BIKE implementation : vulnerabilities and countermeasures

Loïc Demange

THÈSE DE DOCTORAT DE SORBONNE UNIVERSITÉ

Spécialité

Informatique

École doctorale Informatique, Télécommunications et Électronique (Paris)

Présentée par

Loïc Demange

Pour obtenir le grade de

DOCTEUR de SORBONNE UNIVERSITÉ

Mise en œuvre de BIKE, vulnérabilités et contre-mesure

soutenue publiquement le 26 janvier 2024

devant le jury composé de :

Nicolas Sendrier	Inria de Paris	Directeur
Zoé Amblard	Thales	Encadrante
Daniel Augot	Inria Saclay	Rapporteur
Pierre Loidreau	DGA	Rapporteur
Mélissa Rossi	ANSSI	Examinatrice
Jean-Claude Bajard	Sorbonne Université	Examinateur

Remerciements

Sortant d'un stage sur les schémas post-quantique à base de réseaux euclidiens, ce n'était pas quelque chose d'évident d'embrayer sur une thèse traitant d'un schéma à base de codes. Et pourtant, cela a pu se faire grâce à l'équipe crypto de Thales. Je remercie donc Thomas et Sylvain d'avoir cru en mes capacités et de m'avoir permis de poursuivre en doctorat, ainsi que Nicolas de m'avoir proposé un sujet aussi intéressant et de l'avoir encadré pendant ces 3 ans (et 3 mois). Tout cela était encore moins évident que les attaques par canaux auxiliaires ne sont clairement pas une spécialité de l'équipe, ni du côté INRIA ni du côté Thales.
C'est dans cette optique que je remercie aussi très fortement Zoé, Simon et Valentin de m'avoir autant aidé et encadré durant cette thèse. Merci aussi à Mélissa avec qui la quasi-intégralité des travaux ont été réalisés, à Agathe pour nos différents échanges ainsi qu'au CESTI de Toulouse de m'avoir ouvert brièvement leurs portes pour m'expliquer les rudimentaires des attaques par canaux auxiliaires.

Je remercie également Daniel Augot et Pierre Loidreau d'avoir accepté d'être rapporteurs, ainsi que Jean-Claude Bajard d'avoir fait partie du jury.

Je remercie grandement l'intégralité de l'équipe crypto de Thales pour ces moments bien sympas ainsi que l'équipe COSMIQ en globalité avec qui, malheureusement, je n'ai pas eu tant le temps de sociabiliser ni de partager énormément de choses, mais qui ont toujours fait preuve d'une très grande sympathie.

Enfin, dans un cadre plus privé, je remercie énormément ma copine pour sa présence permanente, ainsi que mes chattes et ma chienne d'avoir toujours eu les miaous et les ouafs pour rire.

Introduction

BIKE est un schéma d'encapsulation de clés (KEM) post-quantique sélectionné pour le quatrième tour de la campagne de standardisation du NIST. Sa sécurité repose sur la robustesse du problème de décodage du syndrome pour les codes quasi-cycliques et fournit des performances compétitives par rapport aux autres candidats du 4e tour, ce qui le rend pertinent pour une utilisation dans des cas concrets. La communauté scientifique a fortement encouragé l'analyse de sa résistance aux attaques par canaux auxiliaires et plusieurs travaux ont déjà souligné diverses faiblesses. Pour les corriger, ces derniers ont proposé des contre-mesures ad hoc. Toutefois, contrairement à la ligne de recherche bien documentée sur le masquage des algorithmes basés sur les réseaux, la possibilité de protéger génériquement les algorithmes basés sur des codes par du masquage n'a été étudiée que de manière marginale dans un article de 2016 de Cong Chen et al. À ce stade de la campagne de standardisation, il est important d'évaluer la possibilité de masquer entièrement le schéma BIKE et le coût qui en résulte en termes de performances.

L'objectif de cette thèse est donc de proposer un algorithme de BIKE dont la sécurité a été prouvée, en réalisant l'ensemble du processus de manière masquée, et ce sans jamais manipuler directement les données sensibles. Pour ce faire, nous utilisons des "gadgets", qui sont des sortes de fonctions masquées, identifiées par des niveaux de non-interférence : NI (non-interference) et SNI (strong non-interference). En termes simples, SNI permet aux gadgets d'être composables : ils peuvent être appelés l'un après l'autre, avec les mêmes variables. NI, en revanche, exige plus de précautions en termes de variables manipulées. Les gadgets font l'objet de preuves, basées sur le modèle ISW, permettant de donner un véritable argument de sécurité et de robustesse à l'algorithmique. Si l'algorithme est prouvé sûr de bout-en-bout, cela appuie positivement sa résistance contre des attaques par canaux auxiliaires.
Il convient de noter que le masquage a été initialement développé pour les schémas symétriques et qu'il était basé sur le masquage booléen. Ce n'est que récemment que l'on a commencé à s'intéresser aux schémas asymétriques, et en particulier aux schémas basés sur les réseaux. Dans ce but, le masquage arithmétique a été le principal utilisé, bien que des conversions booléennes aient pu être effectuées pour réaliser certaines choses (comparaison de valeurs entre autres).

Aujourd'hui, nous sommes en mesure de proposer une implémentation masquée de

BIKE, basée sur un algorithme prouvé sûr. Comme BIKE manipule des données binaires, nous nous sommes concentrés sur le masquage booléen. Nous avons dû :

- réutiliser les gadgets existants,
- adapter et optimiser les gadgets de masquage arithmétique existants,
- créer de nouveaux gadgets.

Pour chaque gadget, nous avons dû prouver qu'il répondait bien aux exigences du modèle ISW et à ce que ses compositions au sein des différentes fonctions de BIKE ne pose pas de souci, de façon à arriver au masquage du schéma intégral.
Pour rappel, BIKE repose sur des QC-MDPC, et son arithmétique est basée sur des polynômes denses et creux, il a donc fallu faire des choix concernant la représentation et la manière dont les calculs sont effectués. Nous avons donc décidé d'explorer deux voies (entièrement dense et hybride creux-dense) et de voir laquelle était la plus pertinente. En plus de l'implémentation complète en C, des benchmarks ont été réalisés, ce qui nous a permis de voir où les performances étaient limitées et où se trouvaient les goulots d'étranglement.

Pour récapituler, nous proposons ici un algorithme BIKE masqué intégralement et prouvé sûr, avec son implémentation en C et divers benchmarks permettant de juger de ses performances.

Contents

List of Figures

List of Tables

Chapter 1 Introduction

Cryptography is used to securely exchange information. In simple terms, this means protecting the information you communicate with a third party, so that only they can understand it. This is achieved by modifying the communication with an element that only the participants can know, making the exchange incomprehensible to anyone else. To achieve this rigorously, we use mathematical notions.

1.1 Cryptography

Historically, information technology (IT) was based on three pillars that together formed the CIA triad:

- Confidentiality, to ensure that information is intelligible only to those who have been authorized to access it.
- Integrity, to ensure that information is not altered and is what it should be.
- Authenticity, to ensure that those who have access to the information are in fact the intended recipients.

The model has since been updated, and is now called the CIAAN, to further include:

- Availability, to ensure that information is accessible whenever authorized users want it to be.
- Non-repudiation, to ensure that users cannot deny having sent or received information.

The purpose of cryptography is to be able to communicate information while respecting the CIA triad, availability and non-repudiation being criteria that are very difficult to meet today.

1.1.1 General principle

Encryption ensures that only the sender and legitimate recipient(s) of a message know its contents. This is achieved by "encrypting" the information with a secret, which

prevents anyone who does not have the secret from accessing the message.

The security of cryptographic schemes is based on so-called hard problems. A hard problem is one that cannot, to the best of our knowledge, be solved in polynomial time. In complexity theory, we refer to the **NP** class when we talk about difficult problems. In the case of cryptography, hard problems do not necessarily belong to **NP** and are not necessarily related to **NP**-hard or **NP**-complete problems, although the problems that can be reduced to a **NP**-hard problem provide an additional safety argument.
Indeed, if a cryptographic scheme is to be strong, there must be no algorithm that can find the solution to a random instance of the cryptographic algorithm. So it is not enough to have a few robust instances. This is the difference between **average** and **worst-case** complexity.

When the key is the same for encryption and decryption, we call this private-key encryption (known as symmetric). Otherwise, we have two different keys, one public and one private, and call this public-key encryption (known as asymmetric).

1.1.1.1 Private-key cryptography

Symmetric encryption involves encrypting and decrypting a message using the same key. This raises security questions about how to share the key between all the desired entities. On the other hand, these encryption schemes are highly efficient, and can be very resource-efficient.

The first public standard scheme was DES [77], developed in 1970 by IBM, which was based on a type of structure known as a Feistel cipher. It was later superseded by AES [And97], based on substitution–permutation network (SPN), which remains a cornerstone of symmetric cryptography to this day.

In the context of limited hardware such as embedded systems, NIST launched a lightweight symmetrical cryptography competition, which led to the standardization of the ASCON scheme [Dob+21] in 2023.

1.1.1.2 Public-key cryptography

As mentioned above, public-key cryptography involves encrypting information with a public key, and decrypting it only with the secret key. Public-key cryptography relies on trapdoor functions, which are difficult to “inverse” in the average case.

Definition 1.1. A function f is one-way if f can be computed in polynomial time, but there is no algorithm for computing a preimage in polynomial time.

Definition 1.2. A trapdoor function is a one-way function, which offers the possibility of calculating the preimages of this function in polynomial time if certain additional information is known.

Asymmetric cryptography is based on trapdoor one-way functions, which allow a clear message to be transformed into an encrypted message using public information, for which it is computationally difficult to recover the original message without the trapdoor.

Although asymmetric cryptography can be based on a variety of mathematical structures (which we will describe in more detail later), the most widely used scheme is RSA [RSA78].

RSA (for Rivest, Shamir and Adleman) is an asymmetric encryption scheme based on the difficulty of decomposing large numbers into prime factors. However, the RSA problem could be solved without factoring. Nowadays, it is still used everywhere, especially in banking transactions.

TLS As mentioned previously, asymmetric cryptography is much slower and less resource-efficient than symmetric cryptography. This is why we can combine the two: use an asymmetric scheme to send the key of a symmetric scheme, and thus protect end-to-end communication while being faster. This is the case with the TLS protocol, used in the World Wide Web to secure communications between a client and a web server.

Diffie-Hellman [DH76] was the first key exchange scheme, developed in 1976, and its counterpart ECDH based on elliptic curves is still used today within TLS. The reason why we use ECDH instead of the original Diffie-Hellman scheme is because elliptic curves allow smaller keys to be handled for the same level of security. ECDH is based on the discrete logarithm problem.

Nowadays, factoring is possible in sub-exponential time and RSA factoring challenges are regularly beaten [Bou+20]. This is not yet the case for variants based on elliptic curves, but with the arrival of the quantum computer, other schemes will be favored in the future.

1.1.2 Quantum computer & post-quantum cryptography

As mentioned above, the security of conventional asymmetric cryptography relies on mathematical problems that are reputedly difficult to solve: finding a discrete logarithm or decomposing large numbers into prime factors. However, such cryptography is threatened by a technological breakthrough: the quantum computer. Indeed, the elementary operations of quantum and classical computers are of different natures, making the former capable of solving these problems efficiently thanks to Shor's quantum algorithm [Sho94]. This threat has prompted the scientific community to look for alternatives that can withstand it.

A cryptographic primitive is said to be post-quantum if it can run on a classical computer and if there is no known attack against it even if a sufficiently large quantum computer becomes available. They are classical primitives that are destined to replace the asymmetric algorithms currently in use, particularly for signature and key exchange functions. In this context, the standardization campaign launched by the NIST should define the next post-quantum standards for these two functions by 2024 (drafts have already been published), and initiate hybridation, which consists of combining post-quantum asymmetric schemes with well-known and studied asymmetric schemes based on factorization or discrete logarithm, by 2025.

Post-quantum cryptography has been around for almost as long as asymmetric cryptography, and comprises numerous families based on a wide variety of problems and algorithms, such as:

- lattices, based on the Shortest Vector Problem (SVP, **NP**-hard, worst case to average case reductions) and Closest Vector Problem (CVP, same hardness)
- multivariate, based on the difficulty to solving systems of multivariate polynomial equations (**NP**-complete)
- hash functions, based on the security of hash functions
- isogenies, based on the difficulty of computing isogenies and endomorphism rings
- error-correcting codes, based on the difficulty of decoding random linear codes (**NP**-hard, conjectured hard in average case)

1.2 Side-channel attacks

Side-channel attacks are attacks that compromise a system by exploiting weaknesses not in its theoretical structure and mathematical robustness, but in its implementation and infrastructure. More precisely the adversary is able to obtain information about the secret thanks to the measure of a physical quantity i.e. power, EM radiation, heat, etc. We differentiate between two types of attack: invasive attacks, where we interact directly with the device, and non-invasive attacks, where we make external observations of the device.

Among the non-invasive attacks, two stand out and are widely used:

- the timing attack, where the adversary will use the execution time of the cryptosystem,
- the power consumption attack where the adversary will use the power consumption of the cryptosystem.

