810 | 0.000862 | 862 | 0.000815 | 815 |
| *Z_opt2* | 0.000043 | 43 | 0.000043 | 43 | 0.000044 | 44 |
| One iter. of alg. (1) (no opt.) | 0.933065 | 933065 | 0.900846 | 900845 | 0.880317 | 880317 |
| One iter. of alg. (1) (max. opt.) | 0.000156 | 155 | 0.000153 | 153 | 0.000150 | 150 |

**Table 3.** Average performance for $(r = 24659, w = 206, t = 199)$

| Operation | *O0* | | *O3* | | *Ofast* | |
|---|---|---|---|---|---|---|
| | seconds | cycles | seconds | cycles | seconds | cycles |
| *GF2_noopt* | 0.759139 | 759139 | 0.758156 | 758156 | 0.750196 | 750196 |
| *GF2_opt1* | 0.003079 | 3079 | 0.003022 | 3022 | 0.002988 | 2988 |
| *GF2_opt2* | 0.002202 | 2202 | 0.002172 | 2172 | 0.002171 | 2171 |
| *GF2_opt3* | 0.000047 | 47 | 0.000043 | 43 | 0.000041 | 41 |
| *Z_noopt* | 0.737514 | 737514 | 0.714715 | 714714 | 0.716012 | 716012 |
| *Z_opt1* | 0.002426 | 2426 | 0.002381 | 2381 | 0.002354 | 2354 |
| *Z_opt2* | 0.000170 | 170 | 0.000125 | 125 | 0.000122 | 122 |
| One iter. of alg. (1) (no opt.) | 2.891910 | 2891910 | 2.884335 | 2884334 | 2.907707 | 2907706 |
| One iter. of alg. (1) (max. opt.) | 0.000409 | 409 | 0.000398 | 397 | 0.000403 | 402 |

in seconds and processor cycles. Evaluation of the influence of the applied optimizations in the calculation of convolutions gave the results shown in Tables 2 and 3.

Experimental results show that non-optimized convolution calculations over rings GF(2) and $\mathbb{Z}$ have no significant difference in computation speed. This is due to the fact that both in the case GF(2) and in the case $\mathbb{Z}$, the elements are represented by the same data type (unsigned short). The transition to special representations leads, on the one hand, to a significant acceleration of calculations, and on the other hand, the calculation speed already depends on the ring $\mathcal{R}$. It is also clear that compiler optimization makes it possible to speed up the calculation of convolutions for both rings only in the case $r = 24659$. In the case of estimating the speed of one iteration of the decoder 1, a slight acceleration of the work is observed when using compiler optimization.

## 4. Fast convolutions in security evaluation of BIKE

The security of cryptographic algorithms is defined as the ability to withstand specific unauthorized actions by an attacker. For example, if a cipher is resistant to finding the plaintext by one ciphertext, then the cipher is OW-secure (the cipher has the One-Way property). If the attacker cannot distinguish which of the two plaintexts $m_0$ or $m_1$ corresponds to the given ciphertext $c$, then the cipher is IND-CPA secure (INDistinguishable under Chosen Plain text Attack). The strongest ciphers are those with IND-CCA security (INDistinguishable under Chosen Cipher text Attack), when the attacker cannot distinguish which of the two plaintexts $m_0$ or $m_1$ corresponds to the given ciphertext $c$, even if the attacker can send requests to the decryption oracle (but cannot send $c$). The cipher is said to have security level $\lambda$ (OW, IND-CPA, IND-CCA) if the probability of success of the corresponding attack does not exceed $1/2^{\lambda}$. For example, currently ciphers with a security level of $\lambda \geqslant 128$ are considered secure.

