Thèse n° 7212

Analysis of the BIKE post-quantum cryptographic protocols and the Legendre pseudorandom function

Présentée le 29 octobre 2020

à la Faculté informatique et communications
Laboratoire de cryptologie algorithmique
Programme doctoral en informatique et communications

pour l'obtention du grade de Docteur ès Sciences

par

Dusan KOSTIC

Acceptée sur proposition du jury

Prof. O. N. A. Svensson, président du jury
Prof. A. Lenstra, directeur de thèse
Prof. S. Gueron, rapporteur
Dr J. Bos, rapporteur
Prof. S. Vaudenay, rapporteur

École
polytechnique
fédérale
de Lausanne

2020

Tini…

Acknowledgements

First and foremost I would like to thank Arjen Lenstra for being a wonderful advisor. He gave me freedom to explore a wide range of problems in our field and provided me with constant support and useful advice.

I am also grateful that I had the opportunity to work in LACAL with many bright researchers. Former PhD students and post-doctoral researchers Andrea Miele, Alina Dudeanu, Anja Becker, Jens Zumbrägel, Thorsten Kleinjung, Benjamin Wesolowski, Rob Granger, and Marguerite Delcourt warmly welcomed me to LACAL when I joined EPFL in 2015. I would also like to thank the present PhD students of LACAL, Novak Kaluđerović and Aymeric Genêt, with whom I spent the last three years in LACAL. I very much enjoyed the numerous lunch and coffee breaks we had over the years. Special thanks go to Thorsten and Novak for many insightful and productive discussions we had and for their constructive and useful feedback throughout the years. Also, many thanks go to Monique Amhof for making my life during the PhD easier by helping with all the administrative issues I had.

My sincere thanks go to Shay Gueron and Nir Drucker. I am very happy that since our collaboration started, during my visit to the University of Haifa in 2018, we have been continuously working together. The experience I gained while working with them was invaluable to me. Also, I am glad and excited to be joining Shay and Nir in AWS Cryptography group soon.

On a personal note, I am very grateful for all my friends, both the old friends from Serbia and the new ones I met in Lausanne. They made my PhD journey memorable.

Finally, I am deeply and sincerely grateful to my family. *Veliko hvala mojim roditeljima Vanji i Saši na neizmernoj podršci tokom mog celokupnog školovanja i mojim sestrama Lani i Danici za sve lepe trenutke. Na kraju, hvala Tini koja je sve vreme uz mene!*

Lausanne, August 25, 2020 D. K.

Abstract

The field of post-quantum cryptography studies cryptographic systems that are secure against an adversary in possession of a quantum computer. In 2017, the National Institute of Standards and Technology (NIST) initiated a process to standardize quantum-resistant public-key cryptographic algorithms (NIST PQC Project). In this thesis we analyze the performance and security of the Bit-Flipping Key Encapsulation Mechanism (BIKE) – one of the candidates in the NIST PQC project which advanced to the second round of the standardization process.

BIKE is a code-based cryptographic system featuring three different variants of the protocol. In the first round of the NIST PQC project BIKE offered security only against chosen-plaintext attacks (CPA). In the second round, BIKE introduced three new variants that are claimed to be secure also against chosen-ciphertext attacks (CCA). Firstly, we build a secure implementation of the CCA protocol and show that its performance characteristics are only negligibly worse than the CPA variant. In the key decapsulation phase of the protocol BIKE uses a decoding algorithm which fails with some probability, called the Decoding Failure Rate (DFR). We analyze the DFR of two decoders used in BIKE, Back-Flip and Black-Gray, and propose a new decoder, called Black-Gray-Flip, that achieves the same DFR as the two previously used decoders while being almost twice as fast. Finally, we propose an algorithm for inversion of binary polynomials in a polynomial ring used in BIKE-2, the second variant of BIKE. Our implementation of the inversion significantly outperforms previously used algorithms. With this and the fact that the bandwidth requirement for BIKE-2 is the smallest among the three variants, BIKE-2 is positioned as the preferable variant of BIKE.

The second part of this thesis studies the Legendre pseudorandom function (PRF) which is proposed to be used in the context of blockchains. We present a new algorithm for cryptanalysis of the Legendre PRF. The complexity of our algorithm is lower than the previous best known algorithm. Moreover, we show the results of breaking three Legendre PRF challenges posed by the Ethereum foundation. The most difficult challenge that we solved set the new record which is not broken so far.

Keywords: post-quantum cryptography, BIKE, QC-MDPC codes, QC-MDPC decoders, constant-time implementation, binary polynomial inversion, Legendre PRF, cryptanalysis

Résumé

Le domaine de la cryptographie post-quantique étudie les systèmes cryptographiques qui sont sécurisés contre un adversaire en possession d'un ordinateur quantique. En 2017, le National Institute of Standards and Technology (NIST) a démarré un processus pour standardiser les algorithmes à clé publique résistant au quantique (NIST PQC Project). Dans cette thèse, nous analysons la performance et la sécurité de Bit-Flipping Key Encapsulation Mechanism (BIKE) – un des candidats du NIST PQC Project qui a avancé au deuxième tour du processus de standardisation.

BIKE est un système cryptographique à base de codes qui présente trois variantes différentes du protocole. Au premier tour du NIST PQC Project, BIKE offrait une sécurité contre les attaques à texte clair connu (CPA). En premier lieu, nous construisons une implémentation sécurisée du protocole CCA et montrons que les caractéristiques de ses performances sont affectées seulement d'un ordre de grandeur négligeable par rapport à la variante CPA. Dans la phase de décapsulation de la clé du protocole, BIKE utilise un algorithme de décodage qui échoue avec une certaine probabilité, appelée la Decoding Failure Rate (DFR). Nous analysons la DFR de deux décodeurs dans BIKE, Back-Flip et Black-Gray, puis propose un nouveau décodeur, appelé Black-Gray-Flip, qui atteint la même DFR que les deux décodeurs utilisés précédemment en doublant presque sa rapidité. Finalement, nous proposons un algorithme pour l'inversion de polynômes binaires dans un anneau de polynômes utilisé dans BIKE-2, la deuxième variante de BIKE. Notre implémentation de l'inversion surpasse les algorithmes utilisés précédemment. Grâce à ceci et au fait que l'exigence sur la bande passante pour BIKE-2 est la plus petite parmi les trois variantes, BIKE-2 se place comme la variante préférable de BIKE.

La deuxième partie de cette thèse étudie la fonction pseudo-aléatoire (PRF) de Legendre qui est proposée dans le contexte de blockchains. Nous présentons un nouvel algorithme de cryptanalyse de la PRF de Legendre. La complexité de notre algorithme est inférieure à celle du précédent algorithme le plus connu. En outre, nous montrons les résultats de trois défis de PRF de Legendre lancés par l'Ethereum Foundation que nous avons battus. Le défi le plus difficile que nous avons résolu a établi un nouveau record qui, au jour de cette thèse, n'a pas encore été battu.

Mots-clés : cryptographie post-quantique, BIKE, codes QC-MDPC, décodeurs QC-MDPC, implémentation à temps constant, inversion de polynômes binaires, PRF de Legendre, cryptanalyse

Contents

Acknowledgements **i**

Abstract (English/Français/Deutsch) **iii**

List of Figures **xi**

List of Tables **xiii**

1 Introduction **1**

2 Background **9**

2.1 BIKE 9
2.1.1 Preliminaries 10
2.1.2 QC-MDPC codes 12
2.1.3 McEliece and Niederreiter cryptosystems 20
2.1.4 Key Encapsulation Mechanism 22
2.1.5 IND-CPA and IND-CCA security notions 23
2.1.6 BIKE KEMs 25
2.1.7 BIKE-1 26
2.1.8 BIKE-2 28
2.1.9 BIKE-3 31
2.2 Legendre PRF 34
2.2.1 Legendre symbol 34
2.2.2 Legendre pseudorandom function 36

I BIKE **39**

3 On constant-time QC-MDPC decoding with negligible failure rate **41**

3.1 Preliminaries 44
3.1.1 Bit Flipping Key Encapsulation (BIKE)-1 44
3.1.2 The IND-CCA transformation 44
3.1.3 QC-MDPC Decoders 45
3.2 Idealized schemes and concrete instantiations 47
3.3 Implementing BackFlip^{+} in constant-time 50

3.4 Estimating the Decoding Failure Rate (DFR) of a decoder with a fixed number of iterations . . . 53
3.5 Results . . . 55
3.5.1 Performance studies . . . 58
3.6 Weak keys: a gap for claiming IND-CCA security . . . 59
3.6.1 Constructing weak keys . . . 60
3.7 Discussion . . . 64
3.7.1 Methodologies . . . 65
3.7.2 Practical considerations for BIKE . . . 65

4 QC-MDPC decoders with several shades of gray 69
4.1 Introduction . . . 69
4.2 Preliminaries . . . 70
4.3 New decoders with different shades of gray . . . 71
4.4 DFR evaluations for different decoders . . . 74
4.5 Constant-time implementation of the decoders . . . 75
4.5.1 Optimizing the rotation of an array . . . 78
4.6 Performance studies . . . 81
4.6.1 Decoding and decapsulation: performance studies . . . 82
4.7 Discussion . . . 83

5 Fast polynomial inversion for post quantum QC-MDPC cryptography 87
5.1 Preliminaries and notation . . . 88
5.2 Optimized polynomial inversion in $\mathbb{F}_2[x]/(x^r-1)$ . . . 89
5.3 Our implementation . . . 92
5.3.1 Generating permutation map with SIMD instructions . . . 95
5.3.2 Optimizing the permutation with SIMD instructions . . . 97
5.3.3 Optimizing squaring and multiplication . . . 102
5.3.4 Side-channel protection considerations . . . 105
5.4 Results . . . 107
5.5 Discussion . . . 112

II Legendre PRF 115

6 Improved key recovery on the Legendre PRF 117
6.1 Background and notation . . . 119
6.2 Algorithm . . . 119
6.2.1 Sequence properties . . . 120
6.2.2 Precomputation stage . . . 122
6.2.3 Search stage . . . 122
6.3 Complexity of the algorithm . . . 124
6.3.1 Runtime hypothesis . . . 126
6.3.2 Optimal runtime . . . 126

6.3.3 Runtime with a fixed M 126
6.4 Implementation details 127
6.4.1 Precomputation stage 127
6.4.2 Search stage 128
6.5 Results 131

A Appendix **135**
A.1 Additional information on the experiments and the results of Chapter 3 135
A.1.1 Achieving the same DFR bounds as in [1] 137
A.2 Additional information on the experiments and the results of Chapter 4 138
A.2.1 Optimized and secured implementation of syndrome rotation 142
A.3 Additional information on the experiments and the results of Chapter 5 143
A.3.1 Generating permutation map with **AVX2** instructions 143
A.3.2 Squaring using **PCLMUL** and **VPCLMUL** 143
A.3.3 A 4 × 4 digits multiplication using **VPCLMUL** 144
A.3.4 Example of k-square versus series of k squares 145
A.3.5 Performance results 145

Curriculum Vitae **155**

List of Figures

2.1 Encoding and decoding 9

3.1 Approximating the BackFlip decoder threshold function. 52
3.2 Approximating the threshold function for fixed $\bar{e} = 25$. 52
3.3 BIKE-1 Level-1 extrapolations (see the text for details). 56
3.4 BIKE-1 Level-3 extrapolations (see the text for details). 57
3.5 Comparison of BIKE-CPA flows and BIKE-CCA flows, running with the Black-Gray decoder and $X_{BG} = 3, 4$ for several values of r: $r = 10163$ the original BIKE-1-CPA; $r = 11779$ the original BIKE-1-CCA; r values that correspond to DFR of $2^{-23}, 2^{-64}, 2^{-128}$, according to Table A.1. The vertical axis measures latency. 59
3.6 Histograms showing on the vertical axis the proportion of the experiments that end-up with the corresponding number of error bits on the horizontal axis, after $X_{BG} = 1, 2, 3, 4$ iterations. The decoder is the Black-Gray decoder. Panels a, c, e, g represents the results for $r = 9803$ and Panels b, d, f, h for $r = 10163$ with $f = 0, 20, 30, 40$. A lower error weight is better. Note that the blue line is present in all the histograms, but in some of them it is covered by the green line. 62
3.7 Black-Gray decoder, $r = 9803$, with $f = 0$ and $f = 30$ in Panel (a) and Panel (b), respectively. The horizontal-axis measures the number of decoding failures x out of 10000 experiments with random error vectors. The vertical-axis counts the number of keys that have a DFR of $x/10000$. 63

4.1 DFR comparison of BG with 3 iterations (9 steps) to BGF with: (Left panel) 7 iterations (9 steps); (Right panel) 5 iterations (7 steps). 75
4.2 "Small" rotation of two **AVX512** registers containing consecutive elements of the syndrome (refer to the text for details). 80

5.1 Conversion of a 32-bit value $a_3 a_2 a_1 a_0$ consisting of four bytes from *binary* to *byte* representation with **AVX2** instructions. 100
5.2 **VPCLMUL** instruction. 102
5.3 Squaring eight consecutive 64-bit digits of a binary polynomial with **VPCLMUL** instruction. 103
5.4 Multiplying four 64-bit digits of two binary polynomials using **AVX512** and **VPCLMUL** instructions. 106

A.1 BIKE-1 Level-1 BackFlip$^+$ different extrapolation methods. See the text for details. The sub-captions detail the (DFR; r) for DFR values: 2^{-64}, 2^{-128}. 138
A.2 Extrapolations of BIKE-1 Level-1 using the Black-Gray (BG) decoder. 140
A.3 Extrapolations of BIKE-1 Level-1 using the Black-Gray-Black (BGB) decoder. . . 140
A.4 Extrapolations of BIKE-1 Level-1 using the Black-Gray-Flip (BGF) decoder. . . . 141
A.5 Extrapolations of BIKE-1 Level-1 using the Black (B) decoder. 141

List of Tables

2.1 BIKE-1 IND-CPA/IND-CCA key generation flow. 27
2.2 BIKE-1 IND-CPA/IND-CCA key encapsulation flow. 27
2.3 BIKE-1 IND-CPA/IND-CCA key decapsulation flow. 29
2.4 BIKE-2 IND-CPA/IND-CCA key generation flow. 29
2.5 BIKE-2 IND-CPA/IND-CCA key encapsulation flow. 30
2.6 BIKE-2 IND-CPA/IND-CCA key decapsulation flow. 31
2.7 BIKE-3 IND-CPA/IND-CCA key generation flow. 32
2.8 BIKE-3 IND-CPA/IND-CCA key encapsulation flow. 32
2.9 BIKE-3 IND-CPA/IND-CCA key decapsulation flow. 33

3.1 BIKE-1 block size r (in bits) at security level 1, for which the Black-Gray decoder achieves a target DFR with a specified number of iterations, and the decapsulation performance (in cycles; the precise details of the platform are provided in Section 3.5). A DFR of 2^{-128} is required for the IND-CCA KEM. The IND-CPA used with ephemeral keys can settle for a higher DFR. 43
3.2 Validating the extrapolation results for the Black-Gray decoder with $X_{BG} = 3$ over two values of r. 58
3.3 A performance comparison of the Black-Gray and the BackFlip^{+} decoders for BIKE-1 at NIST security level 1. 59

4.1 The DFR achieved by different decoders. Two extrapolation methods are shown: “best linear fit” (as in [2]) and “two large r’s fit” (as in [2, Appendix C]). The second column shows the number of iterations for each decoder. The third column shows the total number of (performance-wise identical) executed steps. 74
4.2 The EC2 server performance of BIKE-1 when using the BG decoder with 3 iterations. The cycles (in columns 4, 5) are counted in millions. 82
4.3 BIKE-1 using the BG decoder with 3 iterations. Performance in cycles on Ice-Lake using various instruction sets: (a) **AVX512F**; (b) **AVX512F**, **AVX512-VBMI2**, **VPCLMUL**; (c) **AVX512F**, **AVX512-VBMI2**, **VPCLMUL**, **VAES**. 83
4.4 Rotation performance in cycles, comparison of the code snippet given in [3] and our implementations: (a) Listing A.1 with **AVX512F** and (b) Listing. A.1 modified to use **AVX512-VBMI2** as explained in Section 4.5. 83

5.1 Performance of our implementations of inversion in $\mathbb{F}_2[x]/(x^r-1)$ for a set of r values with different $wt(r-2)$. The NTL and OSSL columns denote the runtime of the inversion from the corresponding libraries ([4, 5]). The remaining columns represent our implementation: (a) with **AVX2**; (b) with **AVX512**; (c) with **AVX512** and **VPCLMUL**; columns labeled with "*" denote implementations with pre-computed permutation maps. The runtime is measured in millions of cycles. 110

5.2 Speedup of our implementations of inversion in $\mathbb{F}_2[x]/(x^r-1)$ compared to NTL with GF2X [4]. Columns 3-8 represent the speedup over NTL of the following implementation: (a) **AVX2**; (b) **AVX512**; (c) **AVX512** and **VPCLMUL**; columns labeled with "*" denote implementations with pre-computed permutation maps. The speedup is measured for a set of r values with different $wt(r-2)$. 111

5.3 BIKE-2 key generation performance, for four relevant r values, when our implementation of the inversion algorithm is used (without precomputed maps). Columns 2-4 represent the following implementations: (a) **AVX2**; (b) **AVX512**; (c) **AVX512** and **VPCLMUL**. The runtime is measured in thousands of cycles. 112

6.1 Number of clock cycles required to obtain a Legendre symbol by computation and extraction, amortized. 130

6.2 Comparison of the complexity of our algorithm and those of [6] and [7]. The complexity is given in terms of big-O number of operations on $\Theta(\log p)$-bit words. The time of Legendre symbol evaluation is denoted by t. 132

6.3 Legendre PRF challenges [8] with security levels estimated based on [6] and the new security estimates. 132

6.4 Results and estimates for solving the Legendre PRF challenges. The expected and actual number of core-hours for challenges #0 and #1 is based on measuring the performance of the implementation on our desktop PC with Intel Xeon E5-1650 at 3.5GHZ, while the numbers for the other three challenges are based on performance of Intel Xeon E5-2680 v3 at 2.5GHz CPU available in the EPFL IC cluster. 133

A.1 The linear and the quadratic extrapolation equations, and the computed r values for a given DFR. The cases labeled with N/A are those where the value of r to achieve a target DFR could not be found in the range $[0.7r', 1.3r']$, where r' is the recommended value for IND-CCA security in [9] 136

A.2 The *best linear* and the *two points* extrapolation equations, and the estimated r values for three target DFR. Level is abbreviated to Lvl, the number of iterations is abbreviated to iter, linear is abbreviated to lin., equation is abbreviated to eq. The Lin. start column indicates the index of the first value of r where the linear fit starts. The 5 column (number of steps) is the indication for the overall performance of the decoder (lower is better). 139

A.3 Squaring and k-squaring in $\mathcal{R}$ using our code (**AVX512** and **VPCLMUL**). Columns 2 and 3 count cycles. The threshold is computed by $k_{thr} = k$-square/square. The r values correspond to the IND-CCA variants of BIKE for Level-1/3. 145

A.4 Performance of our implementations of inversion in $\mathbb{F}_2[x]/(x^r-1)$ for a set of r values with different $wt(r-2)$. The NTL and OSSL columns denote the runtime of the inversion from the corresponding libraries ([4, 5]). The remaining columns represent our implementation: (a) with **AVX2**; (b) with **AVX512**; (c) with **AVX512** and **VPCLMUL**; (d) fully portable implementation, independent of any platform; (e) portable with **PCLMUL** instruction used for multiplication and squaring; columns labeled with “*” denote implementations with pre-computed permutation maps. The runtime is measured in millions of cycles. 146
A.5 Speedup of our implementations of inversion in $\mathbb{F}_2[x]/(x^r-1)$ compared to NTL with GF2X [4]. Columns 3-8 represent the speedup over NTL of the following implementation: (a) **AVX2**; (b) **AVX512**; (c) **AVX512** and **VPCLMUL**; (d) PORTABLE; (e) **PCLMUL**; columns labeled with “*” denote implementations with pre-computed permutation maps. The speedup is measured for a set of r values with different $wt(r-2)$. 147
A.6 BIKE-2 key generation performance when our implementation of the inversion algorithm is used. Columns represent the following implementations: (a) **AVX2**; (b) **AVX512**; (c) **AVX512** and **VPCLMUL**; (d) PORTABLE; (e) **PCLMUL**; columns labeled with “*” denote implementations with pre-computed permutation maps. The runtime is measured in thousands of cycles. 148

1 Introduction

In the recent years we are witnessing significant shifts in topics of interest in the field of cryptography. Until a few years ago the research focus of the crypto community were the long established public-key cryptographic systems, such as the well known Rivest-Shamir-Adleman (RSA) algorithm [10] and various algorithms based on the theory of elliptic curve cryptography (ECC) originally proposed by Koblitz and Miller [11, 12].

The security of the RSA algorithm relies on the hardness of the *integer factorization problem* – given a composite integer n that is a product of two different primes $n = uv$, find the factors u and v of n. Although a very simple problem at first glance, integer factorization has turned out to be a remarkably difficult problem to solve when n is large enough and its factors are properly chosen. The fact that there is still no polynomial time factorization algorithm reinforces the widely spread assumption that integer factorization is indeed a hard problem. The best known algorithm so far is the Number Field Sieve (NFS) [13] that has a subexponential complexity in the size of the number to be factored. The invention of the NFS algorithm was a major breakthrough and had significant impact on the security analysis of the RSA cryptosystem. Since then the cryptanalytic efforts invested in the factorization problem resulted in several theoretical and practical improvements, but none of them had a major effect on the security of RSA.

Shortly after the advent of the NFS method that weakened the security of RSA, another type of public-key cryptographic algorithms emerged. Namely, the algorithms based on elliptic curves promised smaller cryptographic key sizes than the RSA algorithm at the same level of security. Thanks to this potential reduction in storage and transmission requirements, the ECC based systems gained in popularity and started to slowly replace the RSA algorithm in practice. The main reason for smaller key sizes is that the security of the ECC based systems relies on the hardness of the *discrete logarithm problem* on elliptic curves (ECDLP). The discrete logarithm problem is defined for a finite group G (written in multiplicative notation) and can be stated as finding an integer a (if it exists) such that $g^a = h$ when given $g, h \in G$. In the case of ECC, the group G is the set of points on an elliptic curve together with a binary operation, called point addition, that sends two points to another point on the curve. The appeal of ECC stems from the fact that the best known way to solve ECDLP is to use a generic algorithm that has running time fully exponential in the size of the group of points on the elliptic curve G (provided that G is properly chosen).

After decades of research on integer factorization and discrete logarithm, the two foundational problems of public-key cryptography, a solution that is sufficiently efficient to threaten the security of cryptographic systems based on these two problems is still not found. Therefore, currently used systems are considered secure against the so-called *classical algorithms*. We call these algorithms *classical* to distinguish them from *quantum algorithms* that are designed to run on a quantum computer. Analyzing the security of classical cryptography in the context of quantum computers paints a very different picture – namely, Shor's algorithm [14] is a quantum algorithm that solves the integer factorization and the discrete logarithm problem in polynomial time. From this perspective the situation with currently used classical cryptography seems pretty grim, provided that a sufficiently powerful quantum computer exists. The prospects of a practical quantum computer being developed in the near future, or being developed at all, are unclear. Arguments and opinions from all around the spectrum can be found in the literature: ranging from very optimistic on one side, i. e., quantum computers will be available in the next few years, to those even doubting the feasibility of a large scale quantum computer on the other side. Nevertheless, as a matter of precaution, from all this uncertainty emerged the field of post-quantum cryptography and quickly grabbed the attention of many researchers.

In 2017, the National Institute for Standards and Technology (NIST) apparently considered the pace of advancement in quantum technology fast enough to initiate the process to standardize quantum-resistant public-key cryptographic algorithms – the NIST Post-Quantum Cryptography (PQC) project [15]. The call for proposals splits the submissions in two categories: key encapsulation mechanisms (KEM) and digital signatures. A KEM is a mechanism to exchange a shared secret key between two parties and it consists of three algorithms: key generation, encapsulation and decapsulation. One party uses public-key cryptographic methods to generate a public and private key pair and publishes the public key. The public key is used by the second party which generates the desired shared key, encapsulates it and transmits it to the first party, which in turn uses its private key to decapsulate the received message and obtain the shared secret key.

Initially, 69 proposals for KEM were accepted by NIST. After the first round of evaluation 17 submissions were deemed worthy by NIST to advance to the next round. In June 2020, NIST stopped receiving comments and modifications for the Round-2 submissions and it is expected to announce the start of the third round, with a further reduced number of candidates, by the end of 2020. The KEM submissions that survived the first round and are being evaluated in Round-2 can be categorized in three different types of cryptographic schemes: lattice-based, code-based, and elliptic curve-based schemes with 9, 7, and 1 submissions, respectively. The lattice-based schemes are further divided in those that use structured lattices and the others that rely on unstructured lattices. Most of the KEMs involving structured lattices have a clear edge in terms of performance and key size compared to the unstructured lattices, and to all the other candidates for that matter. On the other hand, the fact that the lattices contain some structural properties introduces uncertainties about the provable security of those schemes. Nevertheless, the KEMs based on structured lattices are clear favorites for advancing to the next round of the process. The second KEM category, code-based schemes, can be split in two parts, analogously to the lattice schemes. The schemes that use linear codes without any structure have a large disadvantage when it comes to both performance and key size, but at the same time their security is (in part) based on the hardness of decoding a general linear code which is an NP-hard problem [16]. The other code-based schemes introduce an algebraic structure in the used linear code which allows them to represent the code much more compactly and also to perform code operations more efficiently. Similarly to the lattice-based schemes, the structure of the code potentially opens the schemes to new attacks that may exploit the code structure, and therefore these schemes require a more complex and careful security analysis. The final KEM category features only one submission which is based on isogenies between supersingular elliptic curves. This submission offers by far the smallest key size but on the other hand, its performance is not very competitive with (most of) the other schemes. The report published by NIST [17] summarizes all the submitted cryptographic schemes and gives further details on their advantages and disadvantages.

Bit Flipping Key Encapsulation (BIKE) [9] is one of the code-based cryptographic schemes in Round-2 of the NIST PQC project. BIKE works on similar principles as all the other code-based schemes. Namely, an error-correcting code is used to exchange a secret message between two parties. A code is represented in two ways – by a generator matrix and by a parity-check matrix. The generator matrix is used for encoding, i. e., it is used to generate a codeword from a given plain message. The parity-check matrix is used in the other direction, to decode the codeword to the message. Moreover, the properties of the parity-check matrix allow correction of (some number of) errors that might occur in the codeword during transmission. In the context of BIKE's key encapsulation mechanism the code is used in the following way. In the key generation algorithm, the first party chooses a specific error-correcting code, publishes the generator matrix, and keeps the parity-check matrix of the code private. The other party uses the generator to encapsulate the shared secret key by encoding it, obscures the resulting codeword by inserting errors into it, and finally sends the noisy codeword to the first party. Since the first party has the parity-check matrix it is able to correct the errors in the noisy codeword and decode it to recover the original message, i. e., the shared secret key. The security of such protocol relies on the assumption that for the chosen code it is hard to decode a noisy codeword without the knowledge of the parity-check matrix.

The specific error-correcting codes employed in BIKE are binary quasi-cyclic moderate density parity-check (QC-MDPC) codes. The QC part of the code's name comes from the quasi-cyclic property of its generator and parity-check matrices. A cyclic code has its two matrices represented by circular matrices which are matrices such that each row is a cyclic shift (rotation) of its adjacent rows. Therefore, a circular matrix is fully determined by a single row. Moreover, the rows can be equivalently viewed as polynomials in some polynomial ring defined by the parameters of the code. The generator and the parity matrix of a quasi-cyclic code are each composed of several (two in case of BIKE) circular matrices stacked horizontally. The moderate density part of the name reflects the fact that the number of non-zero coefficients in the parity-check matrix rows is moderate, i. e., it is about the square root of the row size.

The BIKE suite entered the NIST PQC project with three variants of the protocol, called BIKE-1, BIKE-2 and BIKE-3. Each variant of the protocol is well suited for a different use case. BIKE-1 for example features a fast key generation procedure but at the cost of fairly large public key and ciphertext size. BIKE-2 on the other hand involves a more costly key generation, while halving the size of the key and ciphertext. BIKE-3 is designed as a middle ground with the performance of the key generation competitive with BIKE-1, the size of the public key almost as small as BIKE-2, and the ciphertext size the same as in BIKE-1. The third variant of BIKE seems to be the most desirable option, however, because of potential patent issues attached to its design it is considered as a less serious candidate. For this reason, only BIKE-1 and BIKE-2 are analyzed in this thesis. In Round-1 of the standardization effort all three variants of BIKE were claimed to be secure against chosen plaintext attacks (CPA) which is considered as a minimum level of security that a cryptographic scheme has to satisfy. The CPA security of a scheme is considered to be sufficient for the *ephemeral key* use case where a new public and private key pair is generated for every execution of the KEM protocol. In Round-2, BIKE introduced the chosen ciphertext attack (CCA) secure versions of the three variants. The CCA security is necessary if the scheme is to be used in the *static key* setting, i. e., a single key pair is used for more than one KEM session.

The latest update of BIKE's specification [18] is prepared for Round-3 of the NIST PQC project in case BIKE advances to the next round. The changes between the Round-2 specification [9] and Round-3 specification [18] are significant and in some part influenced by the work presented in this thesis, which was a joint effort with Shay Gueron and Nir Drucker from the University of Haifa, Israel, and Amazon Web Services, USA. The three chapters of the thesis related to BIKE are based on the following papers:

- Chapter 3 is based on:
  [2] N. Drucker, S. Gueron, and D. Kostic, "On constant-time QC-MDPC decoding with negligible failure rate".
  This paper is accepted at the CBCrypto 2020 International Workshop on Code-Based Cryptography and will be published in the proceedings. The preprint of the paper is available at https://eprint.iacr.org/2019/1289.
- Chapter 4 is based on:
  [19] N. Drucker, S. Gueron, and D. Kostic, "QC-MDPC Decoders with Several Shades of Gray" in *Post-Quantum Cryptography* (J. Ding and J.-P. Tillich, eds.), (Cham), pp. 35-50, Springer International Publishing 2020.
- Chapter 5 is based on:
  [20] N. Drucker, S. Gueron, and D. Kostic, "Fast polynomial inversion for post quan-

tum QC-MDPC cryptography" in *Cyber Security Cryptography and Machine Learning* (S. Dolev, V. Kolesnikov, S. Lodha, and G. Weiss, eds.), (Cham), pp. 110-127, Springer International Publishing 2020.

In Chapter 3 we present the secure implementation of the CCA protocol flows added in BIKE Round-2 and show that the performance overhead, introduced by additional operations required to satisfy the CCA security, does not have a significant impact on the running time of the protocol. Consequently, only the CCA version of BIKE is proposed in the Round-3 specification due to higher security assurances at a negligible performance cost. In addition to the new CCA flows, BIKE has also defined a new decoder, called BackFlip, to be used in the CCA versions of the protocol. This change was due to the increased security requirements. Namely, decoders used for MDPC codes, based on the Bit-Flipping decoder proposed in [21], are on one hand efficient but on the other hand there is some probability that they fail to decode the given noisy codeword. This probability is called the Decoding Failure Rate (DFR). The DFR of a decoder is an especially important property in the static key use case because with every decoder failure the attacker may learn some information about the used private key (as illustrated by the attacks in [22, 23, 24, 25]). Therefore, BackFlip decoder, which achieves sufficiently low DFR, was introduced to be used in the CCA versions of BIKE. In Chapter 3 we show how to build a secure constant-time implementation of BackFlip and analyze its DFR in terms of BIKE's parameters. Moreover, we compare BackFlip with the previously used Black-Gray decoder and discuss some subtleties of the trade-off between performance and DFR of both decoders. Finally, we identify a gap in the proof that CCA instantiations of BIKE are indeed CCA secure. Namely, the proof assumes the equivalence of the DFR of a decoder and the δ-correctness of the scheme and we show that those are not equivalent. We address this gap in [26] and solve the issue by slightly modifying the protocol; subsequently, the solution was applied to the definition of BIKE in Round-3 specification. However, in Chapter 3 we point out that even closing the gap in the proof is not sufficient to claim that BIKE is CCA secure – the remaining condition is that the used decoder has an appropriate DFR. The DFR of the BIKE decoders is determined heuristically by estimating the DFR for small parameters, for which we can run experiments, and then extrapolating from the obtained data points. Therefore, even if the estimated DFR is as low as required, the method by which it is computed does not constitute a proof. For this reason, the Round-3 BIKE specification does not claim CCA security in general but only on the condition that a decoder with appropriately low DFR is used, i. e., if a decoder with a provably low DFR (of the required magnitude) is given then BIKE is CCA secure.

In Chapter 4 we propose three new decoders inspired by the Black-Gray decoder and our observations of its error correcting properties. The goal of the work presented therein was to find a more efficient decoder that can achieve the same DFR levels as the previously considered decoders in BIKE. We identify one of the proposed decoders, called Black-Gray-Flip (BGF), as the most efficient option currently. The BGF decoder is now used in BIKE, as defined in the Round-3 specification [18]. Moreover, we study the performance of *secure* implementation of the four decoders. Ever since cryptography entered the commercial waters and started to be commonly used in various technological applications it was shown time and time again that the main weaknesses of cryptographic algorithms usually do not arise from solving the underlying hard mathematical problems but rather from insecure implementations. The understanding of what constitutes a secure implementation has changed over time with the evolution and ingenuity of practical attacks that can obtain secret keys without solving the hard problems. There are two main criteria that have to be satisfied when implementing a cryptographic algorithm in a secure manner. Firstly, the parts of the implementation that handle secret data have to be *constant-time*, i. e., the running time (the number and order of operations) must not depend on the value of the secrets. The second criterion, more easily and commonly overlooked by implementors, is that the memory access patterns must not depend on the secret data, e. g., accessing a memory location at an address specified by a secret value is considered insecure and vulnerable to practical attacks. Therefore, in Chapter 4 we also show how to securely implement the decoding algorithms without sacrificing too much of the performance.

In Chapter 5 we propose an algorithm for computing an inverse of a polynomial in the polynomial ring defined by BIKE (and used in other code-based schemes submitted to the NIST PQC project). BIKE operates on polynomials in $\mathcal{R} = \mathbb{F}_2[x]/(x^r - 1)$ where parameter r is a prime such that $x^r - 1$ is a product of $x - 1$ and an irreducible polynomial of degree $r - 1$. Our inversion algorithm is an adaptation of the ITI algorithm [27]. The properties of $\mathcal{R}$, such as the irreducibility of $(x^r - 1)/(x - 1)$ and the fact that polynomial coefficients are in $\mathbb{F}_2$, allow us to build a particularly efficient implementation of the inversion algorithm. Furthermore, we explain how to leverage vectorized instruction sets available in modern processors to optimize the implementation and gain considerable performance improvements. Our implementation significantly outperforms the inversion methods from two popular open source libraries, NTL [4] and OpenSSL [5], previously used in BIKE (and commonly used in cryptographic applications). The polynomial inversion is used in the key generation algorithm in BIKE-2. The polynomial being inverted is a part of the private key and therefore we show how to implement the inversion in a secure manner. The running time of the inversion dominates the running time of the key generation procedure in BIKE-2. The slow key generation algorithm was the main reason that BIKE-1 was preferred to BIKE-2 despite the size of the public key and the ciphertext in BIKE-2 being half of that in BIKE-1. The reduction of the running time of the key generation, gained by using our inversion algorithm, alleviated the prohibitive cost of the key generation and made BIKE-2 competitive with BIKE-1. Consequently, and as a result of our work, the Round-3 BIKE specification [18] removes the definitions of BIKE-1 and BIKE-3 and proposes BIKE-2 as the only variant of the KEM (denoted simply by BIKE).

Another completely new field of research in cryptography emerged recently – blockchains are distributed ledgers which are used to record transactions between parties in a verifiable and immutable way by a series of interactions between distributed nodes responsible for maintaining the ledger. The verifiability and the permanence of the records is (in part) ensured by various cryptographic primitives. Blockchains have been used in an attempt to solve different practical problems arising in decentralized systems, such as trust, privacy, anonymity, etc. Because of the wide range of potential applications (not only in purely technological settings), blockchains have attracted the attention of researchers from several different fields such as cryptography, privacy, systems, and even some non-technical disciplines, most notably, the field of economy. The inter-disciplinary nature of the field and the potential of practical use (and abuse) of blockchain based systems has attracted a lot of investment. The usefulness of blockchains is yet to be proven, but nevertheless, just as the field of post-quantum cryptography, arguably even more so, the field of blockchains opened many interesting questions that motivated the cryptographic community to get involved and tackle some of them.

As one of the largest blockchain platforms, Ethereum is actively working on several cryptographic problems. In 2019, the Ethereum research team was exploring options for building a “proof-of-custody” scheme for the next generation of the Ethereum protocol which requires a pseudorandom function that can be efficiently evaluated by a group of parties. The authors of [28] proposed the Legendre PRF as a particularly efficient primitive for multi-party computation. The Legendre PRF is a pseudorandom function based on the Legendre symbol, originally proposed by Damgård [29]. The function is modeled as an oracle parametrized with a prime p and a secret key k that on input $a \in \mathbb{F}_p$ outputs the Legendre symbol $\left(\frac{k+a}{p}\right)$, i. e., outputs a single bit determined by the quadratic residuosity of $k + a$ in $\mathbb{F}_p$. The attractiveness of the Legendre PRF for multi-party applications comes from the multiplicative property of the Legendre symbol, i. e., $\left(\frac{a}{p}\right)\left(\frac{b}{p}\right) = \left(\frac{ab}{p}\right)$. The hard problem associated with the Legendre PRF, conjectured by Damgård, is that given query access to a Legendre PRF oracle it is hard to recover the secret key. This allegedly hard problem coupled with the multiplicative property of the Legendre symbol made the Legendre PRF a promising candidate for Ethereum’s new protocol. Therefore, the Ethereum foundation announced several challenges and bounties for breaking the Legendre PRF [8].

In Chapter 6 we present the best algorithm so far to recover the secret key of an instantiation of the Legendre PRF. For a given p, our algorithm finds the secret key in $O(\sqrt{p \log\log p})$ operations with only $\sqrt[4]{p \log^2 p \log\log p}$ queries of the oracle. This is an improvement compared with the previous best algorithm [7] that achieves the key extraction with complexity $O(\sqrt{p}\, t \log^2 p)$ (where t is the complexity of computing a Legendre symbol). Moreover, when the number of queries to the oracle is limited to M, we reduce the complexity of the key recovery from $O(\frac{p t \log^2 p}{M^2})$ of the algorithm in [7] to $O(\frac{p \log p \log\log p}{M^2})$ of our algorithm. Furthermore, in Chapter 6 we give a detailed explanation of our implementation of the algorithm and describe several techniques that improved the performance and lowered the memory usage of the implementation. Finally, we present the solutions of the first three challenges posed by the Ethereum foundation. The most difficult challenge among the three has so far only been solved by our team. The results presented in Chapter 6 are a joint effort with Novak Kaluđerović and Thorsten Kleinjung from the Laboratory for Cryptologic Algorithms, EPFL, Switzerland. The preprint of our report is available at:

[30] N. Kaluđerović, T. Kleinjung, and D. Kostić, "Improved key recovery on the Legendre PRF", available at eprint.iacr.org/2020/098.

The algorithm presented in [30] is generalized and expanded in our subsequent paper which was accepted at the ANTS 2020 conference (math.auckland.ac.nz/~sgal018/ANTS).

2 Background

2.1 BIKE

Bit Flipping Key Encapsulation (BIKE) is a coding-based cryptographic scheme where certain concepts and algorithms from coding theory are employed to enable a secure cryptographic key exchange [9]. Error correcting codes are commonly and widely used in communication protocols for detecting and potentially correcting errors occurring in the transmitted data while it travels over a noisy channel. The principle behind the ability to correct transmission errors consists of introducing redundancy in the message so that on the receiving end the original message can be recovered correctly even if it was corrupted during the transmission. The typical scenario is depicted in Figure 2.1.

The original message m is *encoded* to the message c which is sent over the noisy channel. The receiver receives the corrupted message $c \oplus e$ which is *decoded* by the decoder to the message m'. Coding schemes are designed such that if the introduced error is *reasonably* low, the recovered and the original messages are identical. The error correction capacity of a system, i. e., the number of errors that can be corrected, depends on the specific scheme and its parameters.

In cryptographic settings, the error is usually intentionally introduced in order to either obfuscate the secret message being transmitted or as the secret information itself. In the following section we present basic definitions and outline the necessary mathematical tools which BIKE relies upon.

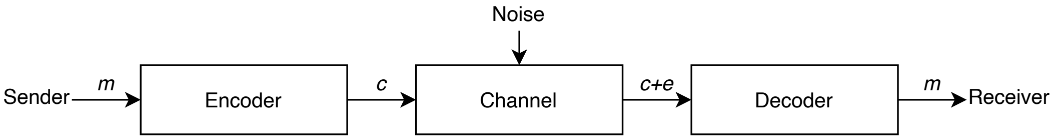


Figure 2.1 – Encoding and decoding

2.1.1 Preliminaries

Let $A = \{a_1, \dots, a_q\}$ be an *alphabet* where a_i values are called *symbols*. A block code is a code where the message is first decomposed in blocks of symbols of fixed length. Hereafter, we assume that the message consists of a single block of length k, i. e., the message is a vector $m \in A^k$. An encoding map is an injective map $A^k \longrightarrow A^n$ for an integer $n > k$. A block code $\mathscr{C}$ of length n over A is a subset of A^n. Vectors belonging to $\mathscr{C}$ are called *codewords*. When $A = \{0, 1\}$, the code is called a binary code. In the remaining part of the thesis, only binary codes are studied, i. e., $A \cong \mathbb{F}_2$, and the symbols are referred to as bits. Codes with ability to detect if an encoded message contains errors are called *error-detecting* codes, while codes which are able to correct certain number of errors are called *error-correction* codes.

The *Hamming weight* of a vector $x = (x_1, \dots, x_n)$ is defined as the number of non-zero elements of the vector, $w_H(x) = |\{i | x_i \neq 0\}|$. The Hamming distance between two vectors x and y is defined as the Hamming weight of their difference, $d_H(x, y) = w_H(x - y)$. The weight of a code $\mathscr{C}$ is defined as the minimum weight of the codewords. The minimum distance of a code $\mathscr{C}$ is defined as the minimum Hamming distance among all the possible pairs of vectors (codewords) in $\mathscr{C}$, $d(\mathscr{C}) = \min_{x,y \in \mathscr{C}, x \neq y} d_H(x, y)$. The distance of a code is an important parameter which determines the capacity for error detection and correction of the code.

Linear codes form a subset of error-correcting codes where each linear combination of two codewords is also a codeword. Linear codes offer several advantages, of which the most important ones are the following: a linear code can be compactly described using its basis (contrary to the non-linear codes where the description may involve the list of all codewords), encoding and decoding (a valid codeword) is straightforward, and the minimum distance of a linear code is equal to the weight of the code.

Definition 1. A binary (n, k)-linear code $\mathscr{C}$ of length n, dimension k, and co-dimension $r = (n - k)$, with $n \geq k$, is a k-dimensional vector subspace of $\mathbb{F}_2^n$ and may be denoted $\mathscr{C}(n, k)$. The dual code of $\mathscr{C}$ is the orthogonal complement of $\mathscr{C}$ in $\mathbb{F}_2^n$ and it is denoted by $\mathscr{C}^\perp$.

The proportion of the useful data in a codeword is called the *code rate*. For an (n, k)-linear code the code rate is equal to $\frac{k}{n}$, i. e., a codeword carries an information about a message of length k, while the size of the redundant information is $n - k$.

There are two ways to describe a binary linear code $\mathscr{C}(n, k)$, by its *generator matrix* or by its *parity-check matrix*. These two different representations correspond to two manners to describe a vector subspace of $\mathbb{F}_2^n$: either by providing its basis or by giving a system of linear equations whose solution space is the code, corresponding to generator matrix and parity-check matrix representation, respectively. A generator matrix G for a linear code $\mathscr{C}$ is a matrix whose rows form a basis for $\mathscr{C}$ as a vector subspace of $\mathbb{F}_2^n$.

Definition 2. (Generator matrix). A matrix $G \in \mathbb{F}_2^{k \times n}$ is a generator matrix of a binary (n, k)-linear code $\mathscr{C}$ if and only if

$$\mathscr{C} = \{mG \mid m \in \mathbb{F}_2^k\}.$$

A parity-check matrix defines a linear application with the code as its kernel.

Definition 3. (Parity-check matrix). A matrix $H \in \mathbb{F}_2^{(n-k)\times n}$ is a parity-check matrix of a binary (n,k)-linear code $\mathscr{C}$ if and only if

$$\mathscr{C} = \{c \in \mathbb{F}_2^n \mid Hc^T = 0\}.$$

Direct consequence of the above stated definitions is that for a linear code $\mathscr{C}$ we have that:

$$Hc^T = H(mG)^T = HG^T m^T = 0$$

for any message m, thus $HG^T = 0$ or equivalently $GH^T = 0$. In other words, G and H are orthogonal and therefore H can be viewed as the generator matrix for the dual code $\mathscr{C}^\perp$ of $\mathscr{C}$.

A generator matrix G and parity-check matrix H are said to be in standard (systematic) form if they are of the form $G = \begin{pmatrix} I_k & P \end{pmatrix}$ and $H = \begin{pmatrix} Q & I_{n-k} \end{pmatrix}$, where I_k and I_{n-k} denote the $k \times k$ and $(n-k)\times(n-k)$ identity matrices, respectively, and P and Q matrices of size $k \times (n-k)$ and $(n-k)\times k$, respectively. A generator matrix G of a systematic code is a full-rank matrix, or in other words, the first k columns of G are linearly independent. Hereafter, we discuss only systematic codes and therefore, omit the prefix systematic for conciseness. Given the generator matrix G of $\mathscr{C}(n,k)$ in its standard form with matrix P as defined above, the parity-check matrix H for $\mathscr{C}$ can be easily determined by $H = \begin{pmatrix} -P^T & I_{n-k} \end{pmatrix}$ since $GH^T = 0$, where in the $\mathbb{F}_2$ case the minus sign in front of P^T can be omitted since $a = -a$ for $a \in \mathbb{F}_2$. We note that, given a generator matrix G in non-standard form, one can always efficiently transform the generator to standard form by computing its reduced row echelon form using ordinary Gaussian elimination since the matrices are defined over a field.

Let $\mathscr{C}(n,k)$ be a code with a generator matrix $G \in \mathbb{F}_2^{k\times n}$. The encoding map is defined as the map:

$$\phi : \begin{cases} \mathbb{F}_2^k & \longrightarrow \mathbb{F}_2^n \\ m & \longmapsto mG. \end{cases}$$

If G is in standard form, then $\phi(m) = mG = m\begin{pmatrix} I_k & P \end{pmatrix} = \begin{pmatrix} m & mP \end{pmatrix}$. Therefore, computing the original message m given $\phi(m)$ is as simple as taking the first k coordinates of $\phi(m)$. Error detection is also efficiently done using the parity-check matrix because as noted above H follows directly from G. Namely, let $x = \phi(m) + e$ be the received message, then if Hx^T is different from zero, the error vector e is also different from zero, indicating that an error was introduced during transmission. However, recovering the original message from a noisy codeword, or in other words correcting the error, is less straightforward.

There are several decoding problems with different hardness levels and space and time complexities which are extensively studied in the literature. Here, we focus only on *syndrome decoding* because it is the one relevant for BIKE.

Definition 4. (Syndrome). Let $\mathscr{C}$ be a binary (n,k)-linear code with parity-check matrix H. For every $x \in \mathbb{F}_2^n$ the *syndrome* s of x determined by H is defined as:

$$s = Hx^T \in \mathbb{F}_2^{n-k}.$$

In literature, the syndrome is sometimes defined as $s = xH^T$, the definitions are equivalent and used interchangeably throughout the text. Following Definition 3, the syndrome of any codeword is a null vector. Note that the syndrome of $x = c + e$ where $c \in \mathscr{C}$ depends only on the error vector since by Definitions 3 and 4:

$$s = Hx^T = H(c+e)^T = Hc^T + He^T = He^T.$$

This property yields a natural algorithm for decoding, referred to as *list decoding*, that can be defined in two steps: precomputation and search. In the precomputation stage generate a list of syndromes He^T for all possible error vectors e and store the pairs (He^T, e) in a hash table. Then for any given $x = c + e$, compute the syndrome He^T and find the error e by a single lookup in the hash table, and recover the codeword by computing $c = x - e$. This approach obviously does not scale well with the maximum number of allowed errors t and the codeword size n since the space complexity depends on the number $\sum_{i=0}^{t} \binom{n}{i}$ of possible error patterns.

2.1.2 QC-MDPC codes

Quasi-cyclic (QC) codes form an important family of codes both in coding theory and practice. They are interesting from a theoretical standpoint because of their rich algebraic structure, while from a practical point of view they have several good properties among which the compactness of their representation stands out as probably the most important one. A code is said to be quasi-cyclic if any cyclic shift of a codeword by some number l of symbols is also a codeword, where if $l = 1$ the code is said to be cyclic.

Definition 5. An (n,k) linear code $\mathscr{C}$ of length $n = rn_0$ and dimension $k = rk_0$ is a quasi-cyclic code if the cyclic shift of any codeword by n_0 symbols yields another codeword.

The generator and the parity-check matrix of the quasi-cyclic code $\mathscr{C}(rn_0, rk_0)$ is completely defined by its first row because every other row is a cyclic shift by n_0 symbols of the preceding one. For example, consider the generator matrix G of the code $\mathscr{C}$ with $n_0 = 2$ and $k_0 = 1$, namely $n = 2r$ and $k = r$:

$$G = \begin{pmatrix} g_0 & g_1 & g_2 & g_3 & \cdots & g_{n-2} & g_{n-1} \\ g_{n-2} & g_{n-1} & g_0 & g_1 & \cdots & g_{n-4} & g_{n-3} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ g_2 & g_3 & g_4 & g_5 & \cdots & g_o & g_1 \end{pmatrix}.$$

Thus G is an $r \times 2r$ matrix. By rearranging the columns of G by grouping together every other n_0-th column, i. e., forming two groups, one containing the even numbered columns and the other with the odd numbered columns, the matrix G is transformed in the following matrix G':

$$G' = \begin{pmatrix} g_0 & g_2 & \cdots & g_{n-2} & g_1 & g_3 & \cdots & g_{n-1} \\ g_{n-2} & g_0 & \cdots & g_{n-4} & g_{n-1} & g_1 & \cdots & g_{n-3} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ g_2 & g_4 & \cdots & g_0 & g_3 & g_5 & \cdots & g_1 \end{pmatrix}.$$

It is easy to see that G' is composed of two circulant $r \times r$ matrices.

Definition 6. A circulant matrix is a square matrix in which every row is a cyclic shift of the adjacent row and every column is a cyclic shift of the adjacent column.

A circulant matrix A:

$$A = \begin{pmatrix} a_0 & a_1 & \ldots & a_{r-1} \\ a_{r-1} & a_0 & \ldots & a_{r-2} \\ \vdots & \vdots & \ddots & \vdots \\ a_1 & a_2 & \ldots & a_0 \end{pmatrix}$$

is fully determined by its first row. The polynomial $a(x) = \sum_{i=0}^{r-1} a_i x^i$ associated with the row vector $a = (a_0, a_1, \ldots, a_{r-1})$ is called the *defining polynomial* of A. A cyclic shift of a corresponds to multiplication of $a(x)$ by x modulo $x^r - 1$. Therefore, there is a natural one-to-one correspondence between circulant matrices of size $r \times r$ and ideals of the quotient ring $\mathbb{F}_2[x]/(x^r - 1)$. This algebraic structure of circulant matrices yields further useful properties. Namely, the sum and product of two circulant matrices A and B is a circulant matrix, where if $AB = C$, the defining polynomial of C is $c(x) = a(x)b(x) \mod x^r - 1$. Furthermore, we have that multiplication $AB = BA$ is commutative since $a(x)b(x) = b(x)a(x)$. The inverse of a circulant matrix A exists if and only if $a(x)$ is invertible modulo $x^r - 1$, i. e., $a(x)$ is relatively prime to $x^r - 1$, and the inverse A^{-1} is then defined by the inverse of the polynomial $a^{-1}(x)$ mod $x^r - 1$. Given a circulant matrix A with $a(x) = a_0 + a_1 x + \cdots + a_{r-1} x^{r-1}$, the associated polynomial of the transposed matrix A^T is defined by $a^T(x) = a_0 + a_{r-1} x + \cdots + a_1 x^{r-1}$. We denote the quotient polynomial ring $\mathcal{R} = \mathbb{F}_2[x]/(x^r - 1)$ hereafter.