Thankfully we can mitigate these attacks by implementing adequate countermeasures. For the first point, the main countermeasure is to realize constant-time implementation, which consists in performing operations that have a cost independent of the secret. For the second point, it is more difficult. In a simplistic way, a device will consume power at the time of a change of state (bit which passes from 0 to 1 or vice versa). The objective is to manipulate data that is not directly correlated with the secret, and therefore to have consumption that is also decorrelated with the secret.

If one does not want to exploit the secret itself, but just want to check that the information given is identical (password, PIN, etc.), one can keep only its hashed form. Alternatively, sensitive values can be masked when manipulated.

In this thesis, we will focus on preventing power consumption attacks with the help of masking techniques.

1.3 Masking

Masking consists in obfuscating sensitive values to prevent them from being recovered through side-channel attacks. To achieve this, we use randomness to alter the value and perform operations on the masked value. The mask is only removed once the operations have been carried out.

There are two ways of doing this:

- use masking only at sensitive points, as a local countermeasure, either as a preventive measure or after an attack is known,
- mask the entire scheme, and have proof of security to say that, theoretically, the scheme is secure (under various conditions, starting with a well-executed implementation).

In the first case, only certain parts are secured, and this does not require a huge sacrifice in terms of performance or implementation complexity. On the other hand, we play a cat-and-mouse game with attackers, by securing only the parts newly discovered as sensitive in the scientific literature. We take the risk to overlook some missing attacks that may be kept secret by governments or criminal attackers and we let our system exposed to future attacks that have yet to be discovered.

In the second case, the choice is made to mask the entirety, and thus add operations and complexity to the original algorithm. However, if it is well done, it provides a solid base and avoids a large proportion of side-channel attacks. To prove that masking is safe, models have already been investigated and defined by decades of scientific litterature. The most commonly used is the ISW model [ISW03], which we will describe in more detail later in Definition 2.11, and which links theoretical proof with robustness in

practice.

Higher order masking was initially implemented on symmetric cryptographic schemes, where Boolean masking is used. It is only very recently that it has begun to be more common on asymmetric schemes, in particular lattice-based like qTesla signature scheme [GR20] (detailed in Section 2.2.1), with adjustments and arithmetic masking. The subject has grown, and has led to the design of schemes adapted for masking, such as the Raccoon signature scheme [Pin+23].

Prior to our work, no code-based scheme had been masked from end-to-end. In fact, some papers have already masked QC-MDPC codes [Che+16], but their approach was different. Masking was only performed on the decoder, which was a simple bitflipping, and had a very practical approach since the goal was to make an FPGA implementation. There is therefore no proof of masking, and the rest of the scheme was not masked.

1.4 Contributions

As a result of this work, we have masked the BIKE scheme in its entirety, with its C implementation.

This consists of:

- creating and modifying various gadgets,
- proving their security,
- assembling them, and proving their composability,
- coding these elements in C,
- benchmarking different parts of the code, according to security level.

This work was published in the first edition of the CiC journal [DR24].

Chapter 2
State of art & preliminary

2.1 Coding theory

2.1.1 General introduction

In this section, we develop the notions of coding theory needed to understand the rest of the work.

Definition 2.1 (Binary linear codes)**.** A binary linear code $\mathcal{C}$ of length n and dimension k is a k-dimensional vector subspace of $\mathbb{F}_2^n$. Is it possible to represent it in two equivalent ways:

- either by using a generator matrix $\mathbf{G} \in \mathbb{F}_2^{k \times n}$ such that each row of $\mathbf{G}$ is an element of a basis of $\mathcal{C}$,
$$\mathcal{C} = \{m \cdot \mathbf{G}, m \in \mathbb{F}_2^k\}.$$
- or with a parity-check matrix $\mathbf{H} \in \mathbb{F}_2^{(n-k) \times n}$ such that for any $\mathbf{c} \in \mathcal{C}$,
$$\mathbf{c} \cdot \mathbf{H}^T = 0.$$

Definition 2.2 (Hamming distance)**.** The Hamming distance between two codewords in a linear code is the number of different coordinates between them.

Definition 2.3 (Weight)**.** The weight of a codeword in a linear code $\mathcal{C}$ is its number of non-zero coordinates.

Definition 2.4 (Minimum distance)**.** The minimum distance of a linear code $\mathcal{C}$ is the minimum weight of the non-zero codewords of $\mathcal{C}$.

Definition 2.5 (Syndrome)**.** Let $\mathcal{C}$ be a linear code, $\mathbf{H}$ its parity-check matrix and $\mathbf{y}$ the received codeword. We call $s = \mathbf{y}\mathbf{H}^T$ the syndrome of $\mathbf{y}$.

In fact, the syndrome is used to locate the errors in the codeword.

Definition 2.6 (Circulant matrix)**.** An $r \times r$ matrix $\mathbf{A}$ is circulant if each row is a cyclic shift of the previous row. More precisely, $\mathbf{A}$ is of the form

$$\begin{pmatrix} a_0 & a_1 & \cdots & a_{r-1} \\ a_{r-1} & a_0 & \cdots & a_{r-2} \\ \vdots & & & \vdots \\ a_1 & a_2 & \dots & a_0 \end{pmatrix}.$$

We say that $\mathbf{A}$ is generated by the vector $(a_0, \cdots, a_{r-1})$.

Remark 2.1. It is possible to define an isomorphism between the ring of polynomials $\mathbb{F}_2[X]/(X^r - 1)$ and the set of circulant matrices of order r. To a vector $(a_0, \cdots, a_{r-1})$ generating a circulant matrix, one can associate the polynomial $\sum_{i=0}^{r-1} a_i X^i$. Multiplication and inversion can then be performed either with matrix multiplication or polynomial multiplication.

Definition 2.7 (Quasi-circulant matrix)**.** A matrix is quasi-circulant if it is composed of circulant blocks.

For example, let

$$\mathbf{A} = \begin{pmatrix} a_1 & a_2 & a_3 \\ a_3 & a_1 & a_2 \\ a_2 & a_3 & a_1 \end{pmatrix} \text{ and } \mathbf{B} = \begin{pmatrix} b_1 & b_2 & b_3 \\ b_3 & b_1 & b_2 \\ b_2 & b_3 & b_1 \end{pmatrix}$$

be two circulant matrices. The matrix $\mathbf{C} = [\mathbf{A}|\mathbf{B}]$ defined as the concatenation of $\mathbf{A}$ and $\mathbf{B}$ is a quasi-circulant matrix.

Remark 2.2. Similarly to Remark 2.1, it is possible to represent quasi-circulant matrices as sets of polynomials.

Definition 2.8 (Quasi-cyclic code)**.** A binary code $\mathcal{C}$ is quasi-cyclic if and only if it admits a quasi-circulant generating matrix. We refer to the order as the size of the circulating blocks, and the index as the number of blocks in a line.

Remark 2.3. Trivially speaking, any code is quasi-cyclic of order 1.

Definition 2.9 (QC-MDPC code)**.** Let n, r, w be integer parameters for length, dimension and minimum code weight (minimum distance). A $[n, r, w]$ QC-MDPC code $\mathcal{C}$ is a quasi-cyclic code that admits a parity-check matrix $\mathbf{H}$ such that $\mathbf{H}$ has a row weight $w = \mathcal{O}(\sqrt{n})$.

Remark 2.4. An low density parity code (LDPC) admits a parity-check matrix $\mathbf{H}$ such that $\mathbf{H}$ has a row weight $w = \mathcal{O}(1)$. [MRS00] showed that LDPCs were not the most suitable for cryptography due to the fact that the small-weight words of the dual are too easy to find. MDPC codes have been designed to balance the hardness of two problems of finding codewords of small weight in the dual, and of decoding t errors, making them very poor correcting codes for communication, but secure enough to be used within a cryptographic scheme.

2.1.2 The BIKE scheme

BIKE (*Bit-Flipping Key Encapsulation*) [Ara+22] is a key encapsulation scheme based on QC-MDPC (Quasi-Cyclic Moderate Density Parity-Check) codes as introduced in Definition 2.9.

BIKE relies on sparse QC-MDPC codes, of order r and index 2. Its private key corresponds to the parity-check matrix. The security of the scheme reduces to quasi-cyclic variants of hard problems from coding theory, decoding and codeword finding in an arbitrary quasi-cyclic code [Ale03; BMT78]. We refer to [Ara+22] for more information about the security and design rationale.

BIKE's first building block is a public key encryption scheme (PKE) based on a variant of the Niederreiter framework [Nie86]. The plaintext is represented by the sparse vector (e_0, e_1), and the ciphertext by its syndrome. The decryption is performed with a decoding procedure that will be presented below in section 2.1.3. Next, this PKE is converted into an IND-CCA KEM with the application of the Fujisaki-Okamoto transformation [FO99; HHK17]. For the scheme to be truly IND-CCA, there must be conditions on the decoding failure rate (also called DFR), which is the case here with the chosen decoder.

Let us detail the key generation (**KeyGen**), Encapsulation (**Encaps**) and Decapsulation (**Decaps**) algorithms in more details. In addition to the parameters r and w, let us define t and ℓ as integer parameters. We denote $\mathcal{R} = \mathbb{F}_2[X]/(X^r - 1)$ the underlying cyclic polynomial ring. Let us define

$$\begin{aligned} \mathcal{H}_w &= \{(h_0, h_1) \in \mathcal{R}^2 \mid |h_0| = |h_1| = w/2\}, \\ \mathcal{E}_t &= \{(e_0, e_1) \in \mathcal{R}^2 \mid |e_0| + |e_1| = t\}, \\ \mathcal{M} &= \{0,1\}^\ell, \\ \mathcal{K} &= \{0,1\}^\ell, \end{aligned}$$

as respectively the private key space, the error space, the message space and the shared key space. In the above, we denote by $|h|$ the Hamming weight of the polynomial h, i.e. the number of non-zero coefficients of h.
The Fujisaki-Okamoto transformation requires several hash functions: $\mathbf{H} : \mathcal{M} \to \mathcal{E}_t$, $\mathbf{L} : \mathcal{E}_t \to \mathcal{M}$ and $\mathbf{K} : \mathcal{M} \times (\mathcal{R} \times \mathcal{M}) \to \mathcal{K}$.

In the following, we write $a \xleftarrow{\$} \mathcal{B}$ when a is sampled uniformly at random from $\mathcal{B}$, and $\leftarrow$ is an assigment of value.

Algorithm 2.1 Keygen

Output: $((h_0, h_1), \sigma) \in \mathcal{H}_w \times \mathcal{M}$, $h \in \mathcal{R}$

1: $(h_0, h_1) \xleftarrow{\$} \mathcal{H}_w$
2: $h \leftarrow h_1 h_0^{-1}$
3: $\sigma \xleftarrow{\$} \mathcal{M}$
4: **return** $((h_0, h_1, \sigma), h)$

Algorithm 2.2 Encaps

Input: $h \in \mathcal{R}$
Output: $K \in \mathcal{K}, c \in \mathcal{R} \times \mathcal{M}$

1: $m \xleftarrow{\$} \mathcal{M}$
2: $(e_0, e_1) \leftarrow \mathbf{H}(m)$
3: $c \leftarrow (e_0 + e_1 h, m \oplus \mathbf{L}(e_0, e_1))$
4: $K \leftarrow \mathbf{K}(m, c)$
5: **return** (K, c)

Algorithm 2.3 Decaps

Input: $(h_0, h_1, \sigma) \in \mathcal{H}_w \times \mathcal{M}$, $c = (c_0, c_1) \in \mathcal{R} \times \mathcal{M}$
Output: $K \in \mathcal{K}$

1: $e' = (e'_0, e'_1) \leftarrow \mathsf{decoder}(c_0 h_0, h_0, h_1)$
2: $m' \leftarrow c_1 \oplus \mathbf{L}(e')$
3: **if** $e' = \mathbf{H}(m')$ **then**
4: $\quad K \leftarrow \mathbf{K}(m', c)$
5: **else**
6: $\quad K \leftarrow \mathbf{K}(\sigma, c)$
7: **return** K

Parameter setting As defined in the specifications, the parameters should satisfy several constraints. The block length r should be a prime number to avoid folding attacks [CT19]. The parameter w should be such that $w = 2d \approx \sqrt{n}$ and the error weight should be such that $t \approx \sqrt{n}$ to balance the costs of attacks on the key and the message. d should be odd and 2 should be primitive modulo r for all blocks of $\mathbf{h}_0$ to be invertible. We present the instantiated parameters in Table 2.1.

Table 2.1: BIKE's proposed parameters [Ara+22]

	Level 1	Level 3	Level 5
r	12323	24659	40973
w	142	206	274
t	134	199	264
ℓ	256	256	256

Security proof BIKE's main security property is its reduction to difficult problems such as decoding and finding small-weight words in QC code. This ensures that BIKE is IND-CPA. If we wish to use the scheme with a static key, it must be IND-CCA, since we can construct error patterns to cause decoding failures, which provide us information

about the secret key [GJS16] (see Section 2.1.3.2 for more details). This is where the DFR (Decoding Failure Rate) value comes into play. Quantifying DFR is a challenging problem, so we have to rely on models supported by simulations. We know that keys with a certain configuration will produce more failures, and avoiding them may be a viable solution.