The disadvantage of BIKE system is that for a QC-MDPC code, the decoder may incorrectly decode the received vector $s$, which means that a legitimate user of the BIKE system cannot decrypt the message $(e_0, e_1)$. Thus, BIKE is characterized by a non-zero decoding failure rate (DFR). However, as shown in the work [10] for binary case and then in [9] for $q$-ary case, with a high DFR an reaction attack is possible that allows to find the secret key of the cryptosystem BIKE. Thus, to guarantee the IND-CCA security of the BIKE system with the security parameter $\lambda$, the DFR value must be less than $1/2^\lambda$. However, at present, for the fast decoder from [6], a theoretical estimate on DFR has not been obtained, and an experimental study of this probability for $\lambda = 128$ means at least $2^{128}$ experiments (one experiment includes generating a QC-MDPC code, encoding a random vector, and decoding), which is currently computationally impossible. Therefore, IND-CCA security cannot yet be guaranteed for BIKE. Note that in [6] the IND-CCA security of the BIKE is proved under the assumption that DFR is less than $1/2^\lambda$ at security level $\lambda$. There are a number of approaches to assessing DFR. In [11], a lower theoretical DFR estimate was obtained for an ML decoder (Maximum Likelihood decoder), which differs from the iterative decoder from [6]. However, this estimate may be redundant: the real DFR may be lower when using modern fast decoders. The second approach to DFR estimation is experimental estimation for small code parameters with subsequent extrapolation to a higher level of security. In this case, extrapolation may lead to the choice of falsely strong parameters of the cryptosystem: the real DFR may be higher than the estimate obtained using an extrapolation. This is due to the fact that there is an inflection point of the decoding error probability versus code length curve: there is such a code length, starting from which the rate of decrease in the error probability decreases. The third approach is an extension of the second and is related to the study of the inflection point. However, it is not computationally possible to estimate the inflection point for $\lambda = 128$, as noted above. In this regard, in [12, 13] the inflection point is studied only for $\lambda = 20$, and for large values of $\lambda$ it is proposed to perform extrapolation. The study, including experimental, of the inflection point for values of $\lambda$ exceeding 20, seems to be an urgent task, since this can allow more accurate extrapolation to the values of $\lambda$ corresponding to the values recommended in practice ($\lambda = 128, 192, 256$). Note that the efficient implementation of the BIKE cryptosystem is beneficial for obtaining estimates of the probability of decryption errors [11–13]. The parallel computations mentioned in [12, 13] alone may not provide significant acceleration. The performance of programs largely depends on data localization [14]. Moreover, one should not rely on good code optimization by an optimizing compiler [15]. One of the main engineering tasks in such a study is the efficient implementation of algorithms for encoding and decoding the QC-MDPC code, which makes it possible to carry out the study in a reasonable time. It seems that the greatest computational resources are consumed during encoding and decoding, while the generation of QC-MDPC code does not require large computational costs. As follows from the description above, the encoder and decoder of the QC-MDPC code can be implemented based on the calculation of convolutions. That is why the study of ways to speed up the calculation of convolutions is important in the study of the cryptosystem BIKE. The use of fast implementations of convolutions developed in this work will make it possible to estimate the time required to conduct $2^\lambda$ experiments on a single processor with parameters $(r, w, t)$ corresponding to the given security level $\lambda$. For example, with $\lambda = 30$, the experiment time on one processor will be about 24 days, taking into account that one decoder operation spends 0.000409 seconds (taken from Table 3) and the decoder uses 5 iterations when decoding. This is a rough estimate, since, on the one hand, for $\lambda = 30$ the decoder operating time will be less than 0.000409, but on the other hand, during decoding, threshold $T$ are usually calculated dynamically (see decoder 1), which can only increase the time of one decoding iteration. However, such approach allows us to estimate in advance the value of $\lambda$ for which experiments can be carried out under conditions of limited computing and time resources.

## Conclusion

It seems that further acceleration of convolutions can be achieved through, for example, the use of PCLMULQDQ instruction of the processor, which performs the multiplication of polynomials of degree no higher than 63. This approach is used, for example, in [16] to accelerate decoder for QC-MDPC codes. Note that the relevance of researching ways to optimize the QC-MDPC encoding/decoding algorithms is associated not only with the problems of accelerating key generation/encryption/decryption algorithms and refining security of BIKE system. It seems that the results of such study can make it possible to formulate requirements for microcircuits that implement these algorithms in hardware, and these results can be transferred to versions of the BIKE cryptosystem, where quasi-cyclic or quasi-group codes over finite fields of higher order are used. The presented algorithms for fast vector convolution, can also be used in other areas as well. For example, it seems that these algorithms can accelerate speed of data embedding in digital images in adaptive steganographic algorithms.