The parameter r defining the ring $\mathcal{R}$ used in BIKE is chosen such that the polynomial defining $\mathcal{R}$ factors as $(x^r - 1) = \Phi_r(x) \cdot (x - 1) \in \mathbb{F}_2[x]$, where the cyclotomic polynomial $\Phi_r(x) = (x^r - 1)/(x - 1) \in \mathbb{F}_2[x]$ is irreducible. One consequence of this property of the ring is that checking if a polynomial in the ring is invertible is straightforward. If a polynomial $a(x) \in \mathscr{R}$ of degree at most $r-1$ is invertible both modulo $\Phi_r(x)$ and modulo $(x-1)$ then it is invertible in $\mathscr{R}$. Since $\Phi_r(x)$ is an irreducible polynomial of degree $r-1$, the sufficient condition for polynomial $a(x)$ to be invertible modulo $\Phi_r(x)$ is that $a(x) \neq \Phi_r(x)$. On the other hand we have that $x = 1 \mod (x-1)$, and consequently that $a(x) \mod (x-1)$ is 0 if the number of terms of $a(x)$ is even and 1 if it is odd. Therefore, $a(x)$ is invertible modulo $x-1$ if and only if it has an odd number of non-zero terms, i. e., the weight of $a(x)$ is odd. We note that the necessary and sufficient condition for the irreducibility of $\Phi_r(x) \in \mathbb{F}_2[x]$ is that r is a prime and that 2 is a primitive root modulo r.

Sum of two binary polynomials that have weights of same parity results in a polynomial of even weight, while summing two binary polynomials of even and odd weight gives an odd weight polynomial. Furthermore, product of two binary polynomials of odd weight is an odd weight polynomial and an even weight polynomial otherwise. For example, consider $a(x), b(x) \in \mathbb{F}_2[x]$, then:

- $wt(a+b)$ is even if $wt(a)$ and $wt(b)$ are both even or both odd, otherwise $wt(a+b)$ is odd.

- $wt(a \cdot b)$ is odd if $wt(a)$ and $wt(b)$ are odd, otherwise $wt(a \cdot b)$ is even.

This is an important observation when dealing with cyclic and quasi-cyclic codes since their generator matrices are defined by a single row. Consider the circulant matrix A as defined above. If the defining polynomial $a(x)$ has even weight, then the sum of all the columns of A is the zero vector because every coordinate of the resulting vector is simply the sum of all the terms of $a(x)$, which is an even number, and therefore, zero modulo 2. This implies that the matrix A is not a full-rank matrix because its columns are not linearly independent. Therefore, A is not systematic and cannot be a generator matrix of a systematic code. Following this observation, we conclude that the weight of the polynomial defining a generator matrix of a systematic cyclic code has to be odd. In case of a quasi-cyclic code, this means that the lefthand minor matrix of the generator has to be defined by an odd weight polynomial.

Consider a QC linear code $\mathscr{C}$ of length $n = rn_0$ and dimension rk_0. The generator and parity-check matrix of the code can be represented by $k_0 \times n_0$ and $(n_0 - k_0) \times n_0$ circulant $r \times r$ matrices, respectively:

$$G = \begin{pmatrix} G_{0,0} & \cdots & G_{0,n_0-1} \\ \vdots & & \vdots \\ G_{k_0-1,0} & \cdots & G_{k_0-1,n_0-1} \end{pmatrix}, \quad H = \begin{pmatrix} H_{0,0} & \cdots & H_{0,n_0-1} \\ \vdots & & \vdots \\ H_{n_0-k_0-1,0} & \cdots & H_{n_0-k_0-1,n_0-1} \end{pmatrix}$$

with all $G_{i,j}$ and $H_{i,j}$ circulant matrices, or equivalently with defining polynomials:

$$G = \begin{pmatrix} g_{0,0} & \cdots & g_{0,n_0-1} \\ \vdots & & \vdots \\ g_{k_0-1,0} & \cdots & g_{k_0-1,n_0-1} \end{pmatrix}, \quad H = \begin{pmatrix} h_{0,0} & \cdots & h_{0,n_0-1} \\ \vdots & & \vdots \\ h_{n_0-k_0-1,0} & \cdots & h_{n_0-k_0-1,n_0-1} \end{pmatrix}$$

where $g_{i,j}, h_{i,j} \in \mathscr{R}$. The two representations are used interchangeably in the rest of the thesis, in such a way that a circulant matrix is denoted by a capital letter and its defining polynomial by the same letter in lower case. When a matrix is composed of several circulant matrices concatenated horizontally, e. g., $G = \begin{pmatrix} G_0 & G_1 \end{pmatrix}$, then it is denoted by a pair of polynomials $g = (g_0, g_1)$. A row vector and its associated polynomial are both denoted by the same lower case letter.

Example. Let the code $\mathscr{C}$ be a binary QC code with the following parameters: $r = 7, n = 2r, k = r$, and the generator and parity-check matrices G and H as defined below.

$$G = \begin{pmatrix} G_0 & G_1 \end{pmatrix} = \left(\begin{array}{ccccccc|ccccccc} 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \end{array}\right),$$

$$H = \begin{pmatrix} H_0 & H_1 \end{pmatrix} = \left(\begin{array}{ccccccc|ccccccc} 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \end{array}\right).$$

Alternatively, G and H can be compactly represented with two pairs of polynomials $g, h \in \mathscr{R}$:

$$g = (g_0, g_1) = (x^4, x + x^2 + x^6)$$

$$h = (h_0, h_1) = (1 + x^2 + x^6, x^2 + x^3 + x^6)$$

It is straightforward to check that G and H are indeed the generator and parity-check matrix for the code by verifying that $GH^T = G_0 H_0^T + G_1 H_1^T = 0$. One can check that the same holds for the polynomials:

$$\begin{aligned}gh^T &= g_0h_0^T + g_1h_1^T\\ &= x^4(1+x+x^5)+(x+x^2+x^6)(x+x^4+x^5)\\ &= x^4+x^5+x^9+x^2+x^5+x^6+x^3+x^6+x^7+x^7+x^{10}+x^{11}\\ &= 0\end{aligned}$$

where the transposition of a polynomial pair is done element wise and all the operations are in $\mathscr{R}$ where $x^7 = 1$.

A message $m = \begin{pmatrix}0 & 1 & 0 & 0 & 1 & 1 & 0\end{pmatrix} \in \mathbb{F}_2^7$ with the associated polynomial $m(x) = x + x^4 + x^5$ is encoded to the codeword c by:

$$c = mG = \begin{pmatrix}mG_0 & mG_1\end{pmatrix} = \begin{pmatrix}0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0\end{pmatrix}$$

or

$$c = mg = (mg_0, mg_1) = (x+x^2+x^5, x^2+x^4+x^5).$$

The syndrome s corresponding to the codeword c is computed as:

$$s = Hc^T = \begin{pmatrix}0 & 0 & 0 & 0 & 0 & 0 & 0\end{pmatrix}^T$$

or

$$\begin{aligned}s &= hc^T\\ &= (h_0,h_1)(c_0,c_1)^T\\ &= h_0c_0^T + h_1c_1^T\\ &= (1+x^2+x^6)(x^2+x^5+x^6)+(x^2+x^3+x^6)(x^2+x^3+x^5)\\ &= 0.\end{aligned}$$

As expected the syndrome of a valid codeword is zero.

QC-MDPC decoding

A binary Moderate Density Parity Check (MDPC) code is a binary linear code determined by a relatively sparse parity-check matrix with typical density of $O(1/\sqrt{n})$.

Definition 7. An (n,k,r,w)-QC-MDPC code is a quasi-cyclic code of length $n = n_0r$, dimension $k = k_0r$, order r, and index n_0 admitting a parity-check matrix with constant row weight $w = O(\sqrt{n})$.

The sparsity of the parity-check matrix allows the use of relatively efficient iterative decoding techniques, such as the *bit-flipping* algorithm proposed in [21]. Bit-flipping decoding is a method of choice in BIKE because it offers good properties while also being very simple.

Let $\mathscr{C}$ be a binary MDPC code and let s be the syndrome of a noisy codeword $y = c + e$ with $c \in \mathscr{C}$ and e an error:

$$\begin{pmatrix} s_0 \\ s_1 \\ \vdots \\ s_{r-1} \end{pmatrix} = s = Hx^T = He^T = \begin{pmatrix} h_{0,0} & h_{0,1} & \dots & h_{0,n-1} \\ h_{1,0} & h_{1,1} & \dots & h_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ h_{r-1,0} & h_{r-1,1} & \dots & h_{r-1,n-1} \end{pmatrix} \cdot \begin{pmatrix} e_0 \\ e_1 \\ \vdots \\ e_{n-1} \end{pmatrix} =$$

$$= \begin{pmatrix} h_{0,0}e_0 + h_{0,1}e_1 + \cdots + h_{0,n-1}e_{n-1} \\ h_{1,0}e_0 + h_{1,1}e_1 + \cdots + h_{1,n-1}e_{n-1} \\ \vdots \\ h_{r-1,0}e_0 + h_{r-1,1}e_1 + \cdots + h_{r-1,n-1}e_{n-1} \end{pmatrix}.$$

Each coordinate of the syndrome vector s represents the value of one parity-check equation which involves coordinates of the error vector e defined by the matrix H. If the parity-check equation for s_i is not satisfied ($s_i \neq 0$), that implies that some of the coordinates of e involved in the equation for s_i are different from zero. This observation leads to the bit-flipping method, where the coordinates of the error vector are *guessed* based on the number of unsatisfied parity-check equations they are involved in. The guessing of the error coordinates and syndrome computation of the newly guessed error are done iteratively until a syndrome which is null vector is found, i. e., the error is corrected.

Consider y, e, s and H as defined above and let p be the probability that a bit of the error vector is set to one and $(1 - p)$ that it is set to zero. We assume that the error has Hamming weight $wt(e) = t$ and that it is uniformly distributed among the vectors of length n and weight t. Coordinate s_i of the syndrome corresponds to the i-th parity-check equation given by the coordinate-wise product $\langle h_i, e\rangle$, namely:

$$s_i = \langle h_i, e\rangle = h_{i,0}e_0 + h_{i,1}e_1 + \cdots + h_{i,n-1}e_{n-1}.$$

Note that since we are working with binary codes, this dot product can be computed as:

$$\langle h_i, e\rangle = \bigoplus_{j=0}^{n-1} h_{i,j}e_j$$

where $\bigoplus$ denotes the "exclusive or" operation (addition modulo 2).

Let h_i be a parity-check equation defined by the matrix H which involves j-th position of the error, i. e., h_i is such that $h_{i,j} = 1$. To illustrate with an example, if we are interested in the position $j = 0$ of the error and we have that $h_{1,0} = 1$ then h_1 defines one equation:

$$s_1 = h_{1,0}e_0 + h_{1,1}e_1 + \cdots + h_{1,n-1}e_{n-1}$$

which involves e_0. We define two conditional probabilities ρ_0 and ρ_1:

$$\rho_0 = \mathbb{P}[\langle h_i, e\rangle = 1 \mid e_j = 0]$$
$$\rho_1 = \mathbb{P}[\langle h_i, e\rangle = 1 \mid e_j = 1] = 1 - \rho_0.$$

Since

$$\langle h_i, e\rangle = \bigoplus_{j'=0}^{n-1} h_{i,j'} e_{j'} = h_{i,j} e_j \oplus \bigoplus_{\substack{j'=0 \\ j' \neq j}}^{n-1} h_{i,j'} e'_j$$

we have that when $e_j = 0$ the parity-check equation $\langle h_i, e\rangle = 1$ if and only if the number of non-zero error positions that participate in the equation excluding the j-th position is odd. Therefore,

$$\begin{aligned}
\mathbb{P}[\langle h_i, e\rangle = 1 \mid e_j = 0] &= \sum_{\substack{l=1 \\ l \text{ odd}}}^{w-1} \binom{w-1}{l} p^l (1-p)^{w-1-l} \\
&= \sum_{l=0}^{w-1} \binom{w-1}{l} \frac{1-(-1)^l}{2} p^l (1-p)^{w-1-l} \\
&= \frac{1}{2}\left(\sum_{l=0}^{w-1} \binom{w-1}{l} p^l (1-p)^{w-1-l} - \sum_{l=0}^{w-1} \binom{w-1}{l} (-p)^l (1-p)^{w-1-l} \right) \\
&= \frac{1}{2}\left(\big(p + (1-p)\big)^{w-1} - \big(-p + (1-p)\big)^{w-1} \right) \\
&= \frac{1-(1-2p)^{w-1}}{2}.
\end{aligned}$$

It follows that

$$\rho_0 = \frac{1-\epsilon}{2} \text{ and } \rho_1 = \frac{1+\epsilon}{2},$$

where $\epsilon = (1-2p)^{w-1}$. If probability $p < \frac{1}{2}$ then $\epsilon > 0$ and consequently, probability that a parity-check equation involving the j-th position of the error is higher when $e_j = 1$ than when $e_j = 0$, i. e., $\rho_1 > \rho_0$. Moreover, this bias ϵ in probability of the equation to evaluate to one or zero depending on the value of e_j depends also on the code density – smaller density w leads to larger bias ϵ. This is exactly the reason why when working with MDPC codes, their "moderate" density can be exploited to guess the error with high probability of success.

The strategy of the bit-flipping algorithm, given in Algorithm 1, is simple: count the number of unsatisfied parity-check (UPC) equations for each bit of the error vector, flip those error bits for which the UPC is higher than some threshold, compute the new syndrome derived from the new error, and repeat these three steps until the error is corrected. The algorithm is probabilistic and with some probability it does not find the right error and runs indefinitely. Therefore, a maximum number of iterations is usually specified such that the decoder stops after that many iterations even if the syndrome is not decoded. The probability of a decoder to fail in finding the correct error is called Decoding Failure Rate (DFR).

The DFR of the bit-flipping decoder and other iterative decoders derived from it depends on several factors, such as the distribution of set bits in the parity-check matrix and in the error vector, and the choice of the threshold. For example, in Section 3.6 we show that not all parity-check matrices are equal in terms of decoding capability and in [22] the authors showed that not all error vectors have the same probability to be corrected. It has been shown in [21] that given a parity-check matrix and a syndrome which are chosen uniformly at random, the probability that the decoder corrects a certain number of errors can be explicitly computed as a function of code density, syndrome weight, and the threshold. However, already after the first iteration of the algorithm the distribution of the set bits in the syndrome is not uniform any more and the failure probability cannot be computed.

The threshold used for the decision to flip an error bit is an important parameter of the algorithm. It determines the balance between the efficiency and the likelihood to fail of the algorithm. In other words, the higher the threshold the algorithm is more “conservative” – it flips fewer error bits per iteration and therefore needs more iterations to decode. On the other hand, with higher threshold the algorithm is less likely to make a mistake, i. e., flip an error bit which is not really an error, and therefore fail to decode. For example, one conservative approach can be to set in each iteration the threshold to the maximum number of counted UPCs. Choosing an optimal threshold is not straightforward because the threshold depends on the error weight. Even if the exact weight of the error corresponding to the given syndrome is fixed in the system and known, after flipping some of the bits in the first iteration, we cannot know how many of the flipped bits are indeed the error bits and how many were mistakenly flipped. Therefore, after the first iteration the number of the remaining error bits is not known and the optimal threshold cannot be computed.

If several conditions are met, e. g., H is sparse enough, there exists an error e of small enough weight which corresponds to the given syndrome s, and if the threshold τ is chosen such that it is not too far from the optimal threshold, then with high probability the algorithm stops, recovers the error e and returns the corrected codeword c, otherwise the algorithm runs indefinitely. Since a closed formula to compute the exact DFR of a decoder is not known, the DFR is usually determined experimentally, i. e., decoding is run many times with different parity-check matrices and error vectors and the number of failures is counted.

In the BIKE proposal, two different decoders are used, “BackFlip” and “Black-Gray”. They are both based on Algorithm 1, however, certain mechanisms are built into them so that the algorithms can recover (to a degree) from making a mistake. Performance and DFR analysis of those two decoders is presented in the subsequent chapters. Further details about the threshold selection rules and the effect on the DFR of a decoder is available in the literature [21, 1, 31, 32, 33].

Algorithm 1 e = BitFlipping(s, H) [21].

Input: Parity-check matrix $H \in \mathbb{F}_2^{r\times n}$, $y \in \mathbb{F}_2^n$ such that $y = c + e$.
Output: Codeword $c \in \mathbb{F}_2^n$.

```
procedure BitFlipping($y$, $H$)
    $s \leftarrow yH^T$
    $s' \leftarrow s$
    $e \leftarrow 0$
    while $s \neq 0$ do
        $\tau \leftarrow$ compute threshold according to a predefined rule
        for $i = 0 \ldots n-1$ do
            if $\mathrm{upc}_i(s', H) \geq \tau$ then
                $e_i \leftarrow e_i \oplus 1$
        $s' \leftarrow s - eH^T$
    return $y - e$
```

$\mathrm{upc}_i(s', H)$ is a function that computes the number of unsatisfied parity-check equations involving e_i, the i-th position of the error vector e.
The parameter τ is the threshold for error bit flip decision, refer to the text for details.

2.1.3 McEliece and Niederreiter cryptosystems

Codes were introduced in public key cryptography by McEliece in his paper [34], where he proposed a scheme based on Goppa codes. The secret key in the scheme is a generator matrix of the selected error-correcting code. The secret generator is then transformed to a generator of a general linear code and set as the public key. Encryption of a message in McEliece's cryptosystem is done by encoding the message with the public matrix and obscuring the codeword by adding some error to it. Decryption of the ciphertext is done by decoding the produced syndrome. The ciphertext is a syndrome of a general linear code for which the decoding problem is known to be NP-hard. Although there is no proof that decoding of a general linear code is hard on average, after many years of research, a subexponential decoding algorithm that solves it still does not exist, even for quantum computers. Thus, it is believed that relying on this problem for the security of a cryptosystem is safe. However, with knowledge of the secret generator matrix and the transformation, it is possible to convert the ciphertext to a syndrome of the chosen error-correcting code where decoding is easy. The McEliece cryptosystem has an advantage in that the encryption and decryption functions are fairly fast. Furthermore, the system is resistant to quantum computers in a sense that no quantum polynomial time algorithm exists which could solve the decoding problem without the knowledge of the secret information about the code. However, the big disadvantage of the system is the size of the public key that is required for a reasonable security level.

McEliece cryptosystem

All parties involved in communication using McEliece encryption share a set of security parameters n, k, t, where n and k determine the length and dimension of a code, respectively, and t is the maximum number of errors the code can correct. The public and secret key are generated by the following procedure:

1. Select a random binary linear code $\mathscr{C}(n,k)$ with generator matrix G,that allows efficient decoding and correction of up to t errors.
2. Select a random $k \times k$ non-singular matrix S called the scrambling matrix.
3. Select a random $n \times n$ matrix P called the permutation matrix.
4. Compute matrix $G' = SGP$.
5. Output the public key G' and the secret key (G,S,P).

Given a message m, the ciphertext is computed as $c = mG' + e$, where e is a randomly generated error vector of length n with Hamming weight t. The decryption process works in the following three steps: compute $c' = cP^{-1}$, then decode c' to m', and finally recover the plaintext by computing $m = m'S^{-1}$.

Niederreiter cryptosystem

The next big development in coding-based cryptography was a paper by Niederreiter [35] in which he proposed a variation of the McEliece cryptosystem. Both schemes admit the same security, but Niederreiter's cryptosystem offers much faster encryption, and furthermore Niederreiter's scheme can be used to create a digital signature scheme.

All the involved parties share the same set of security parameters n, k, t as in McEliece's system. The key generation in the Niederreiter cryptosystem is the following:

1. Select a random binary linear code $\mathscr{C}(n,k)$ with generator matrix G and parity-check matrix H that allows efficient decoding and correction of up to t errors.
2. Select a random $(n-k) \times (n-k)$ non-singular matrix S called the scrambling matrix.
3. Select a random $n \times n$ matrix P called the permutation matrix.
4. Compute matrix $H' = SHP$.
5. Output the public key H' and the secret key (H,S,P).

To encrypt a given plaintext m, it is first mapped to an n-bit string e with Hamming weight t (or several e's if needed). The ciphertext is then computed by $c = H'e^T$. To decrypt a ciphertext, the syndrome is computed $s = S^{-1}c$ which is then decoded and the value $e' = Pe^T$ is recovered. The n-bit string e is then recovered with $e^T = P^{-1}e'$ and finally, the plaintext message m is found by applying the inverse of the map done in the encryption algorithm, i. e., by mapping the n-bit string e of weight t to a message m. Therefore, for this system we need an efficient way to map random information (plaintext) to constant weight words (error) and also an efficient inverse of this map. This problem is studied in [36].

We note that in a *key encapsulation mechanism*, contrary to a cryptosystem, the message itself is not important in a sense that it carries "readable" information. Therefore, mapping a message as an n-bit string with a certain weight can be replaced with generating an n-bit string of the specified weight, thus, sidestepping the potential efficiency issues of the mapping.

2.1.4 Key Encapsulation Mechanism

Public key encryption (PKE) systems or *asymmetric* cryptosystems are a necessary ingredient in any secure communication application in practice. However, they are not used for directly encrypting the communication data for several reasons. One of the main disadvantages of asymmetric cryptosystems is their performance which is usually rather poor compared to *secret key* (*symmetric*) cryptosystems. Another important limitation of public key encryption is that the message space is restricted which leads to various problems in designing secure protocols. In practice, *hybrid* encryption schemes are used, which consist of a public key encryption technique used to encrypt a key which is in turn used to encrypt the actual message with a symmetric key encryption scheme. Secure transmission of the symmetric key is done with a *Key Encapsulation Mechanism* (KEM), which consists of the following three algorithms:

- $\mathit{keygen}(1^\lambda) \rightarrow (pk, sk)$: the key generation algorithm outputs the public and secret key based on the security parameter λ.
- $\mathit{encaps}(pk) \rightarrow (ss, ct)$: the key encapsulation algorithm receives as input the public key pk and outputs the generated shared secret ss which is a symmetric key, and the ciphertext ct which represents the encapsulation of ss.
- $\mathit{decaps}(sk, ct) \rightarrow ss$: the decapsulation algorithm on inputs sk and ct outputs either the shared secret key ss or a failure.

Both *keygen* and *encaps* are probabilistic algorithms in a sense that they require a source of entropy to properly generate the key pair and the shared secret, respectively. Since there is a possibility that the decapsulation fails, we say that a KEM is δ-correct if for all $(pk, sk) \leftarrow \mathit{keygen}(1^\lambda)$ and $(ss, ct) \leftarrow \mathit{encaps}(pk)$ the failure probability satisfies the following condition:

$$\mathbb{P}[\mathit{decaps}(sk, ct) \neq k] \leq \delta.$$

The scenario for a KEM where two parties (A and B) need to derive a shared key is the following:

1. A generates its public and secret key (pk, sk) by using the *keygen* function and sends the public key to B.

2. B generates the shared secret symmetric key *ss* and encapsulates it in ciphertext *ct* by using the function *encaps* with the public key of A. Party B sends the ciphertext to A.

3. A decapsulates the shared key *ss* from the received ciphertext *ct*.

Therefore, in terms of performance it is important to note that one side in the communication performs the key generation and decapsulation algorithms, while the other side performs only the encapsulation algorithm. Regarding the communication bandwidth, in one direction a public key is sent, and a ciphertext in the other one.

2.1.5 IND-CPA and IND-CCA security notions

Indistinguishability (IND-) is the established notion of security of cryptographic schemes. A cryptographic system is modeled in terms of a *game* between a challenger and an adversary. The challenger has its secret key and the adversary is able to interact with the challenger via certain function calls which depend on the adversary's capabilities. The goal of the adversary is to play the game and break the system with some non-negligible probability, in which case the system is not secure under the defined security notion. For a key encapsulation mechanism the indistinguishability property means that an adversary is unable to distinguish between a ciphertext *ct* of a shared secret key *ss* and a random ciphertext ct'.

Based on the capabilities of the adversary two different security levels are defined: indistinguishability under chosen plaintext (IND-CPA) and indistinguishability under chosen ciphertext (IND-CCA) attack. Here we give informal intuitive descriptions of these security notions, full details with various games and adversaries are given in [9].

The IND-CPA notion, usually referred to as semantic security, is a basic requirement for any cryptographic protocol. In an encryption scheme, the IND-CPA game is defined such that the adversary chooses two messages (plaintexts) and provides them to the challenger which then randomly chooses one of the two messages, encrypts it and returns the ciphertext to the adversary. The goal of the adversary is to distinguish from which plaintext the ciphertext was generated. In the KEM schemes the game is slightly modified, namely the IND-CPA game is set up such that the adversary is given access to the encapsulation function, referred to as oracle, of the challenger. Once the challenger generates its key pair and publishes the public key, the adversary queries the encapsulation oracle and the challenger:

- Runs the encapsulation algorithm and generates a key *ss* and the ciphertext *ct* corresponding to *ss*.

- Generates a random ss'.

- Returns the ciphertext *ct* and $ss^\star$ which is one of *ss* and ss' chosen randomly.

The goal of the adversary is then to distinguish if *ct* is indeed the ciphertext corresponding to the received key $ss^{\star}$ or not.

The IND-CCA is a stronger notion of security than IND-CPA in a sense that the adversary has additional capabilities. More precisely, in IND-CCA games in addition to the encapsulation oracle access, the adversary is given access to the decapsulation oracle as well. There are two levels of IND-CCA security:

- IND-CCA1 where the adversary can query the decapsulation oracle only up until it receives the challenge ciphertext.
- IND-CCA2 where the adversary is allowed to query the decapsulation oracle even after receiving the challenge ciphertext, with an obvious limitation that it can ask the challenger to decapsulate any ciphertext except the received challenge ciphertext which would make the game trivial to win for the adversary.

The first level of security, IND-CCA1, is also called the *non-adaptive* IND-CCA and it is a weaker notion than the IND-CCA2 security which is referred to as *adaptive* IND-CCA.

Fujisaki-Okamoto (FO) transformation is a method for converting an IND-CPA secure PKE to an IND-CCA secure KEM. The FO transformation is done in two phases, firstly, the IND-CPA secure PKE is converted to an IND-CCA secure PKE which is then converted to an IND-CCA secure KEM. Consider a simple probabilist PKE,e. g., McEliece cryptosystem, with encryption and decryption functions defined as $ct \leftarrow E_{pk}(m, \ coin)$ and $m \leftarrow D_{sk}(ct)$, respectively. Note that in the encryption algorithm in McEliece's system the message is encoded to a codeword using the public key and the *coin* providing the required entropy is used to generate a random error vector which is added to the codeword to produce the ciphertext. Effectively, this means that the attacker can freely choose the ciphertext to submit to the decryption oracle. To upgrade the security of this PKE from CPA to CCA, in the first phase of the FO transformation we add "de-randomization" and "plaintext checking" to the PKE. De-randomization means that the random coin is transformed such that it is derived from the message. For this purpose, a one-way function is introduced to the system, for example a hash function called **H**, and the new encryption function is defined as $ct \leftarrow \mathscr{E}_{pk}(m, \ \mathbf{H}(m))$. The second change, namely, the plaintext checking is introduced in the decryption algorithm. The new decryption algorithm $\mathscr{D}$, after decrypting the ciphertext to a plaintext message m', re-encrypts the plaintext by applying the $\mathscr{E}$ algorithm to m' and verifies if the received ciphertext indeed corresponds to the obtained plaintext. In case it does, $\mathscr{D}$ outputs the message, otherwise it reports a failure and does not output the resulting plaintext. In the context of McEliece's cryptosystem these two changes result in the inability of the attacker to freely choose the ciphertext because the error vector is derived from a hash of the message during encryption, while the decryption algorithm checks if this was properly done. Therefore, with these two changes applied to the initial PKE, an IND-CCA secure PKE is obtained. The second phase of the FO transform, from CCA secure PKE to CCA secure KEM, is done by introducing another one-way function which is used to derive a shared secret key from a plaintext message. This rather simplistic explanation of the FO transform is meant only to give an intuition about the process of designing a secure KEM, full details of the transformation and its application to various cryptographic schemes in different security scenarios can be found in [37, 38, 39].

2.1.6 BIKE KEMs

The BIKE suite defines three different variants, each with separate definitions for IND-CPA and IND-CCA security. All three versions, called BIKE-1, BIKE-2, and BIKE-3, are based on the McEliece and Niederreiter frameworks and come with certain trade-offs in terms of performance and required communication bandwidth. The CPA versions of BIKE are secure only against a *passive* attacker which is able to monitor exchanged messages between two parties. Therefore, it is used only in a scenario where the keys are *ephemeral*, i. e., any particular set of public/secret key is used only once and a new key pair is generated for every key exchange. On the other hand, the IND-CCA2 secure BIKE can be used in a *static* key setting because it admits a certain level of security even against an *active* attacker which is able to modify the messages being transmitted in a key exchange between the communicating parties. In the rest of the text, the IND-CCA2 security of BIKE is referred to simply as IND-CCA.

There are several global parameters defined by BIKE which are used by the KEM algorithms. The parameters are determined based on the desired security level λ and the error correction capacity of a specific decoder used in the decapsulation method. The fundamental parameter is a prime r which determines the polynomial ring $\mathscr{R} = \mathbb{F}_2[x]/(x^r - 1)$ over which the arithmetic operations are done in BIKE. The prime r is such that the polynomial $(x^r - 1)/(x - 1) \in \mathbb{F}_2[x]$ is irreducible, which is a property mandated by security requirements, i. e., there exist attacks which can exploit the factorization of $(x^r - 1)$ into factors of smaller degree to break the system (full details of the attack are given in [9]). The code used in BIKE is a QC-MDPC code (Definition 7) with order r as defined above, index $n_0 = 2$, length $n = n_0 r = 2r$ and dimension $k = k_0 r = r$. The parameter determining the density of the code is w, thus the parity-check matrix is defined by a polynomial of Hamming weight w. The number of errors that can be corrected by the decoder is denoted by t. The two hash functions used in encapsulation and decapsulation are denoted by $\mathbf{K}$ and $\mathbf{H}$. The first hash function $\mathbf{K}: \{0,1\}^n \rightarrow \{0,1\}^l$, where l is the desired length of the shared secret symmetric key being encapsulated, is used in both CPA and CCA versions. The second hash function $\mathbf{H}$ is used only in the CCA version due to the additional requirements imposed by the CCA security, and it is defined separately for the three variants of BIKE.

Notational conventions. In the next three sections, all the variants of BIKE are presented. For each of the three BIKE versions three tables are presented explaining the key generation, encapsulation and decapsulation procedures. The CPA and CCA versions of a procedure for a variant of BIKE are shown together in a same table. The leftmost column of a table denotes either the input arguments of the method (IN), a step in the algorithm, or one of the algorithm outputs (PK - public key, SK - secret key, CT - ciphertext which encapsulates the shared symmetric key, SS - the shared secret symmetric key). Uniform random sampling from a set W is denoted by $w \xleftarrow{\$} W$. In the explanatory text that accompanies the tables, a block-circulant matrix represented by polynomials is denoted by the capitalization of the letter of the corresponding polynomials. For example, a matrix G is a single block circulant matrix with polynomial g as its first row. Another example is a matrix composed of two circulant matrices $F = \begin{pmatrix} F_0 & F_1 \end{pmatrix}$ where F_0 and F_1 denote matrices with their first row determined by polynomials f_0 and f_1, respectively. A vector and its corresponding polynomial are both denoted by the same small letter, e. g., concatenation of two vectors $\begin{pmatrix} c_0 & c_1 \end{pmatrix}$ is represented by a pair of polynomials (c_0, c_1).

2.1.7 BIKE-1

The first variant of BIKE is based on McEliece's cryptosystem. The key generation flow is shown in Table 2.1. The secret key is a block-circulant matrix composed of two circulant matrices defined by two polynomials $h_0, h_1 \in \mathscr{R}$ such that $H^T = \begin{pmatrix} H_0 \\ H_1 \end{pmatrix}$. The two polynomials are chosen randomly from the set of polynomials in $\mathscr{R}$ with the specified Hamming weight $d = \frac{w}{2}$. The public key (f_0, f_1) is generated by scrambling the secret matrix with a random square matrix G of size r represented by polynomial $g \in \mathscr{R}$, i. e., by multiplying both h_0 and h_1 by g. Note that polynomials h_0, h_1, g are chosen such that they have odd Hamming weight to ensure that the generator and parity-check matrices are systematic, as explained in Section 2.1.2. If we consider the public key matrix $F \simeq (f_0, f_1)$ to be the generator matrix of the code, it is easy to see that the secret matrix H is the parity-check matrix of the code since $FH^T = 0$:

$$\begin{aligned} FH^T &= \begin{pmatrix} F_0 & F_1 \end{pmatrix} \begin{pmatrix} H_0 \\ H_1 \end{pmatrix} = \begin{pmatrix} GH_1 & GH_0 \end{pmatrix} \begin{pmatrix} H_0 \\ H_1 \end{pmatrix} \\ &= GH_1H_0 + GH_0H_1 = G(H_1H_0 + H_0H_1) = 2GH_0H_1 = 0, \end{aligned}$$

exploiting the commutativity mentioned in Section 2.1.1, and noting that we are working over $\mathbb{F}_2$.

In the IND-CCA case two additional polynomials are sampled uniformly at random from $\mathscr{R}$ and attached to the secret key (for a reason that is explained in the decapsulation phase).

Bandwidth requirement for BIKE-1, i. e., the size of the public key and the data (ciphertext), is equal to the length of the code $n = 2r$ in each direction of the communication.

Table 2.2 describes the encapsulation algorithm which receives a public key (f_0, f_1) as input. In both CPA and CCA case a random message m is sampled from $\mathscr{R}$, which is afterwards encoded by computing its product with the public key. The ciphertext c is then generated by adding a random error vector (e_0, e_1) to the encoded message, i. e., $c = (c_0, c_1) = (mf_0 + e_0, mf_1 + e_1)$.

	BIKE-1 IND-CPA	BIKE-1 IND-CCA
1.	$h_0, h_1 \xleftarrow{\$} \mathcal{R}$ both of odd Hamming weight $wt(h_0) = wt(h_1) = w/2$	
2.	-	$\sigma_0, \sigma_1 \xleftarrow{\$} \mathcal{R}$
3.	$g \xleftarrow{\$} \mathcal{R}$ of odd weight	
4.	$(f_0, f_1) = (gh_1, gh_0)$	
PK	(f_0, f_1)	(f_0, f_1)
SK	(h_0, h_1)	$(h_0, h_1, \sigma_0, \sigma_1)$

Table 2.1 – BIKE-1 IND-CPA/IND-CCA key generation flow.

The CPA and CCA versions differ in the way of generating the error vector (step 2) and generating the desired secret shared key (step SS). In the CPA case, the error is generated simply by sampling two random vectors $e_0, e_1 \in \mathcal{R}$ such that their combined Hamming weight is t (the error correction capacity of the code), and the symmetric key is taken as the output of the hash function of the error $ss = \mathbf{K}(e_0, e_1)$. Note that here, the *information* is actually conveyed in the error vector (the key *ss* depends only on (e_0, e_1)), while the *randomness* used to obscure this information is in the message m.

The CCA version is slightly more complex because of the higher security requirements. The place of the *information* and *randomness* is the opposite of the CPA case. Namely, the information is in the message m itself and the randomness comes from the error vector. This reversal of the roles of *information* and *randomness* is due to the FO transform required for CCA security, as explained in the decapsulation algorithm below. Furthermore, the error is not sampled independently but rather generated from m with a custom hash function $\mathbf{H} : \{0,1\}^n \to \{y \in \{0,1\}^n | wt(y) = t\}$. This modification is done for two reasons (mentioned in Section 2.1.5): firstly, to avoid reaction attacks by forcing a uniform distribution of the error, and secondly, to enable plaintext checking necessary for the FO transform.

	BIKE-1 IND-CPA	BIKE-1 IND-CCA
IN	Public key (f_0, f_1)	
1.	$m \xleftarrow{\$} \mathcal{R}$	
2.	$e_0, e_1 \xleftarrow{\$} \mathcal{R}$	$(e_0, e_1) = \mathbf{H}(mf_0, mf_1)$
	where $wt(e_0) + wt(e_1) = t$	
3.	$(c_0, c_1) = (mf_0 + e_0, mf_1 + e_1)$	
CT	$c = (c_0, c_1)$	
SS	$ss = \mathbf{K}(e_0, e_1)$	$ss = \mathbf{K}(mf_0, mf_1, c)$

Table 2.2 – BIKE-1 IND-CPA/IND-CCA key encapsulation flow.

The decapsulation algorithm is shown in Table 2.3. Inputs of the algorithm are the ciphertext c and the secret key (h_0, h_1) in the CPA case and $(h_0, h_1, \sigma_0, \sigma_1)$ in the CCA case. The syndrome

s is computed with the secret parity-check matrix:

$$s = cH^T = \begin{pmatrix} c_0 & c_1 \end{pmatrix} \begin{pmatrix} H_0 \\ H_1 \end{pmatrix} = c_0 H_0 + c_1 H_1.$$

It is easy to verify that s is indeed a syndrome of the code and that it corresponds to the message m and error vector (e_0, e_1) from the encapsulation phase:

$$\begin{aligned} s &= c_0 H_0 + c_1 H_1 \\ &= mF_0 H_0 + e_0 H_0 + mF_1 H_1 + e_1 H_1 \\ &= m(F_0 H_0 + F_1 H_1) + (e_0 H_0 + e_1 H_1) \\ &= mFH^T + \begin{pmatrix} e_0 & e_1 \end{pmatrix} H^T \\ &= \begin{pmatrix} e_0 & e_1 \end{pmatrix} H^T. \end{aligned}$$

Therefore, decoding the syndrome, if it succeeds, recovers the error vector (e'_0, e'_1) which is equal to the original error (e_0, e_1).

In the CPA case, we check if the weight or the error is correct and if the decoder succeeded, and either compute the shared secret key ss or return a failure symbol.

The CCA algorithm contains an additional check - the aforementioned plaintext check, where it is verified that the error obtained by the decoder and the error computed by hashing the plaintext are equal. Plaintext hash (Step 3), is computed in the same way as in encapsulation: $(e''_0, e''_1) = \mathbf{H}(c_0 + e'_0, c_1 + e'_1)$. If the original error (e_0, e_1) and the error recovered by the decoder (e'_0, e'_1) are the same then:

$$\begin{aligned} (e''_0, e''_1) &= \mathbf{H}(c_0 + e'_0, c_1 + e'_1) \\ &= \mathbf{H}(mf_0 + e_0 + e'_0, mf_1 + e_1 + e'_1) \\ &= \mathbf{H}(mf_0, mf_1) \\ &= (e_0, e_1). \end{aligned}$$

If this additional condition is satisfied, then the shared key is computed by $ss = \mathbf{K}(c_0 + e'_0, c_1 + e'_1, c_0, c_1)$. In contrast to the CPA case where if any of the conditions in Step 4 is unsatisfied the algorithm returns a failure, in the CCA decapsulation the algorithm returns a value for the shared key computed with (σ_0, σ_1) values of the secret key: $ss = \mathbf{K}(\sigma_0, \sigma_1, c)$.

2.1.8 BIKE-2

The second variant of BIKE is based on Niederreiter framework and features a parity-check matrix in standard form. The main advantage of BIKE-2 stems directly from this feature – the size of the data being transmitted is halved compared to BIKE-1. On the other hand, the main disadvantage of the scheme comes from the requirement to generate the standard form of the

	BIKE-1 IND-CPA	BIKE-1 IND-CCA
IN	Ciphertext $c = (c_0, c_1)$	
	(h_0, h_1)	$(h_0, h_1, \sigma_0, \sigma_1)$
1.	Compute the syndrome $s = c_0 h_0 + c_1 h_1$	
2.	$(e'_o, e'_1) \leftarrow \text{decode}(s, h_0, h_1)$	
3.	-	$(e''_0, e''_1) \leftarrow \mathbf{H}(c_0 + e'_0, c_1 + e'_1)$
4.	if $wt(e'_0) + wt(e'_1) \neq t$ or decoding failed or $(e'_0, e'_1) \neq (e''_0, e''_1)$ then	
SS	return $\perp$	$ss = \mathbf{K}(\sigma_0, \sigma_1, c)$
5.	else	
SS	$ss = \mathbf{K}(e'_0, e'_1)$	$ss = \mathbf{K}(c_0 + e'_0, c_1 + e'_1, c_0, c_1)$

Table 2.3 – BIKE-1 IND-CPA/IND-CCA key decapsulation flow.

matrix in the key generation algorithm which involves a costly polynomial inversion. This is not a big problem in a static key setup where the key pair is generated only once, but when ephemeral keys are used the key generation might become prohibitively expensive due to the cost of the inversion.

The key generation procedure of BIKE-2 is shown in Table 2.4. The first two steps, where the secret parity-check matrix (h_0, h_1) is generated, are the same as in BIKE-1. The inverse H_0^{-1} of matrix H_0 plays the role of the scrambling matrix S from Niederreiter cryptosystem. Therefore the public key is computed as:

$$F = H_0^{-1}\begin{pmatrix} H_0 & H_1 \end{pmatrix} = \begin{pmatrix} I_r & H_1 H_0^{-1} \end{pmatrix}.$$

Obviously, F is in standard form since the first matrix block is the identity matrix of size r.

	BIKE-2 IND-CPA	BIKE-2 IND-CCA
1.	$h_0, h_1 \xleftarrow{\$} \mathcal{R}$ both of odd Hamming weight $wt(h_0) = wt(h_1) = w/2$	
2.	-	$\sigma_0, \sigma_1 \xleftarrow{\$} \mathcal{R}$
3.	$(f_0, f_1) = (1, h_1 h_0^{-1})$	
PK	(f_0, f_1)	(f_0, f_1)
SK	(h_0, h_1)	$(h_0, h_1, \sigma_0, \sigma_1)$

Table 2.4 – BIKE-2 IND-CPA/IND-CCA key generation flow.

Table 2.5 presents the encapsulation flow of BIKE-2. Following Niederreiter's algorithm the message is first "encoded" as an n bit vector $e = \begin{pmatrix} e_0 & e_1 \end{pmatrix}$ of weight t. Note that there is no actual message here, so e is chosen independently. The ciphertext is then computed by:

$$c = Fe^T = \begin{pmatrix} I_r & H_1 H_0^{-1} \end{pmatrix} \begin{pmatrix} e_0 \\ e_1 \end{pmatrix} = e_0 + e_1 H_1 H_0^{-1}.$$

In the CPA case both e_0 and e_1 are randomly selected from $\mathcal{R}$ such that their combined weight is t.

The BIKE-2 scheme transformation from IND-CPA to IND-CCA is different from BIKE-1. The reason is that in BIKE-2 there are no message and randomness (plaintext and error) like in BIKE-1 so that the roles of a message and randomness could be switched and uniform distribution of the error enforced. Therefore, in the CCA case e is produced from a random seed z with a custom hash function $\mathbf{H} : \{0,1\}^{l_k} \rightarrow \{y \in \{0,1\}^n \mid wt(y) = t\}$.

The required plaintext checking for the FO transform performs the check on the seed instead of on the plaintext as in BIKE-1. Therefore, an additional value needs to be sent with the ciphertext which will allow recovering of the seed in the decapsulation phase. The seed is masked with a simple one-time pad in Step 4 of the algorithm, and the pair of values (c, d) sent as ciphertext.

The shared symmetric key is computed similarly as in BIKE-1: in the CPA version ss is a hash of the error vector, while in the CCA case the input to the hash function is a concatenation of the error and the ciphertext (c, d).

	BIKE-2 IND-CPA	BIKE-2 IND-CCA
IN	Public key (f_0, f_1)	
1.	-	$z \xleftarrow{\$} \{0,1\}^{l_K}$
2.	$e_0, e_1 \xleftarrow{\$} \mathcal{R}^2$	$(e_0, e_1) = \mathbf{H}(z)$
	where $wt(e_0) + wt(e_1) = t$	
3.	$c = e_0 + e_1 f_1$	
4.	-	$d = \mathbf{K}(e_0, e_1) \oplus z$
CT	c	(c, d)
SS	$ss = \mathbf{K}(e_0, e_1)$	$ss = \mathbf{K}(e_0, e_1, c, d)$

Table 2.5 – BIKE-2 IND-CPA/IND-CCA key encapsulation flow.

The decapsulation algorithm is shown in Table 2.6. As defined in Niederreiter's cryptosystem, the syndrome s is computed by multiplying the ciphertext c with the inverse of the scrambling matrix $(H_0^{-1})^{-1} = H_0$ (Step 1), and decoded to obtain the error vector (e'_0, e'_1) in Step 2. The computed syndrome s is

$$\begin{aligned} s &= H_0 c \\ &= H_0(e_0 + e_1 F_1) \\ &= H_0 e_0 + H_0 e_1 H_1 H_0^{-1} \\ &= H_0 e_0 + H_1 e_1 \\ &= He^T \end{aligned}$$

and therefore it is indeed the syndrome corresponding to the error e and as such it is a valid input for the decoder. Note that both the ciphertext and syndrome are regarded as column vectors.

The CPA version continues in a straightforward manner, as in BIKE-1. Upon obtaining (e'_0, e'_1)

it is checked if the combined weight of e'_0 and e'_1 is correct and either the key *ss* is computed or failure is reported.

The additional plaintext checking in the CCA version is done by first computing the new seed as the xor of the received d value and the hash of the obtained error (Step 3), and then comparing it with the original seed. If all the conditions are satisfied the shared key is computed as $ss = \mathbf{K}(e'_0, e'_1, c, d)$, otherwise the algorithm outputs a shared key created based on (σ_0, σ_1).

	BIKE-2 IND-CPA	BIKE-2 IND-CCA
IN	Ciphertext c (h_0, h_1)	Ciphertext (c, d) and seed z $(h_0, h_1, \sigma_0, \sigma_1)$
1.	Compute the syndrome $s = ch_0$	
2.	$(e'_o, e'_1) \leftarrow \text{decode}(s, h_0, h_1)$	
3.	-	$z' = d \oplus \mathbf{K}(e'_0, e'_1)$
4.	if $wt(e_0) + wt(e_1) \neq t$ or decoding failed or $z \neq z'$ then	
SS	return $\perp$	$ss = \mathbf{K}(\sigma_0, \sigma_1, c, d)$
5.	else	
SS	$ss = \mathbf{K}(e'_0, e'_1)$	$ss = \mathbf{K}(e'_0, e'_1, c, d)$

Table 2.6 – BIKE-2 IND-CPA/IND-CCA key decapsulation flow.

2.1.9 BIKE-3

The third variant of BIKE is different from the previous two variants. It is based on the Ouroboros cryptographic protocol [40]. It features fast key generation, comparable with BIKE-1, while in terms of bandwidth it can be easily modified such that the public key size is almost halved, comparable with BIKE-2. Because of the differences in the design compared to BIKE-1 and BIKE-2 which are McEliece and Niederreiter based systems, it requires slightly larger parameters to achieve the same level of security. But the main disadvantage of this variant is that it comes with a patent attached to it, and although the patent owners declared that they are willing to grant non-exclusive license for the purpose of implementing the standard, it is still not regarded as completely safe to use from a legal point of view. As such, BIKE-3 is not treated with the same level of attention as the other variants of BIKE and in the remaining part of the thesis it is often omitted from our analysis. Nevertheless, it is presented here for the sake of completeness.

The key generation algorithm is shown in Table 2.7. Analogously to BIKE-1, the secret key is a parity-check matrix H represented by two polynomials $h_0, h_1 \in \mathcal{R}$ each of weight $d = \frac{w}{2}$ such that $H^T = \begin{pmatrix} H_0 \\ H_1 \end{pmatrix}$. The public key (f_0, f_1) consists of two parts. The second part, f_1, is assigned the value of a randomly generated invertible polynomial $g \in \mathcal{R}$. The value of the first part of the key is then computed as $f_0 = h_1 + gh_0$, which can be considered as a random syndrome of the code defined by H. Note that here it is possible to reduce the size of the public key by replacing f_1 with a random seed which can be used in a pseudo-random function to deterministically derive the actual value of f_1.

	BIKE-3 IND-CPA	BIKE-3 IND-CCA
1.	$h_0, h_1 \xleftarrow{\$} \mathscr{R}$ both of odd Hamming weight $wt(h_0) = wt(h_1) = w/2$	
2.	-	$\sigma_0, \sigma_1, \sigma_2 \xleftarrow{\$} \mathscr{R}$
3.	$g \xleftarrow{\$} \mathscr{R}$ of odd Hamming weight	
3.	$(f_0, f_1) = (h_1 + gh_0, g)$	
PK	(f_0, f_1)	(f_0, f_1)
SK	(h_0, h_1)	$(h_0, h_1, \sigma_0, \sigma_1, \sigma_2)$

Table 2.7 – BIKE-3 IND-CPA/IND-CCA key generation flow.

Table 2.8 outlines the encapsulation flow. Here, three error vectors are sampled from $\mathscr{R}$ such that the combined weight of e_0 and e_1 is t, as before, and the weight of the additional error vector e is $\frac{t}{2}$. The ciphertext in Step 3 is computed in two parts:

$$c_0 = e + e_1 f_0 \ \text{ and } \ c_1 = e_0 + e_1 f_1.$$

If the matrix defined by $\begin{pmatrix} 1 & F_1 \end{pmatrix}$ is considered as a parity-check matrix of some code, then c_1 is a random syndrome in this code. Even though F_1 is public, it defines a general linear code, not a moderate density code, and therefore, it cannot be used to efficiently decode c_1 and recover the secret error vector.

In the IND-CCA version of the key encapsulation, a seed z is introduced to the scheme, from which the errors are generated with a hash function $\mathbf{H}$ to enable plaintext checking required for the FO transform. Moreover, compared to the CPA version where the shared secret key is derived from the error vectors only, in the CCA case, the shared secret depends also on the ciphertext.

	BIKE-3 IND-CPA	BIKE-3 IND-CCA
IN	Public key (f_0, f_1)	
1.	-	$z \xleftarrow{\$} \{0,1\}^{l_K}$
2.	$(e, e_0, e_1) \xleftarrow{\$} \mathscr{R}^3$	$(e, e_0, e_1) = \mathbf{H}(z)$
	where $wt(e_0) + wt(e_1) = t$ and $wt(e) = \frac{t}{2}$	
3.	$(c_0, c_1) = (e + e_1 f_0, e_0 + e_1 f_1)$	
4.	-	$d = \mathbf{K}(e_0, e_1, e) \oplus z$
CT	$c = (c_0, c_1)$	$(c, d) = ((c_0, c_1), d)$
SS	$ss = \mathbf{K}(e_0, e_1, e)$	$ss = \mathbf{K}(e_0, e_1, e, c, d)$

Table 2.8 – BIKE-3 IND-CPA/IND-CCA key encapsulation flow.

The key decapsulation algorithm is presented in Table 2.9. Upon receiving the ciphertext, the syndrome s is computed with:

$$\begin{aligned} s &= c_0 + c_1 H_0 \\ &= e + e_1 F_0 + (e_0 + e_1 G) H_0 \\ &= e + e_1 G H_0 + e_1 H_1 + e_0 H_0 + e_1 G H_0 \\ &= e + e_0 H_0 + e_1 H_1 \\ &= e + \begin{pmatrix} e_0 & e_1 \end{pmatrix} \begin{pmatrix} H_0 \\ H_1 \end{pmatrix} \\ &= e + \begin{pmatrix} e_0 & e_1 \end{pmatrix} H^T. \end{aligned}$$

The computed syndrome s is a *noisy* syndrome of the code defined by H corresponding to the error (e_0, e_1) with additional noise introduced by vector e. Noisy syndrome decoding is a variation of the syndrome decoding problem, and for MDPC codes, the bit flipping decoder is only marginally affected by the added noise. However, the block size r of the code still needs to be slightly increased to achieve the DFR equivalent to standard decoding. Further details about decoding with noisy syndrome can be found in the BIKE specification [9].

Once the errors are recovered, it is checked if their weight is correct, and if it is then the shared secret key is derived in the same way as in encapsulation. If the recovered error is not of the required weight, the CPA protocol reports a failure, while the CCA protocol generates a random shared secret with σ values which are part of the private key. The CCA version includes one more condition, namely, the plaintext checking as defined by the FO transform. If all the conditions are satisfied the CCA algorithm outputs the derived shared secret.