2.1.3 Decoding QC-MDPC codes

The choice of the decoder has a crucial impact on the security and the performances of the scheme. As QC-MDPC codes have sparse parity-check matrices, decoding techniques usually rely on Bit-Flipping algorithms originally introduced in [Gal62] for low density parity-check matrices. Bit-Flipping is based on the hard decision decoding version of "Belief Propagation Decoding", which is a soft decision decoding and has multiple implementations, the best-known being the sum-product algorithm and the min-sum algorithm [FMI99]. By definition, these algorithms are soft-decision decoding, and therefore are complex to implement. They can be implemented on embedded devices, but require dedicated circuits, and for MDPCs, the circuit would be much more important than for LDPCs. Bit-Flipping, requiring much simpler logic and being faster, is more suitable for cryptographic scheme, while having a limited impact on decoder performance.

Technically, the Bit-Flipping algorithm works as follows: over several iterations, we compute the syndrome $c\mathbf{H}^T$ where c is the noisy codeword and $\mathbf{H}^T$ is the transposed parity matrix of the code. Next, we count the number of unsatisfied parity-check equations for each position. If the counter for a position exceeds T, a threshold (computed on the fly according to the weight of the syndrome), the position is flipped and the syndrome is recomputed. Let $\mathsf{syndrome}()$ be the syndrome computation, $\mathsf{counter}()$ the counter computation, and $\mathsf{threshold}()$ the threshold computation function. We refer to [Ara+22] for details.

Algorithm 2.4 Bit-Flipping algorithm

Input: $\mathbf{H}^T$ the sparse parity matrix of a $[2r, r, w]$ MDPC code
$c \in \mathbb{F}_2^n$ a noisy codeword
Output: A codeword c, $c\mathbf{H}^T = 0$ by def.

$s \leftarrow \mathsf{syndrome}(c, \mathbf{H})$
while $|s| \neq 0$ **do**
  $T \leftarrow \mathsf{threshold}(|s|)$
  for $j \in \{1, ..., n\}$ **do**
    if $\mathsf{counter}(s, j, \mathbf{H}) \geqslant T$ **then**
      $c_j \leftarrow c_j \oplus 1$
  $s \leftarrow \mathsf{syndrome}(c, \mathbf{H})$
return c

The authors of BIKE chose a refined Black-Gray-Flip (BGF) technique introduced

in [DGK20b]. This is a bitflipping algorithm that introduces two classification zones, with two different thresholds: the black zone and the gray zone. Two additional iterations are performed to verify the choices made during the classification. The BGF decoding algorithm is presented in algorithm 2.5. This decoder also has a fixed number of iterations (set at 5), to avoid timing attacks.

Algorithm 2.5 `Black-Gray-Flip` (BGF)

Parameters: r, w, t, $d = w/2$, $n = 2r$; Nbr_Iter, τ, `threshold`

$e \leftarrow 0^n$
for $i = 1, \ldots,$Nbr_Iter **do**
    $T \leftarrow \texttt{threshold}(|s + e\mathbf{H}^T|, i)$
    $e, black, grey \leftarrow \texttt{BFIter}(s + e\mathbf{H}^T, e, T, \mathbf{H})$
    if $i = 1$ **then**
        $e \leftarrow \texttt{BFMaskedIter}(s + e\mathbf{H}^T, e, black, (d+1)/2 + 1, \mathbf{H})$
        $e \leftarrow \texttt{BFMaskedIter}(s + e\mathbf{H}^T, e, grey, (d+1)/2 + 1, \mathbf{H})$
if $s = e\mathbf{H}^T$ **then**
    return e
else
    return $\perp$

`procedure BFIter`$(s, e, T, \mathbf{H})$
for $j = 0, \ldots, n-1$ **do**
    if `ctr`$(\mathbf{H}, s, j) \geqslant T$ **then**
        $e_j \leftarrow e_j \oplus 1$
        $black_j \leftarrow 1$
    else if `ctr`$(\mathbf{H}, s, j) \geqslant T - \tau$ **then**
        $grey_j \leftarrow 1$
return $e, black, grey$

`procedure BFMaskedIter`$(s, e, mask, T, \mathbf{H})$
for $j = 0, \ldots, n-1$ **do**
    if `ctr`$(\mathbf{H}, s, j) \geqslant T$ **then**
        $e_j \leftarrow e_j \oplus mask_j$
return e

There has been a few variants of bit-flipping proposed, such as the backflip in 2019 [SV], which consists in giving a time-to-live to the flip to catch up on mistakes, or the "Weight Bit-Flipping" in 2021 [Nil+21], but BGF has remained the classical BIKE decoding algorithm.

2.1.3.1 Existing implementations

The first implementation based on QC-MDPC codes was completed in 2014 [MG14]. It was intended for Cortex-M4, and had an analysis of possible side-channel attacks, with proposals for countermeasures, in particular masking some operations of the decoder. [MOG15] provides optimizations to improve performance. In 2016, an implementation using Niederreiter for the Cortex-M4 which is efficient and, above all, IND-CCA [MHG16], is proposed. At the same time, QcBits came into competition with the latter and was more efficient, offering, among other things, constant-time bitslicing for counter calculations [Cho16].
Some implementation proposals have been made over the years [DG19; GAB19; BOG20], and another constant-time C implementation was introduced in 2020 [DGK20b], especially improving the decoding part, and which is claimed protected against timing and cache attacks. Cortex-M4 optimized implementations of BIKE have been introduced later in [CCK21]. Subsequently, optimizations have been proposed in [Che+22].

2.1.3.2 Existing side-channel attacks & counter-measure

The BIKE scheme has already been subject to vulnerabilities. In fact, a first reaction attack was performed in 2016, where it was found that by causing decoding failures, information about the error and the secret could be recovered [GJS16]. To do this, it uses the notion of spectrum, which is the set of differences between the 1's of a sparse vector. The idea is that when the error spectrum is correlated with the secret spectrum, decoding is more likely to fail. This attack led to a concrete timing attack in 2018 [Eat+18], and on BIKE's word sampler in 2021 [Guo+22], where the authors have highlighted the possibility of using timing information of the constant weight word sampler in the decapsulation in order to apply the reaction attack. Such a vulnerability has been thwarted by redesigning the word sampler in [Sen21].

On the power-consumption attacks side, several works have outlined various side-channel weaknesses and proposed ad-hoc countermeasures.
Indeed, while BIKE's sparse and structured private keys are essential for providing good performances and compactness, this exact structure and redundancy can be exploited by side-channel attacks in order to decrease the difficulty of the underlying decoding problem. For instance, Chou's implementation has been targeted by a differential power analysis attack on the syndrome computation in [Ros+17]. Later, an improvement of the previous attack and a single-trace analysis exploiting leakage in the syndrome computation were provided in [Sim+19].

Very recently, [Che+23] introduced a new single-trace attack on the most recent implementation of BIKE. The authors use unsupervised clustering techniques on the trace during the cyclic shifts computation to recover some bits of the positions of the ones in the private key. They combine such knowledge with classical information set decoding techniques to recover the full key.

2.2 Masking

2.2.1 General introduction

The technique known as masking is the most deployed countermeasure against physical attacks and is widely applied in embedded systems. Masking consists in randomizing any secret-dependent intermediate variable. Each of these secret-dependent intermediate variables, say $\mathbf{x}$, is split into $d+1$ variables $(\mathbf{x}_i)_{0\leqslant i\leqslant d}$ called "shares". The integer d is referred to as the masking order. In the context of BIKE, the only necessary type of masking is *Boolean masking*.

Definition 2.10 (Shared value (boolean masking))**.** A sensitive value (or variable) $\mathbf{x}$ is shared in $(\mathbf{x}_i)_{0\leqslant i\leqslant d}$ such that

$$\mathbf{x} = \mathbf{x}_0 \oplus \cdots \oplus \mathbf{x}_d. \tag{2.1}$$

In the following, we will use “masked variable" to define a shared variable, and a shared variable $(\mathbf{x}_i)_{0\leqslant i\leqslant d}$ will be denoted by $[\![\mathbf{x}]\!]$ for readability.

While $\mathbb{F}_2$-linear operations can straightforwardly be applied share-wise, non-linear operations are more complex and require additional randomness, as shown in [ISW03]. Proving the security of a masked design consists in showing that the joint distribution of any set of at most d intermediate variables is independent of the secrets. But, the bigger the algorithm is, the more dependencies to be considered in the proof. Fortunately, several works have defined intermediate security properties that simplify the security proofs [RP10; Cor+14; Bar+16]: one can focus on proving the properties on small parts of the algorithms, denoted *gadgets*, and it is possible to securely compose the pieces together.

Much effort has been performed on provably masking lattice-based primitives in the past five years and many challenges have been overcome. For example, [Bar+18] introduced a new security notion to justify unmasking certain intermediate steps. In [GR20], the authors proposed a masked implementation of the qTesla signature scheme [Bin+19]. In [Kun+22], a masked Fujisaki-Okamoto transform is introduced for a fully masked Saber KEM implementation [DAn+20]. The NIST post-quantum finalists Crystals-Dilithium [Lyu+22] and Crystals-Kyber [Sch+22] have also been masked in [Azo+23] and [Bos+21].

The picture is less abundant when it comes to code-based schemes. One explanation could come from the large sparse polynomials leading to potential prohibitive performances or the complex counter-based decoder. The authors of [Kra+22] propose a first-order masked inversion in multiplicative masking. Another recent work [Kra+23] presents a way to mask BIKE's key generation with a fixed weight polynomial sampling technique and arithmetic to Boolean conversions.

Definition 2.11 (d-probing Security or ISW security [ISW03])**.** An algorithm is d-probing secure if and only if the joint distribution of any set of at most d internal intermediate values is independent of the secrets.

Even if d-probing security seems far from realistic side-channel protection, it is actually backed-up by theoretical model reductions that relate the d-probing security to side-channel security up to a certain level of noise [DDF14]. Moreover, [Cha+99] showed that the number of measurements required to mount a successful side-channel attack usually increases exponentially in the masking order.

In addition to Definition 2.11, other intermediate security properties were introduced to ease the security proofs [RP10; Cor+14; Bar+16]. The focus can be placed on proving these properties on small parts of the algorithms, denoted gadgets.

Definition 2.12 (Gadget)**.** A gadget is a probabilistic algorithm that takes shared and unshared inputs values and returns shared and un-shared values.

Remark 2.5. In practice, a gadget is nothing more than a function, but in the context of masking.

In fact, we use the notion of NI and SNI to describe gadgets.

Definition 2.13 (Non interference [Bar+16])**.** A gadget is d-non-interfering (d-NI) if and only if any set of at most d observations can be perfectly simulated from at most d shares of each input.

In other words, all combinations of at most d intermediate variables must give at most d shares of each input variable.

As an example, let us define together a 2-NI gadget for XOR in Boolean masking. We could build an equivalent gadget for arithmetic masking, using the $+$. Note that this is a trivial example, since it is a linear operation.

Algorithm 2.6 XOR

Input: $[\![a \in \mathbb{F}_2]\!], [\![b \in \mathbb{F}_2]\!]$
Output: $[\![c \in \mathbb{F}_2 = a \oplus b]\!]$
1: $[\![c]\!]_0 = [\![a]\!]_0 \oplus [\![b]\!]_0$
2: $[\![c]\!]_1 = [\![a]\!]_1 \oplus [\![b]\!]_1$
3: $[\![c]\!]_2 = [\![a]\!]_2 \oplus [\![b]\!]_2$
4: **return** $[\![c]\!] = ([\![c]\!]_0, [\![c]\!]_1, [\![c]\!]_2)$

To check that this gadget is 2-NI, we will perform all combinations of at most two intermediate variables, i.e. :

- $[\![c]\!]_0$, $[\![c]\!]_1$, $[\![c]\!]_2$
- $([\![c]\!]_0, [\![c]\!]_1)$, $([\![c]\!]_0, [\![c]\!]_2)$, $([\![c]\!]_1, [\![c]\!]_2)$

These intermediate variables are correlated with input shares:

- $(\llbracket a \rrbracket_0, \llbracket b \rrbracket_0)$, $(\llbracket a \rrbracket_1, \llbracket b \rrbracket_1)$, $(\llbracket a \rrbracket_2, \llbracket b \rrbracket_2)$
- $((\llbracket a \rrbracket_0, \llbracket b \rrbracket_0), (\llbracket a \rrbracket_1, \llbracket b \rrbracket_1))$, $((\llbracket a \rrbracket_0, \llbracket b \rrbracket_0), (\llbracket a \rrbracket_2, \llbracket b \rrbracket_2))$, $((\llbracket a \rrbracket_1, \llbracket b \rrbracket_1), (\llbracket a \rrbracket_2, \llbracket b \rrbracket_2))$

It can be seen that whatever the combinations, there are at most two shares of each input variable. And since the third part, used to reconstitute the input variables, is random, the information obtained is independent of the inputs, and therefore of the secrets. So the gadget is 2-NI.

However, although a proven NI gadget is independently protected, this does not, in most cases, allow us to compose it. Consider this gadget.