## References

[1] T. Holton, *Digital signal processing: Principles and applications*. Cambridge University Press, 2021, 1058 pp.

[2] D. S. Taubman, M. W. Marcellin, and M. Rabbani, "JPEG2000: Image compression fundamentals, standards and practice", *Journal of Electronic Imaging*, vol. 11, no. 2, pp. 286–287, 2002.

[3] V. Holub, J. Fridrich, and T. Denemark, "Universal distortion function for steganography in an arbitrary domain", *EURASIP Journal on Information Security*, vol. 2014, no. 1, p. 1, 2014.

[4] Intel. "Intel® oneAPI Deep Neural Network Library". (2023), [Online]. Available: https://software.intel.com/content/www/us/en/develop/articles/intel-mkl-dnn-part-1-library-overview-and-installation.html.

[5] N. R. Council *et al.*, *Getting up to speed: The future of supercomputing*. National Academies Press, 2005, 306 pp.

[6] N. Aragon *et al.*, *BIKE: Bit Flipping Key Encapsulation*, Submission to the NIST post quantum standardization process, Dec. 2017. [Online]. Available: https://hal.science/hal-01671903.

[7] T. B. Paiva and R. Terada, "Faster constant-time decoder for MDPC codes and applications to BIKE KEM", *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2022, no. 4, pp. 110–134, 2022.

[8] P. Santini, M. Battaglioni, M. Baldi, and F. Chiaraluce, "Analysis of the error correction capability of LDPC and MDPC codes under parallel bit-flipping decoding and application to cryptography", *IEEE Transactions on Communications*, vol. 68, no. 8, pp. 4648–4660, 2020.

[9] K. Vedenev and Y. Kosolapov, "Theoretical analysis of decoding failure rate of non-binary QC-MDPC codes", in *Code-Based Cryptography*, Springer, 2023, pp. 35–55.

[10] Q. Guo, T. Johansson, and P. S. Wagner, "A key recovery reaction attack on QC-MDPC", *IEEE Transactions on Information Theory*, vol. 65, no. 3, pp. 1845–1861, 2018.

[11] M. Baldi, A. Barenghi, F. Chiaraluce, G. Pelosi, and P. Santini, "Performance bounds for QC-MDPC codes decoders", in *Code-Based Cryptography Workshop*, Springer, 2021, pp. 95–122.

[12] S. Arpin, T. R. Billingsley, D. R. Hast, J. B. Lau, R. Perlner, and A. Robinson, “A study of error floor behavior in QC-MDPC codes”, in *International Conference on Post-Quantum Cryptography*, Springer, 2022, pp. 89–103.

[13] S. Arpin, T. R. Billingsley, D. R. Hast, J. B. Lau, R. Perlner, and A. Robinson. “Raw data and decoder for the paper ”a study of error floor behavior in QC-MDPC codes””. (2022), [Online]. Available: https://github.com/HastD/BIKE-error-floor.

[14] A. Vasilenko, V. Veselovskiy, E. Metelitsa, N. Zhivykh, B. Steinberg, and O. Steinberg, “Precompiler for the acelan-compos package solvers”, in *Parallel Computing Technologies: 16th International Conference, PaCT 2021, Kaliningrad, Russia, September 13–18, 2021, Proceedings 16*, Springer, 2021, pp. 103–116.

[15] Z. Gong *et al.*, “An empirical study of the effect of source-level loop transformations on compiler stability”, *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–29, 2018.

[16] N. Drucker and S. Gueron, “A toolbox for software optimization of QC-MDPC code-based cryptosystems”, *Journal of Cryptographic Engineering*, vol. 9, no. 4, pp. 341–357, 2019.