	BIKE-3 IND-CPA	BIKE-3 IND-CCA
IN	Ciphertext $c = (c_0, c_1)$ (h_0, h_1)	Ciphertext $(c, d) = ((c_0, c_1), d)$ $(h_0, h_1, \sigma_0, \sigma_1)$
1.	Compute the syndrome $s = c_0 + c_1 h_0$	
2.	$(e'_o, e'_1, e') \leftarrow \text{decode}(s, h_0, h_1)$	
3.	-	$z' = d \oplus \mathbf{K}(e'_0, e'_1, e')$
4.	if $wt(e'_0) + wt(e'_1) \neq t$ or $wt(e) \neq \frac{t}{2}$ or decoding failed or $z \neq z'$ then	
SS	return $\perp$	$k = \mathbf{K}(\sigma_0, \sigma_1, \sigma_2, c, d)$
5.	else	
SS	$ss = \mathbf{K}(e'_0, e'_1)$	$ss = \mathbf{K}(e'_0, e'_1, e', c, d)$

Table 2.9 – BIKE-3 IND-CPA/IND-CCA key decapsulation flow.

2.2 Legendre PRF

The Legendre PRF is a pseudorandom function based on the Legendre symbol. The use of Legendre symbols in pseudorandom functions was initially proposed by Damgård [29]. Although very simple and effective, the Legendre PRF has not gained significant traction until recently, the main reason being the existence of faster alternatives. However, in a recent work on cryptographic primitives required for multi-party computation (MPC) [28], the authors conclude that the Legendre PRF is a suitable candidate for pseudorandom generator in MPC settings. This generated new interest in the Legendre PRF and its practical applications, especially in the blockchain community. In the following section we give basic definitions required for building the Legendre PRF and analyzing its security.

2.2.1 Legendre symbol

Let p be an odd prime and denote by $\mathbb{F}_p$ the field of cardinality p. The multiplicative group of $\mathbb{F}_p$ is denoted by $\mathbb{F}_p^*$. We consider the elements of $\mathbb{F}_p$ as integers modulo p.

Definition 8 (Legendre symbol)**.** We define the Legendre symbol by setting

$$\left(\frac{a}{p}\right) = \begin{cases} 1 & \text{if } a \in \mathbb{F}_p^* \text{ is a square mod } p \\ 0 & \text{if } a = 0 \text{ mod p} \\ -1 & \text{if } a \in \mathbb{F}_p^* \text{ is not a square mod } p. \end{cases}$$

Note that some authors prefer to set $\left(\frac{0}{p}\right) = 1$, which makes the Legendre symbol a binary function but breaks the multiplicative property (cf. below).

By definition, the Legendre symbol of a non-zero element of a field tells us if the element is a quadratic residue or a quadratic nonresidue. By Euler's criterion, we can determine if an integer a coprime to p is a quadratic residue modulo p by computing

$$a^{\frac{p-1}{2}} \equiv \begin{cases} 1 \pmod{p} & \text{if there is an integer } b \text{ such that } a \equiv b^2 \pmod{p} \\ -1 \pmod{p} & \text{if there is no such integer.} \end{cases}$$

Therefore, a straightforward way to find the Legendre symbol of $a \in \mathbb{F}_p$ is to compute $\left(\frac{a}{p}\right) = a^{\frac{p-1}{2}}$ which requires $O(\log p)$ arithmetic operations in the field, or $O(\log^3 p)$ bit operations when using naive arithmetic. Note that there are more efficient methods for evaluating the Legendre symbol – one method is described later in this section, while an even faster (asymptotically) algorithm is proposed in [41].

We list here several useful properties of the Legendre symbol. Firstly, it follows directly from the definition that it is a multiplicative function, namely:

$$\left(\frac{ab}{p}\right) = \left(\frac{a}{p}\right)\left(\frac{b}{p}\right).$$

In other words, the product of two residues or two nonresidues is a residue, while the product of a residue and a nonresidue is a nonresidue.

The law of quadratic reciprocity gives us the next three properties. Firstly, we have that

$$\left(\frac{-1}{p}\right) = (-1)^{\frac{p-1}{2}} = \begin{cases} 1 & \text{if } p \equiv 1 \bmod 4 \\ -1 & \text{if } p \equiv 3 \bmod 4, \end{cases}$$

because if $p \equiv 1 \bmod 4$ then $(p-1)/2$ is even and if $p \equiv 3 \bmod 4$ then it is odd. Similarly, we have that the Legendre symbol of 2 is:

$$\left(\frac{2}{p}\right) = (-1)^{\frac{p^2-1}{8}} = \begin{cases} 1 & \text{if } p \equiv 1 \text{ or } 7 \bmod 8 \\ -1 & \text{if } p \equiv 3 \text{ or } 5 \bmod 8. \end{cases}$$

Let q be a prime number different from p, then the quadratic reciprocity law states that

$$\left(\frac{q}{p}\right)\left(\frac{p}{q}\right) = (-1)^{\frac{p-1}{2}\frac{q-1}{2}}.$$

The final property is a useful tool for evaluating the Legendre symbol – it allows us to flip $\left(\frac{q}{p}\right)$ and replace it with $\pm\left(\frac{p}{q}\right)$, reduce p modulo q, and compute the symbol with smaller entries. These steps can be repeated until we get entries as small as desired. However, the problem lies is the fact that $p \bmod q$ may be a composite number meaning that it needs to be factored if we were to continue with the procedure.

The Jacobi symbol is a generalization of the Legendre symbol which can be used to simplify the computation of the Legendre symbol.

Definition 9 (Jacobi symbol)**.** Let a be an integer and m an odd positive integer with prime factorization $m = \prod p_i^{t_i}$. The Jacobi symbol $\left(\frac{a}{m}\right)$ is defined in terms of the factorization of m as

$$\left(\frac{a}{m}\right) = \prod \left(\frac{a}{p_i}\right)^{t_i},$$

where $\left(\frac{a}{p_i}\right)$ are Legendre symbols with $\left(\frac{a}{1}\right) = 1$.

It follows directly from the definition that if a and m are not coprime then $\left(\frac{a}{m}\right) = 0$. The Jacobi symbol is a completely multiplicative function, i. e., for m fixed

$$\left(\frac{ab}{m}\right) = \left(\frac{a}{m}\right)\left(\frac{b}{m}\right),$$

and for a fixed a we have that

$$\left(\frac{a}{mn}\right) = \left(\frac{a}{m}\right)\left(\frac{a}{n}\right).$$

Furthermore, when n and m are odd positive coprime integers the quadratic reciprocity law applies (analogously to the Legendre symbol case):

$$\left(\frac{n}{m}\right)\left(\frac{m}{n}\right) = (-1)^{\frac{m-1}{2}\frac{n-1}{2}},$$

with its supplements:

$$\left(\frac{-1}{m}\right) = (-1)^{\frac{m-1}{2}},$$
$$\left(\frac{2}{m}\right) = (-1)^{\frac{m^2-1}{8}}.$$

The stated properties allow efficient computation of the Jacobi symbol, but also of its special case – the Legendre symbol. The algorithm to compute $\left(\frac{a}{m}\right)$ can be described in the following way:

1. Reduce a modulo m.
2. Extract the 2's from the factorization of a by using the multiplicative property of the symbol and the formula for $\left(\frac{2}{m}\right)$.
3. If $a = 1$ or $\gcd(a, m) \neq 1$ we finish and output the result 1 or 0, respectively. Otherwise, a and m are odd positive coprime integers with $a < m$, so we switch their places and return to the first step.

This algorithm is analogous to Euclid's algorithm for computing the greatest common divisor of two numbers and therefore it has a complexity of $O(\log^2 m)$ bit operations – a $\log p$ improvement from the naive Legendre symbol computation algorithm.

2.2.2 Legendre pseudorandom function

In this section we define Legendre sequences and a pseudorandom function based on the properties of the these sequences together with several hard problems associated with the Legendre PRF.

Definition 10 (Legendre sequence)**.** We define a Legendre sequence with starting point a and length L to be the sequence of Legendre symbols evaluated at L consecutive elements starting from a. We denote it with $\{a\}_L$.

$$\{a\}_L := \left(\frac{a}{p}\right), \left(\frac{a+1}{p}\right), \left(\frac{a+2}{p}\right), \ldots, \left(\frac{a+L-1}{p}\right).$$

Every sequence $\{a\}_L$ is fully determined by the starting value a. However, the statement is not true in the opposite direction. Namely, a sequence of L symbols does not always uniquely determine a starting point – depending on L it may or may not have a unique starting point. For example, when $L = 1$ a given sequence provides only information on quadratic residuosity of its starting point. Moreover, we know that the number of quadratic residues and nonresidues in $\mathbb{F}_p^*$ is equal, meaning that half of the potential starting points of a sequence, elements $a \in \mathbb{F}_p$, give a Legendre symbol 1 and the other half give -1 (ignoring the case $a = 0$). Therefore, the $L = 1$ sequences are well distributed. Similar properties can be attributed to sets of sequences with larger L-values. Following a theorem of Davenport around one in 2^L elements of $\mathbb{F}_p$ is a starting point of a given sequence of length L.

Theorem 1 (Davenport, 1933 [42])**.** *Let S be a finite sequence of $+1$ and -1 values of length L. Then the number of elements of $\mathbb{F}_p$ whose sequence is equal to S satisfies*

$$\#\big\{a \in \mathbb{F}_p \ : \ \{a\}_L = S\big\} = \frac{p}{2^L} + O(p^{\varepsilon})$$

where $0 < \varepsilon < 1$ is a constant depending only on L.

Intuitively, if we want L such that $\{a\}_L$ uniquely defines a, i. e., that the following holds

$$\{a\}_L = \{b\}_L \ \text{ if and only if } \ a = b, \tag{2.1}$$

then L should be of order $\Omega(\log p)$. However, the only provable bound so far comes from the Weil bound [43] which lower bounds L by an exponential function in p, i. e., $L = O(\sqrt{p}\log p)$. Despite this, there is an indication that on average over all sequences S of length L, there are $\frac{p}{2^L} + O(1)$ elements $a \in \mathbb{F}_p$ whose Legendre sequences are $\{a\}_L = S$. In other words for a random sequence S and a random element $a \in \mathbb{F}_p$ we have $\{a\}_L = S$ with probability $\frac{1}{2^L}$. This observation is based both on our computational results and other statistical data on the distribution of Legendre sequences [29].

Definition 11 (Complete Legendre sequence)**.** We define the complete Legendre sequence to be the sequence of p Legendre symbols of all ordered elements of $\mathbb{F}_p$ up to rotation, i.e. $\{0\}_p$ where the tail connects to the head

$$\left(\frac{0}{p}\right), \left(\frac{1}{p}\right), \left(\frac{2}{p}\right), \ldots, \left(\frac{p-1}{p}\right) \Big/ \sim .$$

The Legendre sequences $\{a\}_L$ for all $a \in \mathbb{F}_p$ and $L \geq 0$ are subsequences of the complete Legendre sequence.

Legendre PRF

Pseudorandom functions are deterministic functions of a key and an input that produce an output which is indistinguishable from an output of a truly random function with the same codomain.

Definition 12 (Pseudorandom functions)**.** A pseudorandom function family $\{\mathscr{F}_k\}_k$ is a set of functions with the same domain and codomain indexed by the set of keys k such that a function $\mathscr{F}_k$ chosen randomly over the set of k-values cannot be distinguished from a random function with the same codomain.

Definition 13 (Legendre PRF)**.** The Legendre pseudorandom functions are functions

$$\mathscr{F}_k : \mathbb{F}_p \to \{-1,0,1\}$$

indexed by $k \in \mathbb{F}_p$ and defined as

$$\mathscr{F}_k(a) = \left(\frac{k+a}{p}\right).$$

There are three main problems conjectured to be hard and on which the security of the Legendre PRF is based.

Definition 14 (Shifted Legendre Symbol Problem - SLSP)**.** Let k be a uniformly random value in $\mathbb{F}_p$. Given access to an oracle $\mathscr{F}$ that on input $a \in \mathbb{F}_p$ computes $\mathscr{F}(a) = \left(\frac{k+a}{p}\right)$, find k.

Definition 15 (Decisional Shifted Legendre Symbol Problem - DSLSP)**.** Let k be a uniformly random value in $\mathbb{F}_p$. Let $\mathscr{F}_0$ be an oracle that on input $a \in \mathbb{F}_p$ computes $\mathscr{F}_0(a) = \left(\frac{k+a}{p}\right)$, and let $\mathscr{F}_1$ be an oracle that on input a outputs a random value in $\{-1,+1\}$. Given access to $\mathscr{F}_b$ where b is an unknown random bit, find b.

Definition 16 (Next Symbol Problem - NSP)**.** Given a Legendre sequence $\{a\}_M$ of $M = \mathrm{polylog}(p)$ symbols, find $\left(\frac{a+M}{p}\right)$, or equivalently find $\{a\}_{M+1}$.

It is easy to see that finding the secret shift of a Legendre symbol (SLSP) is at least as hard as finding the next symbol given a sequence (NSP) or distinguishing an output of the Legendre PRF from a truly random function (DSLSP). Moreover, following the result of Yao [44] on general pseudorandom functions, predicting the next bit of a pseudorandom function is equivalently hard as distinguishing it from a truly random one. Therefore, we have that, under polynomial time reductions, NSP = DSLSP ≤ SLSP.

BIKE Part I

3 On constant-time QC-MDPC decoding with negligible failure rate

Bit Flipping Key Encapsulation (BIKE) is a code-based Key Encapsulation Mechanism (KEM) that uses Quasi-Cyclic Moderate Density Parity Check (QC-MDPC) codes. It is one of the Round-2 candidates of the NIST PQC Standardization Project [15]. The BIKE submission includes three variants (BIKE-1, BIKE-2, and BIKE-3) with three security levels for each one. In this chapter, we focus mainly on BIKE-1, at its Category 1 (as defined by NIST) security level.

The decapsulation algorithm of BIKE invokes an algorithm that is called a decoder. The decoder is an algorithm that, given prescribed inputs, outputs an error vector that can be used to extract a message (or the message itself). There are various decoding algorithms and different choices yield different efficiency and Decoding Failure Rate (DFR) properties.

QC-MDPC decoding algorithms. We briefly describe the evolution of several Quasi-Cyclic Moderate-Density Parity-Check (QC-MDPC) decoding algorithms. All of them are derived from the Bit-Flipping algorithm presented in Section 2.1. The Round-1 submission of BIKE describes the “One-Round” decoder. This decoder is indeed implemented in the accompanying reference code [9]. The designers of the constant-time Additional implementation [45] of BIKE Round-1 chose to use a different decoder named “Black-Gray”[1], with rationale as explained in [46]. The study presented in [1] explores two additional variants of the Bit-Flipping decoder: a) a parallel algorithm similar to that of [21], which first calculates some thresholds for flipping bits, and then flips the bits in all of the relevant positions, in parallel. We call this decoder the “Simple-Parallel” decoder; b) a “Step-by-Step” decoder (an enhancement of the “in-place” decoder described in [23]). It recalculates the threshold every time that a bit is flipped.

The Round-2 submission of BIKE uses the One-Round decoder (of Round-1) and a new variant of the Simple-Parallel decoder called BackFlip decoder. The latter introduces a new trial-and-error technique called Time-To-Live (TTL). It is positioned as a derivative of the decoders in [1]. All of these decoders have some nonzero probability to fail in decoding a valid input. The average of the failure probability over all the possible inputs (keys and messages) is

---

[1] This decoder appears in the pre-Round-1 submission “CAKE” (the BIKE-1 ancestor). It is due to N. Sendrier and R. Misoczki. The decoder was adapted to use the improved thresholds published in [9].

called DFR. The KEMs of Round-1 BIKE were designed to offer IND-CPA security and to be used only with ephemeral keys. They had an estimated approximate DFR of 10^{-7}, which is apparently tolerable in real systems, according to [9]. As a result, they enjoyed acceptable bandwidth and performance. Note that the DFR depends on the proportion of the number of error bits in a codeword and the bit length of the codeword. Therefore, decreasing the DFR can be achieved by increasing the code size while leaving the number of error bits the same, which presents a trade-off between the bandwidth of the system and its DFR. The Round-2 submission presented new variants of BIKE Key Encapsulation Mechanism (KEM) that provide IND-CCA security. Such KEM can be used with static keys. The IND-CCA BIKE is based on three changes over the IND-CPA version: a) Fujisaki-Okamoto $FO^{\not\perp}$ transformation (explained in Section 2.1.5) applied to the key generation, encapsulation, and decapsulation of the original IND-CPA flows (see [9, Section 6.2.1]); b) adjusted parameters sizes; c) invoking the BackFlip decoder in the decapsulation algorithm. Note that the requirements for the DFR are different in the CPA and CCA case. Namely, CPA secure schemes are meant to be used only with ephemeral keys and because of that the attacks on BIKE which exploit decoding failures have little effect, in a sense that they need to observe a considerable number of failures (certainly more than one) to recover the key. Therefore, in the CPA case the parameters are chosen such that the DFR is tolerable in the context of system failures, i. e., the impact of failures on the functionality of a network is insignificant. On the other hand, CCA secure schemes are designed for static key use case where a single key is used many times. In this context, decoding failures are important from the security standpoint and DFR of the scheme is required to be negligible. Changes (b) and (c) from above are introduced specifically to decrease the DFR to an acceptable level.

In this chapter, the following contributions are detailed:

- We define BackFlip^{+} decoder as the variant of BackFlip that operates with a fixed number of iterations X_{BF} (for some X_{BF}). We also define the Black-Gray decoder that runs with a given number of iterations X_{BG} (for some X_{BG}). Subsequently, we analyze the DFR of these decoders as a function of X_{BF} and X_{BG} and the block size (which determines the communication bandwidth of the KEM). The analysis results in a new set of parameters where BackFlip^{+} with $X_{BF} = 8, 9, 10, 11, 12$ and the Black-Gray decoder with $X_{BG} = 3, 4, 5$ have an estimated average DFR of 2^{-128}. This offers multiple IND-CCA proper BIKE instantiation options.
- We build an optimized constant-time implementation of the new BIKE CCA flows together with a constant-time implementation of the decoders. This facilitates a performance comparison between the BackFlip^{+} and the Black-Gray decoders. All of our performance numbers are based *only* on constant-time implementations. The comparison leads to interesting results. The BackFlip^{+} decoder has a better DFR than the Black-Gray decoder if both of them are allowed to have very large (practically unlimited) X_{BF} and X_{BG} values. These values do not lead to practical performance. However, for small X_{BF} and X_{BG} values that make the performance practical and DFR acceptable,

Table 3.1 – BIKE-1 block size r (in bits) at security level 1, for which the Black-Gray decoder achieves a target DFR with a specified number of iterations, and the decapsulation performance (in cycles; the precise details of the platform are provided in Section 3.5). A DFR of 2^{-128} is required for the IND-CCA KEM. The IND-CPA used with ephemeral keys can settle for a higher DFR.

DFR		*3 iterations*	*4 iterations*	*5 iterations*
$2^{-23} \approx 10^{-7}$	r	10259	10163	10141
	cycles	3.50M	4.52M	5.53M
$2^{-30} \approx 10^{-9}$	r	10427	10331	10301
	cycles	3.52M	4.56M	5.63M
$2^{-40} \approx 10^{-12}$	r	10667	10589	10501
	cycles	3.55M	4.63M	5.69M
2^{-64}	r	11261	11069	11003
	cycles	3.76M	4.81M	5.96M
2^{-128}	r	12781	12437	12373
	cycles	4.06M	5.22M	6.47M

the Black-Gray decoder is faster (and therefore preferable).

- The BIKE CCA flows require higher bandwidth and more computations compared to the original CPA flows, but the differences as measured on x86-64 architectures are not very significant. Table 3.1 summarizes the trade-off between the BIKE-1 block size (r), the estimated DFR and the performance of BIKE-1 decapsulation (with IND-CCA flows) using the Black-Gray decoder. It provides several instantiation/implementation choices. For example, with $X_{BG} = 4$ iterations and targeting a DFR of 2^{-64} (with $r = 11069$ bits) the decapsulation with the Black-Gray decoder consumes 4.81M cycles. With a slightly higher $r = 11261$ the decoder can be set to have only $X_{BG} = 3$ iterations and the decapsulation consumes 3.76M cycles.

- The $FO^{\not\perp}$ transformation from QC-MDPC McEliece Public Key Encryption (PKE) to BIKE-1 IND-CCA relies on the assumption that the underlying PKE is δ-correct [38] with $\delta = 2^{-128}$. The relation between this assumption and the (average) DFR that is used in [9] is not yet addressed. We identify this gap and illustrate some of the remaining challenges.

This chapter is organized as follows. Section 3.1 offers background, notation and surveys some QC-MDPC decoders. In Section 3.2 we define and clarify subtle differences between schemes using idealized primitives and concrete instantiations of the schemes. In Section 3.3 we present the challenges and the techniques that we used for building a constant-time implementation of IND-CCA BIKE. We explain the method used for estimating the DFR in Section 3.4. Section 3.5 reports our results for the DFR and block size study, and also the performance measurements of the constant-time implementations. The gap between the estimated DFR and the δ-correctness needed for IND-CCA BIKE is discussed in Section 3.6.

Section 3.7 concludes the chapter with several concrete proposals and open questions.

3.1 Preliminaries

Recall the notation defined in Chapter 2.1. The polynomial ring $\mathbb{F}_2[x]/(x^r-1)$ is denoted by $\mathcal{R}$. For every element $v \in \mathcal{R}$ its Hamming weight is denoted by $wt(v)$. The length of a vector w is denoted by $|w|$. Polynomials in $\mathcal{R}$ are viewed interchangeably also as square circulant matrices in $\mathbb{F}_2^{r\times r}$. For a matrix $H \in \mathbb{F}_2^{r\times r}$ let h_j denote its j-th column written as a row vector. We denote null values and protocol failures by $\perp$. Uniform random sampling from a set W is denoted by $w \xleftarrow{\$} W$. For an algorithm A, we denote its output by $out = A()$ if A is deterministic, and by $out \leftarrow A()$ otherwise.

3.1.1 BIKE-1

The computations of BIKE-1-(CPA/CCA) are executed over $\mathcal{R}$, where r is a given parameter. Let w and t be the weights of (h_0, h_1) in the secret key $h = (h_0, h_1, \sigma_0, \sigma_1)$ and the error vector $e = (e_0, e_1)$, respectively. Denote the public key, ciphertext, and shared secret by $f = (f_0, f_1)$, $c = (c_0, c_1)$, and k, respectively. As in [9], we use **H**, **K** to denote hash functions. BIKE-1 parameters for Round 2 of the NIST Post-Quantum Cryptography Standardization project at security level 1 are the following: for BIKE-1-CPA, $r = 10163$, $|f| = |c| = 20326$ and for BIKE-1-CCA, $r = 11779$, $|f| = |c| = 23558$. In both cases, $|ss| = 256$, $w = 142$, $d = w/2 = 71$ and $t = 134$. The key generation, encapsulation and decapsulation flows are presented in Section 2.1.7.

3.1.2 The IND-CCA transformation

Round-2 BIKE submission [9] uses the $FO^{\not\perp}$ transformation described in [38] which is based on Fujisaki-Okamoto transformation [37], to convert the QC-MDPC McEliece public-key encryption scheme into an IND-CCA secure key encapsulation mechanism BIKE-1-CCA. The submission claims that the proof results from [38][Theorems 3.1 and 3.4[2]]. These theorems use the term δ-correct PKE. For a finite message space M, a PKE is called δ-correct when the expected value[3]

$$\mathbb{E}\left[\max_{m\in M} Pr\left[Decrypt(sk, ct) \neq m \mid ct \leftarrow Encrypt(pk, m)\right]\right] \leq \delta \tag{3.1}$$

and a KEM is δ-correct if

$$Pr\left[Decaps(sk, ct) \neq ss \mid (sk, pk) \leftarrow Gen(), (ct, ss) \leftarrow Encaps(pk)\right] \leq \delta \tag{3.2}$$

---

[2] Theorems 3.1 and 3.4 appear only in the ePrint version [47] of [38]. In [38] they appear as Theorems 1 and 4, respectively.

[3] In BIKE-1, the secret key (sk) and public key (pk) are h and f, respectively.

3.1.3 QC-MDPC Decoders

The QC-MDPC decoders discussed in this chapter are variants of the Bit Flipping decoder [21] presented in Algorithm 2. They receive a parity check matrix $H \in \mathbb{F}_2^{r \times n}$ (with column weight d) and a vector $c = mf + e$ as input[4]. Here, $c, mf, e \in \mathbb{F}_2^n$, where the codeword mf is a product of the message m and the public key f (thus, $(mf)H^T = 0$) and e is an error vector with small weight. The algorithm calculates the syndrome $s = eH^T$ and subsequently extracts e' from s. The goal of the Bit Flipping algorithm is to find e' such that $e' = e$.

Algorithm 2 consists of four steps: I) calculate some static/dynamic threshold (τ) based on the syndrome (s) and the error (e) weight; II) compute the number of unsatisfied parity check equations (upc_i) for a given column $i \in \{0, \dots, n-1\}$; III) flip the error bits in the positions where there are more unsatisfied parity-check equations than the calculated threshold; IV) recompute the syndrome. We refer to Algorithm 2 as the Simple-Parallel decoder. The Step-By-Step decoder inserts Steps 4, 9 into the "**for**" loop (Step 5), i. e., it recalculates the threshold and the syndrome for every bit. The One-Round decoder starts with one iteration of the Simple-Parallel decoder, and then switches to the Step-by-Step decoder mode of operation.

Algorithm 2 e=BitFlipping(c, H)

```
    Input: Parity-check matrix $H \in \mathbb{F}_2^{r \times n}$, $c \in \mathbb{F}_2^n$, maxIter  (maximal # of iterations)
    Output: The error $e \in \mathbb{F}_2^n$
    Exception: "decoding failure" return $\perp$
 1: procedure BitFlipping($c$, $H$)
 2:     $s = Hc^T, e = 0$, itr = 0
 3:     while ($wt(s) > 0$) and (itr < maxIter) do
 4:         $\tau$ = computeThreshold($s, e$)                ▷ Step I
 5:         for $i$ in $0 \dots n-1$ do
 6:             Compute $upc_i$                          ▷ Step II
 7:             if $upc_i > \tau$ then                    ▷ Step III
 8:                 $e[i] = e[i] \oplus 1$
 9:         $s = H(c^T + e^T)$                          ▷ Step IV
10:         itr = itr + 1
11:     if ($wt(s) > 0$) then
12:         return $\perp$
13:     else
14:         return $e$
```

The Black-Gray decoder (in the additional code [45]) and the BackFlip decoder [9] use a more complex approach. Similar to the Simple-Parallel decoder, they operate on the error bits in parallel. However, they add a step that unflips the error bits according to some estimation.

Algorithm 3 illustrates the Black-Gray decoder. This Black-Gray variant runs in a fixed number of iterations (X_{BG}). The algorithm involves three main steps: 1) Perform one iteration of the

---

[4] In BIKE-1, $n = 2r$, the parity-check matrix H is formed by the two circulant blocks (h_0, h_1); the vectors c, e, and f are defined as $c = (c_0, c_1)$, $e = (e_0, e_1)$, and $mf = (mf_0, mf_1)$.

Simple-Parallel decoder (Algorithm 2) and define some bit position candidates that should be reconsidered (i. e., bits that were mistakenly flipped). Then, split them into two lists (black, gray); 2) Reevaluate the bits in the black list, and flip them according to the evaluation; 3) Reevaluate the gray error bits and unflip those that meet a certain threshold. Then, recalculate the syndrome.

Algorithm 3 e=Black-Gray(c, H)

```
Input: Parity-check matrix $H \in \mathbb{F}_2^{r \times n}$, $c \in \mathbb{F}_2^n$, $X_{BG}$ (maximal # of iterations)
Output: The error $e \in \mathbb{F}_2^n$
Exception: "decoding failure" return $\perp$
procedure BLACK-GRAY($c$, $H$)
    $s = Hc^T$, $e = 0$, $\delta = 4$
    $B = \emptyset$, $G = \emptyset$                              ▷ Black and Gray position sets

    for $i$ in $1 \ldots X_{BG}$ do
        $\tau$ = computeThreshold($s$)
        $upc[n-1:0]$ = computeUPC($s, H$)
        for $i$ in $0 \ldots n-1$ do                              ▷ Step I
            if $upc[i] \geq \tau$ then
                $e[i] = e[i] \oplus 1$                            ▷ Flip an error bit
                $B = B \cup i$                                    ▷ Update the Black set
            else if $upc[i] > \tau - \delta$ then
                $G = G \cup i$                                    ▷ Update the Gray set
        $s = H(c^T + e^T)$                                        ▷ Update the syndrome

        $upc[n-1:0]$ = computeUPC($s, H$)                         ▷ Step II
        for $b \in B$ do
            if $upc[b] > ((d+1)/2)$ then
                $e[b] = e[b] \oplus 1$                            ▷ Flip an error bit
        $s = H(c^T + e^T)$                                        ▷ Update the syndrome

        $upc[n-1:0]$ = computeUPC($s, H$)                         ▷ Step III
        for $g \in G$ do
            if $upc[g] > ((d+1)/2)$ then
                $e[g] = e[g] \oplus 1$                            ▷ Flip an error bit
        $s = H(c^T + e^T)$                                        ▷ Update the syndrome

    if $wt(s) > 0$ then
        return $\perp$
    else
        return $e$
```

The BackFlip decoder shown in Algorithm 4 has the following steps: 1) Compute the threshold based on the current syndrome and error values, and the number of unsatisfied parity-check equations **upc**. Flip the error bits which correspond to the **upc** values greater or equal to the threshold and for each flipped bit compute the time-to-live value **ttl**. 2) Check the **ttl** values of all the bits and unflip those which reached the corresponding value.

Algorithm 4 e=BackFlip(c, H)

```
    Input: Parity-check matrix $H \in \mathbb{F}_2^{r\times n}$, $c \in \mathbb{F}_2^n$, $maxIter$ (maximal # of iterations)
    Output: The error $e \in \mathbb{F}_2^n$
    Exception: "decoding failure" return $\perp$
 1: procedure BackFlip($c$, $H$)
 2:     $s = Hc^T$, $e = 0$, $itr = 0$
 3:     $time = 1$, $ttl[n-1:0] = 0$
 4:     while ($wt(s) > 0$) and ($itr < maxIter$) do
 5:         $\tau$ = computeThreshold($s, e$)
 6:         $upc[n-1:0]$ = computeUPC($s, H$)
 7:         for $i$ in $0 \ldots n-1$ do                                ▷ Step I
 8:             if $upc[i] \geq \tau$ then
 9:                 $e[i] = e[i] \oplus 1$                          ▷ Flip an error bit
10:                 $ttl[i] = time +$ computeTTL($upc[i], \tau$)
11:         for $i$ in $0 \ldots n-1$ do                                ▷ Step II
12:             if $ttl[i] = time$ then
13:                 $e[i] = e[i] \oplus 1$                          ▷ Flip an error bit
14:                 $ttl[i] = 0$
15:         $s = H(c^T + e^T)$                                  ▷ Update the syndrome
16:         $itr = itr + 1$
17:         $time = time + 1$
18:     if $wt(s) > 0$ then
19:         return $\perp$
20:     else
21:         return $e$
```

The BackFlip$^+$ is a variant of BackFlip that uses a fixed number of iterations as explained in the introduction of this chapter. Technically, the difference is that the condition on the weight of s is removed from the **while** loop in line 3. Therefore, the algorithm performs the appropriate number of mock iterations.

Remark 1. The decoders use the term iterations differently. For example, the runtime of one iteration of the Black-Gray decoder is approximately the same as the runtime of three iterations of the Simple-Parallel decoder. The iteration of the One-Round decoder conssists of multiple (not necessarily fixed) "internal" iterations. Comparison of the decoders needs to take this information into account. For example, the performance is determined by the number of iterations times the latency of an iteration, not just by the number of iterations.

3.2 Idealized schemes and concrete instantiations

We discuss some subtleties related to the requirements from a concrete algorithm in order to be acceptable as substitute for an ideal primitive, and the relation to a concrete implementation.

Cryptographic schemes are often analyzed in a framework where some of the components are modeled as ideal primitives. An ideal primitive is a black-box algorithm that performs a defined flow over some (secret) input and communicates the resulting output (and nothing more). A concrete instantiation of the scheme is the result of substituting the ideal primitive(s) with some specific algorithm(s). We require the following property from the instantiation to consider it *acceptable*: it should be *possible* to implement the algorithm without communicating more information than the expected output. From the practical viewpoint, this implies that the algorithm *could be* implemented in constant-time. Note that a specific implementation of an acceptable instantiation of a provably secure scheme can still be insecure (e. g., due to side channel leakage). Special care is needed for algorithms that run with a variable number of steps.

Remark 2. A scheme can have provable security but this does not imply that every instantiation inherits the security properties guaranteed by the proof, or that there even exists an instantiation that inherits them. Furthermore, an insecure instantiation example does not invalidate the proof of the idealized scheme. For example, an idealized KEM can have an IND-CCA secure proof when using a "random oracle" ideal primitive. An instantiation that replaces the random oracle with a non-cryptographic hash function does not inherit the security proof, but it is commonly acceptable to believe that an instantiation with SHA256 does.

Algorithms with a variable number of steps. Let $\mathscr{A}$ be an algorithm that takes a secret input *in* and executes a flow with a variable number of steps/iterations $v(in)$ that depends on *in*. It is not necessarily possible to implement $\mathscr{A}$ in constant-time. In case ("limited") that there is a public parameter b such that $v(in) \leq b$ we can define an equivalent algorithm ($\mathscr{A}^+$) that runs in exactly b iterations: $\mathscr{A}^+$ executes the $v(in)$ iterations of $\mathscr{A}$ and continues with $b - v(in)$ identical mock iterations, each of which requires the same execution time as a regular iteration. With this definition, we can assume that it is possible to implement $\mathscr{A}^+$ in constant-time. Clearly, details must be provided, and such an implementation needs to be worked out. This could be a challenging task.

Suppose that $v(in)$ is unlimited, i. e., there is no (a-priori) parameter b such that $v(in) \leq b$ (we call this case "unlimited"). It is possible to set a constant parameter b^* and an algorithm $\mathscr{A}^+$ with exactly b^* iterations, such that it emits a failure indication if the output is not obtained after exhausting the b^* iterations. It is possible to implement $\mathscr{A}^+$ in constant-time, but it is no longer equivalent to $\mathscr{A}$, due to the nonzero failure probability. Thus, analysis of $\mathscr{A}^+$ needs to include the dependency of the failure probability on b^*, and consider the resulting implications. Practical considerations would seek the smallest b^* for which the upper bound on the failure probability is satisfactory. Obviously, if $\mathscr{A}$ has originally some nonzero failure probability, then $\mathscr{A}^+$ has a larger failure probability.

Suppose that a cryptographic scheme relies on an ideal primitive. In the limited case an instantiation that substitutes $\mathscr{A}$ (or $\mathscr{A}^+$) is acceptable. However, in the unlimited case, substituting the primitive $\mathscr{A}$ with $\mathscr{A}^+$ is more delicate, due to the failure probability that is either introduced or increased. We summarize the unlimited case as follows.

- To consider $\mathscr{A}$ as an acceptable ideal primitive substitute, $v(in)$ needs to be considered as part of $\mathscr{A}$'s output, and the security proof should take this information into consideration. Equivalently, the incremental advantage that an adversary can gain from learning $v(in)$ needs to be included in the adversary advantage of the (original) proof.
- Considering $\mathscr{A}^+$ as an acceptable ideal primitive substitute requires a proof that it has all the properties of the ideal primitive used in the original proof (in particular, the overall failure probability).

Application to BIKE. The IND-CCA security proof of BIKE relies on the existence of a decoder primitive that has a negligible DFR. This is a critical property of a decoder that is used in the proof. The concrete BIKE instantiation substitutes the idealized decoder with the BackFlip decoding algorithm. BackFlip has the required negligible DFR. By its definition, BackFlip runs in a variable number of steps (iterations) that depends on the input and on the secret key (this property is built into the algorithm's definition).

It is possible to use BackFlip in order to define a corresponding BackFlip$^+$ decoder that has a fixed number of steps: a) fix a number of iterations as a parameter X_{BF}; b) follow the original BackFlip flow but always execute X_{BF} iterations in such a way that if the error vector (e) is extracted after $Y < X_{BF}$ iterations, an additional $(X_{BF} - Y)$ identical mock iterations are executed that do not change e; c) after the X_{BF} iterations are exhausted, output a success/failure indication and e on success or a random vector of the expected length otherwise. The difficulty is that the DFR of BackFlip$^+$ is a function of X_{BF} (and r) and it may be larger than the DFR of BackFlip that is critical for the proof.

It is not clear from [9, 1] whether the BackFlip decoder is an example of the limited or the unlimited case, but we choose to assume the limited case, based on the following indications. BackFlip is defined in [9, Algorithm 4] and the definition is followed by the comment: "The algorithm takes as input [...] and, if it stops, returns an error [...] with high probability, the algorithm stops and returns e.". This comment suggests the unlimited case because the "if it stops" part implies that the algorithm might not stop. Here, it is difficult to accept it as a substitution of the ideal primitive, and claim that the IND-CCA security proof applies to this instantiation. In order to make BackFlip an ideal primitive substitute, the number of executed steps needs to be considered as part of its output as well. As an analogy, consider a KEM where the decapsulation has nonzero failure probability. Here, an IND-CCA security proof cannot simply rely on the (original) Fujisaki-Okamoto transformation [37], because this would model an ideal decapsulation with no failures. Instead, it is possible to use the $FO^{\not\perp}$ transformation suggested in [38] that accounts for failures. This is equivalent to saying that the modeled decapsulation outputs a shared key and a success/fail indication. Indeed, this transformation was used in the BIKE CCA proof.

On the other hand, we find locations in [9], that state: "In all variants of BIKE, we will consider the decoding as a black box running in bounded time" (Section 2.4.1) and "In addition, we will bound the running time (as a function of the block size r) and stop with a failure when this bound is exceeded" (Section 1.3). No bounds and dependency on r are provided. However, if we inspect the reference code [9], we can find that the code sets a *maximal* number of BackFlip iterations to 100 (no explanation for this number is provided and this constant is independent of r). Therefore, we may choose to interpret the results of [1, 9] as if the 2^{-128} DFR was obtained from simulations with this $X_{BF} = 100$ bound[5], although this is nowhere stated and the simulation data and the derivation of the DFR are also not provided (the reference code operates with $X_{BF} = 100$). With this, it is reasonable to hope that if we take BackFlip$^+$ and set $X_{BF} = 100$ we would get a DFR below 2^{-128}. This makes BackFlip with $X_{BF} = 100$ an acceptable instantiation of an IND-CCA secure BIKE (for the studied r values).

The challenge with this interpretation is that the instantiation (BackFlip$^+$ and $X_{BF} = 100$) would be impractical from the performance viewpoint. Our paper solves this by showing acceptable instantiations with a much smaller values of X_{BF}. Furthermore, it also shows that there are decoders with a fixed number of iterations that have better performance at the same DFR level.

Implementation. In order to be used in practice, an IND-CCA KEM should have a proper *instantiation* and *also* a constant-time *implementation* that is secure against side-channel attacks (e.g., [23]). Such attacks were demonstrated in the context of QC-MDPC schemes, e. g., the GJS reaction attack [22] and several subsequent attacks [23, 24, 25]. Other reaction attack examples include [48] for LRPC codes and [49] for attacking the repetition code used by the HQC KEM [50]. This problem is significantly aggravated when the KEM is used with static keys (e. g., [51, 23]).

3.3 Implementing BackFlip$^+$ in constant-time

In [46] the authors explained how to implement the Black-Gray decoder to run in constant-time (i.e., to avoid leaking secret information through branching and memory access patterns). In this chapter, we show how to define and implement a constant-time BackFlip$^+$ decoder. To this end, we use the techniques of [46] in addition to some new considerations.

The BackFlip$^+$ decoder differs from the Black-Gray decoder in two aspects: it uses a new mechanism called Time-To-Live (TTL), and it uses new equations for calculating the thresholds. The TTL mechanism is a "smart queue" where the decoder flips back some error bits when it believes that they were mistakenly flipped in previous iterations. It does so unconditionally and it can unflip bits even several iterations after they were flipped. The Black-Gray decoder uses a different type of TTL, where the black and gray lists serve as the "smart queue". However, the error bits are flipped back after only a single iteration, conditionally, through checking certain thresholds. Indeed, as we report below the differences are observed in cases where the

---

[5] See discussion with some extrapolation methodologies in Appendix A.1.1.

Black-Gray decoder failed to decode after some number of iterations and then with high probability fails completely. The BackFlip decoder shows better recovery capabilities in such cases. Implementing the new TTL queue in constant-time relies mostly on common constant-time techniques.

Handling the new threshold function. The BackFlip decoder thresholds are a function of two variables [9, Section 2.4.3]: the syndrome weight $wt(s)$ as in the Black-Gray decoder and the number of error bits that the decoder believes it flipped (denoted $\bar{e}$). Given the syndrome weight and $\bar{e}$ the threshold is computed as the smallest τ such that:

$$\bar{e}\binom{d}{\tau}\pi_1^{\tau}(1-\pi_1)^{d-\tau} \geq (n-\bar{e})\binom{d}{\tau}\pi_0^{\tau}(1-\pi_0)^{d-\tau},$$

where

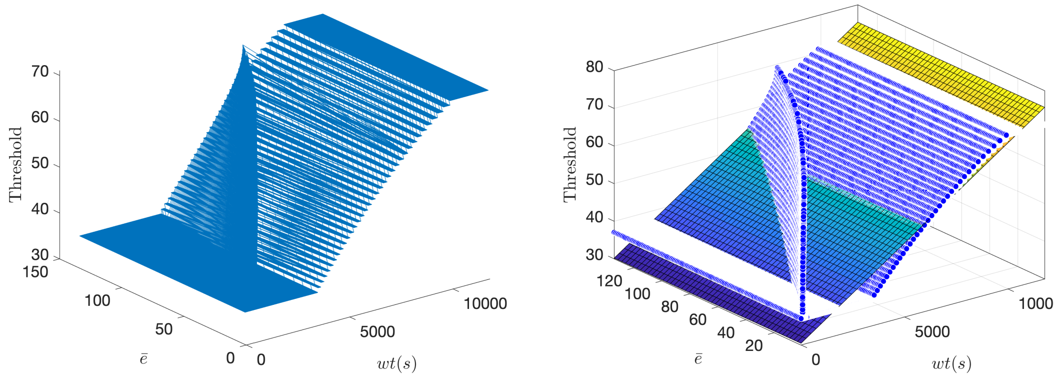
$$\pi_0 = \frac{wt(s)\,w - X}{(n-\bar{e})d} \text{ and } \pi_1 = \frac{wt(s)+X}{\bar{e}d} \text{ with } X = \sum_{l \text{ odd}} (l-1)\frac{r\binom{w}{l}\binom{n-w}{\bar{e}-l}}{\binom{n}{\bar{e}}}.$$

This function outputs higher thresholds compared to the Black-Gray decoder. It is a more conservative approach and by design, the BackFlip decoder, tends to avoid flipping the "wrong" bits so that it would have better recovery capabilities and a lower DFR (assuming that it can execute an un-bounded number of iterations). We point out that evaluating the function involves computing logarithms, exponents, and function minimization, and it is not clear how this can be implemented in constant-time (the reference code [9] is not implemented in constant-time).

One way to address this issue is to pre-calculate the finite number of pairs $(wt(s), \bar{e})$ and the resulting threshold for each pair, store them in a table, and read them from the table in constant-time. Reading from a table in constant-time means that not only the reading has to be executed in exactly the same number of operations every time, but also that the positions of the table which are accessed must not be revealed, i. e., memory access patterns must not depend on the secret input arguments. Therefore, such solution for the threshold function involves very high latency and is not practical.

Another approach is, similarly to the Black-Gray decoder (in BIKE-CPA [9]), to approximate the threshold function – which is here a function of two variables. A first attempt is shown in Figure 3.1. We compute the function over all the valid/relevant inputs and then compute an approximation by fitting it to a plane. Unfortunately, this approximation is not sufficiently accurate. Experiments showed that the DFR of the decoder with threshold approximated in this way is much higher than with exactly computed thresholds.

To improve the approximation we look at the threshold function for a fixed $\bar{e}$. An example for $\bar{e} = 25$ is shown in Figure 3.2. For every valid $0 \leq \bar{e} \leq$ we compute the linear approximation of the data points and obtain a function $\tau(x) = a + bx$, where x is a variable representing $wt(s)$, as shown in Panel (a) of the figure. Given $wt(s)$ and $\bar{e}$, the threshold is then computed by



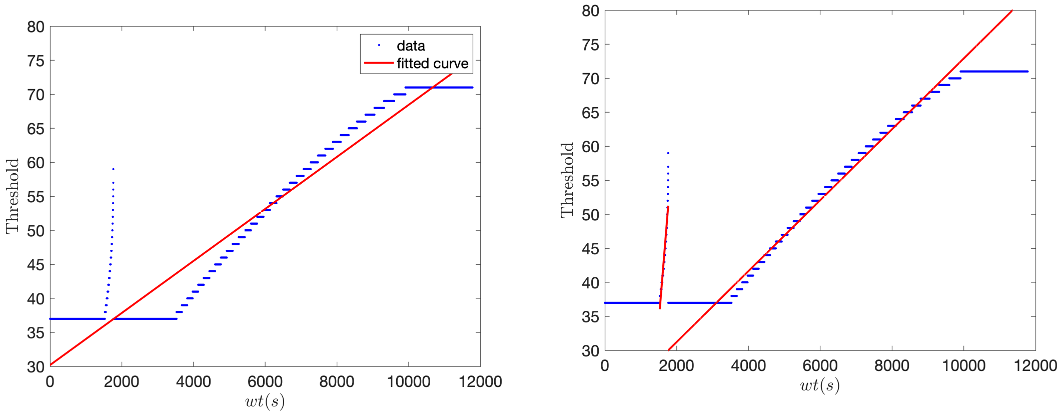
(a) The threshold function.

(b) Approximating the function (blue) using a plane.

Figure 3.1 – Approximating the BackFlip decoder threshold function.

choosing the linear approximation $\tau(x)$ corresponding to $\bar{e}$ and evaluating the function in $wt(s)$. This approach is obviously not good enough because, as it can be seen in Figure 3.2, the threshold function does not have a linear shape.

A refinement can be obtained by partitioning the approximation into five regions, as shown in Panel (b) of Figure 3.2, with three regions being constant and the remaining two regions having a shape which we try to approximate with a linear function. For every valid $\bar{e}$ we determine the boundaries of the five regions $[x_i, x_j]$. The first and the third part of the function are constant, admitting some value $\bar{e}_{min}$, and boundaries $[0, x_0)$ and $[x_1, x_2)$, respectively. The last region of the function evaluates to some $\bar{e}_{max}$, and has boundaries $[x_3, r)$. The remaining two regions with boundaries $[x_0, x_1)$ and $[x_2, x_3)$ are approximated by fitting the values to linear functions $a + bx$ and $c + dx$, respectively.



(a) Linear approximation for $wt(s) \in [0, r-1]$

(b) Five parts of the function depending on $wt(s)$, each with a different approximation .

Figure 3.2 – Approximating the threshold function for fixed $\bar{e} = 25$.

For all values of $\bar{e}$ we precompute the boundaries of the five regions of the threshold function alongside the τ_{min}, τ_{max} and a, b, c, d coefficients and store them in a table. Then for a given pair $(s, e) = (wt(s), \bar{e})$ the threshold is computed by:

```
if   (s < tab[e].x0) threshold = tab[e].thr_min;
elif (s < tab[e].x1) threshold = tab[e].a + tab[e].b * s;
elif (s < tab[e].x2) threshold = min;
elif (s < tab[e].x3) threshold = tab[e].c + tab[e].d * s;
else threshold = tab[e].thr_max;
```

To compute the threshold in constant-time we need to ensure that executed operations do not depend on the input parameters. Therefore, we have to evaluate all the conditions listed above and perform all the operations regardless of which condition is satisfied. Moreover, we have to use a secure constant-time comparison to evaluate the conditions – **secure_le_mask** compares two integers j, k and returns the mask **0x0** if $j < k$ and the mask consisting of all ones otherwise. The threshold computation is now:

```
cond0 = secure_le_mask(tab[e].x0, s);
cond1 = secure_le_mask(tab[e].x1, s) & ~secure_le_mask(tab[e].x0, s);
cond2 = secure_le_mask(tab[e].x2, s) & ~secure_le_mask(tab[e].x1, s);
cond3 = secure_le_mask(tab[e].x3, s) & ~secure_le_mask(tab[e].x2, s);
cond4 = ~secure_le_mask(tab[e].x3,s);

threshold  = cond0 & tab[e].thr_min;
threshold += cond1 & round(tab[e].a + tab[e].b * s1);
threshold += cond2 & tab[e].thr_min;
threshold += cond3 & round(tab[e].c + tab[e].d * s1);
threshold += cond4 & tab[e].thr_max;
```

With this, and the methods described in [46] we can implement BackFlip$^+$ in constant-time, provided that we fix a-priori the number of iterations.

3.4 Estimating the DFR of a decoder with a fixed number of iterations

The IND-CCA BIKE proof assumes a decapsulation algorithm that invokes an ideal decoding primitive. Here, the necessary condition is that the decapsulation has a negligible DFR, e. g., 2^{-128} [38, 9]. Therefore, a technique to estimate the DFR of a decoder is an essential tool.

The extrapolation method of [1]. An extrapolation method technique for estimating the DFR is shown in [1]. It consists of the following steps: a) Simulate proper encapsulation and decapsulation of random inputs for small block sizes (r values), where a sufficiently large number of failures can be observed; b) Extrapolate the observed data points to estimate the DFR for larger r values.

The DFR analyses in [1] and [9] apply this methodology to decoders that have some maximum number of iterations X_{BF} (we choose to assume that $X_{BF} = 100$ was used). In our experiments BackFlip^{+} always succeeds/fails before reaching 100 iterations for the relevant values of r. Practically, it means that setting $X_{BF} = 100$ can be considered equivalent to setting an unlimited number of iterations.

Our goal is to estimate the DFR of a decoder that is allowed to perform exactly X iterations (where X is predefined). We start from small values of X (e.g., $X = 2, 3, \ldots$) and increase it until we no longer see failures (in a large number of experiments) caused by exhausting X iterations. Larger values of X lead to a smaller DFR.

For our study, we used a slightly different extrapolation method. For every combination (scheme, level, decoder, r) we ran a sufficient number N_{exp} of experiments (sufficient in a sense that at least several failures are observed) as follows:

1. Generate, uniformly at random, a secret key and an error vector (e), compute the public key, and perform encapsulation-followed-by-decapsulation (with e).
2. Allow the decoder to run up to $X = 100$ iterations[6].
3. Record the actual number of iterations that were required in order to recover e. If the decoder exhausts the 100 iterations it stops and marks the experiment as a decoding failure.

For every $X < 100$ we say that the X-DFR is the sum of the number of experiments that fail (after 100 iterations) plus the number of experiments that required more than X iterations divided by N_{exp}. Next, we fix the scheme, the level, the decoder, and X, and we end up with an X-DFR value for every tested r. Subsequently, we perform linear/quadratic extrapolation on the data and derive a curve. We use this curve to find the value r_0 for which the X-DFR is our target probability p_0 and use the pair (r_0, p_0) as the BIKE scheme parameters.

We target three p_0 values: a) $p_0 = 2^{-23} \approx 10^{-7}$ that is reasonable for most practical use cases (with IND-CPA schemes); b) $p_0 = 2^{-64}$ also for an IND-CPA scheme but with a much lower DFR; c) $p_0 = 2^{-128}$, which is required for an IND-CCA Level-1 scheme. The linear/quadratic functions and the resulting r_0 values are given in Table A.1 in the appendix.

Our extrapolation methodology. In most cases, we were able to confirm the claim of [1] that the evolution of the DFR as a function of r occurs in two phases: quadratic initially, and then linear. As in [1], we are interested in extrapolating the linear part because it gives a more conservative DFR approximation. We point out that the results are sensitive to the method used for extrapolation (see details in Appendix A.1.1). Therefore, it is important to define it precisely so that the results can be reproduced and verified. To this end, we determine the starting point of the linear evolution as follows: going over the different starting points,

---

[6] Recall that different decoders have different definitions for the term “iterations”, see Section 3.1.3.

computing the fitting line and picking the one for which we get the best fit to the data points. Here, the merit of the experimental fit is measured by the root-mean-square deviation from the data points, which is a good choice in our case, where we believe that the data may have a few outliers.