Algorithm 2.7 NI composition issue (dummy)

Input: $\llbracket a \in \mathbb{F}_2 \rrbracket$
Output: $\llbracket c \in \mathbb{F}_2 = a \oplus a \rrbracket$
1: $\llbracket b \rrbracket \leftarrow$SHUFFLE($\llbracket a \rrbracket$)
2: $\llbracket c \rrbracket \leftarrow$XOR($\llbracket a \rrbracket, \llbracket b \rrbracket$)
3: **return** $\llbracket c \rrbracket$

Let SHUFFLE be the 2-NI gadget that shuffles shares such that SHUFFLE($\llbracket a \rrbracket = \{\llbracket a \rrbracket_0, \llbracket a \rrbracket_1, \llbracket a \rrbracket_2\}$) = $\llbracket b \rrbracket = \{\llbracket a \rrbracket_2, \llbracket a \rrbracket_1, \llbracket a \rrbracket_0\}$. Individually, the two gadgets are 2-NI. However, this composition is not. In fact, b depends on shares of a. Let us go back to the XOR.
With one or two probes, we can observe:

- $(\llbracket a \rrbracket_0, \llbracket b \rrbracket_0)$, $(\llbracket a \rrbracket_1, \llbracket b \rrbracket_1)$, $\llbracket a \rrbracket_2, \llbracket b \rrbracket_2)$
- $((\llbracket a \rrbracket_0, \llbracket b \rrbracket_0), (\llbracket a \rrbracket_1, \llbracket b \rrbracket_1))$, $((\llbracket a \rrbracket_0, \llbracket b \rrbracket_0), (\llbracket a \rrbracket_2, \llbracket b \rrbracket_2))$, $((\llbracket a \rrbracket_1, \llbracket b \rrbracket_1), (\llbracket a \rrbracket_2, \llbracket b \rrbracket_2))$

which is equivalent to

- $(\llbracket a \rrbracket_0, \llbracket a \rrbracket_2)$, $(\llbracket a \rrbracket_1, \llbracket a \rrbracket_1)$, $\llbracket a \rrbracket_2, \llbracket a \rrbracket_0)$
- $((\llbracket a \rrbracket_0, \llbracket a \rrbracket_2), (\llbracket a \rrbracket_1, \llbracket a \rrbracket_1))$, $((\llbracket a \rrbracket_0, \llbracket a \rrbracket_2), (\llbracket a \rrbracket_2, \llbracket a \rrbracket_0))$, $((\llbracket a \rrbracket_1, \llbracket a \rrbracket_1), (\llbracket a \rrbracket_2, \llbracket a \rrbracket_0))$

So we can see that certain combinations allow us to observe all the parts of a, and thus compromise the secret. We could generalize this example in this way.

Algorithm 2.8 NI composition issue

Input: $\llbracket a \in \mathbb{F}_2 \rrbracket$
Output: $\llbracket c \in \mathbb{F}_2 \rrbracket$
1: $\llbracket b \rrbracket \leftarrow f(\llbracket a \rrbracket)$
2: $\llbracket c \rrbracket \leftarrow g(\llbracket a \rrbracket, \llbracket b \rrbracket)$
3: **return** $\llbracket c \rrbracket$

Here, depending on what f is going to do, we can observe parts of a via b, and this can lead to a being compromised when g is executed. The notion of SNI has been introduced to address the problem of composability. In this example, if f or g is SNI, it works.

To achieve compositional security, SNI adds a stronger condition, affecting the gadget's output variables.

Definition 2.14 (Strong non interference [Bar+16])**.** A gadget is d-strong non-interfering (d-SNI) if and only if any set of at most d observations whose d_{int} observations on the internal data and d_{out} observations on the outputs can be perfectly simulated from at most d_{int} shares of each input.

Remark 2.6. Here, $d = d_{out} + d_{int}$.

Proving that a gadget is SNI is very complicated. if you want to prove than your gadget is 2-SNI, you have to check that:

- with 1 or 2 probes for internal data, we can not observe more than 2 shares of the input values
- with 1 probe for internal data and 1 probe for output data, we can not observe more than 1 share of the input values
- with 1 or 2 probes for output data, we can not observe any share of the input values

It is with this difficulty in mind that a proposition has been made in [Bar+16] that allows generic construction of NI and SNI gadgets.

Proposition 2.1 ([Bar+16], Prop. 4)**.** *An algorithm is d-NI provided all its gadgets are d-NI, and all variables are used at most once as argument of a gadget call other than* refresh*. Moreover the algorithm is d-SNI if it is d-NI and one of the following holds:*

- *its return expression is $[\![b]\!]$ and its last instruction is of the form $[\![b]\!] \leftarrow \mathsf{refresh}([\![b]\!])$*
- *its sequence of parameters is $[\![\mathbf{a}]\!] = \{[\![a_0]\!], ..., [\![a_n]\!]\}$, its i-th instruction is $b \leftarrow \mathsf{refresh}([\![a_i]\!])$ for $1 \leqslant i \leqslant n$, and $[\![a_i]\!]$ is not used anywhere else in the algorithm*

Remark 2.7. We will take a closer look at the refresh gadget in the next section (Algorithm 2.10)

So, if we go back to Algorithm 2.8, we note that we only need to refresh $[\![a]\!]$ to obtain an NI gadget according to this proposition, since $[\![a]\!]$ is used twice as an argument.

Algorithm 2.9 NI composition

Input: $[\![a \in \mathbb{F}_2]\!]$
Output: $[\![c \in \mathbb{F}_2]\!]$
1: $[\![b]\!] \leftarrow f([\![a]\!])$
2: $[\![d]\!] \leftarrow \mathsf{refresh}([\![a]\!])$
3: $[\![c]\!] \leftarrow g([\![d]\!], [\![b]\!])$
4: **return** $[\![c]\!]$

It is this proposition that has helped us build BIKE's d-NI gadgets.

2.2.2 Existing masked gadgets

In this section, we list all the gadgets known from the literature and which have been used in our contribution.

2.2.2.1 Refreshing shared value

As we saw earlier, each shared value is made up of a number of shares whose secret value is the XOR of all of them. To prevent an attacker from reconstructing the secret, care must be taken to ensure that the shares are modified each time they are used. To do this, we "refresh" them by drawing some randomness.

The following refresh algorithm was introduced in [Cor14], is d-SNI and is as follows.

Algorithm 2.10 Refresh (refresh)

Input: $[\![x \in \mathbb{F}_2]\!]$
Output: $[\![x \in \mathbb{F}_2]\!]$ with refreshed shares
1: **for** $i \leftarrow 0$ to d **do**
2: **for** $j \leftarrow 1$ to d **do**
3: $r \xleftarrow{\$} \mathbb{F}_2$ ▷ Draw one bit
4: $[\![x]\!]_0 \leftarrow [\![x]\!]_0 \oplus r$
5: $[\![x]\!]_j \leftarrow [\![x]\!]_j \oplus r$
6: **return** $[\![x]\!]$

Remark 2.8. Here, to be SNI, the gadget performs a quadratic number of operations, breaking any correlation between outputs and inputs.

In the signature scheme Raccoon [Pin+23], a quasi-linear-time refresh algorithm has been proposed. The main idea is to recursively refresh the shares, so as to have a complexity in $\mathcal{O}(n \log(n))$.

Although very interesting, we have not explored this possibility, for two reasons: the first being that if we do not have a number of shares to the power of 2 it is not trivial to

show that it is safe, and secondly because the refresh gadget could easily be interchanged if we wanted to.

2.2.2.2 Binary AND between two shared values

The $\mathsf{sec}_{\&}$[Cor+15; Bar+18] is a gadget that performs the binary & between two shared values. During the operation, randomness is drawn to prevent an attacker from obtaining information about the masked values. Since all shares are constantly subject to randomness, the $\mathsf{sec}_{\&}$is d-SNI.

Algorithm 2.11 SecAnd ($\mathsf{sec}_{\&}$)

Input: $[\![x \in \mathbb{F}_2]\!], [\![y \in \mathbb{F}_2]\!]$
Output: $[\![z = x\&y \in \mathbb{F}_2]\!]$

1: $[\![z]\!] \leftarrow [\![x]\!]\&[\![y]\!]$
2: **for** $i \leftarrow 0$ to d **do**
3: **for** $j \leftarrow i + 1$ to d **do**
4: $r \xleftarrow{\$} \mathbb{F}_2$ ▷ Draw one bit
5: $[\![z]\!]_i \leftarrow [\![z]\!]_i \oplus r$
6: $[\![z]\!]_j \leftarrow [\![z]\!]_j \oplus r \oplus ([\![x]\!]_i \& [\![y]\!]_j) \oplus ([\![x]\!]_j \& [\![y]\!]_i)$
7: **return** $[\![z]\!]$

The algorithm is introduced with parameters on $\mathbb{F}_2$, but it can easily be applied to binaries of varying sizes.

2.2.2.3 Addition between two shared values

The sec_{+}[Cor14] is a gadget d-NI that performs the + between two shared values. To do this, it performs the addition using binary &, XOR and shifts to propagate any carries. In 2022, [BC22] proposed an algorithm based on bitslicing, which has not been explored further here, due to lack of time and opportunity.

Algorithm 2.12 SecAdd (sec_+)

Input: $[\![x \in \mathbb{Z}_n]\!], [\![y \in \mathbb{Z}_n]\!]$
Output: $[\![z = x + y \in \mathbb{Z}_n]\!]$
1: $[\![p]\!] \leftarrow [\![x]\!] \oplus [\![y]\!]$
2: $[\![g]\!] \leftarrow \mathsf{sec}_\&([\![x]\!], [\![y]\!])$ ▷ $\mathsf{sec}_\&$ between binary representations
3: **for** $i \leftarrow 0$ to $n-1$ **do**
4: $\quad [\![a]\!] \leftarrow [\![g]\!] \ll 2^i$
5: $\quad [\![b]\!] \leftarrow [\![p]\!] \ll 2^i$
6: $\quad [\![a]\!] \leftarrow \mathsf{sec}_\&([\![a]\!], [\![p]\!])$
7: $\quad [\![g]\!] \leftarrow [\![g]\!] \oplus [\![a]\!]$
8: $\quad [\![b]\!] \leftarrow \mathsf{sec}_\&([\![b]\!], [\![p]\!])$
9: $[\![a]\!] \leftarrow [\![g]\!] \ll 2^{n-1}$
10: $[\![t]\!] \leftarrow \mathsf{sec}_\&([\![a]\!], [\![p]\!])$
11: $[\![g]\!] \leftarrow [\![g]\!] \oplus [\![t]\!]$
12: $[\![z]\!] \leftarrow [\![x]\!] \oplus [\![y]\!] \oplus ([\![g]\!] \ll 1)$
13: **return** $[\![z]\!]$

We introduce $\mathsf{sec}_{+\text{partlymasked}}$ which is almost identical to sec_+ but where the first operation ($\mathsf{sec}_\&$ between the two masked parameters) has been modified to take an unmasked element (& between all parts of the masked value and the public one).

2.2.2.4 Masked conditional branch

This gadget chooses between two masked values based on the value of a masked boolean. It was formalized in [Kra+23].

We define $\mathsf{sec}_\&^{\mathbf{bitwise}}$, which consists in performing a $\mathsf{sec}_\&$ between all the bits of a value and the single bit of another value. Since it's just a succession of $\mathsf{sec}_\&$, which is itself a d-SNI gadget, $\mathsf{sec}_\&^{\mathbf{bitwise}}$ is itself d-SNI.
We also denote by $\neg[\![t]\!]_0$ the fact of bit-reverse the binary value of the first share of t.

Algorithm 2.13 Choose value ($\mathsf{sec}_{\mathsf{if}}$)

Input: $[\![a \in \mathbb{Z}_n]\!]$, $[\![b \in \mathbb{Z}_n]\!]$, $[\![t \in \mathbb{F}_2]\!]$
Output: $[\![a]\!]$ if $[\![t]\!] = 1$, $[\![b]\!]$ otherwise
1: $[\![c]\!] \leftarrow \mathsf{sec}_\&^{\mathbf{bitwise}}([\![a]\!], [\![t]\!])$
2: $[\![t]\!]_0 \leftarrow \neg[\![t]\!]_0$
3: $[\![d]\!] \leftarrow \mathsf{sec}_\&^{\mathbf{bitwise}}([\![b]\!], [\![t]\!])$
4: **return** $[\![c]\!] \oplus [\![d]\!]$ ▷ Coefficient-wise XOR

Since $\mathsf{sec}_\&^{\mathbf{bitwise}}$ is d-SNI and the last $\oplus$ is d-NI, we deduce the theorem below.

Theorem 2.1. *The choose value algorithm 2.13 is d-NI.*

In the same paper, an alternative way of achieving this is presented. In fact, $a\&t\oplus b\&(t\oplus 1)$ is equivalent to $((a\oplus b)\&t)\oplus b$, and we save a $\mathsf{sec}_{\&}$.
However, we have to refresh once more, and experimentally, this does not make any particular difference to performance.

Algorithm 2.14 Choose value (alt.)

Input: $[\![a \in \mathbb{Z}_n]\!]$, $[\![b \in \mathbb{Z}_n]\!]$, $[\![t \in \mathbb{F}_2]\!]$
Output: $[\![a]\!]$ if $[\![t]\!] = 1$, $[\![b]\!]$ otherwise

1: $[\![c]\!] \leftarrow [\![a]\!] \oplus [\![b]\!]$
2: $[\![c]\!] \leftarrow \mathsf{sec}_{\&}^{\mathbf{bitwise}}([\![c]\!], [\![t]\!])$
3: $[\![b]\!] \leftarrow \mathsf{refresh}([\![b]\!])$
4: **return** $[\![c]\!] \oplus [\![b]\!]$ ▷ Coefficient-wise XOR

Chapter 3
Contributions: End-to-end masked implementation of BIKE

As we saw earlier, BIKE is a post-quantum signature scheme, which was present in round 4 of the NIST standardization competition. In sections 2.1.3.1 and 2.1.3.2, we saw that various implementations had been proposed, more or less protected to certain side-channel attacks, and that these attacks had been mostly protected locally.