3.5 Results

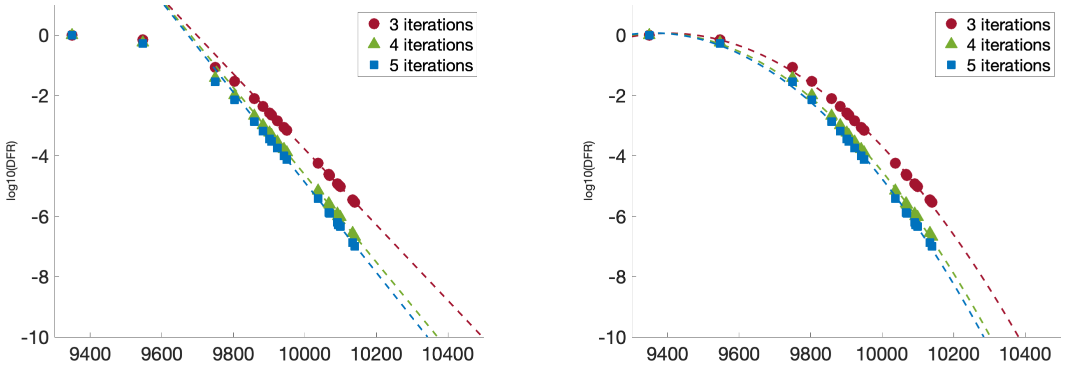
The experimentation platform. All the experiments were executed on an AWS EC2 *m5.metal* instance with the 6^{th} Intel®Core^{TM} Generation (Micro Architecture Codename Sky Lake [SKL]) Xeon®Platinum 8175M CPU 2.50GHz. It has 384 GB RAM, 32K L1d and L1i cache, 1MiB L2 cache, and 32MiB L3 cache, where the Intel® Turbo Boost Technology was turned off.

The code. The core functionality was written in $x86-64$ assembly and wrapped by assisting C code. The code uses the **PCLMULQDQ**, **AES-NI**, and **AVX512** instructions. The code was compiled with gcc (version 7.4.0) in 64-bit mode, using the "O3" Optimization level, and run on a Linux (Ubuntu 18.04.2 LTS) OS. It uses the NTL library [52] compiled with the GF2X library [53].

Figures 3.3 and 3.4 shows the simulation results for BIKE-1, Level-1 and Level-3, using the Black-Gray and BackFlip$^+$ decoders. Note that we use the IND-CCA flows. The left panels present linear extrapolations and the right panels present quadratic extrapolations. The horizontal axis measures the block size r in bits, and the vertical axis shows the simulated $\log_{10}(DFR)$ values. Every panel displays several graphs associated with different X values (number of iterations). The minimal X is chosen so that the extrapolated r value for $DFR = 2^{-128}$ is still considered to be secure according to [9]. The maximal value of X is chosen to allow a meaningful extrapolation. The raw data with equations for each of the performed extrapolations is given in Table A.1 in the appendix.

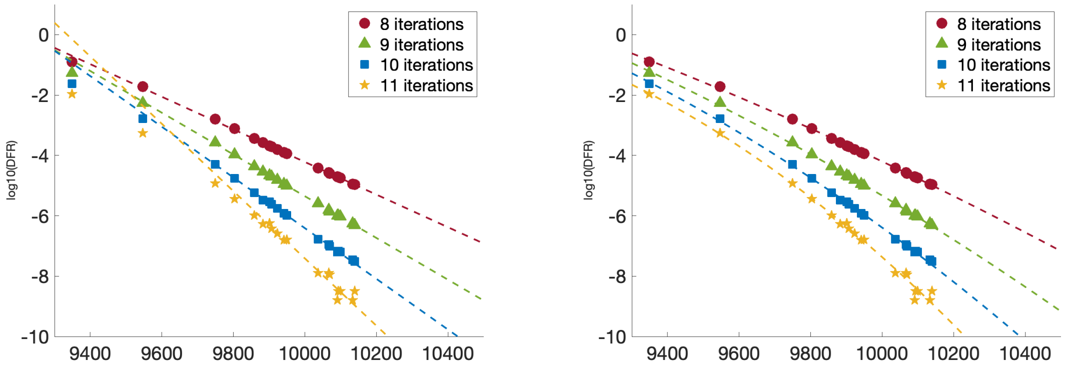
The quadratic approximations shown in Figures 3.3 and 3.4 yield a nice fit to the data points. However, we prefer to use the more pessimistic linear extrapolation in order to determine the target r.

Validating the extrapolation. We validated the extrapolated results for every extrapolation graph. We chose some r that is not a data point on the graph (but is sufficiently small to allow direct simulations). We applied the extrapolation to obtain an estimated DFR value. Then, we ran the simulation for this value of r and compared the results. Table 3.2 shows this comparison for several values of r and the Black-Gray decoder with $X_{BG} = 3$. We note that for 10267 and 10301 we tested at least 960 million and 4.8 billion tests, respectively. In case of 10301 decoding always succeeded after $X_{BG} = 4$ iterations, while for 10267 there were too few failures for meaningful computation of the DFR. Therefore, we use $X_{BG} = 3$ in our experimentation in order to observe enough failures. For example, the extrapolation for the setting (BIKE-1, Level-1, Black-Gray, 10301) estimates 3-DFR= $10^{-7.55}$. This is very close to the experimentally determined DFR of $10^{-7.56}$.



(a) BIKE-1-L1, Black-Gray, lin. ext.

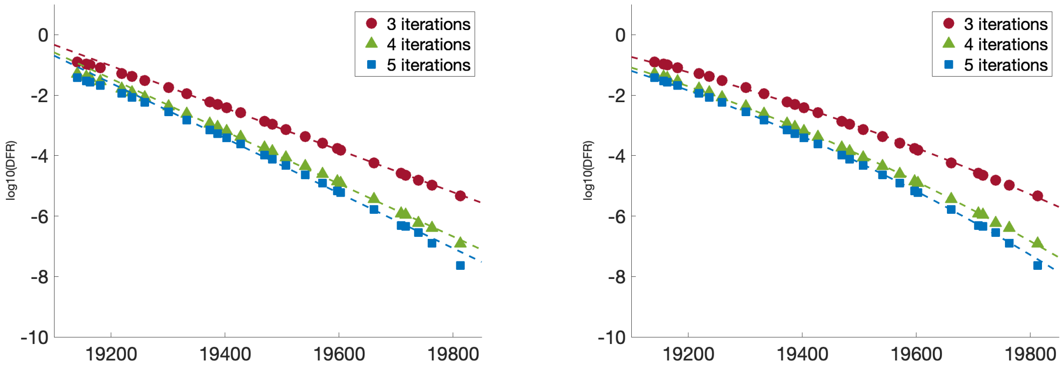
(b) BIKE-1-L1, Black-Gray, quad. ext.



(c) BIKE-1-L1, BackFlip$^+$, lin. ext.

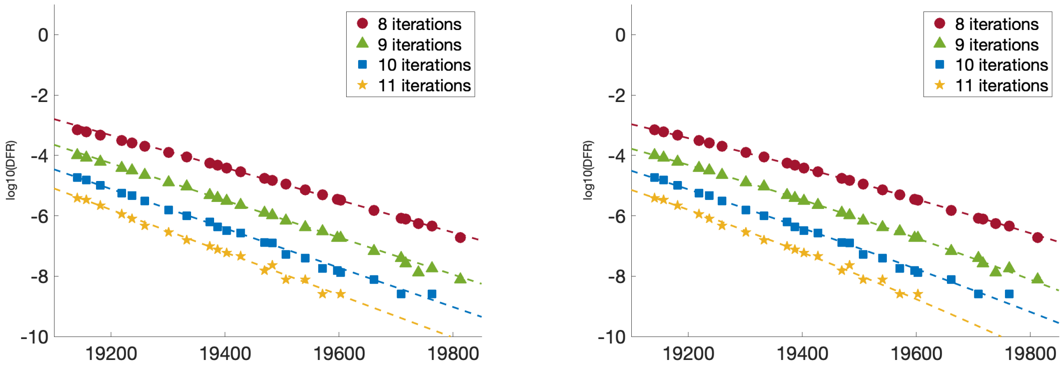
(d) BIKE-1-L1, BackFlip$^+$, quad. ext.

Figure 3.3 – BIKE-1 Level-1 extrapolations (see the text for details).



(a) BIKE-1-L3, Black-Gray, lin. ext.

(b) BIKE-1-L3, Black-Gray, quad. ext.



(c) BIKE-1-L3, BackFlip$^+$, lin. ext.

(d) BIKE-1-L3, BackFlip$^+$, quad. ext.

Figure 3.4 – BIKE-1 Level-3 extrapolations (see the text for details).

Table 3.2 – Validating the extrapolation results for the Black-Gray decoder with $X_{BG} = 3$ over two values of r.

r	Extrapolated DFR	Experimented DFR	Number of tests
10267	$10^{-7.13}$	$10^{-7.26}$	$9.6e8$
10301	$10^{-7.55}$	$10^{-7.56}$	$4.8e9$

Extensive experimentation. To observe that the Black-Gray decoder does not fail in practice with $r = 11779$ (i. e., the recommended r for the BackFlip decoder) we ran extensive simulations. We executed $10^{10} \approx 2^{33}$ tests that generate a random key, encapsulate a message and decapsulate the resulting ciphertext. Indeed, we did not observe any decoding failure (as expected).

3.5.1 Performance studies

The performance measurements reported hereafter are measured in processor cycles (per single core), where lower count is better. All the results were obtained using the same measurement methodology, as follows. Each measured function was isolated, run 25 times (warm-up), followed by 100 iterations that were clocked (using the **RDTSC** instruction) and averaged. To minimize the effect of background tasks running on the system, every experiment was repeated 10 times, and the minimum result was recorded.

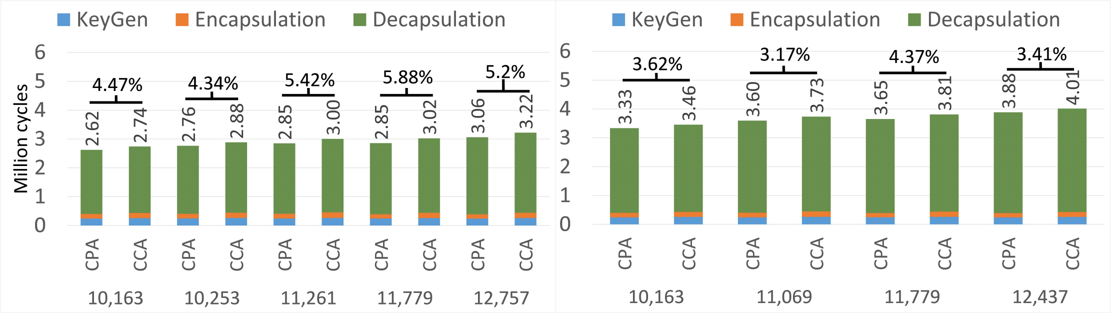
For every decoder, the performance depends on: a) X - the number of iterations; b) the latency of one iteration. Recall that comparing *just* the number of iterations is meaningless. Table 3.3 provides the latency ($\ell_{decoder,r}$) of one iteration and the overall decoding latency ($l_{decoder,r,i} = X_{decoder} \cdot \ell_{decoder,r}$) for the Black-Gray and the BackFlip$^+$ decoders, for several values of r. The first four rows of the table report for the value $r = 10163$ that corresponds to the BIKE-CPA proposal, and for the value $r = 11779$ that corresponds to the BIKE-CCA proposal. The subsequent rows report values of r for which the decoders achieve the same DFR.

Clearly, the constant-time Black-Gray decoder is faster than the constant-time BackFlip$^+$ decoder (when both are restricted to a given number of iterations).

We now compare the performance of the BIKE-CCA flows to the performance of the BIKE-CPA flows, for given r values, using the Black-Gray decoder with $X_{BG} = 3, 4$. Note that values of r that lead to DFR $> 2^{-128}$ cannot give IND-CCA security. Furthermore, even with BIKE-CCA flows and r such that DFR $\leq 2^{-128}$, IND-CCA security is not guaranteed (see the discussion in Section 3.6). The results are shown in Figure 3.5. The bars show the total latency of the key generation (blue), encapsulation (orange), and decapsulation (green) operations. The slowdown imposed by using the BIKE-CCA flows compared to using the BIKE-CPA flows is indicated (in percents) in the figure. We see that the additional cost of using BIKE-CCA flows is only ~ 6% in the worst case.

Table 3.3 – A performance comparison of the Black-Gray and the BackFlip$^+$ decoders for BIKE-1 at NIST security level 1.

DFR	Decoder	r	$X_{decoder}$	$\ell_{decoder,r}$ (cycles)	$l_{decoder,r,i}$ (million cycles)
2^{-19}	Black-Gray	10163	3	702785	2.11
2^{-17}	BackFlip$^+$	10163	8	751246	6.01
2^{-101}	Black-Gray	11779	4	784903	3.14
2^{-58}	BackFlip$^+$	11779	9	841806	7.58
2^{-23}	Black-Gray	10253	3	743168	2.23
2^{-23}	Black-Gray	10163	4	702785	2.81
2^{-23}	BackFlip$^+$	10499	8	777478	6.22
2^{-23}	BackFlip$^+$	10253	9	764959	6.88
2^{-64}	Black-Gray	11261	3	769212	2.31
2^{-64}	Black-Gray	11003	4	769820	3.08
2^{-64}	BackFlip$^+$	12781	8	907905	7.26
2^{-64}	BackFlip$^+$	12011	9	856084	7.70
2^{-128}	Black-Gray	12781	3	849182	2.55
2^{-128}	Black-Gray	12347	4	841310	3.37
2^{-128}	BackFlip$^+$	14797	9	1024798	9.22



(a) $X_{BG} = 3$ (b) $X_{BG} = 4$

Figure 3.5 – Comparison of BIKE-CPA flows and BIKE-CCA flows, running with the Black-Gray decoder and $X_{BG} = 3, 4$ for several values of r: $r = 10163$ the original BIKE-1-CPA; $r = 11779$ the original BIKE-1-CCA; r values that correspond to DFR of $2^{-23}, 2^{-64}, 2^{-128}$, according to Table A.1. The vertical axis measures latency.

3.6 Weak keys: a gap for claiming IND-CCA security

Our analysis of the decoders, the new parameters suggestion, and the constant-time implementation make significant progress towards a concrete instantiation and implementation of IND-CCA BIKE. However, we believe that there is still a subtle missing gap that needs to be addressed before IND-CCA security can be claimed.

The remaining challenge is that a claim for IND-CCA security depends on having an underlying δ-correct PKE (for example with $\delta = 2^{-128}$ for Level-1) [38]. This notion is different from having a DFR of 2^{-128}, and leads to the following problem. The specification [9] defines the DFR as "the probability for the decoder to fail when the input (h_0, h_1, e_0, e_1) is distributed uniformly". The δ-correctness property of a PKE/KEM is defined through Equations (3.1), (3.2) above. These equations imply that δ is the average of the *maximum* failure probability taken over all the possible messages. By contrast, the DFR notion relates to the average probability.

Remark 3. We also suggest to fix a small inaccuracy in the statement of the BIKE proof [9]: "... the resulting KEM will have the exact same DFR as the underlying cryptosystem ...". Theorem 3.1 of [38] states that: "If PKE is δ-correct, then PKE_1 is $\delta 1$-correct in the random oracle model with $\delta 1(qG) = qG \cdot \delta$.[...]". Theorem 3.4 therein states that: "If PKE_1 is $\delta 1$-correct then $\text{KEM}^{\not\perp}$ is $\delta 1$-correct in the random oracle model [..]"[7]. Thus, even if DFR= δ, the statement should be "the resulting KEM is $(\delta \cdot qG)$-correct, where the underlying PKE is δ-correct".

To illustrate the gap between the definitions, we give an example of what can go wrong.

Example 1. Let $\mathscr{S}$ be the set of valid secret keys and let $|\mathscr{S}|$ be its size. Assume that a group of weak keys $\mathscr{W}$ exists, and that $\frac{|\mathscr{W}|}{|\mathscr{S}|} = \bar{\delta} > 2^{-128}$. Suppose that for every key in $\mathscr{W}$ there exists at least one message for which the probability in Eq. (3.1) equals 1. Then, we get that $\delta > \bar{\delta} > 2^{-128}$. By comparison the average DFR can still be upper bounded by 2^{-128}. For instance, let $|\mathscr{S}| = 2^{130}$, $|\mathscr{W}| = 2^4$ and let the failure probability over all messages for every weak key be 2^{-10}. Let the failure probability of all other keys be 2^{-129}. Then,

$$
\begin{aligned}
DFR &= \mathbb{P}(\texttt{fail} \mid k \in \mathscr{W}) \cdot Pr(k \in \mathscr{W}) + Pr(\texttt{fail} \mid k \in \mathscr{S} \setminus \mathscr{W}) \cdot Pr(k \in \mathscr{S} \setminus \mathscr{W}) \\
&= \frac{|\mathscr{W}|}{|\mathscr{S}|} \cdot 2^{-10} + \frac{|\mathscr{S}| - |\mathscr{W}|}{|\mathscr{S}|} \cdot 2^{-129} \\
&= 2^{-126} \cdot 2^{-10} + (1 - 2^{-126}) \cdot 2^{-129} \\
&= 2^{-136} + 2^{-129} - 2^{-255} < 2^{-128}.
\end{aligned}
$$

3.6.1 Constructing weak keys

Currently, we are not aware of studies that classify weak keys for QC-MDPC codes or bound the number of weak keys. To see why this issue cannot be ignored we designed a series of tests that show the existence of a relatively large set of weak keys. Our examples are inspired by the notion of "spectrum" used in [22, 23, 24]. To construct the keys we changed the BIKE key generation. Instead of generating a random h_0, we start by setting the first $f = 0, 20, 30, 40$ bits to one, and then select randomly the positions of the additional $(d - f)$ bits. The generation of h_1 is unchanged.

Since it is difficult to observe failures and weak keys behavior when r is large, we study $r = 10163$ (of BIKE-1 CPA) and also $r = 9803$ that amplify the phenomena.

Figure 3.6 shows the behavior of the Black-Gray decoder for $r = 9803$ and $r = 10163$ with

---

[7] Here, $\text{KEM}^{\not\perp}$ refers to a KEM with implicit rejection, and qG is the number of invocations of the random oracle G (H in the case of BIKE).

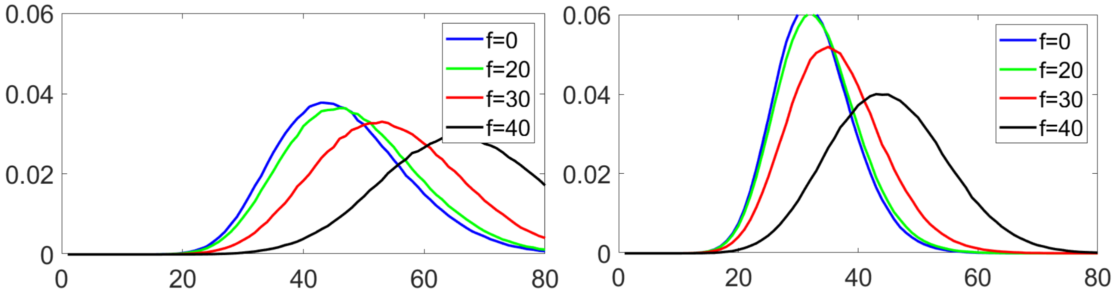
$f = 0, 20, 30, 40$ after $X_{BG} = 1, 2, 3, 4$ iterations. In every case (Panel) we choose randomly 10000 keys. For every key we choose randomly 1000 error vectors. The histograms show on the horizontal axis the weight of an "ideal" error vector e after the X_{BG} iteration, i. e., the number of bits in the error vector that are still not corrected after X_{BG} iterations. The vertical axis of the histogram represents the proportion of experiments that ended up with the corresponding error weight on the x axis. For example, if we follow the blue line ($f = 0$) through panels (a), (c), (e), and (g), we see that after each iteration a higher proportion of the experiments results in smaller error weights. After one iteration approximately 4% of the experiments ended up with 42 bits in the error that are not decoded. Already after the third iteration, most but not all of the experiments completely decode the error, i. e., the weight of the error becomes zero.

Note that the number of bits in the error vector that are not corrected after one iteration can actually increase because the decoder can mistakenly flip a bit which does not belong to the error. This behavior becomes apparent if we follow the black line ($f = 40$) for $r = 9803$ after 3 and 4 iterations, panels (e) and (g), respectively. The line in panel (g) is shifted to the right compared to the line in (e), meaning that the fourth iteration actually introduces new errors instead of correcting the existing ones.

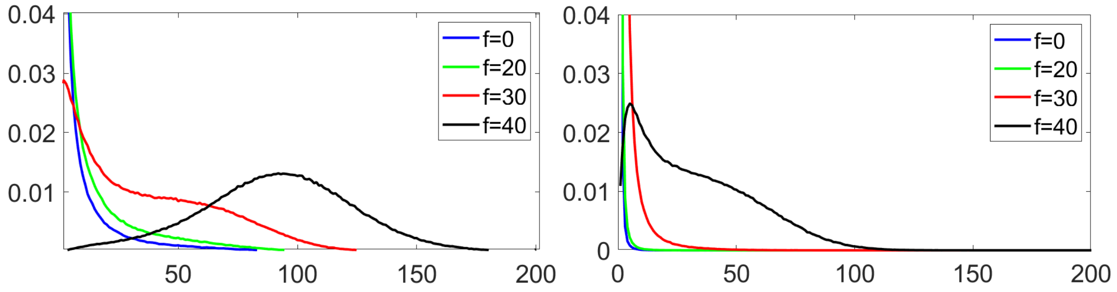
Furthermore, comparing the results of the experiments with different f values we notice that as f increases, the proportion of the experiments that result in more than a given number of un-decoded bits increases as well. This is most obvious after the first iteration, panels (a) and (b) – as f increases the lines shift more to the right side of the histogram. For $f \leq 20$, after 4 iterations decoding always succeeds in the case when $r = 10163$ and almost always when $r = 9803$. However, for $f > 30$ the experiments show that the proportion of decoding failures after 4 iterations is non-negligible in the case of $f = 30$, while for $f = 40$ the failures are even more abundant, e. g., the failure rate when using a weak key with $f = 40$ and $r = 9803$ is almost 100%. This shows that for a given decoder the set of weak keys depends on r and X, and that the issue with the weak keys needs to be properly analyzed when estimating the DFR of a decoder.

Remark 4 (Other decoders). Figure 3.6 shows how the weak keys impact the decoding success probability for chosen r and X_{BG} with the Black-Gray decoder. Note that such results depend on the specific decoder. To compare, BackFlip^{+} calculates the unsatisfied parity checks threshold in a more conservative way, and therefore requires more iterations. Weak keys would lead to a different behavior. On the other hand, when we repeat our experiment with the Simple-Parallel decoder, we see that almost all tests fail even with $f = 19$.

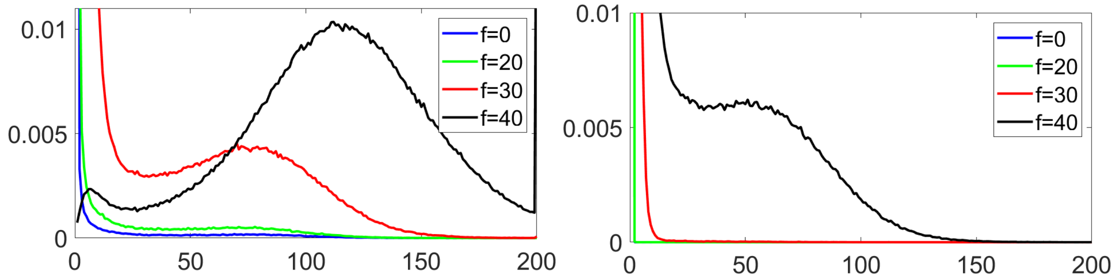
Figure 3.7 shows additional results with the Black-Gray decoder and $r = 9803$. Panel (a) shows the histogram for $f = 0$ (i. e., the standard h_0 generation), and Panel (b) shows the histogram for $f = 30$. The horizontal axis measures the number of failures x out of 10000 random errors. The vertical axis counts the number of keys that had the corresponding x number of failures, i. e., the number of keys with DFR of $x/10000$. For $f = 0$, the expected value and the standard deviation are $\mathbb{E}(x) = 119.06$ and $\sigma(x) = 10.91$, respectively. However, when $f = 30$, the decoder fails much more often and we have $\mathbb{E}(x) = 9900.14$, and $\sigma(x) = 40.68$. This shows the



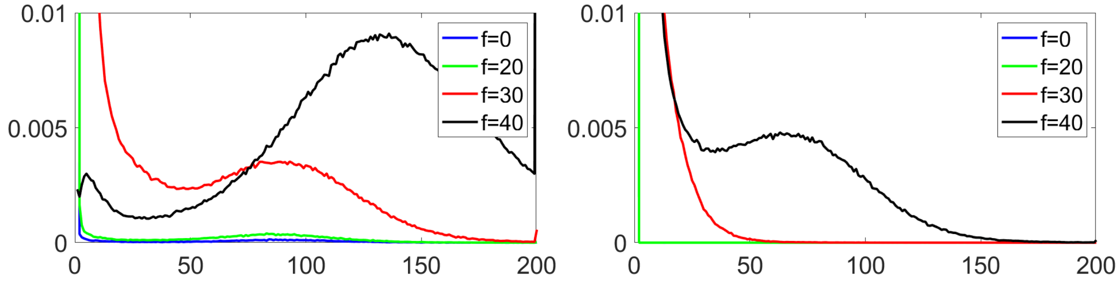
(a) $r = 9803$, iteration 1 (b) $r = 10163$, iteration 1



(c) $r = 9803$, iteration 2 (d) $r = 10163$, iteration 2



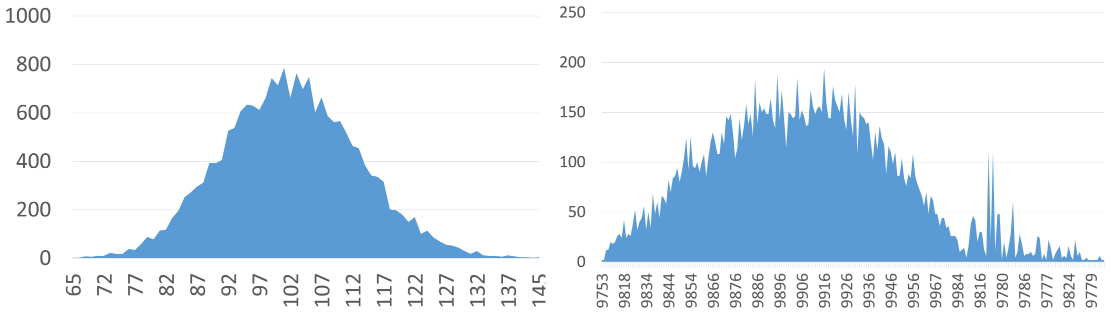
(e) $r = 9803$, iteration 3 (f) $r = 10163$, iteration 3



(g) $r = 9803$, iteration 4 (h) $r = 10163$, iteration 4

Figure 3.6 – Histograms showing on the vertical axis the proportion of the experiments that end-up with the corresponding number of error bits on the horizontal axis, after $X_{BG} = 1, 2, 3, 4$ iterations. The decoder is the Black-Gray decoder. Panels a, c, e, g represents the results for $r = 9803$ and Panels b, d, f, h for $r = 10163$ with $f = 0, 20, 30, 40$. A lower error weight is better. Note that the blue line is present in all the histograms, but in some of them it is covered by the green line.

difference between the weak keys and the “normal” randomized keys and that the DFR cannot be predicted by the “average-case” model. It is also interesting to note that for $f = 30$ we do not get a Gaussian like distribution (unlike the histogram with $f = 0$).



(a) (b)

Figure 3.7 – Black-Gray decoder, $r = 9803$, with $f = 0$ and $f = 30$ in Panel (a) and Panel (b), respectively. The horizontal-axis measures the number of decoding failures x out of 10000 experiments with random error vectors. The vertical-axis counts the number of keys that have a DFR of $x/10000$.

The remaining question is: what is the probability to hit a weak key when the keys are generated randomly as required? Let $\mathscr{W}_f$ be the set of weak keys that correspond to a given value of f. Define $z_{r,f}$ as the relative size of $\mathscr{W}_f$. Then

$$z_{r,f} = \frac{|\mathscr{W}_f|}{|\mathscr{S}|} = \frac{\binom{r-f}{d-f}}{\binom{r}{d}} \tag{3.3}$$

Note that if $f_2 < f_1$ then $\mathscr{W}_{f_1} \subset \mathscr{W}_{f_2}$. Therefore, choosing a larger f decreases the size of $\mathscr{W}_f$. It is easy to compute that

$$z_{9803,0} = z_{10163,0} = 1$$

$$z_{9803,10} = 2^{-72}, \quad z_{9803,20} = 2^{-146}, \quad z_{9803,30} = 2^{-223}, \quad z_{9803,40} = 2^{-304},$$

$$z_{10163,10} = 2^{-72}, \quad z_{10163,20} = 2^{-147}, \quad z_{10163,30} = 2^{-225}, \quad z_{10163,40} = 2^{-306}$$

The conclusion is that while the set $\mathscr{W}_f$ is large, its relative size (from the set of all keys) is still below 2^{-128}. Therefore, this construction *does not* show that BIKE-1 after our fix is necessarily *not* IND-CCA secure. However, it clearly shows that the problem cannot be ignored, and the claim that BIKE *is* IND-CCA secure requires further justification. In fact, there are other patterns and combinations that give sets of weak keys with a higher relative size (e. g., setting every other bit of h_0, f times). We point out again that any analysis of weak keys should relate to a *specific* decoder and a *specific* choice of r.

3.7 Discussion

The Round-2 BIKE [9] represents significant progress in the scheme's design, and offers an IND-CCA version, on top of the IND-CPA KEM that was defined in Round-1. In this chapter we address several difficulties and challenges and solve some of them.

- The BackFlip decoder runs in a variable number of steps that depends on the input and the secret key. We fix this problem by defining a variant, BackFlip$^+$, that, by definition, runs X_{BF} iterations for a parameter X_{BF}. We carry out the analysis to determine the values of X_{BF} where BackFlip$^+$ has DFR of 2^{-128}, and provide all of the details that are needed in order to repeat and reproduce our experiments. Furthermore, we show that for the target DFR, the values of X_{BF} are relatively small e. g., 12 (much less than 100 as implied for BackFlip).
- Inspired by the extrapolation method suggested in [1], we studied the Black-Gray decoder (already used in Round-1 Additional code [45]) that we defined to have a fixed number of steps (iterations) X_{BG}. Our goal was to find values of X_{BG} that guarantee the target DFR for a given r. We found that the values of r required with Black-Gray are smaller than the values with BackFlip$^+$. It seems that achieving the low DFR (2^{-128}) should be attributed to increasing r, independently of the decoding algorithm. The ability to prove this for some decoders is attributed to the extrapolation method.
- After the decoders are defined to run a fixed number of iterations, we could build constant-time software implementations (with memory access patterns and branches that reveal no secret information). This is nowadays the standard expectation from cryptographic implementations. We measured the resulting performance (on a modern x86-64 CPU) to find an optimal "decoder-X-r" combination. Table 3.3 shows that for a given DFR, the Black-Gray decoder is always faster than BackFlip$^+$.
- The analysis in Section 3.6 identifies a gap that needs to be addressed in order to claim IND-CCA security for BIKE. It relates to the difference between average DFR and the δ-correctness definition [38]. A DFR of (at most) 2^{-128} is a necessary requirement for IND-CCA security, which BIKE achieves. However, it is not necessarily sufficient. We show how to construct a "large" set of weak keys, but also show that it is still not sufficiently large to invalidate the necessary δ-correctness bound. This is a positive indication, although there is no proof that the property is indeed satisfied. This gap remains as an interesting research challenge to pursue. The problem of bounding (or eliminating) the number of weak keys is not specific to BIKE. It is relevant for other schemes that claim IND-CCA security and their decapsulation has nonzero failure probability. With this, we can state that BIKE CCA, instantiated with Black-Gray (or BackFlip$^+$) decoder with the parameters that guarantee DFR of 2^{-128}, and with the accompanying constant-time implementation, is IND-CCA secure, under the assumption that its underlying PKE is 2^{-128}-correct.

3.7.1 Methodologies

Our performance measurements were carried out on an x86-64 architecture. Studies on different architectures can give different results and therefore, we point to an interesting study of the performance of other constant-time decoders on other platforms [54]. Note that [54] targets schemes that use ephemeral keys with relatively large DFR and only IND-CPA security.

Differences in the DFR estimations. Our DFR prediction methodology may be (too) conservative and therefore yields more pessimistic results than those of [9]. One example is the combination (BIKE-1, Level-1, BackFlip$^+$ decoder, $r = 11779$, $X_{BF} = 10$). Here, [9] predicts a 100−DFR of 2^{-128} and our linear extrapolation for the 10-DFR predicts only 2^{-71} ($\approx 10^{-21}$). To achieve a 10-DFR of 2^{-128} we need to set $r = 13892$ (> 11779). Although the BackFlip$^+$ decoder with $X_{BF} = 10$ is not optimal, it is important to understand the effect of different extrapolations. Comparing to [1, 9] is difficult because no information is provided that allows us to repeat the experiments. Therefore,we attempt to provide some insight by acquiring data points for BackFlip$^+$ with $X_{BF} = 100$ and applying our extrapolation methodology. Indeed, the results we obtain are still more pessimistic, but if we apply a different extrapolation methodology ("two larger r's fit") we get closer to [9]. The details are given in Appendix A.1.1.

Another potential source of differences is that BackFlip has a recovery mechanism (TTL). For BackFlip$^+$ this mechanism is limited due to setting $X_{BF} \leq 11$. It may be possible to tune BackFlip and BackFlip$^+$ further by using some fine-grained TTL equations that depend on r. Information on the equations that were used for [9] was not published, so we leave tuning for further research.

3.7.2 Practical considerations for BIKE

Our decoder choice. We report our measurements only for Black-Gray and BackFlip$^+$ because other decoders that we are aware of either have a worse DFR (e. g., Parallel-Simple) or are inherently slow (e. g., Step-by-Step). Our results suggest that instantiating BIKE with Black-Gray is recommended. We note that the higher number of iterations required by BackFlip$^+$ is probably because it uses a more conservative threshold function than Black-Gray.

Recommendations for the BIKE flows. Currently, BIKE has two options for executing the key generation, encapsulation, and decapsulation flows. One for an IND-CPA KEM, and another (using the $FO^{\not\perp}$ transformation [38]) for an IND-CCA scheme, to deny a chosen ciphertext attack from the encapsulating party. It turns out that the performance difference is relatively small. As shown in Figure 3.5 for BIKE-1, the overhead of the IND-CCA flows is less than 6% (on x86-64 platforms). With such a low overhead, we believe that the BIKE proposal could gain a great deal of simplification by using *only* the IND-CCA flows. This is true even for applications that intend to use only ephemeral keys in order to achieve forward secrecy. Here, IND-CCA security is not mandatory, and IND-CPA security suffices. However, using the $FO^{\not\perp}$ transformation could be a way to reduce the risk of inadvertent repetition ("misuse") of a supposedly ephemeral key, thus buying some multi-user-multi-key robustness. By applying this approach, the scheme is completely controlled by choosing a single parameter r (with the same implementation).

Choosing r. The choice of r and X_{BG} gives an interesting trade-off between bandwidth and performance. A larger value of r increases the bandwidth but reduces the DFR when X_{BG} is fixed. On the other hand, it allows to reduce X_{BG} while maintaining the same DFR. This could lead to better performance. We give one example. To achieve DFR$= 2^{-23}$ the choice of $X_{BG} = 4$ and $r = 10163$ leads to decoding at $2.8M$ cycles. The choice $X_{BG} = 3$ and a slightly larger $r = 10253$ leads to decoding at $2.22M$ cycles. Complete details are given in Table 3.3.

General recommendations for the BIKE suite. Currently, BIKE [9] consists of 10 variants: BIKE-1 (the simplest and fastest); BIKE-2 (offering bandwidth optimization at the high cost of polynomial inversion); BIKE-3 (simpler security reduction with the highest bandwidth and lowest performance). In addition, there are also BIKE-2-batch and BIKE-3 with bandwidth optimization. Every version comes with two flavors, namely IND-CPA and IND-CCA. On top of this, every option comes with three security levels (L1/L3/L5). Finally, the implementation packages include generic code and optimization for AVX2 and AVX512.

We believe that this abundance of options involves too high complexity and reducing their number would be useful. For Round-3 we recommend to define BIKE as follows: BIKE-1 CCA instantiated with the Black-Gray decoder with $X_{BG} = 3$ iterations. Offer Level-1 with $r = 11261$ targeting DFR$= 2^{-64}$ and $r = 12781$ targeting DFR$= 2^{-128}$, as the main variants. In all cases, use ephemeral keys, for forward secrecy. For completeness, offer also a secondary variant for Level-3 with $r = 24659$ targeting DFR$= 2^{-128}$.

The code that implements these recommendations was contributed to (and already merged into) the open-source library LibOQS [55]. It uses the choice of $r = 11779$, following the block size of the current Round-2 specification (this choice of r leads to a DFR of 2^{-86}).

Vetting keys. We recommend to use BIKE with ephemeral keys and forward secrecy. In this case we do not need to rely on the full IND-CCA security properties of the KEM. However, there may be use cases that prefer the design with static keys. Here, we recommend the following way to narrow the DFR-δ-correctness gap described above by “vetting” the private key. For static keys we can assume that the overall latency of the key generation phase is less significant. Therefore, after generating a key, it would be still acceptable, from the practical viewpoint, to vet it experimentally. This can be done by running encapsulation-followed-by-decapsulation for some number of trials, in the hope to identify a case where the key is obviously weak. A more efficient way is to generate random (and predefined) errors and invoke the decoder. We point out that the vetting process can also be applied offline.

The new specification of BIKE [18], aimed at the Round-3 of the NIST PQC Project, introduced several changes that are based on the results presented in this chapter. Firstly, the CPA versions of the protocol are abandoned and the CCA versions are adopted as the only option because we showed that the CCA flows can be implemented with negligible performance overhead compared to the CPA flows. Moreover, we identified a gap in the proof of CCA security between the δ-correctness and the DFR of the used decoder which led to BIKE not claiming CCA security in Round-3 specification per se. The DFR of the currently used MDPC decoders is determined heuristically, i. e., not in a provable manner. Therefore the new claim is that the protocol is CCA secure only if the used decoder has a DFR of the required level. Finally, the suggestion that BIKE reduces the number of different options to a single one was also implemented in the new specification, granted, the chosen option was not BIKE-1 as suggested in this chapter, but rather BIKE-2 (as a result of the work presented in Chapter 5).

4 QC-MDPC decoders with several shades of gray

4.1 Introduction

BIKE [9] is a key encapsulation mechanism based on QC-MDPC codes, and is one of the Round-2 candidates of the NIST PQC Standardization Project [15]. The submission includes several variants of the KEM and we focus in this chapter on BIKE-1-CCA Level-1 and Level-3. We note that BIKE-2, the second variant of BIKE, has the same parameters as BIKE-1 and therefore, analysis of the performance and the DFR of various decoders in this chapter equally applies to BIKE-2.

The common QC-MDPC decoding algorithms are derived from the Bit-Flipping algorithm [21] and come in two main variants.

- "Step-by-Step": it recalculates the threshold every time that a bit is flipped. This is an enhancement of the "in-place" decoder described in [23].
- "Simple-Parallel": a parallel algorithm similar to that of [21]. It first calculates some thresholds for flipping bits and then flips the bits in all of the relevant positions, in parallel.

BIKE uses a decoder for the decapsulation phase. The specific decoding algorithm is a choice shaped by the target DFR, security, and performance. The IND-CCA version of BIKE Round-2 [9] is specified with the "BackFlip" (BF) decoder, which is derived from Simple-Parallel. The IND-CPA version is specified with the "One-Round" decoder, which combines the Simple-Parallel and the Step-By-Step decoders. In the "additional implementation" [45] we chose to use the "Black-Gray" decoder (BG) [46, 2], with the thresholds defined in [9]. This decoder (with different thresholds) appears in the BIKE pre-Round-1 submission "CAKE" and is due to N. Sendrier and R. Misoczki. We point out that the Backflip decoder that is defined and used in [9] is not the same as the BackFlip$^+$ decoder that is defined and studied in Chapter 3 (see the discussion in Section 4.7 for details).

In this chapter we explore a new family of decoders that combines the BG and the Bit-Flipping algorithms in different ways. Some combinations achieve the same or even better DFR compared to BG with the same block size, and at the same time also have better performance.

For better security we replace the mock-bits technique of the additional implementation [46] with a constant-time implementation that applies rotation and bit-slice-adder as proposed in [56] (and vectorized in [3]), and enhance it with further optimizations. We also report the first measurements of BIKE-1 on the new Intel “Ice-Lake” micro-architecture, leveraging the new **AVX512-VBMI2**, **VAESENC** and **VPCLMUL** instructions [57] (see also [58, 59]).

The chapter is organized as follows. Section 4.2 defines notation and recalls the definition of the BG decoder from the previous chapter. In Section 4.3 we define new decoders (BGF, B and BGB) and report our DFR per block size studies in Section 4.4. We discuss our new constant-time QC-MDPC implementation in Section 4.5. Section 4.6 reports the resulting performance. Section 4.7 concludes the chapter.

4.2 Preliminaries

In the previous chapter we described the BG decoder in Algorithm 3. This algorithm is implemented in the “Additional optimized” package [45] of Round-2 BIKE. Every iteration of BG involves three main steps which we extract and define as separate functions here.

The initial step performs a regular bit flipping based on the number of unsatisfied parity-check equations (*upc*) and a given threshold τ. Moreover, in the first step, two additional sets of bit positions are generated, *black* and *gray* arrays, based on the *upc*, the threshold τ and the parameter δ. The bits that are flipped are added to the *black* list, while the bits that are just below the threshold (*upc* is between τ and $\tau - \delta$) are added to the *gray* list. Algorithm 5 shows the function which implements the described functionality.

The second and the third step of BG flip bits in the received error vector if the unsatisfied parity-check value is at least the threshold and if the corresponding bit in the received mask is set to one. In the second and third step the mask consists of the *black* and *gray* array computed in the first step, respectively. The procedure which performs the required functionality for the last two steps of BG is described in Algorithm 6.

Algorithm 5 BitFlipIter

```
Input: $s \in \mathbb{F}_2^r$ (syndrome), $e \in \mathbb{F}_2^n$ (error vector), $\tau$ (threshold), $\delta$, $H \in \mathbb{F}_2^{r\times n}$ (parity-check matrix)
Output: $s \in \mathbb{F}_2^r$ (updated syndrome), $e \in \mathbb{F}_2^n$ (updated error vector), black and gray arrays
procedure BitFlipIter($s$, $e$, $\tau$, $\delta$, $H$)
    $black[n-1:0] = gray[n-1:0] = 0$
    $upc[n-1:0]$ = computeUPC($s, H$)
    for i in $0 \ldots n-1$ do
        if $upc[i] \geq \tau$ then
            $e[i] = e[i] \oplus 1$                      ▷ Flip an error bit
            $black[i] = 1$                              ▷ Update the Black set
        else if $upc[i] \geq \tau - \delta$ then
            $gray[i] = 1$                               ▷ Update the Gray set
    $s = H(c^T + e^T)$                                  ▷ Update the syndrome
    return $(s, e, black, gray)$
```

Algorithm 6 BitFlipMaskedIter

```
Input: $s \in \mathbb{F}_2^r$ (syndrome), $e \in \mathbb{F}_2^n$ (error vector), mask, $\tau$ (threshold), $H \in \mathbb{F}_2^{r\times n}$ (parity-check matrix)
Output: $s \in \mathbb{F}_2^r$ (updated syndrome), $e \in \mathbb{F}_2^n$ (updated error vector)
procedure BitFlipMaskedIter($s, e, mask, \tau, H$)
    $upc[n-1:0]$ = computeUPC($s, H$)
    for $i$ in $0 \ldots n-1$ do
        if $upc[i] \geq \tau$ then
            $e[i] = e[i] \oplus mask[i]$                ▷ Flip an error bit
    $s = H(c^T + e^T)$                                  ▷ Update the syndrome
    return $(s, e)$
```

4.3 New decoders with different shades of gray

In Algorithm 7, we present BG decoder redefined such that it uses the functions defined in the previous section (Algorithms 5 and 6). In cases where Algorithm 7 can safely run without a constant-time implementation, Step II and Step III are fast. The reason is that the unsatisfied parity-check values can be calculated only for indices that are contained in the *black* and *gray* arrays, and the number of these indices is at most the number of bits that were flipped in Step I (certainly less than n). By contrast, if constant-time and constant memory-access are required, the implementation needs to access all of the n positions uniformly. In that case the performance of Step II and Step III is similar to the performance of Step I. Thus, the overall decoding time of the BG decoder with X_{BG} iterations, where each iteration is executing steps I, II, and III, is approximately $3 \cdot X_{BG}$ times the time required for a single step of the algorithm.

The decoders that are based on Bit-Flipping are not perfect - they can erroneously flip a bit that is not an error bit. The probability to erroneously flip a “non-error” bit is an increasing function of $wt(e)/n$ and also depends on the threshold (note that $wt(e)$ may change during

Algorithm 7 e=BG(c, H, X)

```
    Input: $c \in \mathbb{F}_2^n$ (ciphertext), $H \in \mathbb{F}_2^{r \times n}$ (parity-check matrix), $X$ (maximal number of iterations)
    Output: $e \in \mathbb{F}_2^n$ (error vector)
    Exception: A "decoding failure" returns $\perp$
 1: procedure BG($c$, $H$)
 2:     $s = Hc^T$, $e[n-1:0] = 0$, $\delta = 4$
 3:     for $i$ in $1 \ldots X$ do
 4:         $\tau$ = computeThreshold($s$)
 5:         ($s$, $e$, $black$, $gray$) = BitFlipIter($s, e, \tau, \delta, H$)        ▷ Step I
 6:         ($s, e$) = BitFlipMaskedIter($s, e, black, (d+1)/2, H$)                ▷ Step II
 7:         ($s, e$) = BitFlipMaskedIter($s, e, gray, (d+1)/2, H$)                 ▷ Step III
 8:     if ($wt(s) \neq 0$) then
 9:         return $\perp$
10:     else
11:         return $e$
    Note that $d$ in steps 6 and 7 is the Hamming weight of the two defining polynomials of the parity-check matrix $H$ (as described in Section 2.1), i. e., $d$ is half of the Hamming weight of a row of $H$.
```

the execution). Step II and Step III of BG are designed to fix some erroneously flipped bits and therefore decrease $wt(e)$ compared to $wt(e)$ after one iteration of Simple-Parallel (without the *black*/*gray* masks). Recall from Algorithm 5 that the *black* list is populated by the bits which are flipped, while the *gray* list contains the bits which are not flipped but their *upc* values are just below the threshold. The idea of steps II and III is the following: if a bit which was flipped (*black*) still has a high *upc* value then it is probably not an error bit and therefore it is flipped back in Step II, while the bits in the *gray* list that are potentially error bits, because they had high but not high enough *upc*, are flipped in Step III. Proper analysis of the Black-Gray decoder and the merit of the black/gray technique does not exist in the literature. Therefore, we propose several new variations of the Black-Gray decoder and present them together in the pseudocode in Algorithm 8:

1. (BG) Black-Gray decoder: as defined in Algorithm 7.
2. (B) Black decoder: every iteration consists of only Steps I, II, i. e., there is no gray mask.
3. (BGF) Black-Gray-Flip decoder: it starts with one BG iteration and continues with several Bit-Flipping iterations (without the black and gray masks).
4. (BGB) Black-Gray-Black decoder: it starts with one BG iteration and continues with several B-iterations.

The rationale behind the B decoder is that the third step in BG may be unnecessary because in the first step of the subsequent iteration we will compute the *upc* values again and flip the appropriate bits. However, as presented in the next section, this variation of the decoder shows poor DFR properties. Furthermore, we observed that when $wt(e)/n$ becomes sufficiently small, i. e., enough error bits are corrected, the black/gray technique is no longer needed because erroneous flips have low probabilities. Since most of the error bits are corrected in the first decoding iteration we propose the BGF and BGB variations. The BGF decoder is based on the assumption that after the first iteration, few enough error bits are left that the probability is very low that a bit which is not in the error has high *upc* value. The BGB decoder is meant as a modification of BGF that may be slightly safer, because the "recovery" capability derived from black steps is present in all iterations, as opposed to BGF where it figures only in the first one.

Algorithm 8 e=decoder(D, c, H)

```
    Input: $D$ (decoder type one of $\{B, BG, BGB, BGF\}$), $c \in \mathbb{F}_2^n$ (ciphertext), $H \in \mathbb{F}_2^{r \times n}$ (parity-check matrix), $X$ (maximal number of iterations)
    Output: $e \in \mathbb{F}_2^n$ (error vector)
    Exception: A "decoding failure" returns $\perp$
procedure DECODER($D$, $c$, $H$)
    $s = Hc^T$, $e[n-1:0] = 0$, $\delta = 4$
    for $i$ in $1 \dots X$ do
        $\tau$ = computeThreshold($s$)
        ($s$, $e$, black, gray) = BitFlipIter($s, e, \tau, \delta, H$)        ▷ Step I
        if ($D \in \{B, BG, BGB\}$) or ($D = BGF$ and $i = 1$) then
            $(s, e)$ = BitFlipMaskedIter($s, e, black, (d+1)/2, H$)        ▷ Step II
        if ($D \in \{BG, BGB, BGF\}$ and $i = 1$) then
            $(s, e)$ = BitFlipMaskedIter($s, e, gray, (d+1)/2, H$)        ▷ Step III
    if ($wt(s) \neq 0$) then
        return $\perp$
    else
        return $e$
```

Example 2 (Counting the number of steps). Consider BG with 3 iterations. Here, every iteration involves 3 steps (I, II, and III) which are practically identical from the performance point of view. Thus, the total number of steps is 9. Consider, BGF with 3 iterations. Here, the first iteration involves 3 steps (I, II, and III) and the rest of the iterations involve only one step. The total number of steps is $3 + 1 + 1 = 5$. This is an important observation for discussion on performance of the decoders.

4.4 DFR evaluations for different decoders

In this section we evaluate and compare the B, BG, BGB, and BGF decoders under two criteria:

1. The DFR for a given number of iterations and a given value of the block size r.
2. The value of r that is required to achieve a target DFR with a given number of iterations.

In order to approximate the DFR we use the extrapolation method [1], and apply two forms of extrapolation: "best linear fit" [2] described in the previous section and "two large r's fit" as in [2, Appendix C] and described in Appendix A.1.1. We point out that validity of the results derived by the extrapolation method relies on the assumption that the dependence of the DFR on the block size r is a concave function in the relevant range of r. We point out that this relation between the block size and the DFR is supported by extensive simulations [1, 60, 2], however, in the current state of the art, it is still only an assumption. Table 4.1 summarizes our results. It shows the r-value required for achieving a DFR of $2^{-23}(\approx 10^{-8})$, 2^{-64}, and 2^{-128}. It also shows the approximated DFR for $r = 11779$ (which is the value used for BIKE-1 Level-1 CCA). Appendix A.2 provides the full information on the experiments and the extrapolation analysis.

Table 4.1 – The DFR achieved by different decoders. Two extrapolation methods are shown: "best linear fit" (as in [2]) and "two large r's fit" (as in [2, Appendix C]). The second column shows the number of iterations for each decoder. The third column shows the total number of (performance-wise identical) executed steps.

			Best linear fit				Two large r's fit			
Decoder	#I	#S	DFR = 2^{-23}	2^{-64}	2^{-128}	DFR at 11779	DFR = 2^{-23}	2^{-64}	2^{-128}	DFR at 11779
BG	3	9	10253	11213	12739	2^{-88}	10253	11171	12619	2^{-90}
	4	12	10163	11003	12347	2^{-100}	10163	10909	12107	2^{-110}
	5	15	10133	10909	12107	2^{-111}	10133	10853	11987	2^{-116}
BGB	4	9	10253	11093	12491	2^{-95}	10253	11083	12491	2^{-96}
	5	11	10163	10973	12227	2^{-105}	10163	11027	12413	2^{-99}
	6	13	10133	10973	12269	2^{-104}	10133	10949	12197	2^{-107}
BGF	5	7	10301	11171	12539	2^{-92}	10301	11131	12491	2^{-95}
	6	8	10253	11027	12277	2^{-102}	10253	10973	12197	2^{-107}
	7	9	10181	10949	12149	2^{-108}	10181	10949	12107	2^{-112}
B	4	8	10259	11699	13901	2^{-67}	10301	11813	14221	2^{-63}
	5	10	10133	11437	13229	2^{-79}	10133	11437	13451	2^{-76}
	6	12	10067	11213	13037	2^{-84}	10067	11437	13397	2^{-78}

Interpreting the results of Table 4.1. Based on Table 4.1 we may conclude that it is possible to trade BG with 3 iterations for BGF with 6 iterations. This achieves a better DFR and also a $\frac{9}{8} = 1.125$-fold speedup. Moreover, if the required DFR is at most 2^{-64}, it suffices to use BGF with only 5 iterations (and get the same DFR as BG with 3 iterations). This achieves a

$\frac{9}{7} = 1.28$-fold speedup. The situation is similar for BG with 4 iterations compared to BGB with 5 iterations: this achieves a $\frac{12}{11} = 1.09$-fold speedup. If a DFR of 2^{-128} is required it is possible to trade BG with 4 iterations for BGF with 7 iterations and achieve a $\frac{12}{9} = 1.33$-fold speedup. Another interesting trade off is available if we are willing to slightly increase the value of r. Compare BG with 4 iterations (i. e., 12 steps) and BGF with 6 iterations (i. e., 8 steps). For a DFR of 2^{-64} we have $r_{BG} = 11003$ and $r_{BGF} = 11027$. A very small relative increase in the block size, namely $(r_{BGF} - r_{BG})/r_{BG} = 0.0022$, gives a $\frac{12}{8} = 1.5$-fold speedup.

Example 3 (BGF versus BG with 3 iterations). Figure 4.1 shows a qualitative comparison (the precise details are provided in Appendix A.2). The left panel indicates that BGF has a better DFR than BG for the same number of (9) steps when $r > 9970$. Similarly, the right panel shows the same phenomenon even with a smaller number of BGF steps (7) when $r > 10726$ (with the best linear fit method) and $r > 10734$ (with the two large r's method) that correspond to a DFR of 2^{-43} and 2^{-45}, respectively. Both panels show that the crossover point occurs for values of r below the range that is relevant for BIKE.

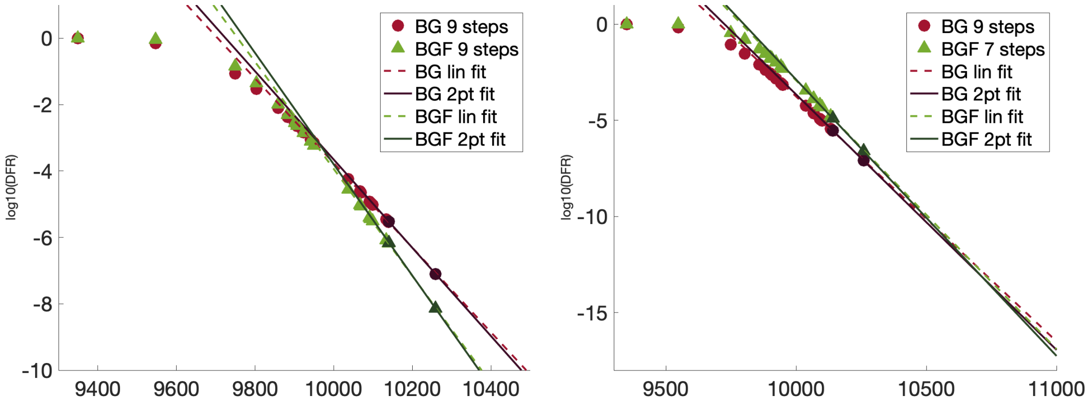


Figure 4.1 – DFR comparison of BG with 3 iterations (9 steps) to BGF with: (Left panel) 7 iterations (9 steps); (Right panel) 5 iterations (7 steps).

4.5 Constant-time implementation of the decoders

Secure implementation of a cryptographic primitive is required to be side-channel resistant. The two most common side-channel attacks are timing attacks and cache (memory) attacks. An implementation is secure against side-channel timing attacks if the execution time, or more precisely the number and order of executed instructions, does not depend on any secret value. Furthermore, the implementation needs to be resistant to attacks that exploit memory accesses of the program (cache attacks). An adversary performing a cache attack is considered to have the ability to see some of the memory addresses that are accessed by the program. Therefore, the memory access pattern of the implementation has to be independent of any secret value as well.

Two functions that are required to implement the three steps in the decoder are described in

Section 4.2, namely in Algorithms 5 and 6. Both functions perform bit flipping – an operation which is inherently constant-time and constant-memory access because all the bits of the error vector are accessed during the execution. Conditional execution of operations in both functions, i. e., **if/else** conditions, can simply be converted to secure implementation by performing both branches of a condition and appropriately masking the result.

The last procedure that has to be securely implemented is counting the number of unsatisfied parity-check equations (computeUPC). Recall that the syndrome is $s \in \mathbb{F}_2^r$, the error vector $e = \begin{pmatrix} e^{(0)} & e^{(1)} \end{pmatrix} \in \mathbb{F}_2^n$ with $e^{(0)}, e^{(1)} \in \mathbb{F}_2^r$, and the parity matrix $H = \begin{pmatrix} H^{(0)} & H^{(1)} \end{pmatrix} \in \mathbb{F}_2^{r \times n}$ is a quasi-cyclic matrix composed of two circulant matrices $H^{(0)}, H^{(1)} \in \mathbb{F}_2^{r \times r}$ which are fully determined by their defining polynomials $h^{(0)}, h^{(1)} \in \mathscr{R}$, respectively. The syndrome is computed by $s = He^T$:

$$\begin{pmatrix} s_0 \\ s_1 \\ \vdots \\ s_{r-1} \end{pmatrix} = \begin{pmatrix} h_0^{(0)} e_0^{(0)} & + & \cdots & + & h_{r-1}^{(0)} e_{r-1}^{(0)} & + & h_0^{(1)} e_0^{(1)} & + & \cdots & + & h_{r-1}^{(1)} e_{r-1}^{(1)} \\ h_{r-1}^{(0)} e_0^{(0)} & + & \cdots & + & h_{r-2}^{(0)} e_{r-1}^{(0)} & + & h_{r-1}^{(1)} e_0^{(1)} & + & \cdots & + & h_{r-2}^{(1)} e_{r-1}^{(1)} \\ & & & & & \vdots & & & & & \\ h_1^{(0)} e_0^{(0)} & + & \cdots & + & h_0^{(0)} e_{r-1}^{(0)} & + & h_1^{(1)} e_0^{(1)} & + & \cdots & + & h_0^{(1)} e_{r-1}^{(1)} \end{pmatrix}$$

The unsatisfied parity-check counters $upc^{(0)}, upc^{(1)} \in \mathbb{Z}^r$ can be computed separately for the two parts of the error vector $e^{(0)}$ and $e^{(1)}$. For the sake of clarity of the notation, we describe algorithms for calculating one of the two *upc* arrays, and in that regard we denote the counters simply by $upc \in \mathbb{Z}^r$, the corresponding part of the error vector by $e \in \mathbb{F}_2^r$, and the corresponding part of the parity matrix by $H \in \mathbb{F}_2^{r \times r}$. With the new notation in mind, we have that the r bits of the syndrome correspond to the parity equations in the following way:

$$\begin{pmatrix} s_0 \\ s_1 \\ \vdots \\ s_{r-1} \end{pmatrix} \leftrightsquigarrow \begin{pmatrix} h_0 e_0 & + & h_1 e_1 & + & \cdots & + & h_{r-1} e_{r-1} \\ h_{r-1} e_0 & + & h_0 e_1 & + & \cdots & + & h_{r-2} e_{r-1} \\ & & & \vdots & & & \\ h_1 e_0 & + & h_2 e_1 & + & \cdots & + & h_0 e_{r-1} \end{pmatrix}$$

The upc_i value for the i-th bit of the error vector e_i is defined as the number of the above shown equations which involve the bit e_i, i. e., the corresponding bit of h and the value of s_i are both one. Given s and H, the *upc* value of the j-th error bit is defined as $upc_j = \sum_{i=0}^{r-1} H_{i,j} s_i$. Therefore, computing upc_j can be done by computing the Hamming weight of the vector calculated by *bitwise-and* of the syndrome s and the j-th column of H (denoted by $H_{:,j}$), i. e., $upc_j = wt(s \star H_{:,j})$ where we denote the *bitwise-and* operation by the "$\star$" symbol. Recall that one of the main advantages of quasi-cyclic codes is their compact representation, and therefore, H is represented by a single vector and never stored as a full matrix. This means that for each of the r bits of the error vector the computation of its *upc* value requires a vector rotation to obtain the right column of H. This solution is good for side-channel resistance, however, it turns out that the performance penalty is too high.

The second approach to computing *upc* tries exploit the moderate density of the code, i. e., the fact that the weight of the parity matrix rows is low. If we have the positions of the non-zero bits of h (denoted by $supp(h)$), then we do not have to compute the *bitwise-and* of s and the columns of H. We can rather compute $upc_j = \sum_{i \in supp(H_{:,j})} s_i$, by accessing the bits of s only at the positions corresponding to the non-zero bits of $H_{:,j}$. This approach is much faster than the previous one, but its memory access pattern makes it unsafe in terms of cache attacks – the accessed positions in the syndrome are exactly the secret values that define the secret key H.

The mock-bits technique was introduced in [46] for side-channel protection in order to obfuscate the secret $supp(h)$. The idea is to introduce a new vector h_m containing a number of mock bits and compute the vector $\overline{h} = h + h_m$. Then the algorithm described in the previous paragraph can be performed with the set of positions $supp(\overline{H}_{:,j})$ defined by the vector $\overline{h}$ instead of positions $supp(H_{:,j})$. In this way, even if the adversary recovers all the accessed positions in the syndrome, namely $supp(\overline{h})$, it is not able to distinguish between the mock-bits and the real bits of the secret vector. Provided that the set $supp(\overline{h}_m)$ is large enough the adversary cannot recover the secret positions by simply guessing which positions are the ones of the secret key. For example, the implementation of BIKE-1 Level-1 used 62 mock-bits and thus $wt(\overline{h}) = 133$. The probability to correctly guess the secret 71 bits of h if the whole set $supp(\overline{h})$ is given is $\binom{133}{71}^{-1} \approx 2^{-128}$. This technique was designed for ephemeral keys but may leak information on the private key if it is used multiple times, i. e., if most of $\overline{h}$ can be obtained by a cache attack. By knowing that $supp(h) \subset supp(\overline{h})$, the adversary can learn that all the other bits of h are zero. Subsequently, the adversary can generate the following system of linear equations $FH^T = 0$, where F is the public key, and set the relevant variables to zero. If the number of zero positions of H that are discovered by the attack is high enough then the adversary can solve the system and recover the remaining part of the secret h. To avoid this, the number of mock bits combined with the number of real bits needs to be at least $r/2$ so the system is sufficiently undetermined. However, using that many mock-bits makes this method impractical in terms of performance (it was used as an optimization to begin with).

Therefore, a different approach is needed here, such as the solution presented in [56]. Let us rearrange the summands in the parity equations in the following way:

$$\begin{pmatrix} s_0 \\ s_1 \\ \vdots \\ s_{r-1} \end{pmatrix} \leftrightsquigarrow \begin{pmatrix} h_0 e_0 & + & h_1 e_1 & + & \cdots & + & h_{r-1} e_{r-1} \\ h_0 e_1 & + & h_1 e_2 & + & \cdots & + & h_{r-1} e_0 \\ & & & \vdots & & & \\ h_0 e_{r-1} & + & h_1 e_0 & + & \cdots & + & h_{r-1} e_{r-2} \end{pmatrix}.$$

Then, for a fixed number j, consider a column vector generated by setting each row to the value of the j-th summand of the corresponding equation. For example, for $j = 1$ we have the following column vector:

$$\begin{pmatrix} s_0 \\ s_1 \\ \vdots \\ s_{r-1} \end{pmatrix} \leftrightsquigarrow \begin{pmatrix} h_1 e_1 \\ h_1 e_2 \\ \vdots \\ h_1 e_0 \end{pmatrix}.$$

This vector consists of the products of h_j and the bits of the error vector rotated by j places. If we now rotate simultaneously both vectors by j places in the opposite direction we obtain the following correspondence:

$$\begin{pmatrix} s_{r-1} \\ s_0 \\ \vdots \\ s_{r-2} \end{pmatrix} \leftrightsquigarrow \begin{pmatrix} h_1 e_0 \\ h_1 e_1 \\ \vdots \\ h_1 e_{r-1} \end{pmatrix}.$$

Therefore, the $upc \in \mathbb{Z}^r$ array can be computed as $upc = \sum_{i=0}^{r-1} h_i \, rot(s,i)$ where for each i vector s is appropriately rotated and multiplied by a scalar representing the i-th bit of h. Obviously, if h_i is zero, then the i-th summand is the zero vector. Thus, $upc = \sum_{i \in supp(h)} h_i \, rot(s,i)$. Therefore, to compute the upc we need only $wt(h)$ rotations of the syndrome and vector additions. Moreover, the operation that performs the *bitwise-and* of two vectors is not required anymore since the involved summands are vectors multiplied by one, i. e., the vectors themselves. Since h is a sparse vector with weight much smaller than r, this algorithm is a promising candidate for computing upc provided that the rotation can be implemented efficiently and securely.

4.5.1 Optimizing the rotation of an array

The algorithm for calculating the number of unsatisfied parity-check equations proposed in [56] and described in the previous section requires a fast and side-channel resistant implementation of the array rotation operation. In [56], the authors propose an optimization based on the *bit-slicing* technique. Vectorized implementation with SIMD instructions of the proposed optimization is described in [3][1].

Consider the rotation of the syndrome s which is stored in memory as a bitstring, i. e., an array of memory words where each word is populated with bits of s. Given a number l that is the number of bits to rotate the syndrome by, we first write it in the word size base $l = l_{hi} \cdot word\_size + l_{lo}$ with $0 \le l_{lo} < word\_size$. The rotation of s is then done in two phases: “big” rotation where s is rotated by the multiple of the word size $l_{hi} \cdot word\_size$ and “small” rotation where the bits inside the words are rotated by l_{lo}.

The “big” rotation is simple since it can be done by rearranging the words of s in a simple manner. For example, let s' be the memory representation of the syndrome, i. e., an array of r_{size} words representing s. Rotation of s' by l_{hi} words is then:

$$rot(s', l_{hi}) = \sum_{i=0}^{r_{size}-1} s'_j \;\text{ where }\; j = (i - l_{hi}) \bmod r_{size}.$$

However, straightforward implementation of this formula is not secure since we would be reading memory locations that depend on the value of the secret l_{hi}. Therefore, we write l_{hi} in binary and pad it to the size equal to the bit-size of the largest possible l_{hi} (denoted by

---

[1] The paper [3] does not point to publicly available code.

r_{max_rot}). Then, for a given l_{hi} we rotate the syndrome by 2^i words for every $i \in [0, r_{max\_rot}]$ and by appropriate masking take into account only the rotations by i for which the i-th bit of l_{hi} is set to one. With this, the implementation of the "big" rotation is both constant-time and constant-memory access.

The "small" rotation is defined as the rotation of the syndrome by a number of places smaller than the word size. Implementing the small rotation in case when the word size is one of the standard sized of the processor architecture, e. g., 64-bit word on x86_64 architecture, is straightforward. Namely, we need only *shift* and *or* operations. Let s' be an array of r_{size} words of size 64 bits, then the rotation by l_{lo} can be performed by the following code snippet:

```
   for (int i = 0; i < r_size; i++)
      out[i] = (in[i] >> l_lo) | (in[i+1] << (word_size - l_lo));
```

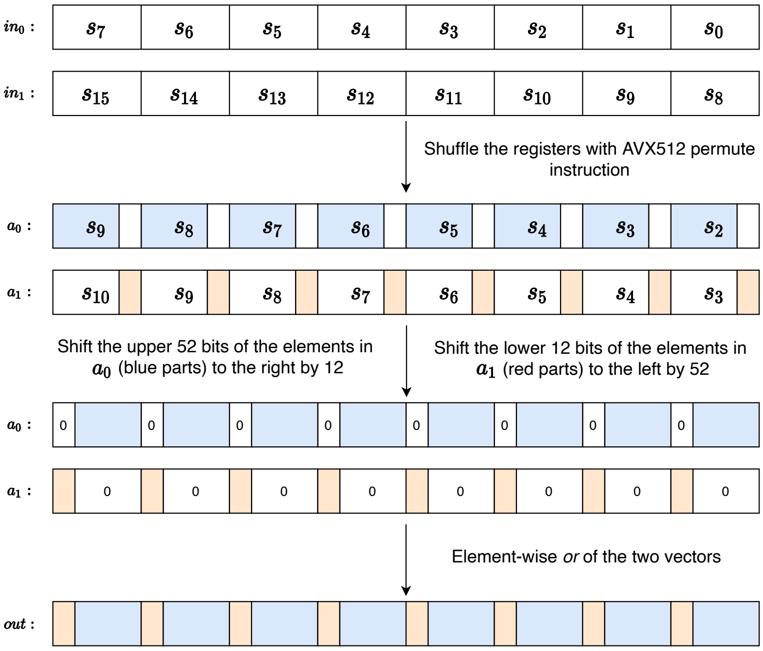
where it is assumed that the input array contains s' with s'_0, the first word of s', appended to the end of the array. Note that the implementation shown in the code snippet above is constant-time and memory access because we iterate over all the words of s' and for each word perform the same operations.

However, it is not so clear how to perform the "small" rotation when using SIMD registers, e. g., **AVX2** and **AVX512** instruction set extensions for x86_64 architectures [57]. The reason is that even though, for example, an **AVX512** register holds 512 bits of data, we can only operate on the standard word size elements of the register. Therefore, there is no *shift* instruction which would shift the 512-bit string inside the register in its entirety, rather the shift instruction shifts each of the eight 64-bit elements of the register. In [3], the authors show a code snippet (for the core functionality) for rotating by a number of positions that is less than the word size. Here, we present our vectorized implementation which achieves better performance by exploiting the **_mm512_permutex2var_epi64** instruction supported by the **AVX512** instruction set [57]. This instruction takes two input registers and produces the output register by shuffling the 64-bit elements in the registers using the values provided in the third input argument, as shown in Listing 4.1.

```
__m512i _mm512_permutex2var_epi64 (__m512i a, __m512i idx, __m512i b) {
  for (int i = 0; i < 8; i++) {
    // Fourth bit of an element of idx serves as the selector
    selector = idx[i] & 0x8;
    // First 3 bits serve as the index of the element to output
    elem_idx = idx[i] & 0x7;
    // Select a or b and copy the appropriate element
    out[i] = selector ? b[elem_idx] : a[elem_idx];
  }
}
```

Listing 4.1 – **AVX512** instruction **_mm512_permutex2var_epi64**

The "small" rotation is performed, analogously to the whole rotation, in two parts. For example, consider two **AVX512** registers in_0 and in_1 holding consecutive words of the syndrome, and let $l_{lo} = 140 = 2 \cdot 64 + 12$. First we shuffle the elements of the **AVX512** registers to obtain register a_0 containing the right shift by $2 \cdot 64$ of the input and additional register a_1 corresponding to the shift by $(2+1) \cdot 64$, as shown in Figure 4.2. Then the elements in a_0 and a_1 are shifted to the right by 12 places and to the left by $64 - 12 = 52$ places, respectively. The result is obtained by computing element-wise *or* of the two registers. The whole function that implements the "small" rotation functionality is given in Listing A.1 in the appendix.



in_0 :
s_7 s_6 s_5 s_4 s_3 s_2 s_1 s_0
in_1 :
s_{15} s_{14} s_{13} s_{12} s_{11} s_{10} s_9 s_8
Shuffle the registers with AVX512 permute instruction
a_0 :
s_9 s_8 s_7 s_6 s_5 s_4 s_3 s_2
a_1 :
s_{10} s_9 s_8 s_7 s_6 s_5 s_4 s_3
Shift the upper 52 bits of the elements in a_0 (blue parts) to the right by 12
Shift the lower 12 bits of the elements in a_1 (red parts) to the left by 52
a_0 :
0 0 0 0 0 0 0 0
a_1 :
0 0 0 0 0 0 0 0
Element-wise *or* of the two vectors
out :

Figure 4.2 – "Small" rotation of two **AVX512** registers containing consecutive elements of the syndrome (refer to the text for details).

The latest Intel micro-architecture "Ice-Lake" introduces a new **_mm512_shrdv_epi64** instruction as part of the new **AVX512-VBMI2** instruction extension set [57]. This instruction receives two 512-bit registers (a, b) together with another 512-bit index register (c) and computes the result as shown in Listing 4.2. It concatenates the corresponding 64-bit elements of a and b to produce a 128-bit intermediate result, and then shifts the result to the right by the value specified in the corresponding element of c, and finally stores the lower 64 bits in the output register. It is easy to see that this instruction can be used directly in the "small" rotation algorithm to replace the two *shift* and the final *or* instruction (lines 19-21 in Listing A.1).

```
__m512i _mm512_permutex2var_epi64 (__m512i a, __m512i idx, __m512i b) {
  for (int i = 0; i < 8; i++) {
    out[i] = concat(b[i], a[i]) >> (c[i] % 64);
  }
}
```

Listing 4.2 – **AVX512-VBMI2** instruction **_mm512_shrdv_epi64**

Using VPCLMUL and VAESENC

The Ice-Lake processors support the new vectorized **PCLMUL** and **AESENC** instructions, **VPCLMUL** and **VAESENC** [57]. We used the multiplication code presented in [61, Figure 2], and the CTR DRBG code of [62, 59], in order to improve the performance of our BIKE implementation. The results are given in Section 4.6.

4.6 Performance studies

We start with describing our experimentation platforms and measurements methodology. The experiments were carried out on two platforms, (Intel® Turbo Boost Technology was turned off on both):

- **EC2 Server:** An AWS EC2 **m5.metal** instance with the 6^{th} Intel®Core^{TM} Generation (Micro architecture Codename "Sky Lake"[SKL]) Xeon®Platinum 8175M CPU 2.50GHz. This platform has 384 GB RAM, 32K L1d and L1i cache, 1MiB L2 cache, and 32MiB L3 cache.
- **Ice-Lake:** Dell XPS 13 7390 2-in-1 with the 10^{th} Intel®Core^{TM} Generation (Micro architecture Codename "Ice Lake"[ICL]) Intel®Core^{TM} i7-1065G7 CPU 1.30GHz. This platform has 16 GB RAM, 48K L1d and 32K L1i cache, 512K L2 cache, and 8MiB L3 cache.

The code. The code is written in C and x86-64 assembly. The implementations use the **VPCLMUL**, **VAES**, **AVX2**, **AVX512** and **AVX512-VBMI2** instructions when available. The code was compiled with gcc (version 8.3.0) in 64-bit mode, using the "O3" Optimization level with the "-funroll-all-loops" flag, and run on a Linux (Ubuntu 18.04.2 LTS) OS.

Measurements methodology. The performance measurements reported hereafter are measured in processor cycles (per single core), where lower count is better. All the results were obtained using the same measurement methodology, as follows. Each measured function was isolated, run 25 times (warm-up), followed by 100 iterations that were clocked (using the **RDTSC** instruction) and averaged. To minimize the effect of background tasks running on the system, every experiment was repeated 10 times, and the average result was recorded. Note that each execution of a measured function is performed with different seeds. This however, does not have any effect on the runtime since the whole code package is implemented in a constant-time manner.

4.6.1 Decoding and decapsulation: performance studies

Performance of BG. Table 4.2 shows the performance of our implementation which uses the rotation and bit-slice-adder techniques of [56, 3], and compares the results to the additional implementation of BIKE [45]. The results show a 3.75 to 6.03-fold speedup for the portable (C code) of the decoder, 1.1-fold speedup for the **AVX512** implementations but a 0.66-fold slowdown for the **AVX2** implementation. This slowdown can be explained by the fact that the **AVX512** implementation can leverage the masked store and load operations that do not exist in the **AVX2** architecture. Note that key generation is faster because generation of mock-bits is no longer needed.

Table 4.2 – The EC2 server performance of BIKE-1 when using the BG decoder with 3 iterations. The cycles (in columns 4, 5) are counted in millions.

Implementation	Level	Op	Additional Impl. [45]	This paper	Speedup
C-portable stand-alone	Level-1	Keygen	1.67	1.37	1.22
		Decaps	60	15.99	3.75
	Level-3	Keygen	4.75	4.03	1.18
		Decaps	242.72	64.09	3.79
C-portable + OpenSSL	Level-1	Keygen	0.86	0.56	1.54
		Decaps	52.38	8.68	6.03
	Level-3	Keygen	2.71	1.98	1.37
		Decaps	218.42	39.82	5.48
AVX2	Level-1	Keygen	0.27	0.15	1.81
		Decaps	3.03	3.62	0.84
	Level-3	Keygen	0.62	0.38	1.64
		Decaps	10.46	15.84	0.66
AVX512	Level-1	Keygen	0.26	0.15	1.79
		Decaps	2.59	1.83	1.42
	Level-3	Keygen	0.57	0.37	1.57
		Decaps	8.97	8.14	1.10

Table 4.3 compares our implementations with different instruction sets (**AVX512F**, **AVX512-VBMI2**, **VPCLMUL**, and **VAES**). The results for BIKE-1 Level-1 show speedup factors of 1.47, 1.28, and 1.26 for key generation, encapsulation, and decapsulation, respectively. Even better speedup factors of 1.58, 1.39, and 1.24, are achieved for BIKE-1 Level-3.

Consider the 6th column and the BIKE-1 Level-1 results. The 93521 cycles of the key generation consist of 13K, 13K, 1K, 1K, 5.5K, 26K, 26K cycles for generating $h_0, h_1, \sigma_0, \sigma_1, g, f_0, f_1$, respectively (and some additional overheads). Compared to the 3rd column of this table (with only **AVX512F** implementation): 13.6K, 13.6K, 2K, 2K, 6K, 46K, 46K, respectively. Indeed, as

Table 4.3 – BIKE-1 using the BG decoder with 3 iterations. Performance in cycles on Ice-Lake using various instruction sets: (a) **AVX512F**; (b) **AVX512F**, **AVX512-VBMI2**, **VPCLMUL**; (c) **AVX512F**, **AVX512-VBMI2**, **VPCLMUL**, **VAES**.

Level	Op	(a)	(b)	Speedup	(c)	Speedup
1	Keygen	137095	95068	1.44	93521	1.47
	Encaps	192123	150860	1.27	150612	1.28
	Decaps	2192433	1711127	1.28	1737912	1.26
3	Keygen	375604	240350	1.56	238198	1.58
	Encaps	432577	310908	1.39	310533	1.39
	Decaps	9019103	7201222	1.25	7277357	1.24

reported in [61], the use of **VPCLMUL** doubles the speed of the polynomial multiplication. Note that the vector-AES does not contribute much, because the bottleneck in generating h_0, h_1 is the constant-time rejection sampling check and not the AES calculations.

Table 4.4 compares our right-rotation method to the snippet shown in [3]. To accurately measure these "short" functionalities, we ported them into separate compilation units and compiled them separately using the "-c" flag. In addition, the number of repetitions was increased to 10000. This small change improves the rotation significantly (by a factor of 2.3) and contributes ~ 2% to the overall decoding performance.

Table 4.4 – Rotation performance in cycles, comparison of the code snippet given in [3] and our implementations: (a) Listing A.1 with **AVX512F** and (b) Listing. A.1 modified to use **AVX512-VBMI2** as explained in Section 4.5.

Level	r	Platform	[3]	(a)	(b)	(a) Speedup	(b) Speedup
L1	11779	EC2 server	128	105	-	1.21	-
L1	11779	Ice-Lake	149	120	63.97	1.24	2.33
L3	24821	EC2 server	250	205	-	1.22	-
L3	24821	Ice-Lake	296	236	121.72	1.25	2.43
L5	40597	EC2 server	404	329	-	1.23	-
L5	40597	Ice-Lake	475	382	194.46	1.24	2.44

4.7 Discussion

Our study shows four shades-of-gray decoders based on combinations of the three steps involved in the Black-Gray decoder. The results show that among the four decoders, BGF offers the most favorable DFR-performance trade off. Indeed, as shown in Table 4.1, it is possible to trade BG, which was our leading option so far, for another decoder and have the same or even better DFR for the same block size. The advantage is either in performance (e. g., BGF with 6 iterations is $\frac{12}{8} = 1.5$ times faster than BG with 4 iterations) or in implementation simplicity (e. g., the B decoder that does not involve gray steps).

The Backflip decoder of [9] and the BackFlip$^+$ decoder of [2]

We explain here why our search for efficient decoders does not include BackFlip$^+$. Our recent work [2] explains why the use of BackFlip, as it is defined in [9], cannot be part of an IND-CCA KEM since, by its definition, it runs with a variable number of steps that depends on the input (for BIKE, this input involves secret data). In this context, defining an algorithm (decoder) that runs a secret-dependent number of steps, and building an IND-CCA claim on top, is a fundamental flaw because the number of steps must be either: a) considered part of the decoder's output in the security proof; or b) a-priori fixed. In such case, the proposed DFR analysis, that is the critical property of the decoder in this context, no longer holds.

To this end, [2] defines a variant *(different)* decoder, named therein BackFlip$^+$ that pursues option #b, and is parametrized by a number X_{BF} of iterations. The difference between BackFlip and BackFlip$^+$ is explicitly defined in:

> [2, Section 2.3], "The BackFlip$^+$ is a variant of Backflip that uses a fixed number of iterations as explained in Section 1. Technically, the difference is that the condition on the weight of s is moved from the **while** loop to the **if** statement (line 10). This performs the appropriate number of mock iterations."

Subsequently, [2] analyzes the resulting DFR (of several decoders, including BackFlip$^+$) as a function of X_{BF} for values where the resulting version is practical ($X_{BF} = 9, 10, 11, 12$), and the DFR is sufficiently low. It also checks for $X_{BF} = 100$ which *implicitly* appears – only in the reference code of BIKE, and as an arbitrary value with no explanation – in order to try and validate the claims on the DFR estimations of [9] (because no sufficient details were given in [9] and also in a preceding work [1]). We point out that by looking at the reference code of [9], it is clear that Backflip executes as many (up to 100) secret-dependent iterations as are needed to decode the syndrome, and is not designed to run exactly 100 iterations. In fact, the reported performance of BIKE reflects this fact exactly. This is why [2, Section 3] states:

> [2, Section 3], "Therefore, we may choose to interpret the results of [2, 18] as if the 2^{-128} DFR was obtained from simulations with this $X_{BF} = 100$ bound, although this is nowhere stated and the simulation data and the derivation of the DFR are also not provided."

Only BackFlip$^+$ (and not the original Backflip) can be used as a decoder which is part of a (potentially) IND-CCA secure KEM, and practically, only for the small X_{BF} values that are possibly useful. Note that this consideration is completely orthogonal to subsequently *building* a real constant time implementation and profiling its performance. This work is also done in [2], and the findings are that *another* decoder, namely, BG has superior properties.

We point out that even with negligible DFR there is still another fundamental problem in assuming that negligible (average) DFR suffices for an IND-CCA claim. The gap in the proof is shown in [2] by constructing a counterexample: a large set of “weak keys”, which is luckily (for BIKE), still small compared to the total number of keys. Thus, determining CCA property for QC-MDPC codes remains an open question. In any case, [2] concludes that the BIKE parameters (block size) need to be increased in order to achieve a DFR of 2^{-128}.

Responsible disclosure. Prior to uploading [2], we communicated it to the authors of BIKE [9]. In particular, we delayed the posting by a month, and engaged in a long discussion with the authors of [1] about the problems in the definition of Backflip and the lacking information in order to reproduce the claimed results. Some statements of the original version of [2] were softened per request.

A comment on the performance of BackFlip$^+$. We note that a BackFlip$^+$ iteration is practically equivalent to Step I of BG with some additional overhead to handle the TTL values. It is possible to improve the constant-time TTL handling with the bit-slicing techniques and reduce this gap. However, we believe that this would not change the DFR-performance trends reported here and in [2].

Parameter choice recommendations for BIKE. BIKE-1 Level-1 (IND-CCA) [9] uses $r = 11779$ with a target DFR of 2^{-128}. We set aside the weak keys gap (identified in [2]) for now and consider a non-weak key. If we limit the number of usages of this key to Q and choose r such that $\mathrm{Q} \cdot \mathrm{DFR} < 2^{-\mu}$ (for some target margin μ), then the probability that an adversary with at most Q queries sees a decoding failure is at most $2^{-\mu}$. We suggest that KEM should use ephemeral keys (i. e., Q= 1) for forward secrecy, and this usage does not mandate IND-CCA security (IND-CPA suffices). Here, from the practical viewpoint, we only need to target a sufficiently small DFR such that decapsulation failures would be a significant operability impediment. However, an important property that *is* desired, even with ephemeral keys, is some guarantee that an inadvertent α times key reuse (where α is presumably not too large) would not affect the security. This suggests the option for selecting r so that $\alpha \cdot \mathrm{DFR} < 2^{-\mu}$. For example, taking $\mu = 32$ and $\alpha = 2^{32}$ (an extremely large number of "inadvertent" reuses), we can target a DFR of 2^{-64}. Using BGF with 5 iterations, we can use $r = 11171$, which is smaller than 11779 that is currently used for BIKE.

Further optimizations. The performance of BIKE’s constant-time implementation is dominated by three primitives: a) polynomial multiplication (it remains a significant portion of the computations even when using the **VPCLMUL** instructions); b) polynomial rotation (that requires extensive memory access); c) the rejection sampling (approximately 25% of the key generation). This paper showed how some of the new Ice-Lake features can already be used for performance improvement. Further optimizations are an interesting challenge.

We note that based on the results presented in this chapter, the new specification of BIKE [18], aimed at Round-3 of the NIST PQC Project, is defined with the proposed BGF decoder because we showed that it offers the best performance for the required DFR levels.

5 Fast polynomial inversion for post quantum QC-MDPC cryptography

The BIKE suite submitted to the Round-2 of the NIST Post-Quantum standardization process proposes three different key encapsulation mechanism (KEM) variants. The first one, BIKE-1 described in Section 2.1.7, is based on McEliece's framework and it offers very good performance in terms of key generation and encapsulation. The consequence of the efficient key generation process is that the public key size, and hence the required bandwidth for the protocol, is fairly big, e. g., the parameters for the first level of security require a bandwidth of about 3000 bytes per key exchange in both directions of the KEM protocol. The third BIKE variant, BIKE-3 described in Section 2.1.9, is based on the Ouroboros cryptographic scheme, and like BIKE-1, it features an efficient key generation procedure. Moreover, the BIKE-3 design offers the possibility to slightly modify the scheme in such a way that the public key size is reduced to almost half the size of the keys in BIKE-1. However, even with this modification the size of the ciphertext stays the same as in BIKE-1, hence the reduction in the amount of data exchanged is only in one direction. Furthermore, BIKE-3 employs a variation of the decoding algorithm (used in the decapsulation phase) of the other two BIKE variants. Consequently, to achieve the same level of security as BIKE-1 and BIKE-2 the size of the BIKE-3 parameters has to be slightly increased. Lastly, the BIKE-2 variant of the KEM (described in Section 2.1.8) is based on the Niederreiter framework and because of that is has one big advantage over the other variants – namely, the required bandwidth in both directions is halved compared to BIKE-1. Another advantage, due to the use of Niederreiter's framework, is that BIKE-2 has a tighter security reduction for the IND-CCA secure variant compared to BIKE-1 and BIKE-3. However, BIKE-2 was not the popular variant, e. g., only BIKE-1 is integrated into LibOQS [55] and s2n [63] libraries. The reason BIKE-2 was neglected so far is its performance. Namely, the key generation algorithm of BIKE-2 involves inversion of a binary polynomial in a polynomial ring. The computational cost of the inversion dominates even the cost of decapsulation which is usually the most expensive part of a code-based scheme due to the decoding procedure. This issue is especially prominent when protocols are designed to achieve forward-secrecy by using ephemeral keys (i. e., where a key has to be generated for every communication session). Considering that the main disadvantage of all the code-based schemes in the NIST standardization process is the key and ciphertext size and that BIKE-2 excels exactly in that sense, we believe that BIKE-2 deserves greater attention. Therefore, in this chapter we present an attempt to reduce the runtime of the BIKE-2 key generation procedure to an acceptable level.

Polynomial inversion over a finite field is a time-consuming operation in several post-quantum cryptosystems (e. g., BIKE [9], HQC [50], ntruhrss701 [64], LEDAcrypt [65]). The literature includes different approaches to the problem, depending on the degree of the polynomial and the field/ring over which the polynomial is defined. For example, the Itoh-Tsuji inversion (ITI) algorithm [27] is designed to be efficient for computing multiplicative inverses in a binary field $\mathbb{F}_{2^k}$. Since one possible way to represent the elements of $\mathbb{F}_{2^k}$ is by using the polynomial representation, the ITI algorithm can be considered as an algorithm for inversion of binary polynomials modulo some irreducible polynomial that defines the field. Another prominent algorithm for inversion is Safegcd [66] which is based on the Extended GCD algorithm modified such that its implementation is constant-time friendly while at the same time the performance of the algorithm remains relatively satisfying. It is demonstrated in [66] as a means for speeding up ntruhrss701 [64] and for elliptic curve cryptography with Curve25519. Furthermore, it is used in the latest implementation of LEDAcrypt [65] which is another code-based submission to the NIST process. Algorithms for inversion of sparse polynomials over binary fields are discussed in [67, 68]. These algorithms are based on the division algorithm of [69].

There are (at least) two popular open-source libraries that provide inversion of polynomials with coefficients in $\mathbb{F}_2$: NTL [4], compiled with the GF2X library [53] and OpenSSL [5] library. We note that the Additional code of BIKE (BIKE-2) [45] can be compiled to use either NTL or OpenSSL. Therefore, we use the performance of these two libraries as our comparison baseline for BIKE-2 key generation. In this chapter, we propose and describe the implementation of a variant of the ITI algorithm (see also [70]) for polynomial inversion that leverages the special algebraic structure in our context. Moreover, we implement the algorithm such that it runs in constant-time and constant-memory access, thus making this implementation a viable option for secure implementation of the key generation procedure in BIKE-2.

The paper is organized as follows. Section 5.1 offers some background and notation. In Section 5.2 we explain our polynomial inversion method. In Section 5.3 we give details of our implementations. Finally, Section 5.4 provides the performance results and Section 5.5 concludes the chapter with several concrete proposals for the BIKE suite.

5.1 Preliminaries and notation

We briefly recall the notation from the previous chapters and the definition of the BIKE-2 KEM. The polynomial ring $\mathcal{R}$ is defined as $\mathbb{F}_2[x]/(x^r-1)$, where r is the block size of the code that is used as a basis for the BIKE-2 protocol. Moreover, r is chosen such that $x^r-1=(x-1)\Phi_{r-1}$, where Φ_{r-1} is irreducible cyclotomic polynomial of degree $r-1$. The set of invertible elements in $\mathcal{R}$ is denoted by $\mathcal{R}^*$. We treat polynomials in $\mathcal{R}$, interchangeably, as vectors of bits. For every element $v \in \mathcal{R}$ its Hamming weight is denoted by $wt(v)$ and its support (i. e., the positions of the non-zero bits) by *supp*(v). In other words, if an element $v \in \mathcal{R}$ is defined by $v = \sum_{i=0}^{r-1} v_i x^i$ then *supp*(v) is the set of positions of the non-zero bits, *supp*$(v) = \{i : v_i = 1\}$. Uniform random sampling from a set U is denoted by $u \xleftarrow{\$} U$, while uniform random sampling of an element with fixed Hamming weight w from a set U is denoted by $u \xleftarrow{w} U$.

The protocol level parameters of BIKE-2 are the block size r and the density of the secret key w, i. e., the weight of a row of the secret parity-check matrix (note that w is such that $d = w/2$ is odd). The key generation procedure performs the following steps:

1. $h_0, h_1 \xleftarrow{d} \mathcal{R}$, where both h_0 and h_1 are sampled such that they have odd Hamming weight $d = w/2$.
2. $\sigma_0, \sigma_1 \xleftarrow{\$} \mathcal{R}$, only in the IND-CCA secure version of the scheme.
3. $(f_0, f_1) = (1, h_1 h_0^{-1})$, where h_0^{-1} is the inverse of $h_0 \in \mathcal{R}$.
4. Output the public key (f_0, f_1) and the secret key (h_0, h_1) or $(h_0, h_1, \sigma_0, \sigma_1)$ in the CPA or CCA case, respectively.

Note that the requirement for d to be odd (and $< r$) has as a consequence the fact that h_0 and h_1 are invertible, $h_0, h_1 \in \mathcal{R}^*$, as explained in Section 2.1.2. Notably, in the second step of the algorithm a polynomial inversion is performed. This is, relatively speaking, a time-consuming operation that can deter adoption when targeting forward-secrecy via ephemeral keys. On the other hand, BIKE-2 has half the communication cost compared to BIKE-1 (and $\sim 2/3$ the communication cost compared to the bandwidth-optimized version of BIKE-3). Specifically, the initiator in BIKE KEM sends *pk* to the responder, i. e., f_1 for BIKE-2 versus (f_0, f_1) for BIKE-1. In the other direction, the responder sends a ciphertext to the initiator. The length of BIKE-2's ciphertext is half the length of BIKE-1's ciphertext, as detailed in Section 2.1.8. Therefore, reducing the computational cost of polynomial inversion can make BIKE-2 more competitive.

5.2 Optimized polynomial inversion in $\mathbb{F}_2[x]/(x^r-1)$

In this chapter, we propose to use an algorithm for inversion that is similar to the ITI algorithm [27]. In both cases, the original ITI algorithm and our proposition, the essence is that raising an element a to the power 2^k (referred to as k-squaring hereafter), can be done efficiently. The ITI algorithm inverts an element $a \in \mathbb{F}_{2^k}$ where the field elements are represented in normal basis. With such representation computing the k-squaring operation, a^{2^k}, consists of k cyclic shifts of a's vector representation. This results in fast implementation of k-squaring. However, we note that the ITI algorithm can be generalized to other cases where k-squaring is efficient. One example is the set of polynomial rings that are used in BIKE and in other QC-MDPC based schemes.

In Algorithm 9 we present an algorithm that on input a, computes a^{2^k-1} for some k which is itself a power of 2, $k = 2^t$. The algorithm exploits the following:

$$2^{2^t} - 1 = 2^{2^t} - 2^{2^{t-1}} + 2^{2^{t-1}} - 1 = 2^{2^{t-1}}(2^{2^{t-1}} - 1) + (2^{2^{t-1}} - 1),$$

and therefore,

$$a^{2^{2^t}-1} = (a^{2^{2^{t-1}}-1})^{2^{2^{t-1}}} a^{2^{2^{t-1}}-1}.$$

To simplify the expression, let $S_t(a) = a^{2^{2^t}-1}$. Then we have that

$$S_t(a) = (S_{t-1}(a))^{2^{2^{t-1}}} S_{t-1}(a),$$

meaning that we can compute $S_t(a)$ by recursively computing $S_i(a)$ for $i \in [0, t-1]$ and multiplying and squaring appropriately, as shown in Algorithm 9.

This algorithm is analogous to [27, Algorithm 2] that computes $a^{-1} \in \mathbb{F}_{2^\ell}$ for $\ell = 2^t + 1$ through Fermat's Little Theorem as:

$$a^{-1} = a^{2^\ell-2} = (a^{2^{\ell-1}-1})^2 = (a^{2^{2^t}-1})^2.$$

Algorithm 9 Computing a^{2^k-1} where $k = 2^t$

```
    Input: $a$
    Output: $a^{2^k-1}$
procedure CUSTOM_EXPONENTIATION($a$)
    $f = a$
    for $i = 0$ to $t-1$ do
        $g = f^{2^{2^i}}$
        $f = f \cdot g$
    return $f$
```

BIKE, on the other hand, operates in the polynomial ring $\mathcal{R}$ with a value r for which the binary polynomial $(x^r - 1)$ factors into two irreducible factors of degree 1 and $r-1$, namely, $(x-1)$ and the cyclotomic polynomial Φ_{r-1} which we denote here by h. In this ring, $ord(a) \mid 2^{r-1} - 1$ for every $a \in \mathcal{R}^*$, and therefore by Fermat's theorem the inverse of a satisfies:

$$a^{-1} = a^{2^{r-1}-2}. \tag{5.1}$$

However, to compute the inverse we cannot use the ITI algorithm directly because $a^{2^{r-1}-2} = (a^{2^{r-2}-1})^2$ and $r-2$ is not a power of 2 in our case. Therefore, we use the following decomposition.

Decomposition of $2^{r-1} - 2$**.** In order to be able to apply Algorithm 9, we write $s = supp(r-2)$

and rewrite $z = 2^{r-1} - 2$ in a convenient way:

$$z = 2\cdot(2^{r-2}-1) = 2\cdot\sum_{i\in s}\left((2^{2^i}-1)\cdot\left(2^{(r-2) \bmod 2^i}\right)\right). \tag{5.2}$$

Example 4. For $r = 11779$ we have that $2^{r-1} - 2$ can be written as:

$$\begin{aligned} 2^{11778} - 2 = 2\cdot(1 + 2(2^{512}-1) + 2^{513}(2^{1024}-1) + {} \\ 2^{1537}(2^{2048}-1) + 2^{3585}(2^{8192}-1)) \end{aligned}$$

With this decomposition and Algorithm 9 we can build the algorithm for inverting an invertible element of the ring $\mathscr{R}$. In Algorithm 10 we present the method.

Algorithm 10 Inversion in $\mathscr{R} = \mathbb{F}_2[x]/((x-1)h)$ with an irreducible h

```
    Input: $a \in \mathscr{R}^*$
    Output: $a^{-1}$
 1: procedure INVERT($a$)
 2:     $f = a$
 3:     $res = a$
 4:     for $i = 1$ to $\lfloor\log(r-2)\rfloor$ do
 5:         $g = f^{2^{2^{(i-1)}}}$                              ▷ As in Alg. 9
 6:         $f = f \cdot g$
 7:         if $((r-2)_i = 1)$ then                     ▷ $i^{th}$ bit of $r-2$
 8:             $res = res \cdot f^{2^{(r-2) \bmod 2^i}}$
 9:     $res = res^2$
10:     return $res$
```

Algorithm 10 requires $\lfloor\log(r-2)\rfloor + wt(r-2) - 1$ multiplications plus $\lfloor\log(r-2)\rfloor + wt(r-2) - 1$ k-squarings and 1 squaring (all operations in $\mathscr{R}$). Therefore, the performance of the inversion depends on $|r-2|$ and on $wt(r-2)$, where, obviously, choices of r with smaller $|r-2|$ and $wt(r-2)$ lead to faster execution. Following Example 4 given above, for $r = 11779$ we can execute the algorithm with 17 polynomial multiplications, 17 k-squarings and 1 squaring.

Remark 5. By changing line 8 of Algorithm 10 into

$$res = res \cdot f^{2^{1+(r-2) \bmod 2^i}}$$

the last square, in line 9, can be removed. This optimization is omitted from the algorithm's description for clarity.

Efficient k-squaring. The straightforward way to implement the k-squaring routine is as a series of k regular squares. The operation of squaring a binary polynomial is very efficient, i. e., it can be done in r bit operations, whereas multiplication takes r^2 bit-operations if implemented naively, or $r^{\log_2 3}$ if the Karatsuba algorithm is used (note that the bit-operation numbers are correct up to a multiplication by a constant). However, the size of k in our k-

squaring operations is $O(r)$ which means that the k-squaring would require r^2 bit-operations. Moreover, modern CPU architectures offer an instruction that multiplies two 64-bit words that represent two binary polynomials in just a few cycles. With this instruction the runtime of multiplication and squaring in $\mathscr{R}$ is actually $(r/64)^{\log_2 3}$ and $r/64$ word-operations, respectively. Note that this improvement does not fully translate to the k-squaring since the number k of required consecutive squares stays the same, resulting in a total of $r^2/64$ word-operations. Therefore, this approach does not yield an efficient algorithm. Furthermore, it underlines the imbalance of the performance of the two operations required for the inversion – multiplication and k-squaring.

Fortunately, in the context of QC-MDPC codes used in post-quantum cryptographic schemes we can perform the k-squaring more efficiently by exploiting the following observation. Let $a = \sum_{j \in supp(a)} x^j \in \mathscr{R}^*$. Then we have that

$$\begin{aligned} a^{2^k} = \left(\sum_{j \in supp(a)} x^j \right)^{2^k} &= \sum_{j \in supp(a)} (x^j)^{2^k} \\ &= \sum_{j \in supp(a)} x^{j \cdot 2^k} = \sum_{j \in supp(a)} x^{j \cdot 2^k \bmod r}. \end{aligned} \tag{5.3}$$

The first step in Equation 5.3 is an identity for polynomials with binary coefficients. The last step stems from the fact that the order of $x \in \mathscr{R}$ is $ord(x) = r$. Therefore, k-square of an element in $\mathscr{R}$ can be computed as a permutation of the bits of the element. The only remaining question is how performant can be a secure implementation of the permutation, while at the same time the implementation admits the standard properties of side-channel protection, i. e., it is constant-time and constant-memory access.

5.3 Our implementation

In this section we discuss our implementation of Algorithm 10 and optimizations that significantly reduce the running time of the algorithm. Moreover, we describe an optimization of the polynomial multiplication code used in the BIKE Additional code package [45].

Element of the ring $\mathscr{R}$ in the source code of BIKE is represented as an array of $r_{size} = \lceil r/8 \rceil$ bytes, where each byte represents eight consecutive coefficients and consecutive bytes represent consecutive blocks of eight bits. The first attempt to implement the k-squaring of a is the naive implementation of the appropriate permutation, as shown in Algorithm 11. Namely, the algorithm iterates over the bits of a and copies each bit to an appropriate position of the output array c, where the position in c is computed as defined by Equation 5.3. The algorithm can be divided into two parts – generating the permutation map $(\pi(i) : i \longrightarrow i \cdot 2^k \bmod r)$ for $i \in [0, r-1]$ and applying the map to the input array.

"Inverted" permutation. Note that in Algorithm 11 for every byte of the result we perform one memory read (in line 5) and eight memory writes (in line 9). Furthermore, with the

Algorithm 11 Computing k-square as permutation

```
    Input: $a$ as an array of $r_{size}$ bytes, $k$
    Output: $c = a^{2^k}$ as an array of $r_{size}$ bytes
 1: procedure K_SQUARE($a$, $k$)
 2:     for $i = 0$ to $r - 1$ do                    ▷ Generate the permutation map
 3:         $map[i] = (i \cdot 2^k)$ % $r$
 4:     for $i = 0$ to $r_{size} - 1$ do             ▷ Apply the permutation map
 5:         $byte = a[i]$
 6:         for $j = 0$ to 7 do
 7:             $bit = (byte >> j)$ & 1
 8:             $pos = map[i \cdot 8 + j]$
 9:             $c[pos/8] \mid= (bit << (pos\%8))$
10:     return $c$
```

required memory reads we are accessing consecutive memory locations ($a[i]$ for i from 0 to r_{size}), while the memory writes are to random locations determined by the value of *pos* variable. However, we can turn things around by computing the "inverted" permutation of a's coefficients $\pi^{-1}(i): \; i \cdot 2^{-k} \bmod r \longrightarrow i$, as shown in Algorithm 12. In terms of implementation we change the map generation such that each element of the map holds the position of the bit of a that should be copied to c at position determined by the index of the map element. This is achieved by simply switching the value and the index that it is written to in line 3 of the algorithm. Then, we iterate over the bits of the output array and read the required values from the appropriate positions in the input array. In this way, for every byte of the result we perform one memory write and eight memory reads. This approach improves the performance of the implementation in a noticeable way, so we choose to use it in the implementation.

Remark 6. Division of a value by 8 and reduction modulo 8 in Algorithms 11 and 12 are implemented as right shift by 3 positions and bitwise *and* of the value with 7, respectively.

Efficient generation of a permutation map. Given k, the permutation map of the corresponding k-square is computed by storing the value $i \cdot 2^k \bmod r$ to $map[i]$ for $i \in [0, r-1]$ (hereafter, for brevity we use *map* to denote the *inverse_map* from Algorithm 12). Firstly, we note that the value $l = 2^k \bmod r$ can obviously be precomputed. Then for each map element we perform only two operations, multiplication and reduction modulo r, that are both costly CPU instructions (especially the reduction). However, we can easily generate the map by using only addition and subtraction. More precisely, the value at position i in the map, $map[i]$, can be computed as $(map[i-1] + l)$. If this sum is greater or equal than r we simply subtract r from it and store it, otherwise, there is no need for subtraction since the value is already smaller than r. Note that the same approach is possible for generating the map of the "inverted" permutation. We compute $l = 2^{-k} \bmod r$ and apply the same algorithm – set $map[i]$ to the value $(map[i-1] + l)$ and subtract r if necessary. Furthermore, this algorithm can easily be vectorized with SIMD instructions, as explained in Section 5.3.1.