The aim here is to propose a BIKE algorithm that has been proven to be secure, by carrying out the entire process in a masked way, without ever manipulating the sensitive data directly. To achieve this, we use "gadgets", which are masked functions of sorts, identified by non-interference levels: NI, and SNI. In simple terms, SNI allows gadgets to be composable: they can be called one after the other, with the same variables. NI, on the other hand, requires greater precaution in terms of the variables manipulated. Gadgets are the subject of proofs, based on the ISW model, and making it possible to give a real argument of safety and solidity of the algorithm. If the algorithm is proven safe from start to finish, we can say it will have a certain level of resistance to side-channel attacks.
It should be noted that masking was initially developed for symmetric schemes, and was based on Boolean masking. Research into asymmetric schemes, and lattice-based schemes in particular, has only recently come to the fore. For this purpose, arithmetic masking was the main one used, although Boolean conversions could be performed to achieve certain things (value comparison among others).

In this work we provide the first provable high-order masked implementation of a code-based algorithm. We detail every masked gadget that is necessary for masking BIKE's key generation, encapsulation and decapsulation. The proofs are given in the d-probing model. Let us detail some aspects of our design.

- **No mask conversion** Mask-conversion gadgets consist in modifying the underlying masking operation, e.g. going from $\oplus$ to an addition in $\mathbb{Z}_q$. Even if the unmasked functionality is the identity function, these gadgets are known to be heavy in terms of computation time. Despite efficiency improvements since their introduction e.g. in [CGV14; Cor17; Cor+15], current secure mask conversion algorithms run in time at least $\mathcal{O}(d^2)$. Contrary to lattices, BIKE is fundamentally relying on binary operations. While the authors of [Kra+23] have included mask conversion in their design, we believe that keeping only Boolean masking would be more natural and

efficient. In this work, we give the first evidence that it is possible to completely mask BIKE without any mask conversion.

- **Sparse versus dense representation**. BIKE's intermediate variables are sparse polynomials with coefficients in $\mathbb{F}_2$. An important question arose rapidly when designing a masked BIKE: *Should we represent the masked polynomials in dense form or keep the sparse structure and mask the indices of the non-zero coefficient instead?* For the dense form, the number of non-zero coefficients is protected but the multiplication requires a masked Karatsuba-based multiplication algorithm. For the sparse form, the number of non-zero coefficients is accessible by timing attacks but a lighter multiplication algorithm based on cyclic shifts is possible. The sparse representation intuitively seems lighter but some parts necessarily required the dense form for security. For completeness, we decided to analyze both following approaches:

  1. A fully-dense implementation where the polynomials are masked in dense form.
  2. A hybrid sparse-dense implementation where the polynomials are represented in sparse form whenever the number of non-zero coefficient is independent from any secret data.

  Interestingly, our experiments showed that a fully-dense approach seems more relevant, especially for high orders. While (2) and (1) seem equivalent for one or two shares, (1) looks indeed more relevant for higher orders. This difference might shrink with more optimizations of the cyclic shift, as it will be discussed in a later section.

- **Many new gadgets**. A lot of new gadgets needed to be introduced for masking BIKE. Although BIKE's bitslice addition technique turned out to operate well with Boolean masking, some other parts of the key generation were more challenging to mask. For example the Fisher-Yates sampling algorithm/technique and the polynomial inversion required many loops and subroutines. More generally, we provide all elementary gadgets that are necessary to mask BIKE, since we need to reuse existing gadgets, adapt and optimize existing arithmetic masking gadgets and create new gadgets. We believe that they can be of independent interest for masking future code-based schemes.

We provide an open C-code implementation of the key-generation, encapsulation and decapsulation algorithms with detailed benchmarks. Although theoretically quadratic [ISW03], several post quantum masked designs can lead to an experimental scale in the masking order that tends to be exponential [Bar+18, Table 1]. The scaling we've obtained is very encouraging, as our experiments seem to indicate a quadratic scaling. We believe that it is even possible to further improve and optimize our code and maybe reach quasi-linearity in the masking order. We hope that this work, published in [DR24], can be a first building block towards masked code-based cryptography and could lead to future analysis and new optimization.

3.1 Notation

In what follows, we take a closer look at the various gadgets designed to create the masked version of BIKE. These gadgets include functions, which we define here.

BIKE's private key $\mathbf{H}$ is a sparse polynomial (see remark 2.2). For masking such polynomials, both approaches are valid: either we represent in its dense form or we keep the sparse structure and mask the indices of the non-zero coefficients instead. Since the number of non-zero coefficient is a public parameter, two approaches are potentially valid. The sparse representation intuitively seems lighter but some part (such as error generation) will require the dense form for security reasons. For completeness, we analyze both approaches: (1) an implementation where $\mathbf{H}$ is masked in dense form and (2) a hybrid-sparse-dense implementation where both dense and sparse forms of $\mathbf{H}$ are stored.

The private key will then be denoted by $\mathbf{h}_0^\circ$, $\mathbf{h}_1^\circ$ when it is represented in sparse form (i.e. the indices of the non-zero coefficients are masked) and it will be denoted by $\mathbf{h}_0$, $\mathbf{h}_1$ when it is represented in dense form. The same convention is applied for other intermediate variables.

Let sparse_to_dense() be an algorithm that converts the sparse representation into a dense representation by multiplying the sparse polynomial by a dense polynomial equal to 1. This procedure is straightforwardly d-NI.

Also, we denote by

- zero_masking() an initialization of a shared value, where the XOR of all shares is zero,
- vector_zero_masking() an initialization of a vector of shared values, where each value of the vector has the XOR of all shares is zero. In fact, we use zero_masking in each coefficient of the vector,
- matrix_zero_masking() an initialization of a d-sharing of a 2-dimensional zero matrix,
- left() (resp. right()) the fact of cutting a vector in two and keeping only its left-hand (resp. right-hand) part,
- subvector() the fact of obtaining a sub-vector of the given coordinates,

As a general rule, all coefficient-wise operations are specified in the pseudo-code.

3.2 Masked gadgets

3.2.1 Auxiliary gadgets

In this section, we enumerate all the auxiliary gadgets required for BIKE masking. We range from the most generic to the most specialized algorithms.

3.2.1.1 Masked equality

The gadget $\text{sec}_=$ is a d-NI gadget that outputs a masked Boolean value corresponding to the equality. The idea is to use Boolean algebra to check if the XOR between the two inputs is 0. For that, we perform a $\text{sec}_\&$ between the negation of each obtained bit. Such a procedure has been outlined in the literature e.g. in [DAn+22].

We decided to optimize it in such a way as to dichotomize the operations about the word binary, and thus achieve better performance.

Algorithm 3.1 Masked equality ($\text{sec}_=$)

Input: $[\![\mathbf{x} \in \mathbb{F}_2^n]\!], [\![\mathbf{y} \in \mathbb{F}_2^n]\!], n$ a power of 2
Output: $[\![z \in \mathbb{F}_2]\!]$ equals 0 if $\mathbf{x} = \mathbf{y}$ and 1 if not

1: $[\![\mathbf{z}]\!] \leftarrow [\![\mathbf{x}]\!] \oplus [\![\mathbf{y}]\!]$
2: $i \leftarrow \frac{n}{2}$
3: **while** $i \neq 0$ **do**
4: $\quad [\![\mathbf{a}]\!] \leftarrow \text{left}([\![\mathbf{z}]\!])$ ▷ Cut in length
5: $\quad [\![\mathbf{b}]\!] \leftarrow \text{right}([\![\mathbf{z}]\!])$ ▷ Cut in length
6: $\quad [\![\mathbf{a}]\!]_0 \leftarrow \neg [\![\mathbf{a}]\!]_0$ ▷ Coefficient-wise not
7: $\quad [\![\mathbf{b}]\!]_0 \leftarrow \neg [\![\mathbf{b}]\!]_0$ ▷ Coefficient-wise not
8: $\quad [\![\mathbf{z}]\!] \leftarrow \text{sec}_\&([\![\mathbf{a}]\!], [\![\mathbf{b}]\!])$ ▷ Coefficient-wise $\text{sec}_\&$
9: $\quad [\![\mathbf{z}]\!]_0 \leftarrow \neg [\![\mathbf{z}]\!]_0$ ▷ Coefficient-wise not
10: $\quad i \leftarrow \frac{i}{2}$
11: **return** $[\![\mathbf{z}_0]\!]$

Given that the only operation manipulating the data is $\text{sec}_\&$, and that it is a d-SNI function, we can deduce that the algorithm is d-NI.
In fact, as negation only manipulates the first share, it is not able to leak anything (given that values are updated at each loop turn).

Theorem 3.1. *The equality algorithm 3.1 is* d-NI.

3.2.1.2 Masked maximum computation

This gadget returns the largest of the two masked values.

Remark 3.1. To calculate the negative of a shared value, we bit-reverse the first share (which will bit-reverse the actual value) and use the sec_+ gadget to add 1, thus calculating the 2's complement.

Algorithm 3.2 Max ($\mathsf{sec}_{\mathsf{max}}$)

Input: $[\![a \in \mathbb{Z}_n]\!]$, $[\![b \in \mathbb{Z}_n]\!]$
Output: $[\![c = \max(a, b) \in \mathbb{Z}_n]\!]$
1: $[\![t]\!] \leftarrow \mathsf{sec}_+([\![a]\!], [\![-b]\!])$
2: **return** $\mathsf{sec}_{\mathsf{if}}(\mathsf{refresh}([\![b]\!]), \mathsf{refresh}([\![a]\!]), \mathsf{sign\_bit}([\![t]\!]))$

Since the variables a and b are used within the sec_+ gadget, which is d-NI, we need to refresh them (d-SNI gadget) before reusing them in the call to the $\mathsf{sec}_{\mathsf{if}}$ function. This yields the following theorem.

Theorem 3.2. *The max algorithm 3.2 is* d-NI.

3.2.1.3 Filling a matrix in masked form

Our gadget fills each column of a matrix with the binary representation of a masked value. This gadget is used within the Algorithm 3.23 to fill the matrix with the negative threshold.

Algorithm 3.3 Fill matrix ($\mathsf{sec}_{\mathsf{fill}}$)

Input: $[\![v \in \mathbb{Z}_n]\!]$
Output: $[\![\mathbf{X} \in \mathbb{F}_2^{(\lfloor \log_2(n) \rfloor + 1)} \times k]\!]$ a matrix filled with the binary representation of $[\![v]\!]$
1: **for** $i \leftarrow 0$ to $k - 1$ **do**
2: **for** $j \leftarrow 0$ to $\lfloor \log_2(n) \rfloor$ **do**
3: $[\![\mathbf{X}_{j,i}]\!] \leftarrow [\![v]\!][j]$
4: $[\![v]\!] \leftarrow \mathsf{refresh}([\![v]\!])$
5: **return** $[\![\mathbf{X}]\!]$

Since we just initialize $[\![\mathbf{X}]\!]$ with $[\![v]\!]$ binary, we just refresh $[\![v]\!]$ to avoid to get same mask in two different lines.

We then get the following theorem.

Theorem 3.3. *The fill algorithm 3.3 is* d-NI.

3.2.1.4 Adders and carries

Adder gadgets are needed for bitslicing, and therefore for most of BIKE's calculations.

Algorithm 3.4 Half Adder SecHalf_Adder

Input: $[\![x \in \mathbb{F}_2]\!]$, $[\![y \in \mathbb{F}_2]\!]$
Output: $[\![z = x \oplus y \in \mathbb{F}_2]\!]$, $[\![c \in \mathbb{F}_2]\!]$ the carry

1: $[\![z]\!] \leftarrow [\![x]\!] \oplus [\![y]\!]$
2: $[\![c]\!] \leftarrow \mathsf{sec}_{\&}([\![x]\!], [\![y]\!])$
3: **return** $[\![z]\!], [\![c]\!]$

Algorithm 3.5 Adder SecAdder

Input: $[\![x \in \mathbb{F}_2]\!]$, $[\![y \in \mathbb{F}_2]\!]$, $[\![c_0 \in \mathbb{F}_2]\!]$
Output: $[\![z = x \oplus y \oplus c_0 \in \mathbb{F}_2]\!]$, $[\![c \in \mathbb{F}_2]\!]$ the carry

1: $([\![t]\!], [\![s]\!]) \leftarrow \mathsf{SecHalf\_Adder}([\![x]\!], [\![y]\!])$
2: $([\![z]\!], [\![u]\!]) \leftarrow \mathsf{SecHalf\_Adder}([\![t]\!], [\![c_0]\!])$
3: $[\![c]\!] \leftarrow [\![s]\!] \oplus [\![u]\!]$
4: **return** $[\![z]\!], [\![c]\!]$

Since the $\oplus$ enjoys the d-NI property and $\mathsf{sec}_{\&}$ takes the same variables as input but is d-SNI, their combination leads to an d-NI algorithm. Thus, we introduce the following stating the probing security of the half adder algorithm.

Theorem 3.4. *The half adder algorithm 3.4 is d-NI.*

Since the adder uses only two calls to SecHalf_Adder (itself d-NI), handling different variables, we can infer the following.

Theorem 3.5. *The adder algorithm 3.5 is d-NI.*

3.2.1.5 Multiplications

Within BIKE, we need to perform multiplication between integers and between polynomials. First, we look at a gadget for multiplying between a masked and unmasked value, then at naive multiplication between two masked polynomials, allowing the use of Karatsuba algorithm.

Algorithm 3.6 Partly masked multiplication ($\mathsf{SecMult}_{\mathrm{partlymasked}}$)

Input: $[\![x \in \mathbb{Z}_n]\!]$, $y \in \mathbb{Z}$
Output: $[\![z = x \cdot y \in \mathbb{Z}_n]\!]$

1: $[\![z]\!] \leftarrow \mathsf{zero\_masking}()$
2: $[\![t]\!] \leftarrow [\![x]\!]$
3: **for** $i \leftarrow 0$ to $\lfloor \log_2(y) \rfloor$ **do**
4: **if** $y[i] = 1$ **then** ▷ $y[i] = (y \gg i)\ \&\ 1$
5: $[\![z]\!] \leftarrow \mathsf{sec}_{+}([\![z]\!], [\![t]\!])$
6: $[\![t]\!] \leftarrow \mathsf{refresh}([\![t]\!])$
7: $[\![t]\!] \leftarrow [\![t]\!] \ll 1$
8: **return** $[\![z]\!]$

Theorem 3.6. *The partly masked multiplication algorithm 3.6 is d-NI.*

Proof. There are two possible blocks in the for loop: if $y[i] = 0$, the only operation is the shift, which enjoys the d-NI property. If $y[i] = 1$, we use the sec_+ gadget which is also d-NI. Since we reuse the t in the shift, we need to refresh it before.
So the two blocks are d-NI, and their sequential combination leads to a d-NI algorithm □

Algorithm 3.7 SecPolymul: Naive Polynomial multiplication (parameterized by B, the degree of the polynomials)

Input: $[\![\mathbf{x} \in \mathbb{F}_2^B]\!]$, $[\![\mathbf{y} \in \mathbb{F}_2^B]\!]$
Output: $[\![\mathbf{z} = \mathbf{x} \cdot \mathbf{y} \in \mathbb{F}_2^{2B}]\!]$

```
[[z]] ← vector_zero_masking()
for i ← 0 to B − 1 do
    for j ← 0 to B − 1 do
        [[u]] ← sec_&([[x_i]], [[y_j]])
        [[z_(i+j)]] ← [[z_(i+j)]] ⊕ [[u]]
return [[z]]
```

Since we only use a SNI gadget and we update the $\mathbf{z}$ vector on the other hand, the algorithm is d-NI.

Theorem 3.7. *The polynomial multiplication* SecPolymul *algorithm 3.7 parametered with B is d-NI.*

3.2.1.6 Modular random number

To generate keys and errors, we need to be able to draw random numbers modulo n. For this, we are using a method formalized by Lemire [Lem19], where we perform the calculation $\frac{r \times n}{\mathbb{F}_2^p}$, with r randomly drawn in $\mathbb{F}_2^p$. It is easy to see that the result ranges from 0 and $n-1$, with the same distribution as a modulo without performing any division other than with a power of 2. We will only need the gadgets already introduced (masked multiplication see Algorithm 3.6) and the shift, which is a linear operation.

Remark 3.2. It is assumed that the bits can be drawn safely, since the p bits can be drawn on each of the shares of the shared value. In the context of an implementation, the choice of algorithm for effectively drawing these bits is up to the developer.

Algorithm 3.8 Modular random number ($\mathsf{sec}_{\mathsf{rand}}$)

Input: $n \in \mathbb{N}^*$, $p \in \mathbb{N}^*, 2^p \geqslant n$
Output: $[\![r \xleftarrow{\$} \mathbb{Z}_n]\!]$

```
[[r]] ←$ F_{2^p}                          ▷ Draw p bits on each share
[[r]] ← SecMult_partlymasked([[r]], n)
[[r]] ← [[r]] ≫ p                         ▷ Shift on each share
return [[r]]
```

Theorem 3.8. *The modular random number algorithm 3.8 is d-NI.*

Proof. Since p and n are public values, we do not need to mask them.
Since it operates on each share individually, the shift operation is d-NI.
$\mathsf{SecMult}_{\text{partlymasked}}$ is d-NI, by the previous proof. Finally, the random draw is also d-NI since it operates on each share.

The algorithm is d-NI. □

3.2.2 Karatsuba

Several multiplication algorithms are necessary for masking BIKE. As opposed to many other masked designs, the multiplication often takes two masked inputs instead of only one. In addition, the underlying $\mathbb{F}_2$ structure makes NTT-based multiplications irrelevant in BIKE's context, because the NTT is only applicable when the characteristic of the finite field is greater than the degree of the polynomials, which is not the case in BIKE.

We could have considered using a finite field of characteristic greater than the degree of the polynomials and then returning to $\mathbb{F}_2$ when necessary, but this would have prevented us from using specific techniques. The choice was made to use bitslicing in this case (see section 3.2.3), allowing us to continue manipulating binary values, and to remain within Boolean masking. There was also the possibility of using additive FFT, but we did not go any further.

Let B be a parameter denoting the recursion depth. It is fixed experimentally to allow performance optimization. In our experiments, we have fixed $B = 64$. We also set a parameter $s \in \mathbb{N}$ as a power of two corresponding to the degree of the polynomials. Let split be a subroutine that splits the $s/2$ high order and $s/2$ low order bits into two variables.

Algorithm 3.9 Karatsuba multiplication

Input: $[\![\mathbf{p}1 \in \mathbb{F}_2^s]\!]$, $[\![\mathbf{p}2 \in \mathbb{F}_2^s]\!]$
Output: $[\![\mathbf{z} = \mathbf{p}1 \cdot \mathbf{p}2 \in \mathbb{F}_2^{2s}]\!]$

1: **if** $s = B$ **then**
2: **return** SecPolymul($[\![\mathbf{p}1]\!], [\![\mathbf{p}2]\!]$) ▷ Naive polynomial multiplication, see Algorithm 3.7 in Section 3.2.1.5
3: $([\![\mathsf{left}1]\!], [\![\mathsf{right}1]\!]) \leftarrow \mathsf{split}([\![\mathbf{p}1]\!])$ ▷ Splitting the $s/2$ high order and $s/2$ low order bits
4: $([\![\mathsf{left}2]\!], [\![\mathsf{right}2]\!]) \leftarrow \mathsf{split}([\![\mathbf{p}2]\!])$ ▷ Splitting the $s/2$ high order and $s/2$ low order bits
5: $[\![\mathbf{z}1]\!] \leftarrow \mathsf{SecKaratsuba}([\![\mathsf{right}1]\!], [\![\mathsf{right}2]\!])$
6: $[\![\mathbf{z}2]\!] \leftarrow \mathsf{SecKaratsuba}([\![\mathsf{left}1]\!], [\![\mathsf{left}2]\!])$
7: $[\![\mathsf{left}1]\!] \leftarrow \mathsf{refresh}([\![\mathsf{left}1]\!])$
8: $[\![\mathsf{right}1]\!] \leftarrow \mathsf{refresh}([\![\mathsf{right}1]\!])$
9: $[\![\mathsf{left}2]\!] \leftarrow \mathsf{refresh}([\![\mathsf{left}2]\!])$
10: $[\![\mathsf{right}2]\!] \leftarrow \mathsf{refresh}([\![\mathsf{right}2]\!])$
11: $[\![\mathbf{t}1]\!] \leftarrow [\![\mathsf{left}1]\!] \oplus [\![\mathsf{right}1]\!]$ ▷ Coefficient-wise XOR
12: $[\![\mathbf{t}2]\!] \leftarrow [\![\mathsf{left}2]\!] \oplus [\![\mathsf{right}2]\!]$ ▷ Coefficient-wise XOR
13: $[\![\mathbf{z}3]\!] \leftarrow \mathsf{SecKaratsuba}([\![\mathbf{t}1]\!], [\![\mathbf{t}2]\!])$
14: **return** $[\![\mathbf{z}]\!] \leftarrow [\![\mathbf{z}1]\!] \oplus ([\![\mathbf{z}2]\!] \ll s/4) \oplus ([\![\mathbf{z}3]\!] \ll s/2)$ ▷ Coefficient-wise

Theorem 3.9. *The Karatsuba algorithm is d-NI for any power of two s and any bound $B \leqslant s$.*

Proof. Let us prove this theorem by induction on the parameter s. If $s \leqslant B$, the d-NI property is directly inherited from the d-NI property of SecPolymul (Theorem 3.7 in Section 3.2.1.5). Let us assume that the Karatsuba algorithm is d-NI for $s > B$ and let us sketch a proof that is it d-NI for the next power of two: $2 \cdot s$. The algorithm first computes $[\![\mathbf{z}_1]\!], [\![\mathbf{z}_2]\!]$ with d-NI gadgets. Then, the dependencies are broken by the d-SNI refresh before computing $[\![\mathbf{z}_3]\!]$. Finally, the recombination of $[\![\mathbf{z}_1]\!], [\![\mathbf{z}_2]\!]$ and $[\![\mathbf{z}_3]\!]$ uses only coefficient-wise $\mathbb{F}_2$-linear operations. Thus, Karatsuba algorithm is d-NI for $2 \cdot s$ which concludes the proof. □

Remark 3.3 (Generalization to arbitrary s). Note that it is possible to generalize Karatsuba for multiplying two polynomials of any degree s. This generalization can be obtained with an extra padding before the multiplication and a modulo application afterwards. Since the size of polynomials and padding is public and the padding will itself be masked, this does not raise any security concerns. We use the same notation "SecKaratsuba".

3.2.3 Bitslicing

Bitslicing for QC-MDPC was introduced in [Cho16] for QcBits, which is a cryptographic implementation based on QC-MDPC. By way of comparison, BIKE is a much more complete scheme, especially when it comes to security reductions.

The matrices manipulated by bitslice can be seen as vectors of integers, but where each integer is decomposed in binary form. Each column of this matrix therefore represents the binary of a number. The SecBitslice function (Algorithm 3.11) takes two matrices, and performs an addition between the two matrices, i.e. each integer in the first matrix is added with the integer corresponding to the same position in the second matrix. To perform these calculations, we need to use binary operations, hence the use of adders and half-adders. The SecHalf_Bitslice (Algorithm 3.10) function, on the other hand, takes a matrix and a vector, and performs the same operation, with the difference that the vector can be seen as a vector of integers represented on a single bit.
In both cases, we note that the number of rows will be the $\log_2$ of the largest integer to be represented, and that in the case of additions, we need to think about the possibility of having a carry and having to add a row.

These techniques allow computations to be performed very efficiently and in constant time by focusing on the binary representation. In Algorithms 3.10 and 3.11, we present two versions of this BitSlice procedure depending on the type of the input. Both versions will be used in our implementation.

Algorithm 3.10 SecHalf_Bitslice

Input: $[\![\mathbf{X} := (\mathbf{X}_0, \cdots, \mathbf{X}_l) \in \mathbb{F}_2^{k \times l}]\!]$, $[\![\mathbf{y} \in \mathbb{F}_2^l]\!]$
Output: $[\![\mathbf{X} \in \mathbb{F}_2^{l \times k}]\!]$ the result of the bitsliced addition between $[\![\mathbf{X}]\!]$ and $\mathbf{y}$
1: **for** $i := 0$ to $\ell - 1$ **do**
2: $\quad [\![r]\!] := [\![\mathbf{y}_i]\!]$
3: $\quad$ **for** $j := 0$ to $k - 1$ **do**
4: $\quad\quad ([\![\mathbf{X}_{ij}]\!], [\![r]\!]) \leftarrow$ SecHalf_Adder$([\![\mathbf{X}_{ij}]\!], [\![r]\!])$
5: **return** $[\![\mathbf{X}]\!]$

Algorithm 3.11 SecBitslice

Input: $[\![\mathbf{X} := (\mathbf{X}_0, \cdots, \mathbf{X}_k) \in \mathbb{F}_2^{l \times k}]\!]$, $[\![\mathbf{Y} := (\mathbf{Y}_0, \cdots, \mathbf{Y}_k) \in \mathbb{F}_2^{l \times k}]\!]$
Output: $[\![\mathbf{X} \in \mathbb{F}_2^{l \times k}]\!]$ the result of the bitsliced addition between $[\![\mathbf{X}]\!]$ and $[\![\mathbf{Y}]\!]$
1: **for** $i := 0$ to $l - 1$ **do**
2: $\quad [\![r]\!] \leftarrow$ zero_masking()
3: $\quad$ **for** $j := 0$ to $k - 1$ **do**
4: $\quad\quad ([\![\mathbf{X}_{ij}]\!], [\![r]\!]) \leftarrow$ SecAdder$([\![\mathbf{X}_{ij}]\!], [\![\mathbf{Y}_{ij}]\!], [\![r]\!])$
5: **return** $[\![\mathbf{X}]\!]$

Since both SecHalf_Adder and SecAdder are d-NI and all loop iterations use different or updated variables, their sequential combination leads to a d-NI algorithm. Hence the following theorem.

Theorem 3.10. *The* SecHalf_Bitslice *and* SecBitslice *algorithms are* d-NI.