Precomputed maps. The actual values of k in all the k-squarings of Algorithm 10 depend only

Algorithm 12 Computing k-square as "inverted" permutation

```
Input: $a$ as an array of $r_{size}$ bytes, $k$
Output: $c = a^{2^k}$ as an array of $r_{size}$ bytes
procedure K_SQUARE($a$, $k$)
    for $i = 0$ to $r - 1$ do                          ▷ Generate the permutation map
        $inverse\_map[(i \cdot 2^k) \% r] = i$
    for $i = 0$ to $r_{size} - 1$ do                   ▷ Apply the permutation map
        $val = 0$
        for $j = 0$ to $7$ do
            $pos = inverse\_map[i \cdot 8 + j]$
            $byte = a[pos/8]$
            $bit = (byte >> (pos\%8))\ \&\ 1$
            $val \mid= (bit << j)$
        $c[i] = val$
    return $c$
```

on r, not on the input a. Since r is a fixed public parameter of the BIKE cryptosystem, all the relevant values of k involved in the inversion algorithm can be precomputed. Moreover, for each k we can precompute the whole permutation map. Using the precomputed maps instead of generating them on the fly speeds up the implementation. However, this performance improvement comes at a cost of storing all the maps in memory and accessing them frequienlty. The required storage is $\lfloor \log(r-2) \rfloor + 1 + wt(r-2)$ tables where each one holds r values. The trade-off between the performance and the memory footprint of the code is discussed in Section 5.4.

k-square versus k squares. As already noted, squaring a polynomial in $\mathcal{R}$ is very efficient and significantly faster than a k-squaring (see Appendix A.3.4). This observation leads to the following optimization: for values of k smaller than a certain threshold k_{thr} we compute a^{2^k} as a series of k regular squares instead of executing the k-square routine. The k_{thr} value should be chosen such that it gives the faster of the two options for a given k. The choice of k_{thr} obviously depends on the actual performance of the implementation of square and k-square on a specific processor. To this end, in Table A.3 in the appendix we provide an example of the optimal k_{thr}.

The consequence of this optimization is that in addition to $r-2$ and $wt(r-2)$, the efficiency of inversion depends on the number of k-squares that can be replaced with a series of regular squares. For example, consider $r_1 = 11779$ and $r_2 = 12347$. Here, inverting a polynomial in $\mathcal{R}_1$ is expected to be faster than in $\mathcal{R}_2$, because $wt(r_1 - 2) = 5 < 6 = wt(r_2 - 2)$, and the number of required k-squares is smaller. However, from the binary representations $r_1 - 2 =$ `0b10111000000001` and $r_2 - 2 =$ `0b11000000111001`, we see that the set bits in $r_2 - 2$ are positioned close to the LSB, and the set bits in $r_1 - 2$ are positioned close to the MSB. Therefore, if $k_{thr} = 64$, then for the inversion in $\mathcal{R}_1$ we can replace only one k-square with a chain of squares, while in case of $\mathcal{R}_2$ we can replace 4 such k-squares. This is another consideration that should be taken into account when choosing the r parameter for the scheme (as discussed in Section 5.5).

5.3.1 Generating permutation map with SIMD instructions

The first target for optimization with SIMD instructions is the permutation map generation. We have already explained in the previous section how to generate the map without using time consuming instructions such as multiplication and modular reduction which are also very inefficient in SIMD settings. We proceed by implementing this approach with **AVX** instructions.

In Listing 5.1, we present the implementation of the map generation function with the **AVX512** instruction set which is quite straightforward. The function receives the l parameter as input, where $l = 2^{-k} \bmod r$, as previously explained. Note that the values of r that are relevant for BIKE are $r < 2^{15}$. Therefore, the map can be represented by an array of length r of 16-bit unsigned integers and a single **AVX512** register can hold 32 map elements. Importantly, since $r < 2^{15}$, a sum of two values smaller than r is less than 2^{16}, thus the sum fits in a 16-bit register. In the first step we initialize one 512-bit register (*prev*) with values representing the first 32 elements of the map and broadcast values $l * 32 \bmod r$ and r to registers *inc* and *rval*, respectively (broadcasting a value to a vector register means setting the value in all elements of the register). Then we proceed with generating the remaining part of the map, 32 map values at a time. To obtain the currently processed map values in register *curr* we add the increment *inc* to the previous vector *prev*. Then we compare the corresponding elements of the two vectors *curr* and *rval* to generate the 32-bit mask where a bit is set to one if the corresponding elements of *curr* and *rval* satisfy the condition, otherwise the bit is set to zero (the condition being $curr[i] \geq rval[i]$). Finally, by using a convenient *masked* version of the **AVX512** subtraction instruction and the generated mask we subtract r from the appropriate elements of the vector and store the values in the map.

Remark 7. Note that all the **AVX** instructions operate on vectors element-wise. Also, **AVX512** offers a *masked* version of almost every instruction, where a mask can be supplied to the instruction to denote which vector elements the operation should or should not be applied to.

```c
void gen_permutation_map (uint16_t map[R], uint16_t ell) {
  __m512i curr, prev, inc, rval;
  uint32_t mask;
  // Initialization: compute the first 32 map elements
  for (int i = 0; i < 32; i++)
    map[i] = (i * ell) % R;
  prev = LOAD(map); // Load the 32 values into the register
  inc  = BCAST_U16((ell * 32) % R);
  rval = BCAST_U16(R);

  // Generate the rest of the map elements
  for (int i = 1; i < ceil(R / 32); i++) {
    curr = ADD_U16(prev, inc);
    mask = CMP_U16(curr, rval, CMP_GEQ);
    curr = SUB_U16(curr, rval, mask);
    STORE(&map[i * 32], curr);
  }
}
```

Listing 5.1 – Permutation map generation with **AVX512** instructions (the actual names of **AVX512** instructions are replaced with upper-case macro names for clarity)

We point out that in the case of **AVX2**, the older SIMD standard that uses 256-bit vectors, the implementation requires a few modifications. The first reason is that **AVX2** does not support *masked* instructions. Moreover, the comparison of two vectors (element-wise) does not produce a bitmask as in **AVX512** but rather another vector with the corresponding elements appropriately set to zero or all ones based on the supplied condition. Hence, the subtraction has to be done with two instructions – first we *and* the mask vector and *rval* and then subtract the result from *curr* vector. Another problem is that the comparison function compares vector elements as signed integers, not as unsigned integers as in **AVX512** case. This means that, if $r > 2^{14}$, adding two values can result in a number greater than 2^{15} which is actually a negative number when interpreted as a signed 16-bit integer, and therefore, the comparison (which works only on signed numbers) would not produce the desired result. To sidestep this limitations we use the following trick: in the initialization phase we subtract r from the elements of the increment vector; in the second phase, when we compute *curr* by adding *inc* to *prev* we also produce a *mask* register by comparing the elements of *curr* with zero (i. e., checking which elements are negative) and finally add *rval* to the appropriate elements (by using the *and* of *mask* and *rval*). The code implementing the described algorithm is given in Appendix A.3.1.

We note that the implementations can be further optimized by processing two (or more) **AVX** registers (containing map elements) at a time, i. e., if in Listing 5.1 we use several *curr* and *prev* registers. This allows the processor to eliminate any latency coming from the sequential nature of the instructions in the for loop by executing the instructions in a more favourable order and thus filling the execution pipeline. Furthermore, vector processing units usually have two input and output ports which allows them to execute some pairs of instructions in parallel and achieve higher throughput (for example, simple arithmetic instructions such as addition, subtraction, etc., can be executed concurrently). With this, our implementation achieves further performance improvements.

To conclude, SIMD implementations of the function that generates a permutation map are fairly efficient. In the **AVX2** case we need an order of $r/16$ vector instructions to generate the whole map, while the performance of the **AVX512** implementation is even better since the required number of instructions is an order of $r/32$.

5.3.2 Optimizing the permutation with SIMD instructions

The next optimization target is the second phase of the k-square algorithm (Algorithm 12) – the application of a given permutation map. Recall that both the input polynomial and the result of the algorithm are given in *binary representation*, i. e., a polynomial is represented by a byte array of length r_{size}, each byte holding 8 bits of data. Because of that, applying the permutation map to a single coefficient involves several instructions. Namely, given a position *pos* of a bit we need to compute the byte position in the array that the bit belongs to (one shift instruction), to extract the bit from the byte (one shift and two and instructions), and finally write the extracted bit to the byte of the output (one shift and one or instruction). These are all "light" operations, i. e., most processors perform them in a single cycle. However, they are executed for each of the r coefficients of the polynomial, and therefore, eliminating even some of them can significantly reduce the runtime of the algorithm.

We note that if a polynomial was given in *byte representation* as array of r bytes (instead of $r_{size} = \lceil r/8 \rceil$ bytes) where each byte holds one binary coefficient of the polynomial, then the application of the map would require a single memory transfer without any arithmetic operation. To illustrate, given a map and the input and output polynomials stored as described, the permutation is applied by the following procedure: $out[i] = in[map[i]]$ for $i \in [0, r-1]$. In this way we would be able achieve the goal of eliminating as many instructions as possible.

Unfortunately, in between calls to the k-square function in the inversion algorithm we perform polynomial multiplication (and reduction) which operates on polynomials in *binary representation* and which would be very inefficient if it operated on polynomials in *byte representation*. Because of this we can not perform the whole inversion with *byte represented* polynomials. Hence, we are forced to convert the polynomials from one to the other representation and back for each invocation of the k-square function. Fortunately, the required conversions can be done very efficiently with SIMD instructions.

AVX512 implementation

To convert from *binary* to *byte* representation we use the **_mm512_maskz_set1_epi8** instruction [57], as shown in Listing 5.2. This instruction accepts two parameters: a 64-bit *mask* and an 8-bit *val*, and produces a vector register that contains 64 elements each of size 8 bits by broadcasting *val* to all elements of the resulting vector using zeromask *mask*, i. e., element of the vector is zeroed out when the corresponding bit of the mask is not set. Therefore, we process the input polynomial 64 bits at a time, where we use the 64 bits as the *mask* and set $val = 1$. In this way, when a bit of *mask* is set (i. e., the polynomial coefficient is one) then the corresponding byte in the output vector is set to one, otherwise it is set to zero. With this, the conversion is done.

```
void convert_bin_byte (uint8_t out[R], uint8_t in[R_SIZE]) {
  // Consider the input as an array of 64-bit elements
  uint64_t *in64 = (uint64_t*)in;

  for (int i = 0; i < ceil(R / 64); i++) {
    // Convert 64 bits to byte representation
    __m512i t = _mm512_maskz_set1_epi8(in64[i], 1);
    STORE(&out[i * 64], t); // Store the resulting 64 bytes to the output
  }
}
```

Listing 5.2 – Conversion of a polynomial from *binary* to *byte* representation using **AVX512** instructions.

The conversion in the opposite direction, *byte* to *binary* representation, can be done with **_mm512_cmp_epi8_mask** instruction [57], as shown in Listing 5.5. Recall from the previous section that the comparison instructions in **AVX512** receive two vectors and produce the output mask by comparing the corresponding elements of the vector. More precisely, the specified instruction takes two vectors viewed as arrays of 64 bytes compares the bytes in the corresponding positions and if they are equal sets the corresponding bit in the output mask to one. To realize the conversion, we use this instruction and provided it with a vector register containing bytes of the polynomial (each byte is zero or one) and a register where we set all bytes to one.

```
void convert_byte_bin (uint8_t out[R_SIZE], uint8_t in[R]) {
  // Consider the output as an array of 64-bit elements
  uint64_t *out64 = (uint64_t*)out;
  for (int i = 0; i < ceil(R / 64); i++) {
    // Convert 64 bytes of the input
    // and store the resulting 64 bits to the output
    __m512i one = BCAST_U8(1);
    __m512i t = LOAD(&in[i * 64]);
    out64[i] = _mm512_cmp_epi8_mask(t, one, CMP_EQ);
  }
}
```

Listing 5.3 – Conversion of a polynomial from *byte* to *binary* representation using **AVX512** instructions.

AVX2 implementation

In the case when only **AVX2** is available the conversion is slightly more complicated because the two instructions we used for the **AVX512** implementation are not available in the **AVX2** specification. Therefore, we need to resort to various tricks to build an efficient implementation of the two conversion functions.

The register size in **AVX2** extension is 256 bits, e. g., a register can be viewed as holding 32 byte-size elements or 8 elements of size 32 bits. We implement the conversion from *binary* to *byte* representation by converting 32 bits of the input to 32 bytes of the output at a time. The algorithm is depicted in Figure 5.1. Let $val = a_3 a_2 a_1 a_0$ be the 32-bit value (consisting of four bytes a_i) that we convert to *byte* representation. We start by broadcasting *val* to the eight elements of the vector register t. Ideally, we would then shuffle the byte-size elements in t such that the i-th element contains the byte of *val* which contains the i-th bit of *val*, e. g., elements of t at positions 0 to 7 are set to a_0, at positions 8 to 15 are set to a_1, etc. Once we have a_i's ordered like this we can obtain the result by appropriately shifting each element of t such that the desired bit is shifted to the most significant position of the element, e. g., shift i-th element of t to the left by $(7-i)$ mod 8 bit positions. Unfortunately, **AVX2** does not support shift operations on byte-sized elements of a register, but only on 32-bit and 64-bit sized elements.

Remark 8. Note that, contrary to the **AVX512** implementation where each byte in the *byte* representation holds the corresponding bit in the least significant bit position, here, we want the bit to be in the most significant position. The reason for this change is that it allows simple and efficient implementation of the conversion in the opposite direction, as explained later in the text.

The above stated limitations of the **AVX2** instruction set are overcome in the following way. The explanation follows the procedure in Figure 5.1. Note that for clarity we depict only the conversion of a_0, the first 8 bits of *val*, however, the remaining part of *val* is simultaneously processed. First, we shift the 32-bit elements of t by the values provided in register q, e. g., the first two elements, $t[0]$ and $t[1]$ are shifted to the left by 6 and 4 places, respectively. Note that by shifting, for example, the 32-bit value $t[0] = a_3 a_2 a_1 a_0$ to the left by 6 places, we obtain $t[0] = a_3' a_2' a_1' a_0'$ where $a_i' \neq a_i << 6$, but the most significant bit of a_i' is equal to the most significant bit of $a_i << 6$ (we denote this by the "$\sim$" sign in Figure 5.1). Then we use the **AVX2** shuffle instruction to reorder the byte-sized elements of t as shown in the figure. Thus, we obtain register t with the bytes in odd positions exactly as we need them for the output – they hold values with most significant bit set to the value of the corresponding bit in *val*, e. g., byte at position 1 holds the value $t_0 \sim (a_0 << 6)$. The bytes of t in even positions have to be shifted once more to the left by one place, to obtain register s that has even positioned elements filled with the right values. The resulting register is then simply generated by blending t and s. For this we use the **AVX2** blend instruction which we provide with a mask such that it copies elements at odd positions from t and at even positions from s.

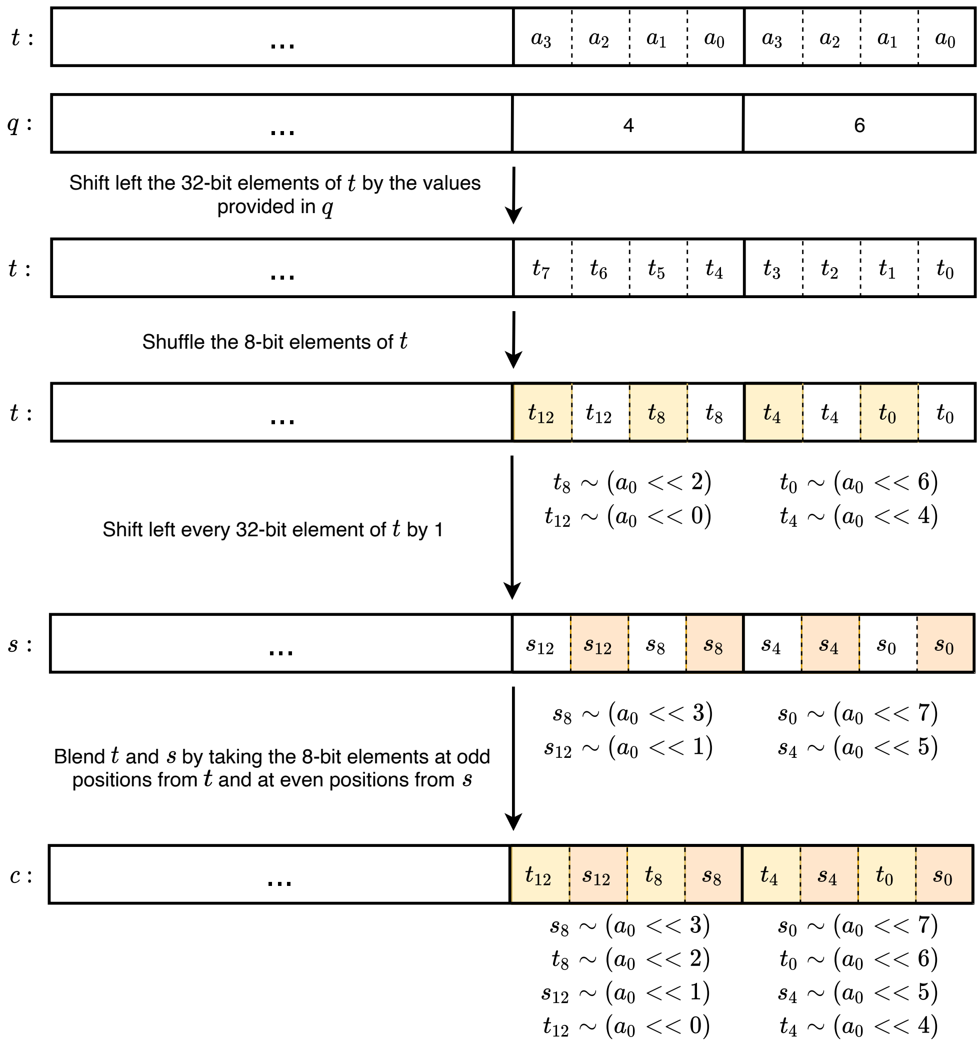

t :
...
a_3 a_2 a_1 a_0 a_3 a_2 a_1 a_0
q :
...
4
6
Shift left the 32-bit elements of t by the values provided in q
t :
...
t_7 t_6 t_5 t_4 t_3 t_2 t_1 t_0
Shuffle the 8-bit elements of t
t :
...
t_{12} t_{12} t_8 t_8 t_4 t_4 t_0 t_0
$t_8 \sim (a_0 << 2)$
$t_{12} \sim (a_0 << 0)$
$t_0 \sim (a_0 << 6)$
$t_4 \sim (a_0 << 4)$
Shift left every 32-bit element of t by 1
s :
...
s_{12} s_{12} s_8 s_8 s_4 s_4 s_0 s_0
$s_8 \sim (a_0 << 3)$
$s_{12} \sim (a_0 << 1)$
$s_0 \sim (a_0 << 7)$
$s_4 \sim (a_0 << 5)$
Blend t and s by taking the 8-bit elements at odd positions from t and at even positions from s
c :
...
t_{12} s_{12} t_8 s_8 t_4 s_4 t_0 s_0
$s_8 \sim (a_0 << 3)$
$t_8 \sim (a_0 << 2)$
$s_{12} \sim (a_0 << 1)$
$t_{12} \sim (a_0 << 0)$
$s_0 \sim (a_0 << 7)$
$t_0 \sim (a_0 << 6)$
$s_4 \sim (a_0 << 5)$
$t_4 \sim (a_0 << 4)$

Figure 5.1 – Conversion of a 32-bit value $a_3a_2a_1a_0$ consisting of four bytes from *binary* to *byte* representation with **AVX2** instructions.

Therefore, *binary* to *byte* conversion of a polynomial is performed by applying the described algorithm to every 32 consecutive bits of the polynomial, as shown in Listing 5.4.

```
void convert_bin_byte (uint8_t out[R], uint8_t in[R_SIZE]) {
  // shift values (q), shuffle mask (p), blend mask (w)
  __m256i p, q, w, t;
  q = SET_I32(0, 2, 4, 6, 0, 2, 4, 6);
  p = SET_I8(15, 15, 11, 11, 7, 7, 3, 3, 14, 14, 10, 10, 6, 6, 2, 2
             13, 13, 9, 9, 5, 5, 1, 1, 12, 12, 8, 8, 4, 4, 0, 0);
  w = BCAST_I16(0x00ff);

  // Consider the input as an array of 32-bit elements
  uint32_t *in32 = (uint32_t*)in;
  for (int i = 0; i < ceil(R / 32); i++) {
    // Convert 32 bits to byte representation
    t = BCAST_I32(in32[i]);
    t = _mm256_sllv_epi32(t, q); // shift left elements of t by vals in q
    t = _mm256_shuffle_epi8(t, p);
    s = _mm256_slli_epi32(t, 1); // shift left each element of t by 1
    t = _mm256_blendv_epi8(t, s, w); // blend t and s
    STORE(&out[i * 32], t); // Store the resulting 32 bytes to the output
  }
}
```

Listing 5.4 – Conversion of a polynomial from *binary* to *byte* representation using **AVX2** instructions.

Conversion from *byte* to *binary* representation is straightforward thanks to the fact that, as previously noted, the *byte* representation is such that each byte holds the corresponding bit in the most significant position. To convert 32 consecutive bytes we use **_mm256_movemask_epi8** instruction available in the **AVX2** instruction set [57]. The instruction takes a vector register as input (consisting of 32 byte-sized elements) and creates a 32-bit mask from the most significant bit of each element of the register. Since this is the exact functionality that we need for the conversion, we simply iterate over the coefficients of the *byte* represented polynomial, 32 bytes at a time, and generate the desired 32 bits of output (the implementation is shown in Listing 5.5).

```
void convert_byte_bin (uint8_t out[R_SIZE], uint8_t in[R]) {
  // Consider the output as an array of 32-bit elements
  uint32_t *out32 = (uint32_t*)out;
  for (int i = 0; i < ceil(R / 32); i++) {
    // Convert 32 bytes of the input
    // and store the resulting 32 bits to the output
    __m512i t = LOAD(&in[i * 32]);
    out32[i] = _mm256_movemask_epi8(t);
  }
}
```

Listing 5.5 – Conversion of a polynomial from *byte* to *binary* representation using **AVX2** instructions.

5.3.3 Optimizing squaring and multiplication

Modern CPUs offer a fast carry-less multiplication instruction (**PCLMUL**) that can be used for multiplying two elements of a field with characteristic 2. We note that **PCLMUL** multiplies two 64-bit inputs and produces a 128-bit result. Since **AVX512** and **AVX2** offer many instructions that can operate on wider registers (512-bit and 256-bit, respectively), **PCLMUL** can be a bottleneck when polynomial multiplication is implemented with one of these SIMD instruction extension sets.

In the recent 10^{th} generation CPUs (codename "Ice Lake") Intel introduced a vectorized version of the **PCLMUL** instruction, namely **VPCLMUL**, which can multiply simultaneously four pairs of 64-bit inputs (in a SIMD manner). We leverage the new instruction to improve the performance of the existing polynomial multiplication in BIKE and also to implement polynomial squaring required for the inversion. Figure 5.2 shows how **VPCLMUL** instruction works. It receives two 512-bit registers a and b, each containing eight 64-bit elements, and a mask. The elements of a and b are grouped into four groups of two elements. The mask determines which elements (lower or higher) of the corresponding groups will be multiplied, as illustrated in the figure. Finally, the specified elements are multiplied and four 128-bit products are stored in the output register.

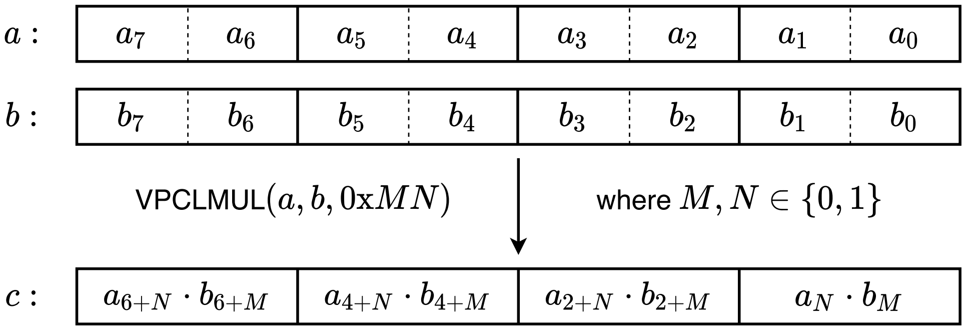


Figure 5.2 – **VPCLMUL** instruction.

Binary polynomial squaring with VPCLMUL

Squaring a polynomial with binary coefficients is particularly efficient. Let $a \in \mathscr{R}$ be the polynomial $a = \sum_{i=0}^{r-1} a_i x^i$. Then its square is $a^2 = \left(\sum_{i=0}^{r-1} a_i x^i\right)^2 = \sum_{i=0}^{r-1} \left(a_i x^i\right)^2$ since the coefficients of a are in $\mathbb{F}_2$. Therefore, squaring can be performed by squaring every term of the input polynomial. We note that this can be implemented with the non-vectorized **PCLMUL** instruction in a straightforward manner. Consider the polynomial as consisting of sub-polynomials with 64 terms, i. e., $a = \sum_{i=0}^{\lceil r-1/64\rceil} a_i(x)x^{i\cdot 64}$ where $a_i(x) \in \mathbb{F}_2[x]$ with degree at most 63. Hereafter, we refer to polynomials $a_i(x)$ as 64-bit digits of a. Squaring is then implemented by iterating over the digits of a and squaring them with **PCLMUL**. The code that implements this functionality is given in Appendix A.3.2. In this section we focus on using **VPCLMUL** instruction for squaring.

In Figure 5.3 we show how to square a binary polynomial of 512 bits length to obtain the 1024-bit result with **VPCLMUL** instruction. We consider the polynomial as consisting of eight 64-bit digits which occupy **AVX512** register a. To be able to get the digits in the resulting registers in correct order, we first need to permute the elements of a, as shown in the figure. Then we invoke **VPCLMUL** instruction twice, with mask **0x00** and **0x11**, to square the lower and the higher elements of the four 128-bits parts of a, respectively. In this way we obtain the result in two registers c_{lo} and c_{hi} with their elements appropriately ordered such that we can simply store them in memory. Squaring a polynomial of size r is done by iterating over it, 512 bits at a time, and applying the described algorithm (the source code of this function is given in Listing A.4 in the appendix). After computing the square of a polynomial (or a product of two polynomials) we need to reduce the result modulo $x^r - 1$. The implementation of the reduction is not presented, but we note that since $x^r = 1$ the reduction can be done by shifting to the right by r places the higher part of the result (bits at positions $\geq r$) and adding it to the lower part of the result (bits at positions 0 to $r - 1$).

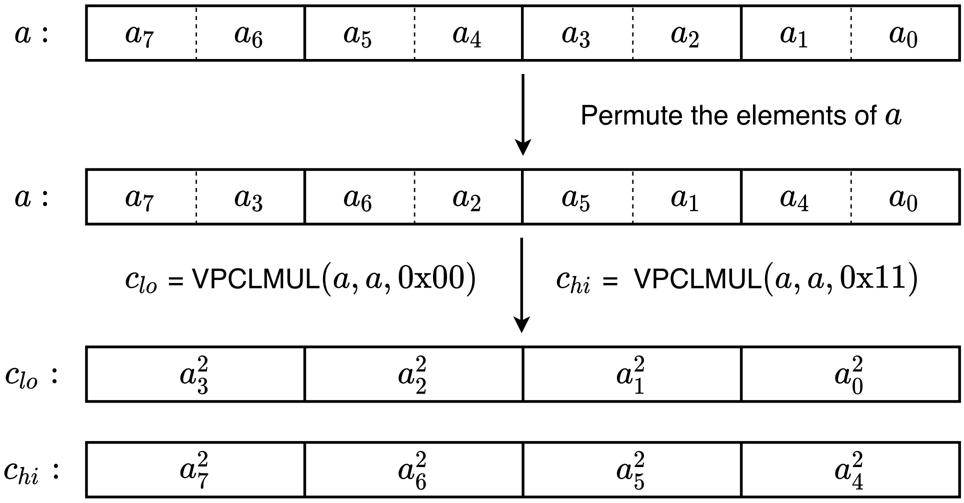


Figure 5.3 – Squaring eight consecutive 64-bit digits of a binary polynomial with **VPCLMUL** instruction.

Binary polynomial multiplication with VPCLMUL

The “Additional implementation” of BIKE [45], submitted to the second round of the NIST Post-Quantum standardization project, implements polynomial multiplication with the recursive Karatsuba algorithm. The recursion splits the input into two equally sized parts, proceeds with multiplying the new parts individually in the same manner, and stops when inputs of size four 64-bit digits are encountered. Then, the *base case* multiplication is performed with a 4 × 4 64-bit digits schoolbook multiplication algorithm (using the **PCLMUL** instruction). Since the new **VPCLMUL** instruction operates on 512-bit registers, we replaced the existing *base case* multiplication with the code described in [61] that multiplies two binary polynomials of size eight 64-bit digits. This yields some improvements. However, we further optimize the code by implementing the *base case* as 16 × 16 digits multiplication using the Karatsuba algorithm with **AVX512** and **VPCLMUL** instructions.

We start by making a function that multiplies four digits of a with four digits of b:

$$c = a_3 a_2 a_1 a_0 \cdot b_3 b_2 b_1 b_0.$$

Recall that in Karatsuba's algorithm we would split the terms in half and compute the product as $c = x \cdot 2^{256} + y \cdot 2^{128} + z$ (first level of Karatsuba), with

$$\begin{aligned} x &= a_3 a_2 \cdot b_3 b_2 \\ y &= (a_3 a_2 + a_1 a_0) \cdot (b_3 b_2 + b_1 b_0) + x + z \\ z &= a_1 a_0 \cdot b_1 b_0, \end{aligned}$$

where each of the three products (in x, y, z) would be computed in the same way (second level), i. e., by splitting the terms and computing three sub-products. Therefore, to compute c we need in total nine single digit multiplications, which if done with **VPCLMUL** instruction (which performs four single digit multiplications in parallel), we would need three calls to **VPCLMUL**. This was our initial approach. However, since we are using **VPCLMUL** three times it means that we can actually perform twelve single digit multiplications instead of nine at the same cost. We use this fact to our advantage to implement a hybrid between Karatsuba and schoolbook multiplication which simplifies the algorithm and removes some additions (and register permutations). The idea is the following: replace the first level of Karatsuba by schoolbook multiplication and compute the sub-products by Karatsuba. Namely, we compute:

$$c = (a_3 a_2 \cdot b_3 b_2) \cdot 2^{256} + (a_3 a_2 \cdot b_1 b_0 + a_1 a_0 \cdot b_3 b_2) \cdot 2^{128} + a_1 a_0 \cdot b_1 b_0.$$

To compute the four sub-products with Karatsuba we need to obtain the following twelve products:

	$a_3 a_2 \cdot b_3 b_2$:	$a_3 a_2 \cdot b_1 b_0$:	$a_1 a_0 \cdot b_3 b_2$:	$a_1 a_0 \cdot b_1 b_0$:
(1)	$a_2 \cdot b_2$	$a_2 \cdot b_0$	$a_0 \cdot b_2$	$a_0 \cdot b_0$
(2)	$a_3 \cdot b_3$	$a_3 \cdot b_1$	$a_1 \cdot b_3$	$a_1 \cdot b_1$
(3)	$(a_2 + a_3)(b_2 + b_3)$	$(a_2 + a_3)(b_0 + b_1)$	$(a_0 + a_1)(b_2 + b_3)$	$(a_0 + a_1)(b_0 + b_1)$

After all the products are computed we have to perform several more additions. Firstly, since we are using Karatsuba's method to compute the four products $a_i a_j \cdot b_k b_l$, we need to add the first two computed terms to the third one, and then shift the third term (multiply by 2^{64}) and add it to the result. For example,

$$a_1 a_0 \cdot b_1 b_0 = a_1 \cdot b_1 \cdot 2^{128} + ((a_0 + a_1)(b_0 + b_1) + a_1 \cdot b_1 + a_0 \cdot b_0) 2^{64} + a_0 \cdot b_0.$$

Finally, we need to sum the two middle $a_i a_j \cdot b_k b_l$ products, and again, shift appropriately and add to obtain the final result.

To illustrate how the whole procedure is done with **AVX512** and **VPCLMUL** instructions we present Figure 5.4. The plan is to compute the four products in each row, denoted by (1), (2), (3) in the equations above, in parallel. Let the **AVX512** registers a and b hold the corresponding four 64-bit digits in the order as shown in the figure (this can be achieved with **AVX512** permutation function). First, we obtain the sums that are required to compute the products in the third row. This is done by shuffling the elements of a and b to get them ordered as shown in *sa* and *sb* in the figure and then by simply adding the values of a and b to *sa* and *sb*, respectively.

Now we can use the **VPCLMUL** instruction to compute all the required products and store them in registers u, v, and w. Note that w holds the products of the third row which represent the middle term in Karatsuba's algorithm so we add both u and v to w. For example, the lowest 128 bits of w hold two digits $w_1 w_0 = (a_0 + a_1)(b_0 + b_1) + a_1 \cdot b_1 + a_0 \cdot b_0$. Recall that to compute $(a_1 a_0 \cdot b_1 b_0)$ we need to add $w_1 w_0 \cdot 2^{64}$ to the sum $(a_1 \cdot b_1 \cdot 2^{128} + a_0 \cdot b_0)$. Since $w_1 w_0 \cdot 2^{64} = w_1 \cdot 2^{128} + w_0 \cdot 2^{64}$ this means that we can add $w_0 \cdot 2^{64}$ to the $a_0 \cdot b_0$ product (basically add w_0 to the higher digit of $a_0 \cdot b_0$) and add w_1 to the $a_1 \cdot b_1$ product. Products $a_0 \cdot b_0$ and $a_1 \cdot b_1$ are stored in $u_1 u_0$ and $v_1 v_0$, respectively. Therefore, we shuffle w to *sw* to obtain the elements of w ordered as shown in the figure and perform two additions – we add the elements of *sw* at odd positions to u, and the elements at even positions to v. With this the four Karatsuba multiplications are done. The only thing left to do is to permute the elements of u and v in the right order and store the result (this step is not shown in the figure).

The described algorithm to compute a 4 × 4 digits product is used as a function that is called inside the 8 × 8 digits Karatsuba multiplication. The 8 × 8 multiplication function takes care of providing correctly ordered input registers and also handles the output of the 4 × 4 multiplication. The source code which implements the 4 × 4 multiplication function is given in Listing A.5 in the Appendix A.3.3.

5.3.4 Side-channel protection considerations

The proposed polynomial inversion algorithm (Algorithm 10) is used during the key generation process in BIKE where a polynomial, which is a part of the secret key, has to be inverted. Because we are dealing with secret data the inversion has to be implemented securely. On the high level, the algorithm involves several polynomial multiplications and several k-squarings. We note that the number of these operations and the order in which they are performed depend only on the public parameter r (not on a given input). Therefore, the algorithm is inherently constant-time and if the required subroutines are implemented securely, then the algorithm itself is secure without any modification.

The subroutines used in the inversion algorithm are the following: multiplication, squaring,

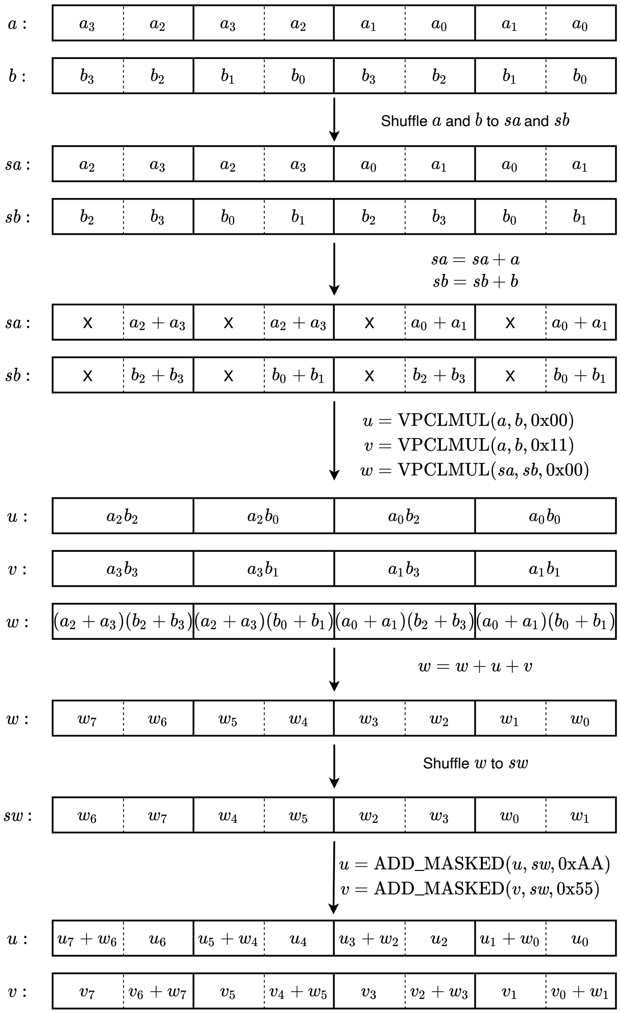

$a:$ a_3 a_2 a_3 a_2 a_1 a_0 a_1 a_0
$b:$ b_3 b_2 b_1 b_0 b_3 b_2 b_1 b_0
Shuffle a and b to sa and sb
$sa:$ a_2 a_3 a_2 a_3 a_0 a_1 a_0 a_1
$sb:$ b_2 b_3 b_0 b_1 b_2 b_3 b_0 b_1
$sa = sa + a$
$sb = sb + b$
$sa:$ X $a_2 + a_3$ X $a_2 + a_3$ X $a_0 + a_1$ X $a_0 + a_1$
$sb:$ X $b_2 + b_3$ X $b_0 + b_1$ X $b_2 + b_3$ X $b_0 + b_1$
$u = \mathrm{VPCLMUL}(a, b, 0x00)$
$v = \mathrm{VPCLMUL}(a, b, 0x11)$
$w = \mathrm{VPCLMUL}(sa, sb, 0x00)$
$u:$ a_2b_2 a_2b_0 a_0b_2 a_0b_0
$v:$ a_3b_3 a_3b_1 a_1b_3 a_1b_1
$w:$ $(a_2 + a_3)(b_2 + b_3)$ $(a_2 + a_3)(b_0 + b_1)$ $(a_0 + a_1)(b_2 + b_3)$ $(a_0 + a_1)(b_0 + b_1)$
$w = w + u + v$
$w:$ w_7 w_6 w_5 w_4 w_3 w_2 w_1 w_0
Shuffle w to sw
$sw:$ w_6 w_7 w_4 w_5 w_2 w_3 w_0 w_1
$u = \mathrm{ADD\_MASKED}(u, sw, 0xAA)$
$v = \mathrm{ADD\_MASKED}(v, sw, 0x55)$
$u:$ $u_7 + w_6$ u_6 $u_5 + w_4$ u_4 $u_3 + w_2$ u_2 $u_1 + w_0$ u_0
$v:$ v_7 $v_6 + w_7$ v_5 $v_4 + w_5$ v_3 $v_2 + w_3$ v_1 $v_0 + w_1$

Figure 5.4 – Multiplying four 64-bit digits of two binary polynomials using **AVX512** and **VPCLMUL** instructions.

and k-squaring. The Additional code of BIKE [45] already implements multiplication in constant-time and the optimizations we introduce in this chapter follow the same secure implementation practices. Likewise, our implementation of polynomial squaring is constant-time and memory access. The k-squaring function is implemented as a permutation of the coefficients of the input polynomial. During the permutation we scan every bit of the input and update the appropriate bit in the output. Hence, k-squaring is constant-time. Memory locations that are accessed when copying the bits of input to the output are fully determined by the parameter k, which is itself derived solely from the public value r. Therefore, our implementation of k-squaring is also secure against side-channel attacks which exploit the knowledge of memory locations that are accessed by the program.

5.4 Results

In this section we provide performance results of different implementations of the inversion function. Namely, we implement and benchmark the following versions of the function:

1. PORTABLE – fully portable version of the code, implemented in C without any platform specific instructions.
2. PCLMUL – the same as PORTABLE with the exception that the **PCLMUL** instruction is used for polynomial multiplication and squaring.
3. AVX2 – implementation leveraging the instructions offered by the **AVX2** instruction set.
4. AVX512 – implementation leveraging the instructions offered by the basic **AVX512** instruction set, called **AVX512F**, which is supposed to be supported by any x86_64 platform with 512-bit wide SIMD capabilities.
5. VPCLMUL – the same as AVX512 with the exception that the new **VPCLMUL** instruction is used for polynomial multiplication and squaring.

For each of these implementation flavors we benchmark two variants of inversion based on the k-squaring implementation – generating permutation maps on the fly or using precomputed maps. Moreover, we measure and present the runtime of the BIKE-2 key generation algorithm that uses the described implementations of inversion.

The comparison baseline. The performance of our implementations is compared with two popular open-source libraries NTL (compiled with GF2X) [4, 53] and OpenSSL [5]. We do not compare to [67, 68, 27, 69] because they are all slower than NTL: a) the inversion algorithm of [68] is reported to be twice faster than [69] and 12 times faster than [27], but 1.7 times slower than NTL; b) the implementation in [67] is reported to be 3 times slower than NTL. Another reason for choosing NTL and OpenSSL for comparison is that the "Additional implementation" [45] of BIKE protocol submitted to the second round of the NIST standardization project uses the inversions from these two libraries.

Remark 9. We also measured the inversion function of the LEDAcrypt optimized code [65] that implements safegcd algorithm [66]. This code uses AVX2, so for fair comparison, we compile our code with AVX2 instructions only and compare the runtime. The performance of the LEDAcrypt inversion is: a) using gcc: 4.05 and 12.43 million cycles for Level-1 and 3, respectively; b) using clang: 3.29 and 10.30 million cycles for Level-1 and 3, respectively. The performance of our inversion on the same platform is: 0.57 and 2.08 million cycles for Level-1 and 3, respectively. The code of [65] runs in constant time and is faster than NTL. On the other hand, it is significantly slower than our implementation even when we use only the AVX2 code.

Blinding a non-constant time inversion. Binary polynomial inversion does not operate in constant-time in either NTL [4] or OpenSSL [5] because these libraries use the extended

GCD based algorithms to compute the inverse. To address this issue, a recent change in OpenSSL (between version 1.0.2 to version 1.1.0) protects the implementation by *blinding* the inversion as follows. The function **BN_GF2m_mod_inv**(a, s) computes a^{-1} mod s by the following sequence: 1) choose a random b; 2) compute $c = ab$; 3) invert c); 4) multiply by b. Unfortunately, this does not work in the general case, where s is not necessarily an irreducible polynomial (see discussion in [71]). If s is reducible, $c = ab$ may be non-invertible modulo s. This is exactly the case of BIKE-2 where $x^r - 1$ is reducible. Although the OpenSSL function **BN_GF2m_mod_inv**(a, $x^r - 1$) is called with invertible a, the internal blinding may select a random non-invertible polynomial b and then inverting $c = ab$ would fail. In the polynomial ring $\mathscr{R}$ a randomly selected b has probability $\frac{1}{2}$ to be non-invertible. For a fair comparison (of constant-time implementations), we use the same blinding technique for NTL as well. For correctness, we always choose b such that $wt(b)$ is odd, and therefore b is invertible in $\mathscr{R}$.

The platform. We carried out performance measurements on a platform which supports all the required instructions for the five versions of the code specified above. The platform is a Dell XPS 13 7390 2-in-1 laptop. It has the latest, 10^{th} generation Intel®Core^{TM} processor (microarchitecture codename "Ice Lake"[ICL]). The specifics are Intel®Core^{TM} i7-1065G7 CPU 1.30GHz. This platform has 16 GB RAM, 48K L1d cache, 32K L1i cache, 512K L2 cache, and 8MiB L3 cache and it supports **AVX512** and **VPCLMUL** instructions. For the experiments, we turned off the Intel® Turbo Boost Technology (in order to work with a fixed frequency and measure performance in cycles).

Measurements methodology. The performance reported hereafter is measured in processor cycles (per single core). We obtain the results using the following methodology. Every measured function was isolated, run 25 times (warm-up), followed by 100 iterations that were clocked (using the **RDTSC** instruction) and averaged. To minimize the effect of background tasks running on the system, every experiment was repeated 10 times, and the minimum result was recorded.

The code. Our code is written in C with intrinsic functions [57] for AVX functionality. The code is compiled with gcc (version 9.3.0), using the "-O3" optimization flag, and ran on a Linux OS (Ubuntu 20.04).

Performance of inversion

The performance of the inversion algorithm depends on the Hamming weight of $r-2$ (recall that r defines the polynomial ring $\mathscr{R}$), as explained in Section 5.2. Therefore, we generate a set of r values, with different $wt(r-2)$, from the range of values relevant for BIKE. Then, we choose one representative for every value of $wt(r-2)$, and measure the runtime of the algorithm for the chosen parameters. Note that only r values for BIKE level 1 and 3 are considered since NIST announced that the highest level of security, Level-5, is not critical for standardization.

Tables that contain all the measurements that were performed and performance improvements over the NTL library, i. e., all the different implementation variants listed in the introduction of this section, are given in the appendix (Table A.4 and A.5). Here, we present only the most interesting and relevant data points. The **AVX2** instruction set was introduced with the Haswell lineup of Intel processors in 2013. We assume that most of the CPUs in use today support at least **AVX2**, and therefore we present the performance numbers for **AVX2** and **AVX512** implementations. The third option, **AVX512** plus **VPCLMUL**, is included to showcase the improvements that can be achieved on the latest generation of Intel CPUs and to get a glimpse of what can be expected from future processor architectures.

In Table 5.1 we show the runtime of the two baseline implementations, NTL and OpenSSL, together with our implementations. Firstly, we note that all the different variants of the algorithm that we implemented significantly outperform the baseline. While NTL is an order of magnitude faster than OpenSSL, our implementations are an order of magnitude faster than NTL.

It is interesting to note that for the GCD based inversion algorithms the runtime increases with the size of r, while this is not necessarily the case for our algorithm. For example, if we take $r_1 = 12323$ (first row) and $r_2 = 12157$ (last row), both NTL and OpenSSL are faster for the smaller r_2, while our implementations show a better performance for the larger r_1. This is due to the fact that $wt(r_1 - 2) < wt(r_2 - 2)$ and therefore, the algorithm proposed in this chapter performs fewer operations for r_1 than for r_2. Note that this does not hold in general for $r_1 > r_2$, especially when the corresponding weights $wt(r_1 - 2) < wt(r_2 - 2)$ are close, because even though with r_1 we perform a smaller number of operations, the operations themselves are more time consuming since the polynomial ring we work in is larger.

The use of pre-computed permutation maps for k-squaring provides an interesting trade-off. It improves the overall performance at a cost of occupying some memory space. The maps that we need to store hold $r \cdot (\lfloor \log(r-2) \rfloor + 1 + wt(r-2))$ entries of size r bits (for all security levels of BIKE the entries can be stored in 2 bytes of memory). For example, BIKE-2 IND-CCA version (as proposed to the Round-2 NIST project) requires 450KB and 1.1MB of memory to store the maps for parameter sizes defined for Level-1 and Level-3 security, respectively. However, the performance improvements when using the maps are not very impressive – the difference in the runtime with and without precomputed maps is always around five percent. For example, the **AVX2** implementations for $r = 11779$ invert a polynomial in 560K and 590K cycles with and without the precomputed maps, respectively, showing a difference of 30K cycles. The small contribution of the precomputed maps to the performance of the inversion can be attributed to the heavily optimized functions for generating permutation maps on the fly (described in Section 5.3.1).

It is also interesting to note the differences in the performance of the three SIMD implementations. For example, consider the columns (a), (b), and (c) of Table 5.1. The jump from **AVX2**, in (a), which operates on 256-bit wide registers to the **AVX512** implementation, in (b), which works with registers of twice the size, does not improve the performance as much as we would

Table 5.1 – Performance of our implementations of inversion in $\mathbb{F}_2[x]/(x^r - 1)$ for a set of r values with different $wt(r-2)$. The NTL and OSSL columns denote the runtime of the inversion from the corresponding libraries ([4, 5]). The remaining columns represent our implementation: (a) with **AVX2**; (b) with **AVX512**; (c) with **AVX512** and **VPCLMUL**; columns labeled with "*" denote implementations with pre-computed permutation maps. The runtime is measured in millions of cycles.

r	$wt(r-2)$	NTL	OSSL	(a)	(a)*	(b)	(b)*	(c)	(c*)
12323	4	6.75	49.19	0.59	0.56	0.54	0.52	0.43	0.41
11779	5	5.86	42.61	0.57	0.54	0.54	0.51	0.44	0.41
12347	6	6.52	48.67	0.64	0.63	0.60	0.58	0.47	0.45
11789	7	6.10	43.83	0.62	0.59	0.58	0.55	0.45	0.44
11821	8	5.99	44.98	0.66	0.62	0.61	0.59	0.48	0.46
11933	9	6.22	43.31	0.69	0.65	0.64	0.63	0.52	0.49
12149	10	6.37	46.60	0.75	0.71	0.70	0.67	0.55	0.52
12157	11	6.30	47.00	0.78	0.74	0.72	0.70	0.58	0.55
25603	4	9.00	213.84	1.75	1.72	1.65	1.61	1.28	1.24
24659	5	8.67	188.42	1.77	1.71	1.66	1.61	1.30	1.24
24677	6	8.61	193.27	1.88	1.83	1.74	1.71	1.35	1.32
24733	7	8.77	204.55	1.93	1.89	1.79	1.77	1.40	1.35
24821	8	9.07	185.17	2.08	2.02	1.92	1.87	1.51	1.49
25453	9	8.86	197.20	2.26	2.20	2.09	2.06	1.61	1.54
24547	10	8.32	182.11	2.13	2.08	1.99	1.95	1.61	1.53
24533	11	8.79	175.41	2.21	2.14	2.08	2.00	1.67	1.60
24509	12	8.47	181.95	2.27	2.20	2.13	2.07	1.66	1.61

expect. The **AVX512** is slightly faster than the **AVX2** implementation with the difference in performance around five percent. One possible reason for this is that the **AVX512** instructions that we use have higher latency compared to the used **AVX2** instructions. Another likely culprit for the unimpressive performance of **AVX512** is the platform used for the experiments which has a low-powered mobile processor designed for portable devices. Based on the previous generations of Intel CPUs, one of the ways that the power demand is lowered is by crippling the SIMD unit because it is one of the most power hungry parts of a processor. Unfortunately, these are the only 10^{th} generation IceLake Intel CPUs available on the market currently, but we expect to see higher performance improvements on the desktop and server versions of IceLake once they are released. Nevertheless, the contribution of the **VPCLMUL** instruction to the reduction in the runtime is more noticeable. For example, inversion time for $r = 11779$ drops by 100K cycles, from 540K to 440K, when **VPCLMUL** is used in addition to **AVX512**. The reason for the ~ 20 percent performance improvement here is that the bottleneck in the polynomial multiplication function when implemented with basic **AVX512** instruction set is the use of the (non-vectorized) **PCLMUL** instruction for multiplying two 64-bit digits, while the remaining part of the function is able to use the 512-bit vector registers offered by **AVX512**. The use of **VPCLMUL** does improve the situation, but it is difficult to leverage its full power when implementing the multiplication with Karatsuba's method (as explained in Section 5.3.3).

In Table 5.2 we show the relative speedups over the NTL inversion achieved by our various implementations. Depending on the specific implementation the measurements show an 8-fold to 16-fold speedup for Level-1 parameter sizes, while the improvements for Level-3 parameters are more modest, exhibiting 3-fold to 7-fold speedup over NTL. The proposed parameters in Round-2 of NIST project for CCA secure BIKE-2 are $r = 11779$ and $r = 24821$ for the first two security levels. Our most efficient implementation (**AVX512** with **VPCLMUL**) is able to invert a polynomial 14.37 times faster than NTL when $r = 11779$, and 6.1 times faster when $r = 24821$. It is interesting to note that the relative speedups for Level-1 parameter sizes are much higher than those for Level-3, meaning that NTL's implementation of the inversion scales better with the polynomial size than our implementation. This may be attributed to the fact that NTL's implementation is a GCD based inversion with linear complexity in r of the number of ring operations that are required, together with the fact that these ring operations are fairly efficient. On the other hand, our implementation requires $\lfloor \log(r-2) \rfloor + 1 + wt(r-2) - 1$ polynomial multiplications which themselves require an order of $(r/64)^{\log_2 3}$ processor instructions, and therefore might scale worse than NTL's implementation. However, we leave the investigation of this phenomenon for future research.