3.2.4 Hamming Weight

We introduce a masked Hamming weight computation. It has been optimized and involves the masked bitslice algorithm presented in Algorithm 3.11.

To obtain the Hamming weight, we need to add all the bits together. We can do this in an optimized way, using bitslice. Indeed, if we split our $\mathbb{F}_2^n$ vector in two and use bitslice between the left and right parts, we'll end up with an $\mathbb{F}_2^{2\times\frac{n}{2}}$ matrix, each column of two rows containing the binary resulting from our addition on each of the positions. Here, position does not matter, as we want the result of the addition of all bits. Iterating in this way, we will end up with a column vector containing the binary of the Hamming weight. All we have to do now is reconstitute it.

Similarly to Karatsuba, we denote by right and left the cut in length of the matrix. For example, if $\mathbf{T} \in \mathbb{F}_2^{l\times k}$, $\mathsf{right}(\mathbf{T})$ and $\mathsf{left}(\mathbf{T}) \in \mathbb{F}_2^{l\times\frac{k}{2}}$.

$[\![\mathbf{T}]\!]$ is a matrix that starts with one row, and will gain one more row per loop turn (call to bitslice). So we initialize $[\![\mathbf{T}_0]\!]$ as a vector, then at each iteration, $[\![\mathbf{T}]\!]$ will gain a row.

Algorithm 3.12 Hamming weight ($\mathsf{sec_{hw}}$)

Input: $[\![\mathbf{x} \in \mathbb{F}_2^n]\!]$
Output: $[\![y \in \mathbb{Z}_n]\!]$ the hamming weight of $\mathbf{x}$

$[\![\mathbf{T}_0]\!] \leftarrow [\![\mathbf{x}]\!]$ ▷ We initialize the first line of the $[\![\mathbf{T}]\!]$ matrix with $[\![\mathbf{x}]\!]$ vector
$j \leftarrow 1$
$i \leftarrow \frac{n}{2}$
while $i \neq 0$ **do**
    $[\![\mathbf{T}]\!] \leftarrow \mathsf{SecBitslice}(\mathsf{left}([\![\mathbf{T}]\!]), \mathsf{right}([\![\mathbf{T}]\!]))$ ▷ Cut in length
    $j \leftarrow j + 1$
    $i \leftarrow \frac{i}{2}$
$[\![y]\!] \leftarrow \mathsf{zero\_masking}()$
for $i \leftarrow 0$ to $j - 1$ **do**
    $[\![y]\!] \leftarrow [\![y]\!] \oplus ([\![\mathbf{T}_{0,i}]\!] \ll i)$
return $[\![y]\!]$

Theorem 3.11. *The hamming weight algorithm 3.12 is* d-NI.

Proof. Since as SecBitslice has been proved d-NI in Theorem 3.10 and all loops use updated variables, their composition leads to a d-NI algorithm. □

3.2.5 Cyclic shifting

This is a masked version of the barrel shifter algorithm. The principle of the barrel shifter is to shift the polynomial of each power of 2, and to keep only the shifts corresponding to a position at 1 in the binary of the number of shifts to be performed.

We define SecCyclic_Shift the function that allows to shift a masked polynomial with a public value. Since all we need to do to realize this are binary shifts with public values (linear), & with known masks (to recover the overflowing part and apply the modulo) and XOR (linear), it is safe and not a concern.

Algorithm 3.13 Secure masked cyclic shift ($\mathsf{sec}_{\gg}$)

Input: $[\![\mathbf{x} \in \mathbb{F}_2^n]\!]$, $[\![s \in \mathbb{N}]\!]$
Output: $[\![\mathbf{y}]\!] = [\![\mathbf{x} \gg s \in \mathbb{F}_2^n]\!]$

1: $[\![\mathbf{y}]\!] \leftarrow [\![\mathbf{x}]\!]$
2: **for** $i \leftarrow 0$ to $\lfloor \log_2(n) \rfloor$ **do**
3: $\quad [\![v]\!] \leftarrow [\![s]\!][i]$ ▷ $[\![s]\!][i] = ([\![s]\!] \gg i)$ & 1
4: $\quad [\![\mathbf{t}]\!] \leftarrow \mathsf{SecCyclic\_Shift}([\![\mathbf{y}]\!], 2^i)$
5: $\quad$ **for** $j \leftarrow 0$ to $n-1$ **do**
6: $\quad\quad [\![s1]\!] \leftarrow \mathsf{sec}_{\&}([\![\mathbf{t}_j]\!], [\![v]\!])$
7: $\quad\quad [\![s2]\!] \leftarrow \mathsf{sec}_{\&}([\![\mathbf{y}_j]\!], \neg[\![v]\!])$
8: $\quad\quad [\![\mathbf{y}_j]\!] \leftarrow [\![s1]\!] \oplus [\![s2]\!]$
9: **return** $[\![\mathbf{y}]\!]$

Theorem 3.12. *The secure cyclic shift algorithm 3.13 is d-NI.*

Proof. In the most nested for loop, we used two $\mathsf{sec}_{\&}$, which are d-SNI. The $\oplus$ being d-NI, the block is d-NI.
Since the i loop is composed by d-NI gadgets, and the $\mathbf{y}$ vector is updated in the for i loop, all of this is d-NI.
So their sequential combination leads to a d-NI algorithm. □

3.2.6 Fisher-Yates

The generation of sparse polynomials is performed using the Fisher-Yates technique. The Fisher-Yates shuffle designed for computer use was first introduced by Richard Durstenfled in 1964 [Dur64]. It consists in generating a random permutation. It has been adapted here for use in BIKE, where a vector of n random elements is drawn and each i position contains a value between 0 and $n-i$. Since it is important to avoid any duplicates, we go through the array backwards and we replace the value by the index i in case of duplicates. Despite a bias in the distribution, this does not affect the security of the scheme as proved in [Sen21].

This procedure can be masked as presented in Algorithm 3.14. It uses $\mathsf{sec}_{\mathsf{rand}}$, presented in Section 3.2.1.6.

Algorithm 3.14 Fisher-Yates (SecFisherYates)

Input: $s \in \mathbb{N}$, $n \in \mathbb{N}$
Output: $[\![\mathbf{r} \in \mathbb{Z}_n^s]\!]$ a randomly generated vector without repeated values

1: **for** $i \leftarrow s-1$ to 0 **do**
2: $\quad [\![\mathbf{r}_i]\!] \leftarrow \mathsf{sec}_{\mathsf{rand}}(n-i)$
3: $\quad$ Initialize $[\![i]\!]$ as a Boolean sharing of i
4: $\quad [\![\mathbf{r}_i]\!] \leftarrow \mathsf{sec}_{+\text{partlymasked}}([\![\mathbf{r}_i]\!], i)$
5: $\quad$ **for** $j \leftarrow i+1$ to $s-1$ **do**
6: $\quad\quad [\![\mathbf{r}_j]\!] \leftarrow \mathsf{refresh}([\![\mathbf{r}_j]\!])$
7: $\quad\quad [\![b]\!] \leftarrow \mathsf{sec}_{=}([\![\mathbf{r}_i]\!], [\![\mathbf{r}_j]\!])$
8: $\quad\quad [\![\mathbf{r}_i]\!] \leftarrow \mathsf{refresh}([\![\mathbf{r}_i]\!])$
9: $\quad\quad [\![i]\!] \leftarrow \mathsf{refresh}([\![i]\!])$
10: $\quad\quad [\![\mathbf{r}_i]\!] \leftarrow \mathsf{sec}_{\mathsf{if}}([\![i]\!], [\![\mathbf{r}_i]\!], [\![b]\!]))$
11: **return** $[\![\mathbf{r}]\!]$

Theorem 3.13. *The Fisher-Yates algorithm 3.14 is d-NI.*

Proof. The Fisher-Yates algorithm involves many dependency loops. Indeed, each random $[\![\mathbf{r}_i]\!]$ is compared to all the previously derived ones. However, each value is refreshed before being used. Thus, the loop in lines 6 to 10 can be seen itself as a d-SNI gadget outputting $[\![\mathbf{r}_i]\!]$. Besides, the operations in lines 2 to 4 are d-NI. Hence, the outer loop can be seen as a sequential combination of NI gadgets and a d-SNI gadget for lines 6 to 10. In consequence, the algorithm is d-NI. □

3.2.7 Sparse-dense operations

In BIKE, it is often possible to leverage the fact that some masked polynomials are stored in sparse format. Using the $\mathsf{sec}_{\gg}$ gadget introduced earlier, we can develop a gadget to perform multiplication between a sparse polynomial and a dense polynomial. The main idea behind it is that in $\mathbb{F}_2$, with $\mathbf{y} \in \mathbb{Z}_n^c$ in sparse representation, the multiplication of the two polynomials $\mathbf{x}$ and $\mathbf{y}$ is equal to $\sum_{i=0}^{c-1}(\mathbf{x}x^{\mathbf{y}_i})$

Algorithm 3.15 Sparse-dense multiplication ($\mathsf{SecMult}_{\text{sparsedense}}$)

Input: $[\![\mathbf{x} \in \mathbb{F}_2^n]\!]$, $[\![\mathbf{y} \in \mathbb{Z}_n^c]\!]$
Output: $[\![\mathbf{z} = \mathbf{x} \cdot \mathbf{y} \in \mathbb{F}_2^n]\!]$

1: $[\![\mathbf{z}]\!] \leftarrow \mathsf{vector\_zero\_masking}()$
2: **for** $i \leftarrow 0$ to $c - 1$ **do**
3: $\quad [\![\mathbf{t}]\!] \leftarrow \mathsf{sec}_{\gg}([\![\mathbf{x}]\!], [\![\mathbf{y}_i]\!])$ ▷ See Algorithm 3.13 in Section 3.2.5
4: $\quad [\![\mathbf{x}]\!] \leftarrow \mathsf{refresh}([\![\mathbf{x}]\!])$
5: $\quad [\![\mathbf{z}]\!] \leftarrow [\![\mathbf{z}]\!] \oplus [\![\mathbf{t}]\!]$ ▷ Coefficient-wise XOR
6: **return** $[\![\mathbf{z}]\!]$

Theorem 3.14. *The* $\mathsf{SecMult}_{\text{sparsedense}}$ *algorithm 3.15 is* d-NI.

Proof. Since the gadgets $\mathsf{sec}_{\gg}$ and $\oplus$ are d-NI. And even though if $\mathbf{x}$ is reused in each loop, $\mathbf{x}$ is refreshed (d-SNI). □

3.2.8 Inversion

A masked polynomial inversion is needed for inverting $\mathbf{h}_0$ inside the key generation. For this, we use the fast polynomial inversion method introduced in [DGK20a].

We note $\mathsf{sec}_{\mathsf{pow}}$ a d-NI gadget allowing to raise a polynomial to the given (known) power. Since we only perform elevations of powers of 2, it boils down to permutations as the underlying ring is $\mathbb{F}_2$.

Algorithm 3.16 SecInversion

Input: $[\![\mathbf{x} \in \mathbb{F}_2^n]\!]$
Output: $[\![\mathbf{y} = \mathbf{x}^{-1} \in \mathbb{F}_2^n]\!]$

1: $[\![\mathbf{f}]\!] \leftarrow [\![\mathbf{x}]\!]$
2: $[\![\mathbf{y}]\!] \leftarrow [\![\mathbf{x}]\!]$
3: $[\![\mathbf{y}]\!] \leftarrow \mathsf{refresh}([\![\mathbf{y}]\!])$
4: **for** $i \leftarrow 0$ to $\lfloor \log_2(n) \rfloor - 1$ **do**
5: $\quad [\![\mathbf{g}]\!] \leftarrow \mathsf{sec}_{\mathsf{pow}}([\![\mathbf{f}]\!], 2^{2^i})$
6: $\quad [\![\mathbf{f}]\!] \leftarrow \mathsf{refresh}([\![\mathbf{f}]\!])$
7: $\quad [\![\mathbf{f}]\!] \leftarrow \mathsf{SecKaratsuba}([\![\mathbf{f}]\!], [\![\mathbf{g}]\!])$
8: $\quad$ **if** the $(i+1)^{\text{th}}$ bit of $n-2$ is 1 **then**
9: $\quad\quad [\![\mathbf{t}]\!] \leftarrow \mathsf{sec}_{\mathsf{pow}}([\![\mathbf{f}]\!], 2^{(n-2) \pmod{2^{(i+1)}}})$
10: $\quad\quad [\![\mathbf{y}]\!] \leftarrow \mathsf{SecKaratsuba}([\![\mathbf{y}]\!], [\![\mathbf{t}]\!])$
11: $[\![\mathbf{y}]\!] \leftarrow \mathsf{sec}_{\mathsf{pow}}([\![\mathbf{y}]\!], 2)$
12: **return** $[\![\mathbf{y}]\!]$

Theorem 3.15. *The masked inversion algorithm 3.16 is* d-NI.

Proof. The first iteration of the algorithm is presented below. One can graphically conclude that each iteration is d-NI as all the observations can be simulated with at

most d shares of $([\![\mathbf{f}]\!], [\![\mathbf{y}]\!])$. Thus, the full loop is d-NI. In addition, the final operation is d-NI. And, since both $[\![\mathbf{f}]\!]$ and $[\![\mathbf{y}]\!]$ are initialized with the same input $[\![\mathbf{x}]\!]$, one of them should be refreshed to end up with a full d-NI gadget.