Table 5.2 – Speedup of our implementations of inversion in $\mathbb{F}_2[x]/(x^r-1)$ compared to NTL with GF2X [4]. Columns 3-8 represent the speedup over NTL of the following implementation: (a) **AVX2**; (b) **AVX512**; (c) **AVX512** and **VPCLMUL**; columns labeled with "*" denote implementations with pre-computed permutation maps. The speedup is measured for a set of r values with different $wt(r-2)$.

r	$wt(r-2)$	(a)	(a)*	(b)	(b)*	(c)	(c*)
12323	4	11.51	12.15	12.50	13.02	15.68	16.55
11779	5	10.26	10.80	10.85	11.45	13.32	14.37
12347	6	10.11	10.36	10.86	11.26	13.87	14.42
11789	7	9.85	10.37	10.44	11.03	13.44	13.96
11821	8	9.10	9.61	9.89	10.15	12.42	13.10
11933	9	8.97	9.55	9.67	9.93	12.03	12.70
12149	10	8.48	8.99	9.09	9.46	11.54	12.23
12157	11	8.10	8.48	8.72	9.04	10.91	11.46
25603	4	5.15	5.23	5.45	5.59	7.06	7.23
24659	5	4.89	5.06	5.22	5.40	6.66	6.98
24677	6	4.58	4.71	4.96	5.04	6.38	6.54
24733	7	4.54	4.65	4.91	4.97	6.25	6.48
24821	8	4.37	4.49	4.72	4.84	6.01	6.10
25453	9	3.92	4.03	4.23	4.31	5.51	5.74
24547	10	3.91	4.00	4.18	4.27	5.18	5.44
24533	11	3.97	4.11	4.23	4.39	5.28	5.49
24509	12	3.73	3.85	3.98	4.10	5.10	5.27

The speedups shown in the table highlight again the difference in the GCD based inversion algorithm of NTL and ITI based algorithm proposed in this chapter. Namely, the performance of the former one depends only on the size of the polynomials (determined by r), while the performance of the latter depends also on the value of $wt(r-2)$. This effect is embodied in the fact that the relative speedups of our implementations decrease as the corresponding weight of r increases.

Remark 10. We note that our implementations are two orders of magnitude faster than the inversion from the OpenSSL library. The exact numbers can be found in Tables A.4 and A.5 in the appendix.

Performance of BIKE key generation

In Table 5.3 we report the performance of BIKE key generation procedure that uses our implementation of inversion. The table contains only data for those implementations that during the inversion generate permutation maps, required for k-squaring, on the fly, i. e., implementations without precomputed maps. For details about all our implementations for the full set of r values refer to Table A.6 in the appendix.

Table 5.3 present the numbers for $r = 11779$ and $r = 24821$ which are the parameters proposed for security Levels 1 and 3, respectively, in BIKE submission to the second round of the NIST PQ project. Additionally, we show the runtime of the inversion when $r = 12323$ and $r = 24659$. With these two r values we can achieve an improvement in performance of the key generation while maintaining the DFR at the level required for the corresponding security level.

Table 5.3 – BIKE-2 key generation performance, for four relevant r values, when our implementation of the inversion algorithm is used (without precomputed maps). Columns 2-4 represent the following implementations: (a) **AVX2**; (b) **AVX512**; (c) **AVX512** and **VPCLMUL**. The runtime is measured in thousands of cycles.

r	(a)	(b)	(c)
11779	630	590	480
12323	644	587	473
24821	2222	2061	1607
24659	1913	1781	1408

It is evident from the data in Tables 5.1 and 5.3 that the cost of key generation is dominated by the cost of inversion. For example, the **AVX2** implementation for $r = 11779$ takes 570K cycles to invert the secret key polynomial, while the rest of the operations performed during the key generation take only 60K cycles, yielding in total 630K cycles to generate a key pair. This signifies the importance of having an efficient inversion in the implementation of BIKE-2 protocol.

5.5 Discussion

In this chapter we proposed an algorithm for polynomial inversion in the context of code-based cryptographic schemes submitted to the NIST Post-Quantum Cryptography Standardization Project. The algorithm is based on the ITI algorithm [27], with some modifications that make it particularly efficient and applicable in the context of inverting elements of a polynomial ring $\mathbb{F}_2[x]/(x^r - 1)$ used for example in BIKE KEM. We also explain how this algorithm can be implemented, and indeed implement the algorithm such that it offers a very competitive performance. Moreover, our experiments show that it can substitute the NTL and OpenSSL inversion, which is used in BIKE Round-2 NIST submission, and achieve significant performance improvements.

In general, the parameter r determines the size of the public and private key and the ciphertext, and thus the overall latency and bandwidth of BIKE. So far, r was chosen as the minimum value that satisfies the security target [9] and the target DFR of the decoder [1, 19]. We propose an additional consideration, namely $wt(r-2)$ because the inversion Algorithm 10 is more efficient when $wt(r-2)$ is smaller. Tables X and Y in Appendix X list r values according to their respective weight and their security levels 1 and 3. BIKE submission in the second round of NIST project recommends $r = 11779$ for Level-1 security, for which $wt(r-2) = 5$. Interestingly, a considerably larger $r = 12323$ has $wt(r-2) = 4$, and therefore offers faster key generation than $r = 11779$. We also note that [19] shows that $\sim r = 12323$ is needed and sufficient in order to achieve the required DFR of 2^{-128} for the first level of security.

Before this, BIKE-1 seemed to be a more appealing option than BIKE-2. This was the result of the prohibitive cost of BIKE-2 key generation that seemed to be an obstacle for adoption, especially when ephemeral keys are desired. This left out BIKE-2's main advantage – the amount of data that needs to be exchanged between two parties in a single execution of the key exchange protocol. BIKE specification [9] addresses this difficulty by using a "batch inversion" approach that requires pre-computation of a batch of key pairs. Such solutions require that other protocols are adapted to using batched key pairs, and this raises additional complications.

Our improved inversion and hence faster key generation avoids the difficulty. For Level-1 ($r = 11779$) BIKE-2 has key generation / encapsulation / decapsulation at 480K/180K/1.2M cycles, and requires 1.4KB of data to be sent in each direction. By comparison, BIKE-1 (after using our latest multiplication implementation) has key generation / encapsulation / decapsulation at 67K/230K/1.3M cycles, with 2.8KB of data sent in each direction. We believe that our results position BIKE-2 as an appealing design choice among the BIKE variants.

Based on the results presented in this chapter, the new specification of BIKE [18], aimed at Round-3 of the NIST PQC Project, adopts BIKE-2 as the only variant of BIKE.

Legendre PRF Part II

6 Improved key recovery on the Legendre PRF

The Legendre PRF is a pseudorandom function (PRF) based on the properties of the Legendre symbol proposed by Damgård [29]. More precisely, for a given prime p, we model the function as an oracle $\mathscr{F}_k$ parametrized by the secret key k, which on input $a \in \mathbb{F}_p$ outputs the Legendre symbol of $k+a$, $\mathscr{F}_k(a) = \left(\frac{k+a}{p}\right)$. Damgård conjectured that given a sequence of Legendre symbols of consecutive elements it is hard to predict the next one. Similar problems conjectured to be hard were also proposed in [28], such as finding the secret key k while being given access to $\mathscr{F}_k$ and distinguishing $\mathscr{F}_k$ from a random function. A polynomial time algorithm for solving either of these problems is not yet found and it is believed that the problems are indeed hard. Until recently practical applications of the Legendre PRF have been limited, primarily due to availability of much faster alternatives.

Recent results on cryptographic primitives for multi-party computation (MPC) [28] positioned the Legendre PRF as a promising candidate for randomness generation in MPC settings. The main reasons the Legendre PRF is suitable for MPC are the multiplicative property of the Legendre symbol and very efficient evaluation of the symbol which can be performed with only three modular multiplications in arithmetic circuit multi-party computations. Motivated by these results, the Ethereum blockchain developers are considering to use the Legendre PRF in a construction for the Ethereum 2.0 protocol [8] (due to be launched in 2020). To incentivize research in the security of the Legendre PRF the Ethereum foundation announced a number of challenges where the goal is to recover the secret key given $M = 2^{20}$ consecutive Legendre symbols, for primes of size varying from 64 to 148 bits [8].

Previous work on attacks on the Legendre PRF includes [6] and [7]. In [6] the authors give an attack on the Legendre PRF that has complexity $O(\sqrt{p}\; t \log p)$, where t is the number of operations needed to compute a Legendre symbol or query an oracle. Subsequently, a better attack was published in [7][1] with complexity $O(\sqrt{p}\; t \log^2 p)$.

---

[1] We note that the preprint of [7] was published (eprint.iacr.org/2019/1357) while we were independently working on the algorithms presented in this chapter. Our report was posted online [30] after their preprint. Subsequently, we wrote another paper where we generalize and expand the methods presented in [30]. This paper was submitted and accepted at the ANTS 2020 conference (www.math.auckland.ac.nz/~sgal018/ANTS).

Contributions. In this chapter we present an algorithm that recovers the secret key of the Legendre PRF in $O(\sqrt{p\log\log p})$ operations on a $\Theta(\log p)$-bit architecture and by using only $\sqrt[4]{p\log^2 p\log\log p}$ queries of the PRF oracle. There are two advantages of our algorithm with respect to the previous best algorithm [7]. Firstly, in [7] the runtime of the algorithm depends linearly in the cost of Legendre symbol evaluations and queries, while in our algorithm this cost can be ignored. Our algorithm lowers the key extraction effort from $O(\sqrt{p}\ t\log^2 p)$ in [7] to $O(\sqrt{p\log\log p})$. Secondly, if the number of oracle calls is bounded by M, we reduce the number of operations from $O(\frac{p\ t\log^2 p}{M^2})$ in [7] to $O(\frac{p\log p\log\log p}{M^2})$.

We give a rigorous analysis of the runtime of our algorithm. The main bottleneck is the number of simple operations on $\log p$-bit words, such as word comparisons, shifts, ANDs, ORs and XORs. Therefore we analyze the total number of such operations and ignore the cost of Legendre symbol computations as the amount of work spent on computing them is negligible compared to the rest of the algorithm.

Furthermore, we explain the details of our implementation of the algorithm and various optimizations that significantly improved the performance. Finally, we give the solutions of challenges 0, 1 and 2 of the Ethereum foundation Legendre PRF challenge. Challenges 0, 1, and 2 feature a prime of size 64, 74 and 84 bits , respectively. The most difficult challenge, number 2, has so far only been solved by us.

Structure. In Section 6.1 we recall the notation and basic concepts about the Legendre symbol.

In Section 6.2 we present our algorithm for attacking the Legendre PRF. The algorithm is based on the birthday attack and it is divided in two parts – the precomputation phase where a table of sequences of Legendre symbols is generated and the search phase where random sequences are generated until a collision is found. The similarity of the precomputation and the search phase can lead to some problems which we address and show how to fix.

In Section 6.3 we analyze and give the complexity of the algorithm in terms of number of operations on a $\Theta(\log p)$-bit word machine. The costs of Legendre symbol computation and oracle queries are ignored as they are negligible with respect to the rest of the algorithm. The precomputation and search stage are treated separately and the optimal runtime is given under reasonable heuristic assumptions. We also show how the runtime changes if a limited number of queries is available.

Section 6.4 presents the implementation details and outlines differences between theoretical and practical considerations of the runtime. We also give some implementational tricks that give very valuable constant performance improvements.

In Section 6.5 we give the results of the performed experiments and the secret keys that we recovered for the first three challenges posted on the Ethereum website [8]. Moreover, we discuss the difference between the expected and the observed runtime.

6.1 Background and notation

Throughout the paper we consider p to be an odd prime number, $\mathbb{F}_p$ the finite field of cardinality p with elements represented by integers modulo p. We denote by $\left(\frac{a}{p}\right)$ the Legendre symbol:

$$\left(\frac{a}{p}\right) = \begin{cases} 1 & \text{if } a \in \mathbb{F}_p^* \text{ is a square mod } p \\ 0 & \text{if } a = 0 \text{ mod p} \\ -1 & \text{if } a \in \mathbb{F}_p^* \text{ is not a square mod } p. \end{cases}$$

A sequence of Legendre symbols of L consecutive elements with a starting point a is denoted by:

$$\{a\}_L := \left(\frac{a}{p}\right), \left(\frac{a+1}{p}\right), \left(\frac{a+2}{p}\right), \ldots, \left(\frac{a+L-1}{p}\right).$$

In this chapter we assume that L is such that a given Legendre sequence of length L uniquely defines the starting point, i. e., that $\{a\}_L = \{b\}_L$ iff $a = b$. As already explained in Section 2.2 the provable bound for such L is $O(\sqrt{p}\log p)$. However, for all practical purposes L can be selected to be $\Omega(\log p)$.

The Legendre PRF is a pseudorandom function parametrized with the prime p and the secret key k that on input a outputs $\left(\frac{k+a}{p}\right)$. Several problems related to the Legendre PRF are conjectured to be hard. In the Shifted Legendre Symbol Problem (SLSP) we are given access to a Legendre PRF oracle $\mathscr{F}_k$ that we can query with arbitrary values and the goal is to recover the secret k. Another supposedly hard problem, Decisional Shifted Legendre Symbol Problem (DSLSP), is to distinguish between a Legendre PRF and a truly random oracle. The third problem that is hard to solve is the Next Symbol Problem (NSP) where we are given a sequence of $M = \text{polylog}(p)$ symbols and the goal is to find the next Legendre symbol in the sequence.

6.2 Algorithm

We give our algorithm in the scenario of attacking the Shifted Legendre Symbol Problem. The attack is easily generalized to the other two stated problems – DSLSP and NSP. The assumption is that we are given access to an oracle $\mathscr{F}$ that computes $\left(\frac{k+a}{p}\right)$ on input a, and we want to find k. Let M be the number of oracle calls that we make and L as defined in the previous section. It is assumed that M is larger than L.

The general idea is to execute the algorithm in two phases – precomputation and search phase. In the precomputation phase we invoke the oracle multiple times in order to obtain many Legendre sequences $\{k_i\}_L$, such that if we recover k_i we can easily compute k, and store these sequences in a table. Then in the second phase we compute Legendre sequences of random elements j until we find $\{j\}_L = \{k_i\}_L$ for some k_i. By the assumption that L is large enough we will have $j = k_i$, and subsequently we can recover the secret k. This is a simple birthday attack which has optimal runtime when the table contains $\sqrt{p}$ sequences. However one needs to take into account the cost of creating the table – which depends on the number of oracle calls, the cost of computing $\{j\}_L$ – which depends on the cost of Legendre symbol computations, and the cost of table lookups – which is a couple of operations on an L-bit word machine. The naive way to populate the table with sequences is to generate each sequence individually by performing L queries to the oracle. Likewise, in the second phase where we generate random sequences to find a collision with those in the table, we can create every random sequence by evaluating the Legendre symbol for L different values.

However we show that one can do much better. Firstly, we can reduce the cost of populating the table to $o(1)$ oracle calls per sequence. By doing this we decrease the number of necessary oracle calls to create a table of a desired size, which is particularly important in the cases where we are allowed to make a limited number of queries to the oracle. More precisely, we show how to create a table of size $O(M^2/L)$ with M oracle calls. For example, in the Legendre PRF challenge we are given only the first $M = 2^{20}$ consecutive Legendre symbol starting from the secret k. Secondly, we explain how to reduce the cost of generating random sequences in the search phase to $o(1)$ Legendre symbol computations per sequence. With this the bottleneck of the algorithm becomes the cost of simple bit operations, such as sequence comparison, which are much cheaper than Legendre symbol computations.

6.2.1 Sequence properties

Given a Legendre sequence $\{a\}_M$ of length $M \geq L$, we can trivially extract $\{a\}_L$ from it by taking only the first L elements. However, the following three properties allow us to extract additional sequences from $\{a\}_M$, and moreover, to have a special relation between the starting point a and the extracted sequences.

Shifting property. Each subsequence of $\{a\}_M$ of L consecutive symbols corresponds to the Legendre sequence of $(a+i)$ for some shift i. As long as $0 \leq i \leq M-L$, then $\{a+i\}_L$ is a subsequence of $\{a\}_M$:

$$\{a+i\}_L = \{a\}_M \text{ from } i^{th} \text{ to } (L-1+i)^{th} \text{ element,}$$

or equivalently,

$$\{a+i\}_L = \{a\}_{L+i} \text{ from } i^{th} \text{ to the last element.}$$

This allows us to extract the sequences $\{a+i\}_L$ for $i = 0, 1, \ldots, M-L$ from $\{a\}_M$.

Multiplicative property. It is well known that the Legendre symbol is a totally multiplicative function, or in other words $\left(\frac{a}{p}\right)\left(\frac{d}{p}\right) = \left(\frac{ad}{p}\right)$. This relates to Legendre sequences of a and d in the following way. The sequence of Legendre symbols of length L with starting point ad and common difference $d \geq 1$ between the "numerators" in the sequence:

$$\left(\frac{ad}{p}\right), \left(\frac{ad+d}{p}\right), \left(\frac{ad+2d}{p}\right), \ldots, \left(\frac{ad+(L-1)d}{p}\right)$$

is equal to the Legendre sequence of length L with starting point a and common difference 1, multiplied by the Legendre symbol of d, i. e.,

$$\left(\frac{d}{p}\right)\{a\}_L := \left(\frac{d}{p}\right)\left(\frac{a}{p}\right), \left(\frac{d}{p}\right)\left(\frac{a+1}{p}\right), \dots, \left(\frac{d}{p}\right)\left(\frac{a+(L-1)}{p}\right).$$

This property can be expressed in a different manner. Namely, that a sequence of L Legendre symbols starting from a with common difference d is equal to the Legendre sequence of a/d where every element is multiplied by $\left(\frac{d}{p}\right)$:

$$\left(\frac{d}{p}\right)\{a/d\}_L = \left(\frac{a}{p}\right), \left(\frac{a+d}{p}\right), \left(\frac{a+2d}{p}\right), \dots, \left(\frac{a+(L-1)d}{p}\right).$$

Note that $a, d \in \mathbb{F}_p$ and a/d refers to division in $\mathbb{F}_p$, i. e., it is computed as $ad^{-1} \bmod p$.

The sequence $\left(\frac{d}{p}\right)\{a/d\}_L$ is a subsequence of $\{a\}_M$ as long as $(L-1)d \le M-1$, or in other words as long as $d \le D_M := \left\lfloor \frac{M-1}{L-1} \right\rfloor$. This allows us to obtain $\{a/d\}_L$ for $d = 1, 2 \dots, D_M$ by computing $\left(\frac{d}{p}\right)$ for all d's, and extracting symbols from $\{a\}_M$ at positions $0, d, 2d, \dots, (L-1)d$.

Reverse sequence property. Suppose that we have the following Legendre sequence:

$$\{a\}_L = \left(\frac{a}{p}\right), \left(\frac{a+1}{p}\right), \left(\frac{a+2}{p}\right), \dots, \left(\frac{a+L-1}{p}\right).$$

Then, the reverse sequence, after multiplying it element-wise by $\left(\frac{-1}{p}\right)$, is the Legendre sequence of $-(a+L-1) = -a-(L-1)$:

$$\{-a-(L-1)\}_L = \left(\frac{-a-L+1}{p}\right), \left(\frac{-a-L+2}{p}\right), \dots, \left(\frac{-a-1}{p}\right), \left(\frac{-a}{p}\right)$$

We may think of this property as of a generalization of the homomorphic property to negative denominators. This observation allows us to obtain one extra sequence gratis for each sequence that we have.

Combining all properties. The three properties can be combined to vastly increase the number of Legendre sequences that can be extracted from $\{a\}_M$. Consider an arithmetic sequence of length L starting from $a+i$ and of common difference d. Legendre symbols of this sequence are:

$$\left(\frac{a+i}{p}\right), \left(\frac{a+i+d}{p}\right), \left(\frac{a+i+2d}{p}\right), \dots, \left(\frac{a+i+(L-1)d}{p}\right),$$

which can all be obtained from $\{a\}_M$ if $0 \le i$ and $i+(L-1)d \le M-1$. Furthermore, this sequence, multiplied (divided) by $\left(\frac{d}{p}\right)$ is equal to

$$\left\{\frac{a+i}{d}\right\}_L = \left(\frac{\frac{a+i}{d}}{p}\right), \left(\frac{\frac{a+i}{d}+1}{p}\right), \left(\frac{\frac{a+i}{d}+2}{p}\right), \ldots, \left(\frac{\frac{a+i}{d}+L-1}{p}\right).$$

Therefore, from $\{a\}_M$ we can extract the Legendre sequences of $\frac{a+i}{d}$ for $d = 1, 2, \ldots, D_M = \lfloor \frac{M-1}{L-1} \rfloor$ and $i = 0, 1, \ldots, M-1-(L-1)d$. Furthermore, applying the reverse sequence property, we can obtain the sequence of $-\frac{a+i}{d} - (L-1)$. This increases the total number of Legendre sequences that can be extracted from $\{a\}_M$ to

$$\sum_{d=1}^{D_M} \sum_{i=0}^{M-1-(L-1)d} 2 = 2MD_M - (L-1)D_M(D_M+1) = \frac{M^2}{(L-1)} - M + O(L)$$

where the constant in $O(L)$ is at most 2. For all these sequences, if we know their starting points, $\frac{a+i}{d}$ or $-\frac{a+i}{d} - (L-1)$, together with i and d, then we can compute a.

6.2.2 Precomputation stage

The first part of the algorithm is the precomputation stage which is itself done in two steps. Firstly we query $\mathscr{F}(x)$ for $x = 0, 1, \ldots, M-1$ in order to obtain $\{k\}_M$. Then we use the described sequence properties to extract $\frac{M^2}{(L-1)} + O(M)$ Legendre sequences out of $\{k\}_M$. These sequences are of the following two types:

$$\left\{\frac{k+i}{d}\right\}_L \text{ and } \left\{-\frac{k+i}{d} - (L-1)\right\}_L.$$

They are saved in a hash table, together with the corresponding i, d, and one extra bit to differentiate $\frac{k+i}{d}$ from $-\frac{k+i}{d} - (L-1)$. With this the precomputation stage is finished.

6.2.3 Search stage

During the second phase of the algorithm we compute $\{j\}_L$ sequences for many random j's in an attempt to find a collision with a sequence stored in the hash table. Once the collision is found we have that $\{j\}_L = \{\frac{k+i}{d}\}_L$, and by the assumption that L is such that a given sequence uniquely determines its starting point, it follows that $j = \frac{k+i}{d}$, which allows us to compute the secret key by $k = dj - i$. The key can be recovered in the same way in the case of collision with one of the reverse sequences $-\frac{k+i}{d} - (L-1)$.

The table contains $\frac{M^2}{L-1} + O(M)$ sequences, and therefore, a collision is expected to happen after $p\frac{L-1}{M^2}$ trials. Note that each trial involves generating a random sequence of L symbols. If this was to be done in a naive manner we would need to compute L Legendre symbols for each trial. However, by using the same sequence properties as in the precomputation phase we can greatly reduce the number of required Legendre symbol evaluations.

Similarly as in the precomputation stage, we proceed by choosing a random $j \in \mathbb{F}_p$ and computing the Legendre sequence of length N with starting point j. Once $\{j\}_N$ is obtained we can extract $\frac{N^2}{L-1} + O(N)$ sequences of type $\frac{j+a}{b}$ and $-\frac{j+a}{b} - (L-1)$ from it, with a and b satisfying similar constrains as i and d in the precomputation stage. However, here we need to be more careful. The sequences that we extract from $\{j\}_N$ are highly correlated with the sequences extracted from $\{k\}_M$. Therefore, they may not be considered as sequences obtained from uniformly random elements in $\mathbb{F}_p$ which is the assumption that the collision search runtime is based on. To illustrate, if we extract two sequences from $\{j\}_N$, which are correlated such that if one of them is in the hash table then the other one is as well, then the trials performed with these sequences cannot be considered as two trials but rather as a single trial, i. e., either we get two collisions or none. Therefore, the goal is to extract as many sequences as possible from $\{j\}_N$ which are not "correlated".

There are three main types of correlation described below.

Reverse sequence correlation. If we have that

$$\frac{j+a}{b} = \frac{k+i}{d}$$

then

$$-\frac{j+a}{b} - (L-1) = -\frac{k+i}{d} - (L-1)$$

and vice-versa. Therefore we avoid computing the reverse sequences of the ones we extract from $\{j\}_L$.

Shifting correlation. If we have that

$$\frac{j+a}{b} = \frac{k+i}{d}$$

then

$$\frac{j+a+b}{b} = \frac{k+i+d}{d}.$$

In other words, if we generate the sequence from $\frac{j+a}{b}$ and it does not produce a hash table collision, then there is a lower chance for a collision for the sequence generated from $\frac{j+a+b}{b}$. Therefore, we want to avoid wasting time on such sequences since they give lower probability of a collision. To combat this issue we reduce the number of sequences that are extracted from $\{j\}_N$ by only considering sequences for $\frac{j+a}{b}$ with $0 \leq a < b$. In this way the sequence starting from $\frac{j+a+b}{b}$ is never tested. However, this reduces the number of sequences that can be extracted to

$$\sum_{b=1}^{D_N} \sum_{i=0}^{b-1} 1 = \frac{N^2}{2(L-1)^2} + O\left(\frac{N}{L-1}\right).$$

Multiplicative correlation. If we have that

$$\frac{j+a}{b} = \frac{k+i}{d}$$

then

$$\frac{j+a}{b/f} = \frac{k+i}{d/f}$$

for each divisor f of $\operatorname{lcm}(d,b)$. Similarly as before, we want to avoid producing sequences correlated in such way. Therefore, when extracting sequences from $\{j\}_N$ we do not allow any common divisors between the denominators d and b. This can be done simply by choosing b's from a different set, i. e., instead of taking be in $1,2,\dots,D_N = \lfloor \frac{N-1}{L-1} \rfloor$ we take

$$b \in \{D_M+1, D_M+2, \dots, D_N\} \cap \mathbb{P}$$

where $\mathbb{P}$ is the set of prime numbers, giving in total $O\left(\frac{N-M}{L} / \log\left(\frac{N}{L}\right)\right)$ different b-values.

By putting all the pieces together we obtain the following algorithm: given the sequence $\{j\}_N$, we extract from it all sequences of type $\left\{\frac{j+a}{b}\right\}_L$ with $b \in \{D_M+1, D_M+2, \dots, D_N\} \cap \mathbb{P}$ and $0 \le a < b$. This gives rise to a total of

$$\sum_{\substack{b=D_M+1 \\ b \text{ prime}}}^{D_N} b = O\left(\frac{N^2}{L^3}\right)$$

Legendre sequences where we consider $N > 2M$ and $L = O(\log N)$, which is our use case. Therefore, by computing N Legendre symbols we are able to obtain $O(\frac{N^2}{L^3})$ sequences, implying that we compute $O(\frac{L^3}{N})$ Legendre symbols per sequence. Since N is exponential in L, this cost becomes negligible and most of the runtime is spent on extracting the sequences out of $\{j\}_N$ and lookups in the hash table.

6.3 Complexity of the algorithm

In order to give a precise estimate of the complexity of our algorithm we measure the runtime in number of operations on an L-bit word processor architecture. When L is slightly larger than 64, for example as in the case of Ethereum challenges [8], the presented complexities are fairly exact considering that 64-bit architectures are standard for a number of years already.

We assume that the following operations can be performed in $O(1)$, i. e., in a constant number of processor instructions: accessing a memory location; comparing strings of length L bits; copying, shifting or writing a bit in an L-bit string; a single look up in a hash table. Moreover, we assume that a hash table with n entries can be generated and stored in $O(n)$ instructions.

Precomputation phase. In the precomputation phase we perform several operations. Firstly, the Legendre PRF oracle is queried M times. We can either assume that every oracle query takes the time of a Legendre symbol computation or that the M symbols are given as in the Ethereum challenges case. In both cases the cost of obtaining the M consecutive symbols ($\{k\}_M$) is negligible compared to the rest of the algorithm.

Recall that during the precomputation, we extract from $\{k\}_M$ sequences of type $\left\{\frac{k+i}{d}\right\}_L$, where d ranges from 1 to $D_M = \lfloor \frac{M-1}{L-1} \rfloor$. To extract those sequences we need to compute the Legendre symbol of all d-values. This computation also takes negligible time within the whole algorithm.

On the other hand, the number of sequences of length L that are extracted form $\{k\}_M$ is $O(M^2/L)$. If done naively, i. e., by taking L symbols from $\{k\}_M$ for each sequence, this extraction takes $O(M^2)$ instructions. However, we can do this in a more efficient way. For each value of d we can extract $\left\{\frac{k+i}{d}\right\}_L$ for $0 \le i < d$ by performing L instructions for each sequence. Then, we note that for sequential values of i we need to extract only one extra bit per sequence because $\left\{\frac{k+i+d}{d}\right\}_L = \left\{\frac{k+i}{d} + 1\right\}_L$, and this can be obtained from $\left\{\frac{k+i}{d}\right\}_L$ by one shift and one extra symbol extraction. Therefore the total cost of sequence extraction part of the algorithm is $O((M^2/L^2)L + M^2/L) = O(M^2/L)$. Finally, the hash table is made on the fly as the sequences are extracted, so this cost is also $O(M^2/L)$.

The total runtime of the precomputation stage is $O(M)$ Legendre symbol computations and $O(M^2/L)$ instructions on L-bit words.

Search phase. In the search phase we select a random $j \in \mathbb{F}_p$, compute the sequence $\{j\}_N$, extract subsequences from $\{j\}_N$ and check if there is a collision in the hash table. If the collision is not found, we generate a new j and repeat the steps. To obtain the $\{j\}_N$ sequence we have to perform N Legendre symbol evaluations.

Recall that we extract from $\{j\}_N$ sequences of type $\left\{\frac{j+a}{b}\right\}_L$ for prime b-values in the range $[D_M+1, D_M+2, \ldots, D_N]$, where $D_M = \lfloor \frac{M-1}{L-1} \rfloor$ and $D_N = \lfloor \frac{N-1}{L-1} \rfloor$. Therefore, we sieve the required range to obtain the prime denominators b, and finally, compute the Legendre symbol of each b. A rough estimate of the sieving cost is $O(N \log N \log\log N)$ instructions, which is negligible compared to the last step of the algorithm. In total we need to compute $O(N/(L \log N/L)) = O(N/L^2)$ Legendre symbols.

The number of sequences that are extracted (and looked up in the hash table) from a single $\{j\}_N$ is $O(N^2/L^3)$. The required number of instructions to perform the whole extraction procedure is $O((N^2/L^3)\log L)$.

Once a collision is found we have that either $\frac{j+a}{b} = \frac{k+i}{d}$ or $\frac{j+a}{b} = -\frac{k+i}{d} - (L-1)$ from which we compute k in $O(1)$ modular operations.

Denoting by c the number of different j-values that are selected before the collision is found, the total runtime of the search phase takes $O(c(N^2/L^3)\log L)$ instructions on an L-bit word machine.

6.3.1 Runtime hypothesis

We conjecture that each sequence extracted from $\{j\}_N$ has probability of $(M^2/L)/p$ of being inside the hash table, in other words we assume that the sequences extracted from $\{j\}_N$ behave as if they were Legendre sequences of uniformly random elements of $\mathbb{F}_p$. The heuristic results indicate that this is indeed the case. Therefore, the number of trials required until a hit is found is $p/(M^2/L)$, and so if the following formula is satisfied

$$\frac{M^2}{L}\frac{cN^2}{L^3} = p$$

we find a hit with constant probability.

6.3.2 Optimal runtime

The total runtime is the sum of runtimes for both stages of the algorithm:

$$\frac{M^2}{L} + \frac{cN^2}{L^3}\log L,$$

under the hypothesis that

$$\frac{M^2}{L}\frac{cN^2}{L^3} = p.$$

The optimal runtime can then be obtained for the following values of M and N:

$$\begin{aligned} M &= \sqrt[4]{p}\sqrt{L}\sqrt[4]{\log L}, \\ N &= \sqrt[4]{p}\frac{L\sqrt{L}}{\sqrt{c}\sqrt[4]{\log L}}, \\ \text{runtime} &= \sqrt{p\log L}. \end{aligned}$$

In this scenario we have to additionally compute $O(M + M/L + cN + N/(\log N)^2)$ Legendre symbols using either oracle calls or by directly computing them. However, the cost of this is negligible with the above choices of M and N. We also note that the variable c can be chosen freely as long as $c < L^2/\log L$.

6.3.3 Runtime with a fixed M

In the special case where we are allowed to make only a fixed number M of queries to the oracle the runtime of the algorithm is a function of M. If $M \geq \sqrt[4]{p}\sqrt{L}\sqrt[4]{\log L}$, i. e., if M is larger than the number of queries we need to achieve the optimal runtime of the algorithm, then it is enough to do $\sqrt[4]{p}\sqrt{L}\sqrt[4]{\log L}$ queries, discard the rest, and achieve a $\sqrt{p\log L}$ runtime. Otherwise, when $M < \sqrt[4]{p}\sqrt{L}\sqrt[4]{\log L}$, the runtime is dominated by the search stage which has complexity

$$O\left(\frac{cN^2}{L^3}\log L\right) = O\left(\frac{pL}{M^2}\log L\right) = O\left(\frac{p\log p\log\log p}{M^2}\right)$$

assuming $L = O(\log p)$. Therefore the runtime of the algorithm for a fixed M is

$$O\left(\min\left\{\sqrt[4]{p}\sqrt{\log\log p},\, \frac{p\log p\log\log p}{M^2}\right\}\right).$$

6.4 Implementation details

In this section we explain the concrete implementation and some subtle optimizations of the presented algorithm which we used to break the Legendre PRF challenges [8]. For each challenge, a prime p and a sequence $\{k\}_M$ of $M = 2^{20}$ bits of output from the Legendre PRF were given. The challenge was to find the correct secret key k.

The proposed algorithm works in two stages, the *precomputation* and the *search* stage. During the *precomputation* stage a big hash table is generated, containing the short subsequences extracted from the given sequence of M bits. Later in the *search* stage many random short sequences are produced and checked against the entries in the hash table. Every collision that is found gives the correct key with a certain probability.

6.4.1 Precomputation stage

Since we are given the same number M of Legendre PRF oracle calls for every challenge, the *precomputation* stage is exactly the same for each instance of the challenge. In all the cases we set the length L of the short subsequences to be $L = 64$. We note that the primes in the challenges we solve have bit length larger than 64, and theoretically L should be set to approximately the bit length of the prime (as explained in Section 6.1), but for practical reasons we opt for $L = 64$ even for larger primes. The advantage of this decision is that we are working with 64-bit processor architectures so naturally all the operations as well as memory storage of the subsequences are much faster if their size is limited to the word size of the architecture. On the other hand, the proportion of false-positive collisions is increased for larger p's. However, the additional cost of validating the fake collisions is negligible and heavily outweighed by the memory and runtime savings obtained by choosing this trade-off.

From the given M bits we extract approximately $\frac{M^2}{L} = 2^{34}$ subsequences of length L, as explained in Section 6.2.2. We define the hash table with 32-bit keys (i.e., a hash table key is an integer between 0 and $2^{32} - 1$). The table is then generated by considering the 32 least significant bits of each subsequence as the hash key and storing only the remaining 32 bits (the most significant bits) in the hash entry with address determined by the key (this is simply done by setting `table[seq & (`$2^{32}-1$`)] = seq ≫ 32`). By storing not the full sequence, but only the last 32 bits we halve the space required for the hash table. Obviously, since the number of sequences we have is larger than the number of keys in the hash table, some of the entries in the table may hold more than one subsequence.

In order to minimise the memory usage we generate the table in two passes. In the first one, we extract all the subsequences and only count and store the number of different values for each hash key. After this, we get an array denoted by *positions* which for each key holds its starting position in the table. Then in the second pass we allocate the required memory for the table, extract the subsequences again and populate the table based on the *positions* array.

Each entry in the table contains one or more values that is compared with many random values later in the *search* stage. Therefore, we have to either sort the values in each individual entry and do a binary search among the sorted values in the *search* stage, or leave the values as is in the table and perform a linear search with the guessed value in the *search* stage. We decided to sort the values. However, we note that either way does not affect the performance of the algorithm because in the concrete instances of the challenge we have that the average number of values sharing the same hash key is 4, which is small enough that all the values get stored in the CPU cache memory so both linear and binary search run in approximately the same number of operations.

The *positions* array stores 2^{32} pointers each pointing to a number in the range $[0, 2^{34} - 1]$. The i'th pointer points to roughly $4 \cdot i$ since each hash key holds 4 values on average. Therefore, in order to reduce memory usage, we only store the last 32 bits of the pointer, and then we choose the full 34 bit value that is closest to $4 * i$ and has the last 32 bit equal to the ones saved. This allows us to save the *positions* array in 16GB of memory instead of 32GB if we used 64-bit values for pointers.

For the given parameter $M = 2^{20}$ and the chosen $L = 64$, the 2^{34} extracted sequences are stored in the hash table of size about 65GB, while the *positions* array occupies another 16GB. We also note that contrary to the theoretical algorithm given in Section 6.2.2 where the i and d of an extracted subsequence are stored in the hash table alongside the actual hash value, in practice we do not store i and d to minimize the memory required for the program to run. In the next section we explain how this is handled in the *search* stage. Additionally, if the amount of available memory permits we generate another data structure during the precomputation to further optimize memory accesses, namely a *bitmap*, as explained in the next section.

6.4.2 Search stage

In the *search* stage we generate random sequences of length L and query the hash table for collisions. As already explained in Section 6.2.3, computation of a single short sequence of L Legendre symbols is computationally expensive. Therefore, we apply the same technique for sequence extraction as in the *precomputation* stage with some rather important differences, vastly lowering the number of clock cycles per Legendre symbol.

Firstly, a random $j \in \mathbb{F}_p$ is selected and the Legendre symbols of N consecutive values starting from j are computed. The number N of symbols to be computed is such that $N > 2M$, as explained in Section 6.2.3. We proceed by extracting subsequences from the obtained $\{j\}_N$ sequence. Recall that subsequences extracted from $\{k\}_M$ and $\{j\}_N$ are of the form:

$$\left\{\frac{k+i}{d}\right\}_L \text{ and } \left\{\frac{j+a}{b}\right\}_L,$$

respectively, where $d \in \{1,2,\ldots,D_M\}$ for $D_M = \lfloor\frac{M-1}{L-1}\rfloor$. On the j side the denominators b are chosen from the range $[D_M+1, D_N]$ such that b is prime and $D_N = \lfloor\frac{N-1}{L-1}\rfloor$. To select prime numbers in the relevant range we implemented a simple sieve. We note that there is no need for a more sophisticated algorithm because the sieving range determined by M is rather small and the sieving is done only once.

Given the $\{j\}_N$ list as an array of $\frac{N}{8}$ bytes and an (a,b) pair, the subsequence $\left\{\frac{j+a}{b}\right\}_L$ is extracted as shown in Algorithm 13, where we use 0 for quadratic residues, and 1 for non-residues. The same algorithm is used for extracting the subsequences in the precomputation stage. However, in the precomputation stage the extraction of multiple subsequences can be further optimized. Recall from Section 6.2 that the i and a parameters used for computing the $\left\{\frac{k+i}{d}\right\}_L$ and $\left\{\frac{j+a}{b}\right\}_L$ sequences are such that $i \in [0,\ldots,M-1-(L-1)d]$ and $a \in [0,\ldots,b-1]$. Hence, when computing for example the $\left\{\frac{k+i}{d}\right\}_L$ and $\left\{\frac{k+i+d}{d}\right\}_L$, we note that those two sequences share $L-1$ Legendre symbols. This allows us to amortize the cost of extraction of "consecutive" sequences by basically extracting only one bit per sequence for each string of such "consecutive" sequences.

Algorithm 13 Extract the subsequence $\left\{\frac{j+a}{b}\right\}_L$ from sequence $\{j\}_N$

```
Input: Byte array j_list, pair $(a,b)$, length $L$
Output: Sequence $\left\{\frac{j+a}{b}\right\}_L$
procedure EXTRACT_SEQ(j_list, a, b, L)
    $seq = 0$
    for $k = 0$ to $L-1$ do
        $idx = a + k * b$
        $byte\_idx = idx >> 3$
        $bit\_idx = 7 - (idx \& 7)$
        $bit = (j\_list[byte\_idx] >> bit\_idx) \& 1$
        $seq = seq \mid (bit << j)$
    if Legendre$(b)$ == 1 then
        $seq = \sim seq$
    return seq
```

In Table 6.1 we show the performance results of our implementation for different sizes of prime p. Obtaining a single Legendre symbol by computing it takes 460 to 2700 processor cycles depending on the size of p. On the other hand, amortized cost of extracting a single symbol from sequences $\{k\}_M$ and $\{j\}_N$ requires 0.25 and 5.95 cycles, respectively, regardless of the size of p.

Each obtained subsequence is checked against the hash table for a potential collision. The

Table 6.1 – Number of clock cycles required to obtain a Legendre symbol by computation and extraction, amortized.

Prime size [bits]	64	74	84	100	148
computing	460	650	780	950	2700
extracting from $\{k\}_M$	0.25	0.25	0.25	0.25	0.25
extracting from $\{j\}_N$	5.95	5.95	5.95	5.95	5.95

check is performed in the following three steps:

1. Compute the hash key of the sequence by taking only the 32 least significant bits, and the hash value by taking the 32 most significant bits of the sequence.
2. Read two values from the *positions* array (generated in the *precomputation* stage) – value corresponding to the computed hash key and the succeeding value. Recall that these two values are the address of the hash table entry corresponding to the key and the address of the next entry, respectively.
3. If the two addresses are different, it means that the table entry of the computed key is not empty. This happens with probability $1-1/e^4$. In that case we perform a binary search for the hash value in the range specified by the starting address of the entry and the address of the succeeding entry.
4. In case of a collision, the subsequence and the variables that determine it (j, a, b) are stored for later validation.

After all the subsequences, produced by a randomly selected j, are checked against the hash table and all the collisions are saved, we proceed to the last step where the collisions are validated. Note that due to memory and performance optimizations described in Section 6.4.1 the collisions may not result in key recovery (false-positive collisions). The hash table holds only the actual sequences $\left\{\frac{k+i}{d}\right\}_L$ without the (i,d) values. On the other hand, for each collision that happened in the *search* stage we save the following data: the sequence $\left\{\frac{j+a}{b}\right\}_L$, j and (a,b). Furthermore, the data for all the collisions is sorted by the value of the sequence. Then again, in the same way as in the precomputation stage, we sequentially produce the $\left\{\frac{k+i}{d}\right\}_L$ and $\left\{-\frac{k+i}{d}-(L-1)\right\}_L$ subsequences and compare them to the sorted colliding sequences. Once a collision is obtained we can compute the guessed key with:

$$k=\frac{(j+a)\cdot d}{b}-i \quad \text{or} \quad k=-\frac{(j+a)\cdot d}{b}-d\cdot(L-1)-i,$$

and check if it is indeed the correct key by computing $\{k\}_{192}$ and comparing to the first 192 symbols given in the challenge. If the key is not found, the algorithm simply chooses the next random j and repeats all the steps above.

Optimizing the algorithm with a bitmap

For each hash table lookup we perform two random memory accesses – one access to the *positions* array and another one to the hash table. In order to decrease the number of random memory accesses we generate a bitmap during the *precomputation* stage. Each bit in the bitmap denotes if a sequence with a certain property appears in the hash table. More precisely, the m least significant bits of the sequence form an address, and the bit at this address in the bitmap signifies the existence of such a sequence in the table. Therefore, after Step 1 from the above algorithm, we proceed by reading the appropriate bit in the bitmap and only if this bit is set we continue with reading the *positions* array as before.

If we suppose that the size of the bitmap is 2^m and the size of the hash table 2^h, the probability that a sequence produces a hit in the bitmap is 2^{h-m}. Consequently, instead of accessing the memory twice per trial we access the bitmap once per trial and with probability 2^{h-m} proceed with accessing the *positions* array twice more. As a result, the number of accesses per trial is reduced from 2 to $1 + \frac{2}{2^{m-h}}$. The size of the bitmap is determined based on the available memory, for example in the solution for the 84-bit prime challenge, we have used a bitmap consisting of 2^{37} bits so in particular we had $m = 37$ and $h = 34$.

6.5 Results

In this section we compare our algorithm with the two previously published algorithms for attacking the Legendre PRF and give the results of our attempt to break the Legendre PRF challenges posed by the Ethereum foundation [8].

Table 6.2 presents the complexity of algorithms proposed in [6] and [7], and our algorithm. The algorithm of Khovratovich [6] computes sequences with on-the-fly queries on one side, i. e., each sequence of length L is obtained by querying the oracle L times, and obtains sequences on the other side by computing Legendre symbols. The advantage of this algorithm is that it does not require memory for storing sequences. This approach was improved by Beullens et al. [7] by extracting sequences instead of obtaining them by querying the oracle or computing the symbols. In this way they produce $O(M^2/L^2)$ sequences from the given sequence of length M (note that in the table we set $L = \log p$). However, they extract sequences only for (i, d) pairs with $i < d$ which results in a factor of L smaller yield than we obtain with our algorithm. Moreover, with the approach described in Section 6.2.2 the extraction of all sequences in the precomputation phase can be done in $O(M^2/\log p)$ operations. The difference in the number of extracted sequences leads to the difference in the expected number of trials that need to be performed in the search stage. However, because of the limitations imposed by the correlation properties (Section 6.2.3), we are not able to fully exploit the L times bigger table, thus the additional $\log\log p$ factor in the search stage complexity of our algorithm.

Table 6.2 – Comparison of the complexity of our algorithm and those of [6] and [7]. The complexity is given in terms of big-O number of operations on $\Theta(\log p)$-bit words. The time of Legendre symbol evaluation is denoted by t.

Algorithm	Search	Precomputation	Memory	Optimal runtime
Khovratovich [6]	$\frac{pt\log^2 p}{M}$	M	$\log p$	$\sqrt{p}\,t\log p$
Beullens et al. [7]	$\frac{p\log^2 p}{M^2}$	M^2	$\frac{M^2}{\log p}$	$\sqrt{p}\log p$
Our algorithm	$\frac{p\log p\log\log p}{M^2}$	$\frac{M^2}{\log p}$	M^2	$\sqrt{p\log\log p}$

Ethereum challenge

In each challenge we are given a prime p and $M = 2^{20}$ bits of the sequence $\{k\}_M$ as defined in Section 6.1, where k is the secret key. The challenge is to recover the key k. The five challenges and their corresponding security levels are shown in Table 6.3. We note that security levels in the table are computed based on the complexity of the attack by Khovratovich [6], and that the algorithm presented in this chapter lowers those bounds. Finally, we successfully solved the challenges #0, #1 and #2.

Table 6.3 – Legendre PRF challenges [8] with security levels estimated based on [6] and the new security estimates.

Challenge	Prime size [bits]	Security old [bits]	Security new [bits]
0	64	44	32
1	74	54	40
2	84	64	50
3	100	80	66
4	148	128	114

Our algorithm was implemented in C and compiled with the gcc compiler. All the parallelization was done with OpenMP primitives. Testing and experiments were conducted on a desktop PC equipped with an Intel Xeon E5-1650 processor with 6 cores running at 3.5GHz, and 128GB of RAM. The first two challenges (#0 and #1) were solved on this PC, while for the third challenge (#2) we used 16 nodes of the EPFL IC cluster. Each node has two Intel Xeon E5-2680 v3 processors with 12 cores each running at 2.5GHz, and 192GB of RAM.

The precomputation stage in all three cases took less than 20 minutes on the desktop PC. During the precomputation we produce three files for each challenge:

- 65GB file containing the hash table,
- 16GB file containing the positions array,
- 16GB file containing the bitmap which we set at 2^{37} bits.

In the search stage, all three files are loaded in RAM so the program requires in total almost 100GB of memory.

In Table 6.4 we show the results of the experiment. The complexity of the search stage of our attack expressed as the expected number of trials that need to be done before the solution is found is $\frac{p \cdot L}{M^2}$, where by trial we denote a single hash table collision check with a random subsequence generated as explained in 6.4.1. We show the expected number of trials for each challenge in the second column of Table 6.4, while the third column shows the actual number of trials performed by the algorithm before the solutions are found. For the first two challenges we run the algorithm several times with different seeds in order to record the required number of trials and validate the expected numbers given in the complexity analysis in Section 6.3. The numbers shown in the third column are the average of all the conducted experiments ($2^{30.78}$ and $2^{39.81}$ for challenges #0 and #1 respectively). We note that the observed variance is considerable, which can be explained by the fact that the expected number of trials for a prime p is pL/M^2 and the variance is about p^2L^2/M^4.

Table 6.4 – Results and estimates for solving the Legendre PRF challenges. The expected and actual number of core-hours for challenges #0 and #1 is based on measuring the performance of the implementation on our desktop PC with Intel Xeon E5-1650 at 3.5GHZ, while the numbers for the other three challenges are based on performance of Intel Xeon E5-2680 v3 at 2.5GHz CPU available in the EPFL IC cluster.

	Expected # trials	Observed # trials	Expected core-hours	Observed core-hours	k
#0	2^{30}	$2^{30.78}$	0.08	0.14	650282827113560997
#1	2^{40}	$2^{39.53}$	82	59	1661947092456596025913 3
#2	2^{50}	$2^{46.97}$	1.4e5	1.72e4	187320452088744099523844
#3	2^{66}	-	9.1e9	-	-
#4	2^{114}	-	2.5e24	-	-

The solution for challenge #2 was the first and so far the only one that is published. As shown in Table 6.4, the actual number of trials that were done before the key was found is $2^{46.97} = 1.38e14$ which is far less than expected. This can be explained by the large variance and by sheer luck. The implementation version that was used for challenge #2 can perform $2.2e6$ trials per second on a single core of a processor in the EPFL IC cluster. The number of trials per second is slightly higher on the desktop PC since its CPU is working at a higher frequency. In Table 6.4 we also give estimates for the two most difficult challenges (#3 and #4), which are out of reach with the proposed attack and its implementation.

A Appendix

A.1 Additional information on the experiments and the results of Chapter 3

Table A.1 gives the equations for the linear and the quadratic extrapolation together with the extrapolated values of r for a DFR of 2^{-23}, 2^{-64}, and 2^{-128}. It covers the tuple (scheme, level, decoder, X), where decoder $\in$ {BG=Black-Gray, BF=BackFlip^{+}}.

The BIKE specification [9] chooses r to be the minimum required for achieving a certain security level, and the best bandwidth trade-off. It also indicates that it is possible to increase r by "plus or minus 50%" (leaving w, t fixed) without reducing the complexity of the best known key/message attacks. This is an interesting observation. For example, increasing the BIKE-1 Level-3 $r = 19853$ by 50% gives $r = 29779$ which is already close to the BIKE-1 Level-5 that has $r = 32749$ (of course with different w and t). We take a more conservative approach and restrict r values to be at most 30% above their CCA values stated in [9]. Table A.1 labels values beyond this limit as N/A.

Table A.1 – The linear and the quadratic extrapolation equations, and the computed r values for a given DFR. The cases labeled with N/A are those where the value of r to achieve a target DFR could not be found in the range $[0.7r', 1.3r']$, where r' is the recommended value for IND-CCA security in [9]

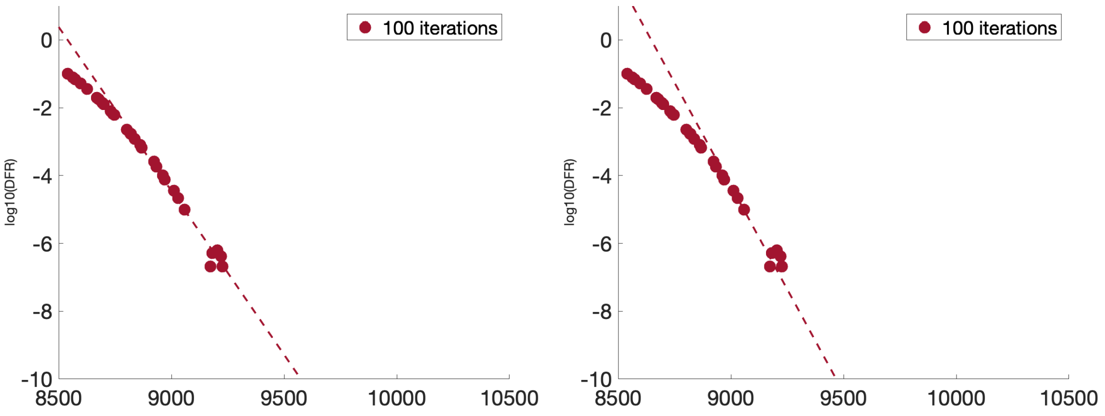
KEM	Lev.	Decoder	Iter.	Lin. start	Lin. eq. (a,b) s.t. $\log_{10}\texttt{DFR} = ar + b =$	2^{-23}	2^{-64}	2^{-128}	Quad. eq. (a,b,c) s.t. $\log_{10}\texttt{DFR} = ar^2 + br + c =$	2^{-23}	2^{-64}	2^{-128}
BIKE-1	1	BG	3	10	$(-1.25\text{e}{-2}, 121)$	10253	11261	12781	$(-1.05\text{e}{-5}, 1.97\text{e}{-1}, -927)$	10253	10789	11317
BIKE-1	1	BG	4	9	$(-1.45\text{e}{-2}, 140)$	10163	11003	12347	$(-1.16\text{e}{-5}, 2.18\text{e}{-1}, -1020)$	10139	10667	11197
BIKE-1	1	BG	5	9	$(-1.49\text{e}{-2}, 144)$	10133	10973	12251	$(-1.18\text{e}{-5}, 2.20\text{e}{-1}, -1030)$	10133	10667	11171
BIKE-1	1	BF	8	9	$(-5.40\text{e}{-3}, 49.8)$	10499	12781	N/A	$(-6.86\text{e}{-7}, 8.11\text{e}{-3}, -16.8)$	10459	12149	14107
BIKE-1	1	BF	9	6	$(-6.92\text{e}{-3}, 63.8)$	10253	12011	14797	$(-1.16\text{e}{-6}, 1.62\text{e}{-2}, -50.9)$	10253	11579	13109
BIKE-1	1	BF	10	8	$(-8.40\text{e}{-3}, 77.6)$	10067	11549	13829	$(-1.88\text{e}{-6}, 2.90\text{e}{-2}, -108)$	10067	11197	12437
BIKE-1	1	BF	11	7	$(-1.12\text{e}{-2}, 104)$	9949	11069	12781	$(-3.41\text{e}{-6}, 5.77\text{e}{-2}, -243)$	9949	10883	11867
BIKE-1	3	BG	3	10	$(-6.97\text{e}{-3}, 133)$	20051	21821	24659	$(-2.39\text{e}{-6}, 8.64\text{e}{-2}, -780)$	19997	21059	22189
BIKE-1	3	BG	4	10	$(-8.70\text{e}{-3}, 166)$	19853	21269	23459	$(-3.34\text{e}{-6}, 1.22\text{e}{-1}, -1110)$	19813	20717	21683
BIKE-1	3	BG	5	10	$(-9.10\text{e}{-3}, 173)$	19813	21139	23251	$(-3.67\text{e}{-6}, 1.34\text{e}{-1}, -1220)$	19763	20627	21557
BIKE-1	3	BF	8	10	$(-5.36\text{e}{-3}, 99.7)$	19867	22171	25771	$(-9.08\text{e}{-7}, 3.02\text{e}{-2}, -248)$	19853	21523	23339
BIKE-1	3	BF	9	9	$(-6.14\text{e}{-3}, 114)$	19661	21661	24781	$(-1.37\text{e}{-6}, 4.71\text{e}{-2}, -403)$	19661	21059	22613
BIKE-1	3	BF	10	5	$(-6.51\text{e}{-3}, 120)$	19469	21379	24371	$(-8.64\text{e}{-7}, 2.69\text{e}{-2}, -204)$	19469	21011	22787
BIKE-1	3	BF	11	6	$(-7.05\text{e}{-3}, 130)$	19373	21101	23869	$(-1.66\text{e}{-6}, 5.69\text{e}{-2}, -488)$	19373	20693	22067

A.1.1 Achieving the same DFR bounds as in [1]

We ran experiments with BackFlip$^+$ and $X_{BF} = 100$ for BIKE-1 Level-1, scanning all the 34 legitimate $r \in [8500, 9340]$ (prime r values such that $x^r - 1$ is a primitive polynomial) with a sufficient number of tests for every value (sufficient in a sense that at least several failures are observed). Applying our extrapolation methodology (see Section 3.4) to the acquired data leads to the results illustrated in Figure A.1 Panels (a) and (b). The figure highlights the pairs (DFR; r) for DFR 2^{-64} and 2^{-128} with the smallest possible r. For example, with $r = 12539$ the linear extrapolation gives DFR of 2^{-128}. Note that [9] claims a DFR of 2^{-128} for a smaller $r = 11779$. For comparison, with $r = 11779$ our methodology gives a DFR of 2^{-104}. We can guess that either different TTL values were used for every r, or that other r values were used, or that a different extrapolation methodology was applied.

We show one possible methodology ("two larger r's fit") that gives a DFR of $\sim 2^{-128}$ with $r = 11779$ when applied to the acquired data: a) Ignore the points from the data-set for which 100−DFR is too low to be calculated reliably (e. g., the five lower points in Figure A.1); b) Draw a line through the last two remaining data points with the highest values of r. The rationale is that the "linear regime" of the DFR evolution starts for values of r that are beyond those that can be estimated in an experiment. Under a concavity assumption, a line drawn through two data points where r is smaller than the starting point of the linear regime leads to an extrapolation that is lower-bounded by the "real" linear evolution. With this approach, the question is how to choose the two points for which experimental data is obtained and from which the DFR is extrapolated.

This shows that different ways to acquire and interpret the data give different upper bounds for the DFR. The gap between the values of r for which we get the DFR by performing the experiments and the values of r for which we obtain the DFR by extrapolation is large. Therefore, the extrapolated DFR results are sensitive to the chosen methodology. It is interesting to note that if we take our data points for Black-Gray and $X_{BG} = 5$ and use the two larger r's fit extrapolation, we can find two points that would lead to 2^{-128} and $r = 11779$, while more conservative methodology gives only 2^{-101}.



(a) lin. ext., our method. (DFR, r) = (2^{-64}; 10589), (2^{-128}; 12539)

(b) lin. ext. two larger r's fit. (DFR, r) = (2^{-64}; 10253), (2^{-128}; 11813)

Figure A.1 – BIKE-1 Level-1 BackFlip$^+$ different extrapolation methods. See the text for details. The sub-captions detail the (DFR; r) for DFR values: 2^{-64}, 2^{-128}.

A.2 Additional information on the experiments and the results of Chapter 4

Table A.2 shows the DFR extrapolation results for BIKE-1 at security level 1 for different decoders. The number of tests for every value of r is $3.84M$ for $r \in [9349, 9901]$ and $384M$ for $r \in [9907, 10139]$. For the "two larger r's fit" extrapolation method (see Appendix A.1.1) we chose: $r = 10141$ running 384M tests, and $r = 10259$ running 7.296 billion tests.

Figures A.2, A.3, A.4, and A.5 present the experimental data points and the extrapolation lines given in Table A.2.

Table A.2 – The *best linear* and the *two points* extrapolation equations, and the estimated r values for three target DFR. Level is abbreviated to Lvl, the number of iterations is abbreviated to iter, linear is abbreviated to lin., equation is abbreviated to eq. The Lin. start column indicates the index of the first value of r where the linear fit starts. The 5 column (number of steps) is the indication for the overall performance of the decoder (lower is better).

KEM	Lvl	Decoder	Iter	Steps	Lin. start	Best lin. fit eq. s.t. $\log_{10}\mathrm{DFR} = ar + b =$	2^{-23}	2^{-64}	2^{-128}	Two points line eq. (a,b) $\log_{10}\mathrm{DFR} = ar + b =$	2^{-23}	2^{-64}	2^{-128}
BIKE-1	1	BG	3	9	15	$(-1.27\mathrm{e}{-2}, 124)$	10253	11213	12739	$(-1.33\mathrm{e}{-2}, 129)$	10253	11171	12619
BIKE-1	1	BG	4	12	8	$(-1.45\mathrm{e}{-2}, 140)$	10163	11003	12347	$(-1.63\mathrm{e}{-2}, 158)$	10163	10909	12107
BIKE-1	1	BG	5	15	13	$(-1.61\mathrm{e}{-2}, 156)$	10133	10909	12107	$(-1.70\mathrm{e}{-2}, 165)$	10133	10853	11987
BIKE-1	1	BGB	4	9	13	$(-1.38\mathrm{e}{-2}, 134)$	10253	11093	12491	$(-1.40\mathrm{e}{-2}, 136)$	10253	11083	12491
BIKE-1	1	BGB	5	11	13	$(-1.52\mathrm{e}{-2}, 147)$	10163	10973	12227	$(-1.41\mathrm{e}{-2}, 136)$	10163	11027	12413
BIKE-1	1	BGB	6	13	7	$(-1.48\mathrm{e}{-2}, 143)$	10133	10973	12269	$(-1.54\mathrm{e}{-2}, 149)$	10133	10949	12197
BIKE-1	1	BGF	5	7	14	$(-1.40\mathrm{e}{-2}, 137)$	10301	11171	12539	$(-1.44\mathrm{e}{-2}, 141)$	10301	11131	12491
BIKE-1	1	BGF	6	8	13	$(-1.53\mathrm{e}{-2}, 149)$	10253	11027	12277	$(-1.61\mathrm{e}{-2}, 157)$	10253	10973	12197
BIKE-1	1	BGF	7	9	13	$(-1.61\mathrm{e}{-2}, 157)$	10181	10949	12149	$(-1.68\mathrm{e}{-2}, 164)$	10181	10949	12107
BIKE-1	1	B	4	8	15	$(-8.69\mathrm{e}{-3}, 82.4)$	10259	11699	13901	$(-8.05\mathrm{e}{-3}, 75.8)$	10301	11813	14221
BIKE-1	1	B	5	10	15	$(-1.02\mathrm{e}{-2}, 96.3)$	10133	11437	13229	$(-9.56\mathrm{e}{-3}, 89.9)$	10133	11437	13451
BIKE-1	1	B	6	12	14	$(-1.08\mathrm{e}{-2}, 101)$	10067	11213	13037	$(-9.52\mathrm{e}{-3}, 88.8)$	10067	11437	13397

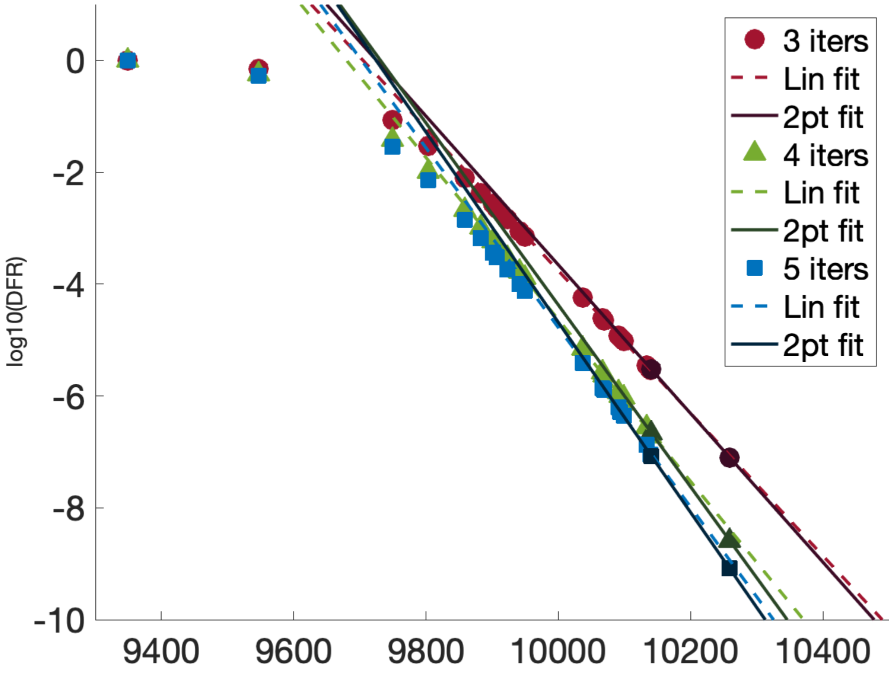


Figure A.2 – Extrapolations of BIKE-1 Level-1 using the Black-Gray (BG) decoder.

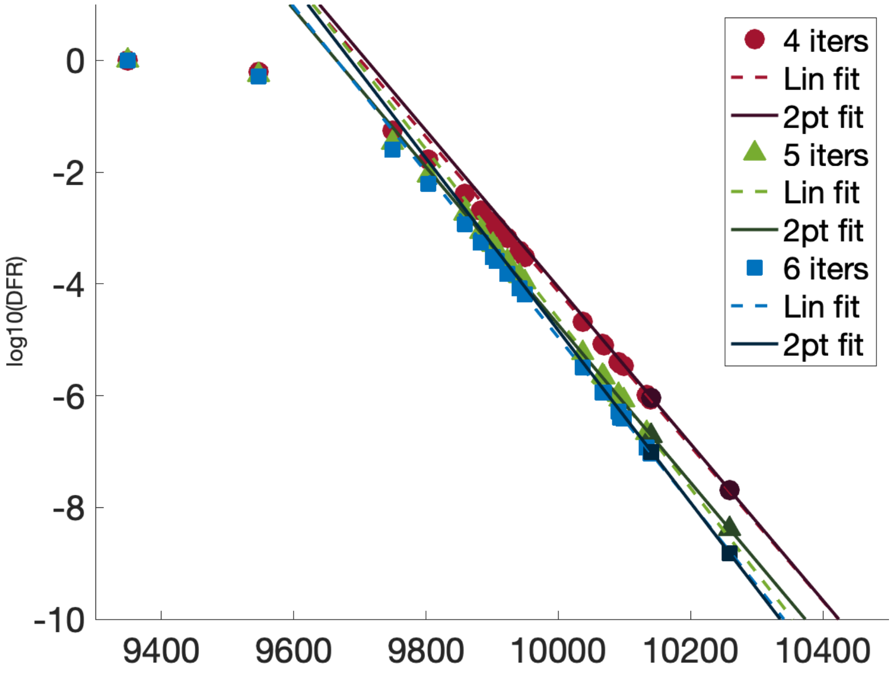


Figure A.3 – Extrapolations of BIKE-1 Level-1 using the Black-Gray-Black (BGB) decoder.

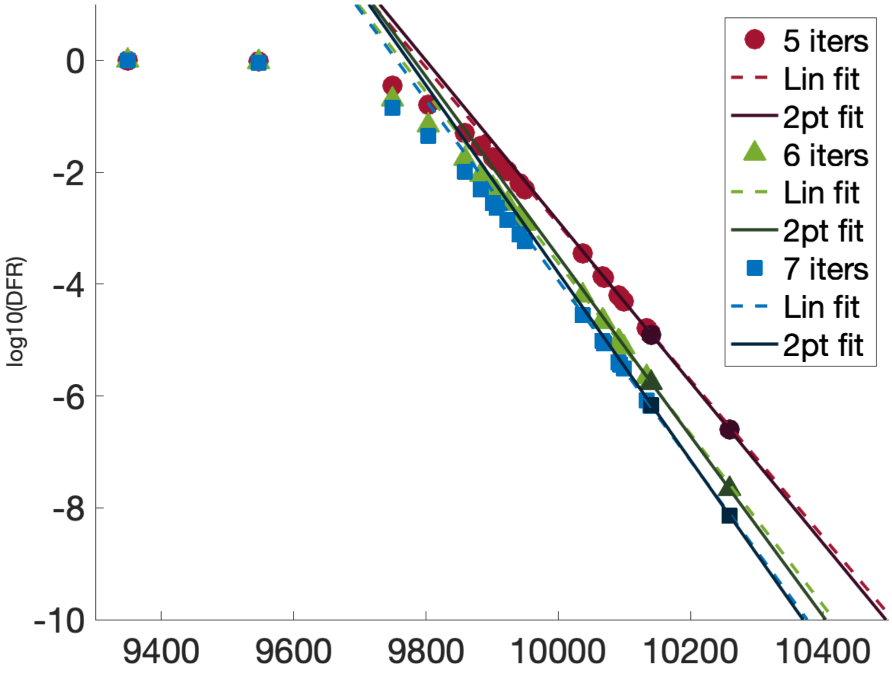


Figure A.4 – Extrapolations of BIKE-1 Level-1 using the Black-Gray-Flip (BGF) decoder.

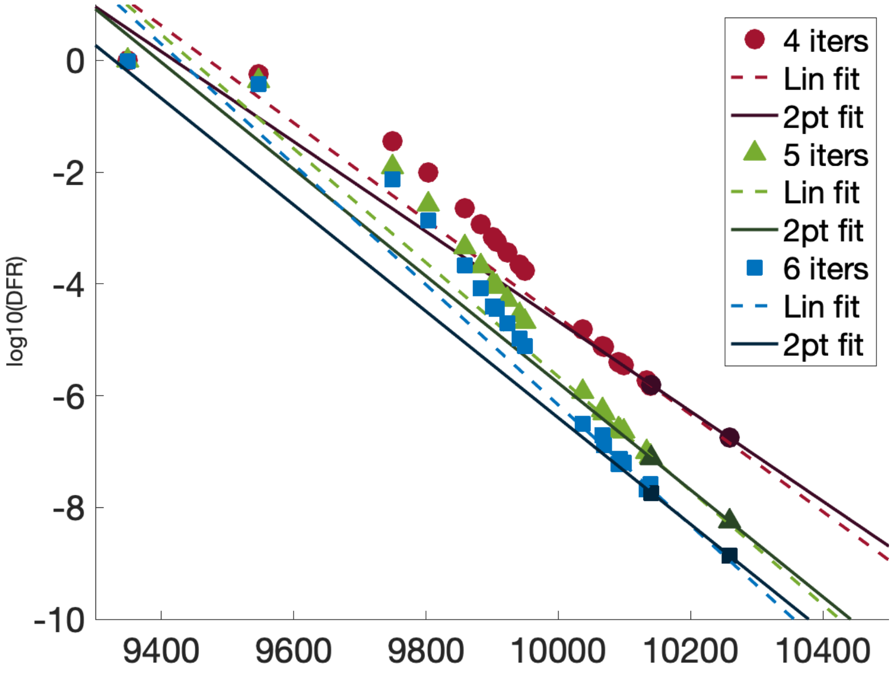


Figure A.5 – Extrapolations of BIKE-1 Level-1 using the Black (B) decoder.

A.2.1 Optimized and secured implementation of syndrome rotation

Listing A.1 presents the implementation of the “small” rotation function required for rotating a syndrome (more details can be found in Section 4.5. The implementation is using the **AVX512** instruction set.

```
__m512i curr, next, a0, a1, idx0, idx1, num_full_qw;
uint64_t bitscount0 = bitscount / 64;
uint64_t bitscount1 = bitscount % 64;

num_full_qw = _mm512_set1_epi8(bitscount0);
one         = _mm512_set1_epi64(1);
previous    = _mm512_setzero_si512();
idx0        = _mm512_setr_epi64(0, 1, 2, 3, 4, 5, 6, 7);
idx0        = _mm512_add_epi64(idx0, num_full_qw);
idx1        = _mm512_add_epi64(idx0, one);

next = _mm512_load_si512(in[0]);

for(int i = 0; i < R_ZMM; i++) {
  curr = next;
  next = _mm512_load_si512(in[i+1]);
  a0 = _mm512_permutex2var_epi64(curr, idx, next);
  a1 = _mm512_permutex2var_epi64(curr, one, next);
  a0 = _mm512_srli_epi64(a0, bitscount1);
  a1 = _mm512_slli_epi64(a1, 64 - bitscount1);
  _mm512_store_si512(out[i], _mm512_or_si512(a0, a1));
}
```

Listing A.1 – Right rotate of bits stored in **R_ZMM** input 512-bit registers by **bitscount** places using **AVX512** instructions.

A.3 Additional information on the experiments and the results of Chapter 5

A.3.1 Generating permutation map with AVX2 instructions

In Listing A.2 we show the **AVX2** implementation of a function that generates the permutation map given the l parameter, as explained in Section 5.3.1.

```c
void gen_permutation_map (uint16_t map[R], uint16_t l) {
  __m512i curr, inc, rval, zero;
  uint32_t mask;
  // Initialization: compute the first 16 map elements
  for (int i = 0; i < 16; i++)
    map[i] = (i * l) % R;

  rval = BCAST_I16(R);
  zero = BCAST_I16(0);
  inc  = BCAST_I16((l * 16) % R);
  inc  = SUB_I16(inc, rval);

  // Load the initial 16 values into the register
  curr = LOAD(map);

  // Generate the rest of the map elements
  for (int i = 0; i < ceil(R / 16); i++) {
    curr = ADD_I16(curr, inc);
    mask = CMP_I16(zero, curr, CMP_GT);
    curr = ADD_I16(curr, rval & mask);
    STORE(&map[i * 16], curr);
  }
}
```

Listing A.2 – Permutation map generation with **AVX2** instructions (the actual names of **AVX2** instructions are replaced with upper-case macro names for clarity)

A.3.2 Squaring using PCLMUL and VPCLMUL

As explained in Section 5.3.3, squaring of a binary polynomial a can be performed by squaring every digit of a. On platforms that offer the **PCLMUL** instruction, this instruction can be used to multiply two 64-bit digits, and therefore, can be used to square a digit. The **PCLMUL** instruction takes as an input two 128-bit values and an additional parameter denoting which 64-bit words of the input should be multiplied. For example, mask 0x00 instructs **PCLMUL** to multiply the two lower 64-bit words of the inputs, while the mask 0x01 requests multiplication of the lower word of the first input and the higher word of the second input. The code that implements polynomial squaring with **PCLMUL** is shown in Listing A.3.

```
void gf2x_sqr(uint64_t *c, const uint64_t *a) {
  for (size_t i = 0; i < ceil(R / 128); i++) {
      __m128i va = LOAD(&a[i*2]);
      STORE(&c[i*4],   PCLMUL(va, va, 0x00));
      STORE(&c[i*4+2], PCLMUL(va, va, 0x11));
  }
}
```

Listing A.3 – Squaring a polynomial in $\mathcal{R}$ with **PCLMUL** instruction.

When **VPCLMUL** instruction is available, the vectorized version of **PCLMUL**, we can execute four 64-bit multiplications in parallel. In Section 5.3.3, we explain how **VPCLMUL** works and moreover, how to apply it to compute a square of a binary polynomial. In Listing A.4 we present the described implementation.

```
void gf2x_sqr(uint64_t *c, const uint64_t *a) {

  __m512i perm_mask = _mm512_set_epi64(7, 3, 6, 2, 5, 1, 4, 0);

  for (size_t i = 0; i < ceil(R / 512); i++) {
      __m512i va = LOAD(&a[i*8]);
      va = PERMUTE(va, perm_mask);
      STORE(&c[i*16],   VPCLMUL(va, va, 0x00));
      STORE(&c[i*16+8], VPCLMUL(va, va, 0x11));
  }
}
```

Listing A.4 – Squaring a polynomial in $\mathcal{R}$ with **VPCLMUL** instruction.

A.3.3 A 4×4 digits multiplication using VPCLMUL

Listing A.5 shows the implementation of the function that multiplies four by four 64-bit digits of a binary polynomial. The algorithm is explained in details in Section 5.3.3 and Figure 5.4.

```c
void mul4x4(__m512i *h, __m512i *l, __m512i a, __m512i b) {

    __m512i sa = PERM64(a, _MM_SHUFFLE(2, 3, 0, 1));
    __m512i sb = PERM64(b, _MM_SHUFFLE(2, 3, 0, 1));

    sa = sa ^ a;
    sb = sb ^ b;

    __m512i u = VPCLMUL(a,  b,  0x00);
    __m512i v = VPCLMUL(a,  b,  0x11);
    __m512i w = VPCLMUL(sa, sb, 0x00);

    w = w ^ u ^ v;
    w = PERM64(w, _MM_SHUFFLE(2, 3, 0, 1));

    *l = XOR_MASKED(u, w, 0xaa);
    *h = XOR_MASKED(v, w, 0x55);
}
```

Listing A.5 – Multiplying four 64-bit digits of two binary polynomials using **AVX512** and **VPCLMUL** instructions as explained in Section 5.3.3 and Figure 5.4

A.3.4 Example of k-square versus series of k squares

In Section 5.2 we explain that for values of $k < k_{thr}$ instead of performing k-squaring we perform k regular squares. The threshold k_{thr} depends on the implementation and the platform. For example, in Table A.3 we compare the performance of squaring and k-squaring in $\mathcal{R}$ using **AVX512** and **VPCLMUL** instructions, and compute the thresholds.

Table A.3 – Squaring and k-squaring in $\mathcal{R}$ using our code (**AVX512** and **VPCLMUL**). Columns 2 and 3 count cycles. The threshold is computed by k_{thr} = k-square/square. The r values correspond to the IND-CCA variants of BIKE for Level-1/3.

r	k-square	square	k_{thr}
11779	16000	230	69
24821	35000	510	68

A.3.5 Performance results

Table A.4 – Performance of our implementations of inversion in $\mathbb{F}_2[x]/(x^r - 1)$ for a set of r values with different $wt(r-2)$. The NTL and OSSL columns denote the runtime of the inversion from the corresponding libraries ([4, 5]). The remaining columns represent our implementation: (a) with **AVX2**; (b) with **AVX512**; (c) with **AVX512** and **VPCLMUL**; (d) fully portable implementation, independent of any platform; (e) portable with **PCLMUL** instruction used for multiplication and squaring; columns labeled with "*" denote implementations with pre-computed permutation maps. The runtime is measured in millions of cycles.

r	$wt(r-2)$	NTL	OSSL	(a)	(a)*	(b)	(b)*	(c)	(c*)	(d)	(d)*	(e)	(e)*
12323	4	6.75	49.19	12.79	0.56	0.54	0.52	0.43	0.41	12.64	0.95	0.78	0.59
11779	5	5.86	42.61	11.81	0.54	0.54	0.51	0.44	0.41	11.42	1.15	0.79	0.57
12347	6	6.52	48.67	15.03	0.63	0.60	0.58	0.47	0.45	14.67	1.15	0.86	0.64
11789	7	6.10	43.83	12.95	0.59	0.58	0.55	0.45	0.44	12.74	1.05	0.84	0.62
11821	8	5.99	44.98	14.04	0.62	0.61	0.59	0.48	0.46	13.98	1.10	0.89	0.66
11933	9	6.22	43.31	14.50	0.65	0.64	0.63	0.52	0.49	14.28	1.18	0.94	0.69
12149	10	6.37	46.60	15.60	0.71	0.70	0.67	0.55	0.52	15.31	1.29	1.02	0.75
12157	11	6.30	47.00	16.57	0.74	0.72	0.70	0.58	0.55	16.23	1.33	1.06	0.78
25603	4	9.00	213.84	39.10	1.72	1.65	1.61	1.28	1.24	38.62	2.78	2.33	1.75
24659	5	8.67	188.42	41.75	1.71	1.66	1.61	1.30	1.24	40.94	3.10	2.35	1.77
24677	6	8.61	193.27	44.41	1.83	1.74	1.71	1.35	1.32	43.53	3.19	2.48	1.88
24733	7	8.77	204.55	46.47	1.89	1.79	1.77	1.40	1.35	45.65	3.30	2.56	1.93
24821	8	9.07	185.17	49.16	2.02	1.92	1.87	1.51	1.49	49.00	3.24	2.73	2.08
25453	9	8.86	197.20	51.42	2.20	2.09	2.06	1.61	1.54	50.86	3.93	2.97	2.26
24547	10	8.32	182.11	45.81	2.08	1.99	1.95	1.61	1.53	44.46	4.07	2.88	2.13
24533	11	8.79	175.41	47.10	2.14	2.08	2.00	1.67	1.60	46.14	4.11	3.00	2.21
24509	12	8.47	181.95	50.24	2.20	2.13	2.07	1.66	1.61	50.06	3.67	3.05	2.27

Table A.5 – Speedup of our implementations of inversion in $\mathbb{F}_2[x]/(x^r-1)$ compared to NTL with GF2X [4]. Columns 3-8 represent the speedup over NTL of the following implementation: (a) **AVX2**; (b) **AVX512**; (c) **AVX512** and **VPCLMUL**; (d) PORTABLE; (e) **PCLMUL**; columns labeled with "*" denote implementations with pre-computed permutation maps. The speedup is measured for a set of r values with different $wt(r-2)$.

r	$wt(r-2)$	(a)	(a)*	(b)	(b)*	(c)	(c*)	(d)	(d)*	(e)	(e)*
12323	4	11.51	12.15	12.50	13.02	15.68	16.55	0.53	0.53	7.12	8.68
11779	5	10.26	10.80	10.85	11.45	13.32	14.37	0.50	0.51	5.11	7.46
12347	6	10.11	10.36	10.86	11.26	13.87	14.42	0.43	0.44	5.64	7.61
11789	7	9.85	10.37	10.44	11.03	13.44	13.96	0.47	0.48	5.79	7.26
11821	8	9.10	9.61	9.89	10.15	12.42	13.10	0.43	0.43	5.46	6.75
11933	9	8.97	9.55	9.67	9.93	12.03	12.70	0.43	0.44	5.29	6.59
12149	10	8.48	8.99	9.09	9.46	11.54	12.23	0.41	0.42	4.93	6.23
12157	11	8.10	8.48	8.72	9.04	10.91	11.46	0.38	0.39	4.75	5.94
25603	4	5.15	5.23	5.45	5.59	7.06	7.23	0.23	0.23	3.23	3.87
24659	5	4.89	5.06	5.22	5.40	6.66	6.98	0.21	0.21	2.80	3.69
24677	6	4.58	4.71	4.96	5.04	6.38	6.54	0.19	0.20	2.69	3.47
24733	7	4.54	4.65	4.91	4.97	6.25	6.48	0.19	0.19	2.66	3.43
24821	8	4.37	4.49	4.72	4.84	6.01	6.10	0.18	0.19	2.80	3.32
25453	9	3.92	4.03	4.23	4.31	5.51	5.74	0.17	0.17	2.25	2.98
24547	10	3.91	4.00	4.18	4.27	5.18	5.44	0.18	0.19	2.05	2.88
24533	11	3.97	4.11	4.23	4.39	5.28	5.49	0.19	0.19	2.14	2.93
24509	12	3.73	3.85	3.98	4.10	5.10	5.27	0.17	0.17	2.31	2.78

Table A.6 – BIKE-2 key generation performance when our implementation of the inversion algorithm is used. Columns represent the following implementations: (a) **AVX2**; (b) **AVX512**; (c) **AVX512** and **VPCLMUL**; (d) PORTABLE; (e) **PCLMUL**; columns labeled with "*" denote implementations with pre-computed permutation maps. The runtime is measured in thousands of cycles.

r	$wt(r-2)$	(a)	(a)*	(b)	(b)*	(c)	(c*)	(d)	(d)*	(e)	(e)*
12323	4	642	644	581	587	470	473	13829	13574	1344	1121
11779	5	625	630	585	591	477	479	12651	12530	1454	1304
12347	6	709	708	642	650	512	516	15882	15773	1431	1332
11789	7	672	674	623	629	500	510	13959	13635	1516	1226
11821	8	709	715	656	658	520	529	15101	14786	1556	1272
11933	9	743	751	692	696	551	561	15552	15226	1659	1334
12149	10	806	806	743	748	594	593	16746	16291	1818	1451
12157	11	829	842	769	772	616	621	17659	17288	1865	1493
25603	4	1907	1906	1773	1762	1440	1391	42186	41515	3910	3241
24659	5	1944	1913	1777	1781	1406	1408	44485	44131	3881	3550
24677	6	2024	1994	1865	1892	1474	1454	47236	46788	3979	3659
24733	7	2126	2097	1918	1908	1509	1504	49141	48942	4107	3769
24821	8	2246	2222	2064	2061	1648	1607	52216	51642	4392	3697
25453	9	2420	2414	2241	2230	1732	1703	54726	53957	4786	4388
24547	10	2324	2299	2159	2172	1700	1706	48211	47725	5001	4493
24533	11	2367	2348	2213	2190	1763	1733	49638	49292	5037	4587
24509	12	2454	2432	2271	2239	1781	1765	53145	52372	5004	4115

Bibliography

[1] N. Sendrier and V. Vasseur, “On the Decoding Failure Rate of QC-MDPC Bit-Flipping Decoders,” in *Post-Quantum Cryptography* (J. Ding and R. Steinwandt, eds.), vol. 2, (Cham), pp. 404–416, Springer International Publishing, 2019.

[2] N. Drucker, S. Gueron, and D. Kostic, “On constant-time QC-MDPC decoding with negligible failure rate.” Cryptology ePrint Archive, Report 2019/1289, 2019.

[3] A. Guimarães, D. F. Aranha, and E. Borin, “Optimized implementation of QC-MDPC code-based cryptography,” vol. 31, no. 18, p. e5089, 2019.

[4] V. Shoup, “Number theory c++ library (ntl) version 11.3.2.” http://www.shoup.net/ntl, November 2018.

[5] The OpenSSL Project, “OpenSSL 1.1.1: The open source toolkit for SSL/TLS.” https://github.com/openssl/openssl.

[6] D. Khovratovich, “Key recovery attacks on the Legendre PRFs within the birthday bound.” Cryptology ePrint Archive, Report 2019/862, 2019. https://eprint.iacr.org/2019/862.

[7] W. Beullens, T. Beyne, A. Udovenko, and G. Vitto, “Cryptanalysis of the Legendre PRF and Generalizations,” *IACR Transactions on Symmetric Cryptology*, vol. 2020, pp. 313–330, May 2020.

[8] D. Feist, “Legendre pseudo-random function,” 2019. https://legendreprf.org.

[9] N. Aragon, P. S. L. M. Barreto, S. Bettaieb, L. Bidoux, O. Blazy, J.-C. Deneuville, P. Gaborit, S. Gueron, T. Güneysu, C. A. Melchor, R. Misoczki, E. Persichetti, N. Sendrier, J.-P. Tillich, and G. Zémor, “BIKE: Bit Flipping Key Encapsulation,” 2019. Submission to the Round-2 of the NIST PQC Standardization Project,
https://bikesuite.org/files/round2/spec/BIKE-Spec-2019.06.30.1.pdf.

[10] R. L. Rivest, A. Shamir, and L. Adleman, “A method for obtaining digital signatures and public-key cryptosystems,” *Commun. ACM*, vol. 21, p. 120–126, Feb. 1978.

[11] N. Koblitz, “Elliptic curve cryptosystems,” *Mathematics of Computation*, vol. 48, pp. 203–209, Jan. 1987.

[12] V. S. Miller, "Use of elliptic curves in cryptography," in *Advances in Cryptology*, CRYPTO '85, (Berlin, Heidelberg), p. 417–426, Springer-Verlag, 1985.

[13] A. K. Lenstra and H. W. Lenstra, *The development of the number field sieve*, vol. 1554 of *Lecture notes in mathematics.* Berlin [etc.: Springer-Verlag, 1993.

[14] P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM J. Comput.*, vol. 26, p. 1484–1509, Oct. 1997.

[15] NIST, "Post-Quantum Cryptography." https://csrc.nist.gov/projects/post-quantum-cryptography, 2019. Last accessed 20 Aug 2019.

[16] E. Berlekamp, R. McEliece, and H. van Tilborg, "On the inherent intractability of certain coding problems (corresp.)," *IEEE Transactions on Information Theory*, vol. 24, no. 3, pp. 384–386, 1978.

[17] NIST, "Status Report on the First Round of the NIST Post-Quantum Cryptography Standardization Process," 2019. https://nvlpubs.nist.gov/nistpubs/ir/2019/NIST.IR.8240.pdf.

[18] N. Aragon, P. S. L. M. Barreto, S. Bettaieb, L. Bidoux, O. Blazy, J.-C. Deneuville, P. Gaborit, S. Gueron, T. Güneysu, C. A. Melchor, R. Misoczki, E. Persichetti, N. Sendrier, J.-P. Tillich, and G. Zémor, "BIKE: Bit Flipping Key Encapsulation," 2020. Prospective submission to the Round-3 of the NIST PQC Standardization Project,
https://bikesuite.org/files/v4.0/BIKE_Spec.2020.05.03.1.pdf.

[19] N. Drucker, S. Gueron, and D. Kostic, "QC-MDPC Decoders with Several Shades of Gray," in *Post-Quantum Cryptography* (J. Ding and J.-P. Tillich, eds.), (Cham), pp. 35–50, Springer International Publishing, 2020.

[20] N. Drucker, S. Gueron, and D. Kostic, "Fast polynomial inversion for post quantum qc-mdpc cryptography," in *Cyber Security Cryptography and Machine Learning* (S. Dolev, V. Kolesnikov, S. Lodha, and G. Weiss, eds.), (Cham), pp. 110–127, Springer International Publishing, 2020.

[21] R. Gallager, "Low-density parity-check codes," *IRE Transactions on Information Theory*, vol. 8, pp. 21–28, January 1962.

[22] Q. Guo, T. Johansson, and P. Stankovski, *A Key Recovery Attack on MDPC with CCA Security Using Decoding Errors*, pp. 789–815. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016.

[23] E. Eaton, M. Lequesne, A. Parent, and N. Sendrier, "QC-MDPC: A Timing Attack and a CCA2 KEM," in *Post-Quantum Cryptography* (T. Lange and R. Steinwandt, eds.), vol. 1, (Cham), pp. 47–76, Springer International Publishing, 2018.

[24] A. Nilsson, T. Johansson, and P. Stankovski Wagner, "Error Amplification in Code-based Cryptography," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, no. 1, pp. 238–258, 2019.

[25] P. Santini, M. Battaglioni, F. Chiaraluce, and M. Baldi, "Analysis of Reaction and Timing Attacks Against Cryptosystems Based on Sparse Parity-Check Codes," in *Code-Based Cryptography* (M. Baldi, E. Persichetti, and P. Santini, eds.), (Cham), Springer International Publishing, 2019.

[26] N. Drucker, S. Gueron, D. Kostic, and E. Persichetti, "On the Applicability of the Fujisaki-Okamoto Transformation to the BIKE KEM." Cryptology ePrint Archive, Report 2020/510, 2020.

[27] T. Itoh and S. Tsujii, "A fast algorithm for computing multiplicative inverses in GF(2m) using normal bases," *Information and Computation*, vol. 78, no. 3, pp. 171–177, 1988.

[28] L. Grassi, C. Rechberger, D. Rotaru, P. Scholl, and N. P. Smart, "Mpc-friendly symmetric key primitives," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, (New York, NY, USA), pp. 430–443, ACM, 2016.

[29] I. Damgård, "On the randomness of legendre and jacobi sequences," in *Proceedings of the 8th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '88, (London, UK, UK), pp. 163–172, Springer-Verlag, 1990.

[30] N. Kaluđerović, T. Kleinjung, and D. Kostic, "Improved key recovery on the Legendre PRF." Cryptology ePrint Archive, Report 2020/098, 2020. https://eprint.iacr.org/2020/098.

[31] J. Chaulet and N. Sendrier, "Worst case QC-MDPC decoder for mceliece cryptosystem," in *IEEE International Symposium on Information Theory, ISIT 2016, Barcelona, Spain, July 10-15, 2016*, pp. 1366–1370, IEEE, 2016.

[32] J. Chaulet, *Etude de cryptosystèmes à clé publique basés sur les codes MDPC quasi-cycliques. (Study of public key cryptosystems based on quasi-cyclic MDPC codes).* PhD thesis, Pierre and Marie Curie University, Paris, France, 2017.

[33] J. Tillich, "The decoding failure probability of mdpc codes," in *2018 IEEE International Symposium on Information Theory (ISIT)*, pp. 941–945, 2018.

[34] R. J. McEliece, "A Public-Key Cryptosystem Based On Algebraic Coding Theory," *Deep Space Network Progress Report*, vol. 44, pp. 114–116, Jan 1978.

[35] H. Niederreiter, "Knapsack-type cryptosystems and algebraic coding theory," *Prob. Contr. Inform. Theory*, vol. 15, no. 2, pp. 157–166, 1986.

[36] N. Sendrier, "Encoding information into constant weight words," in *IEEE Conference, ISIT 2005*, (Adelaide, Australia), pp. 435–438, Sept. 2005.

[37] E. Fujisaki and T. Okamoto, "Secure Integration of Asymmetric and Symmetric Encryption Schemes," in *Advances in Cryptology – CRYPTO '99* (M. Wiener, ed.), (Berlin, Heidelberg), pp. 537–554, Springer Berlin Heidelberg, 1999.

[38] D. Hofheinz, K. Hövelmanns, and E. Kiltz, "A Modular Analysis of the Fujisaki-Okamoto Transformation," in *Theory of Cryptography* (Y. Kalai and L. Reyzin, eds.), (Cham), pp. 341–371, Springer International Publishing, 2017.

[39] V. Shoup, "Using hash functions as a hedge against chosen ciphertext attack," in *International Conference on the Theory and Applications of Cryptographic Techniques*, pp. 275–288, Springer, 2000.

[40] J.-C. Deneuville, P. Gaborit, and G. Zémor, "Ouroboros: A simple, secure and efficient key exchange protocol based on coding theory," in *Post-Quantum Cryptography* (T. Lange and T. Takagi, eds.), (Cham), pp. 18–34, Springer International Publishing, 2017.

[41] R. P. Brent and P. Zimmermann, "An o(m(n) logn) algorithm for the jacobi symbol," in *Algorithmic Number Theory* (G. Hanrot, F. Morain, and E. Thomé, eds.), (Berlin, Heidelberg), pp. 83–95, Springer Berlin Heidelberg, 2010.

[42] H. Davenport, "On the distribution of quadratic residues (mod p)," *Journal of the London Mathematical Society*, vol. s1-8, no. 1, pp. 46–52, 1933.

[43] A. Weil, "On some exponential sums," *Proceedings of the National Academy of Sciences*, vol. 34, no. 5, pp. 204–207, 1948.

[44] A. C. Yao, "Theory and applications of trapdoor functions," in *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*, pp. 80–91, IEEE Computer Society, 1982.

[45] N. Drucker, S. Gueron, and D. Kostic, "Additional implementation of BIKE." https://bikesuite.org/additional.html, 2019.

[46] N. Drucker and S. Gueron, "A toolbox for software optimization of QC-MDPC code-based cryptosystems," *Journal of Cryptographic Engineering*, pp. 1–17, jan 2019.

[47] D. Hofheinz, K. Hövelmanns, and E. Kiltz, "A modular analysis of the fujisaki-okamoto transformation." Cryptology ePrint Archive, Report 2017/604, 2017.

[48] S. Samardjiska, P. Santini, E. Persichetti, and G. Banegas, "A Reaction Attack Against Cryptosystems Based on LRPC Codes," in *Progress in Cryptology – LATINCRYPT 2019* (P. Schwabe and N. Thériault, eds.), (Cham), pp. 197–216, Springer International Publishing, 2019.

[49] G. Wafo-Tapa, S. Bettaieb, L. Bidoux, and P. Gaborit, "A Practicable Timing Attack Against HQC and its Countermeasure," Tech. Rep. Report 2019/909, aug 2019.

[50] C. Aguilar Melchor, N. Aragon, S. Bettaieb, B. Lo ic, O. Blazy, J.-C. Deneuville, P. Gaborit, E. Persichetti, and G. Zémor, "Hamming Quasi-Cyclic (HQC)," 2017.

[51] J. Chaulet and N. Sendrier, “Worst case QC-MDPC decoder for McEliece cryptosystem,” in *2016 IEEE International Symposium on Information Theory (ISIT)*, pp. 1366–1370, July 2016.

[52] V. Shoup, “Number theory c++ library (ntl) version 11.3.4.” http://www.shoup.net/ntl, September 2019.

[53] P. Z. Pierrick Gaudry, Richard Brent and E. Thome, “gf2x-1.2.” https://gforge.inria.fr/projects/gf2x/, July 2017.

[54] I. V. Maurich, T. Oder, and T. Güneysu, “Implementing QC-MDPC McEliece encryption,” *ACM Trans. Embed. Comput. Syst.*, vol. 14, pp. 44:1–44:27, Apr. 2015.

[55] Open Quantum Safe Project, “liboqs.” https://github.com/open-quantum-safe/liboqs, 2020. Last accessed 16 Feb 2020.

[56] T. Chou, “QcBits: Constant-Time Small-Key Code-Based Cryptography,” in *Cryptographic Hardware and Embedded Systems – CHES 2016* (B. Gierlichs and A. Y. Poschmann, eds.), (Berlin, Heidelberg), pp. 280–300, Springer Berlin Heidelberg, 2016.

[57] –, “Intel®64 and IA-32 architectures software developer’s manual,” *combined volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4*, November 2019.

[58] N. Drucker and S. Gueron, “Fast multiplication of binary polynomials with the forthcoming vectorized VPCLMULQDQ instruction,” in *2018 IEEE 25th Symposium on Computer Arithmetic (ARITH)*, June 2018.

[59] N. Drucker, S. Gueron, and V. Krasnov, “Making AES Great Again: The Forthcoming Vectorized AES Instruction,” in *16th International Conference on Information Technology-New Generations (ITNG 2019)* (S. Latifi, ed.), pp. 37–41, Springer International Publishing, 2019.

[60] N. Sendrier and V. Vasseur, “About low DFR for QC-MDPC decoding,” in *Post-Quantum Cryptography - 11th International Conference, PQCrypto 2020, Paris, France, April 15-17, 2020, Proceedings* (J. Ding and J. Tillich, eds.), vol. 12100 of *Lecture Notes in Computer Science*, pp. 20–34, Springer, 2020.

[61] N. Drucker, S. Gueron, and V. Krasnov, “Fast multiplication of binary polynomials with the forthcoming vectorized VPCLMULQDQ instruction,” in *2018 IEEE 25th Symposium on Computer Arithmetic (ARITH)*, pp. 115–119, jun 2018.

[62] N. Drucker and S. Gueron, “Fast CTR DRBG for x86 platforms,” March 2019. https://github.com/aws-samples/ctr-drbg-with-vector-aes-ni.

[63] Amazon Web Services, “s2n.” https://github.com/awslabs/s2n, 2020. Last accessed 16 Feb 2020.

[64] A. Hülsing, J. Rijneveld, J. Schanck, and P. Schwabe, "High-Speed Key Encapsulation from NTRU," in *Cryptographic Hardware and Embedded Systems – CHES 2017* (Fischer, Wieland and Homma, Naofumi, ed.), (Cham), pp. 232–252, Springer International Publishing, 2017.

[65] M. Baldi, A. Barenghi, F. Chiaraluce, G. Pelosi, and P. Santini, "LEDAcrypt," 2019.

[66] D. J. Bernstein and B.-Y. Yang, "Fast constant-time gcd computation and modular inversion," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2019, pp. 340–398, May 2019.

[67] A. Guimarães, D. F. Aranha, and E. Borin, "Optimized implementation of QC-MDPC code-based cryptography," *Concurrency and Computation: Practice and Experience*, vol. 31, no. 18, p. e5089, 2019.

[68] A. Guimar, E. Borin, D. F. Aranha, A. Guimarães, E. Borin, and D. F. Aranha, "Introducing Arithmetic Failures to Accelerate QC-MDPC Code-Based Cryptography," *Code-Based Cryptography*, vol. 2, pp. 44–68, 2019.

[69] Chien-Hsing Wu, Chien-Ming Wu, Ming-Der Shieh, and Yin-Tsung Hwang, "High-speed, low-complexity systolic designs of novel iterative division algorithms in $gf(2^m)$," *IEEE Transactions on Computers*, vol. 53, pp. 375–380, March 2004.

[70] J. W. Bos, T. Kleinjung, R. Niederhagen, and P. Schwabe, "ECC2K-130 on Cell CPUs," in *Progress in Cryptology – AFRICACRYPT 2010* (D. J. Bernstein and T. Lange, eds.), (Berlin, Heidelberg), pp. 225–242, Springer Berlin Heidelberg, 2010.

[71] Gueron, Shay. https://github.com/open-quantum-safe/openssl/issues/42#issuecomment-433452096, October 2018.

Dusan **Kostic**

☎ (+41) 78-6733-606 | ✉ dkostic@protonmail.com

Education

PhD Thesis, École Polytechnique Fédérale de Lausanne

ADVISOR ARJEN LENSTRA, LABORATORY FOR CRYPTOLOGIC ALGORITHMS *2016 - PRESENT*

- *Analysis of the BIKE post-quantum cryptographic protocol and the Legendre pseudorandom function*

Master's degree, Computer Engineering and Informatics

SCHOOL OF ELECTRICAL ENGINEERING, UNIVERSITY OF BELGRADE *2013 - 2015*

- Thesis title: *OpenCL implementation of Parallel Pollard Rho algorithm for ECDLP targeting GPU architectures*
- Implementation of parallel Pollard Rho algorithm for elliptic curve discrete logarithm problem in C++ and OpenCL.
- One and two orders of magnitude speedup achieved on integrated and dedicated GPUs respectively, in comparison to a sequential implementation on CPU

Bachelor's degree, Electrical Engineering and Computing

SCHOOL OF ELECTRICAL ENGINEERING, UNIVERSITY OF BELGRADE *2009 - 2013*

- Thesis title: *Analysis and implementation of asymmetric cryptography algorithms*

Experience

Applied Scientist - intern

AMAZON WEB SERVICES, SEATTLE, USA *May 2019 - Sep. 2019*

- Work on two post-quantum cryptographic schemes Bit Flipping Key Encapsulation (BIKE) and Supersingular Isogeny Based Key Exchange (SIKE) submitted to the NIST PQCrypto process
- Define and analyze constant-time algorithms for the NIST PQCrypto Round 2 submission of BIKE, specifically constant-time QC-MDPC decoding
- Improving the performance of the Round 2 submission of SIKE by using the new Intel instruction set

Visiting researcher

UNIVERSITY OF HAIFA, ISRAEL *Sep. 2018 - Dec. 2018*

- Improving the performance of the key exchange mechanism (SIKE) by designing the finite field arithmetic functions to use the new Intel processor VPMADD instructions

Teaching assistant

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE *Jan. 2016 - PRESENT*

- CS-101: Advanced Information, Computation, Communication I (fall 2016, 2017, 2018, 2019)
- COM-402: Information Security and Privacy (spring 2017)
- Supervisor of student semester projects

Research assistant

SCHOOL OF ELECTRICAL ENGINEERING, BELGRADE *Jan. 2015 - Sep. 2015*

- Parallelization of existing implementations of routing algorithms in software router packages
- Implementation of routing protocols, security standards and quality of service methods

Software development engineer - intern

INTEL CORPORATION, BELGRADE *Jan. 2014 - Oct. 2014*

- Implementation of digital image processing algorithms on Intel Atom processor in C, using Intel Parallel Primitives library and parallelization with Intel's Cilk+
- Parallelization of digital image processing algorithms in C++ and OpenCL targeting Intel integrated GPUs architecture
- Exposing application through Android software stack, from Linux Device Driver to Camera Application, and enabling usage on smartphone devices

Publications

BIKE CT DEC	N. Drucker, S. Gueron, and D. Kostic, “On constant-time QC-MDPC decoding with negligible failure rate”. Accepted at the CBCrypto 2020 International Workshop on Code-Based Cryptography and will be published in the proceedings. The preprint of the paper is available at `eprint.iacr.org/2019/1289`.
BIKE DEC	N. Drucker, S. Gueron, and D. Kostic, “QC-MDPC Decoders with Several Shades of Gray” in *Post-Quantum Cryptography* (J. Ding and J.-P. Tillich, eds.), (Cham), pp. 35-50, Springer International Publishing 2020.
POLY INV	N. Drucker, S. Gueron, and D. Kostic, “Fast polynomial inversion for post quantum QC-MDPC cryptography” in *Cyber Security Cryptography and Machine Learning* (S. Dolev, V. Kolesnikov, S. Lodha, and G. Weiss, eds.), (Cham), pp. 110-127, Springer International Publishing 2020.
BIKE CCA	N. Drucker, S. Gueron, and D. Kostic, E. Persichetti “On the Applicability of the Fujisaki-Okamoto Transformation to the BIKE KEM”, available at `eprint.iacr.org/2020/510`
Legedre PRF	N. Kaluđerović, T. Kleinjung, and D. Kostić, “Improved key recovery on the Legendre PRF”, available at `eprint.iacr.org/2020/098`
SIKE VPMADD	D. Kostic and S. Gueron, ”Using the New VPMADD Instructions for the New Post Quantum Key Encapsulation Mechanism SIKE,” 2019 IEEE 26th Symposium on Computer Arithmetic (ARITH), Kyoto, Japan, 2019, pp. 215-218, doi: 10.1109/ARITH.2019.00050
Lattice sieve	Speeding up lattice sieve with Xeon Phi coprocessor, available at `eprint.iacr.org/2017/592`
Network graphs	M. Vesović, A. Smiljanić, D. Kostić, “Performance of shortest path algorithms based on parallel vertex traversal” in *Serbian Journal of Electrical Engineering*, Volume 13, pp. 31-43 (link)

Awards

2019	Teaching Assistant Reward for teaching excellence, EPFL
2017	Teaching Assistant Reward for teaching excellence, EPFL
2015	Doctoral EDIC Fellowship, EPFL

Skills

Programming	Advanced knowledge of C, C++, Python, Sage, OpenCL, OpenMPI, CUDA, Assembly
Languages	Serbian - first language, English - professional proficiency, French - basic knowledge