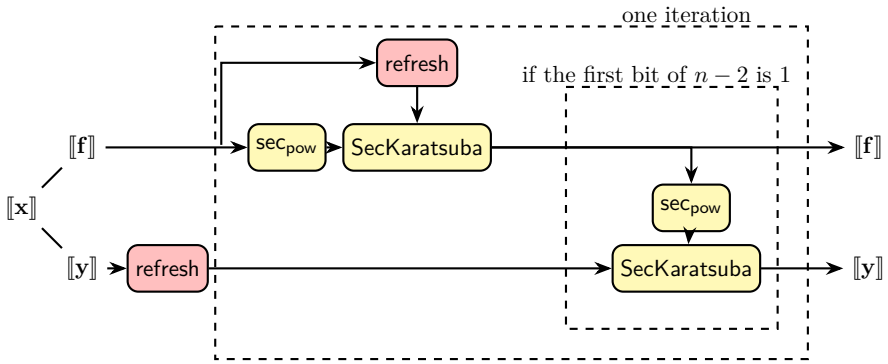

Figure 3.1: Structure of the polynomial inversion algorithm

□

3.3 Masked BIKE

In this section, the parameters used are those of the BIKE scheme. You can find them in Table 2.1.

3.3.1 Key generation

The masked key generation consists of generating private keys $\mathbf{h}_0$ and $\mathbf{h}_1$, and computing the public key $\mathbf{h}$. To do this, we use the Fisher-Yates and inversion algorithms presented above.

Algorithm 3.17 Masked key generation

Output: $[\![\mathbf{h}_0^\circ \in \mathbb{Z}_r^{\frac{w}{2}}]\!]$, $[\![\mathbf{h}_1^\circ \in \mathbb{Z}_r^{\frac{w}{2}}]\!]$, $[\![\mathbf{h}_0 \in \mathbb{F}_2^r]\!]$, $[\![\mathbf{h}_1 \in \mathbb{F}_2^r]\!]$, $[\![\mathbf{h} \in \mathbb{F}_2^r]\!]$, $[\![\vec{\sigma} \in \mathbb{F}_2^\ell]\!]$

1: $[\![\mathbf{h}_0^\circ]\!] \leftarrow \mathsf{SecFisherYates}(\frac{w}{2}, r)$ ▷ Algorithm 3.14
2: $[\![\mathbf{h}_1^\circ]\!] \leftarrow \mathsf{SecFisherYates}(\frac{w}{2}, r)$
3: $[\![\mathbf{h}_0]\!] \leftarrow \mathsf{sparse\_to\_dense}([\![\mathbf{h}_0^\circ]\!])$
4: $[\![\mathbf{h}_1]\!] \leftarrow \mathsf{sparse\_to\_dense}([\![\mathbf{h}_1^\circ]\!])$
5: $[\![\mathbf{h}_0^{-1}]\!] \leftarrow \mathsf{SecInversion}([\![\mathbf{h}_0]\!], r)$ ▷ Algorithm 3.16
6: $[\![\mathbf{h}]\!] \leftarrow \mathsf{SecKaratsuba}([\![\mathbf{h}_0^{-1}]\!], [\![\mathbf{h}_1]\!])$ ▷ Algorithm 3.9
7: $[\![\vec{\sigma}]\!] \xleftarrow{\$} \mathbb{F}_2^\ell$
8: **return** $sk = ([\![\mathbf{h}_0^\circ]\!], [\![\mathbf{h}_1^\circ]\!], [\![\mathbf{h}_0]\!], [\![\mathbf{h}_1]\!])$, $pk = [\![\mathbf{h}]\!]$, $[\![\vec{\sigma}]\!]$

Provided that all the gadgets enjoy the d-NI property, their sequential combination leads to a d-NI algorithm. Thus we have the following result.

Theorem 3.16. *The masked key generation algorithm 3.17 is d-NI.*

3.3.2 Encapsulation

In this section, we take a closer look at encapsulation which consists in generating the error, then encrypting it using the public key. A few additional steps are taken to make the algorithm IND-CCA.

IND-CCA masked implementation The IND-CCA security of the scheme is achieved thanks to the Fujisaki-Okamoto transformation. This transformation consists in XORing the seed used to generate the secret with the hashed secret. This will allow, during the decryption, to recover the seed and thus to check if the secret has been honestly generated. This transformation prevents active chosen ciphertext attack. In BIKE [Ara+22], the $\mathbf{K}$, $\mathbf{L}$ and $\mathbf{H}$ hash functions (see Algorithm 2.3) are instantiated with SHAKE256 and SHA384. These functions have already been protected in the masked implementation of Saber (see [DAn+20] for more information about Saber) in [Kun+22]. This framework is easily adaptable for BIKE without major modification. Masking is done in a similar way, keeping the same masking order.

3.3.2.1 Error generation

The error generation algorithm is necessary for both encapsulation and decapsulation. Its masked version is introduced below in Algorithm 3.18. It consists in generating a masked error vector $[\![\mathbf{e}_0]\!], [\![\mathbf{e}_1]\!]$.

Since we know that $|\mathbf{e}_0| + |\mathbf{e}_1| = t$ but not $|\mathbf{e}_0|$ and $|\mathbf{e}_1|$ individually, representing them in sparse representation gives information about them. The error must be in dense representation, to avoid leaking this sensitive information.

Algorithm 3.18 Masked Error generation SecErrorGen

Input: $[\![s]\!] \in \mathbb{F}_2^\ell$ the seed for SecFisherYates
Output: $[\![\mathbf{e}_0]\!] \in \mathbb{F}_2^r$, $[\![\mathbf{e}_1]\!] \in \mathbb{F}_2^r$

1: $[\![\mathbf{e}_0]\!] \leftarrow$ vector_zero_masking()
2: $[\![\mathbf{e}_1]\!] \leftarrow$ vector_zero_masking()
3: $[\![\mathbf{e}^\circ]\!] \leftarrow$ SecFisherYates$(t, 2 \times r)$ ▷ algorithm 3.14, t the error weight
4: **for** $i \leftarrow 0$ to $t-1$ **do**
5: $\quad [\![v]\!] \leftarrow \mathsf{sec}_{+\text{partlymasked}}([\![\mathbf{e}_i^\circ]\!], -r)$ ▷ see section 2.2.2.3
6: $\quad [\![\mathbf{e}_i^\circ]\!] \leftarrow$ refresh$([\![\mathbf{e}_i^\circ]\!])$
7: $\quad [\![\mathbf{t}^\circ]\!] \leftarrow \mathsf{sec}_{\mathsf{if}}([\![\mathbf{e}_i^\circ]\!], [\![v]\!],$ sign_bit(refresh$([\![v]\!]))$) ▷ see section 2.2.2.4
8: $\quad [\![\mathbf{t}]\!] \leftarrow$ sparse_to_dense$([\![\mathbf{t}^\circ]\!])$ ▷ see section 3.1, polynomial with only one coefficient
9: $\quad [\![\mathbf{e}_0]\!] \leftarrow [\![\mathbf{e}_0]\!] \oplus \mathsf{sec}_{\mathsf{if}}([\![\mathbf{t}]\!],$ vector_zero_masking(), sign_bit(refresh$([\![v]\!]))$)
10: $\quad [\![\mathbf{e}_1]\!] \leftarrow [\![\mathbf{e}_1]\!] \oplus \mathsf{sec}_{\mathsf{if}}($vector_zero_masking(), $[\![\mathbf{t}]\!],$ sign_bit(refresh$([\![v]\!]))$)
11: **return** $[\![\mathbf{e}_0]\!], [\![\mathbf{e}_1]\!]$

Remark 3.4. In the context of error generation, we use the $[\![\mathbf{s}]\!]$ seed to generate our $[\![\mathbf{e}]\!]$ vector using SHAKE256 hash function (see Section 3.3.2) which is then processed in the same way as Fisher-Yates. Since we have defined Fisher-Yates (Algorithm 3.14) with random generation within it, this would require us to redefine it to take a random vector, which would complicate its understanding. This does not change the nature of the algorithm, so to avoid making it unnecessarily complicated, we call Fisher-Yates directly.

In this algorithm, the intermediate values are used only once within d-NI gadgets, the only exception being $[\![\mathbf{e}_i^\circ]\!]$, which is refreshed ($d$-SNI) before its new use. We can therefore conclude with the following theorem.

Theorem 3.17. *The error generation algorithm 3.18 is* d-NI.

3.3.2.2 Encapsulation algorithm

Algorithm 3.19 Encapsulation

Input: $[\![\mathbf{h} \in \mathbb{F}_2^r]\!]$
Output: $[\![\mathbf{c} \in \mathbb{F}_2^{r+\ell}]\!]$

1: $[\![\mathbf{m}]\!] \xleftarrow{\$} \mathbb{F}_2^\ell$
2: $[\![\mathbf{e}]\!] = ([\![\mathbf{e}_0]\!], [\![\mathbf{e}_1]\!]) \leftarrow$ SecErrorGen$([\![\mathbf{m}]\!])$ ▷ Algorithm 3.18
3: $[\![\mathbf{m}]\!] \leftarrow$ refresh$([\![\mathbf{m}]\!])$
4: $[\![\mathbf{c}]\!] \leftarrow$ SecKaratsuba$([\![\mathbf{e}_1]\!], [\![\mathbf{h}]\!])$
5: $[\![\mathbf{c}]\!] \leftarrow [\![\mathbf{c}]\!] \oplus [\![\mathbf{e}_0]\!]$ ▷ Coefficient-wise XOR
6: $[\![\mathbf{e}]\!] \leftarrow$ refresh$([\![\mathbf{e}]\!])$
7: $[\![\mathbf{c}_r]\!] \leftarrow \mathsf{L}([\![\mathbf{e}]\!]) \oplus [\![\mathbf{m}]\!]$ ▷ see Section 3.3.2, Coefficient-wise XOR
8: **return** $[\![\mathbf{e}_0]\!], [\![\mathbf{e}_1]\!]$

All the functions used are d-NI.

Since the only variable that has been reused is the seed $\mathbf{m}$ and the generated error $\mathbf{e}$, we have to refresh them. We can conclude that the algorithm is itself d-NI.

Theorem 3.18. *The encapsulation algorithm 3.19 is d-NI.*

3.3.3 Decapsulation

Finally, we take a closer look at decapsulation. As this is a fairly dense process, it has been decomposed into lots of smaller gadgets.

- The syndrome computing. The syndrome must be calculated at each decoding iteration.
- The threshold computing. Like syndrome, it must be calculated at each iteration and depends on the hamming weight of the syndrome.
- The counters computing. At each iteration, we need to use bitslicing to obtain the counter of each position.

With these gadgets, we can construct the decoder (and the grey zone, which is an additional decoding performed at the first decoder iteration).

The decapsulation is composed of the BGF decoder, and a few operations to make decapsulation IND-CCA.

3.3.3.1 Computing the syndrome

Within decoding, we need to calculate the syndrome, which corresponds to the use of Karatsuba between the error construction $[\![\mathbf{e}_0]\!]$ and $[\![\mathbf{e}_1]\!]$ and the private key $[\![\mathbf{h}_0]\!]$ and $[\![\mathbf{h}_1]\!]$, which we will XOR with the initial syndrome $[\![\mathbf{c}_0]\!][\![\mathbf{h}_0]\!]$.

Algorithm 3.20 compute_syndrome

Input: $[\![\mathbf{h}_0 \in \mathbb{F}_2^r]\!]$, $[\![\mathbf{h}_1 \in \mathbb{F}_2^r]\!]$, $[\![\mathbf{e}_0 \in \mathbb{F}_2^r]\!]$, $[\![\mathbf{e}_1 \in \mathbb{F}_2^r]\!]$, $[\![\mathbf{s} = \mathbf{c}_0\mathbf{h}_0 \in \mathbb{F}_2^r]\!]$
Output: $[\![\mathbf{s}1 = \mathbf{c}_0\mathbf{h}_0 + \mathbf{e}_0\mathbf{h}_0 + \mathbf{e}_1\mathbf{h}_1 \in \mathbb{F}_2^r]\!]$

1: $[\![\mathbf{s}2]\!] \leftarrow \mathsf{SecKaratsuba}([\![\mathbf{e}_0]\!], [\![\mathbf{h}_0]\!])$
2: $[\![\mathbf{s}3]\!] \leftarrow \mathsf{SecKaratsuba}([\![\mathbf{e}_1]\!], [\![\mathbf{h}_1]\!])$
3: $[\![\mathbf{s}1]\!] \leftarrow [\![\mathbf{s}]\!] \oplus [\![\mathbf{s}2]\!] \oplus [\![\mathbf{s}3]\!]$ ▷ Coefficient-wise XOR
4: **return** $[\![\mathbf{s}1]\!]$

Since different variables are used in each of the function calls (all d-NI), we get the following theorem.

Theorem 3.19. *The syndrome computing algorithm 3.20 is d-NI.*

3.3.3.2 Computing the threshold

The threshold is an integer value that needs to be recomputed several times during decoding. To find out whether a position will be flipped, we need to check whether a value exceeds a threshold. To calculate this threshold, we need to use the Hamming weight of the syndrome. Initially, the calculation of the threshold is done with floats, which is a concern for the masking. We have therefore reduced this to simple operations on integers such as threshold is equal to $\mathsf{max}(\lfloor \frac{T_0 \cdot S + T_1}{2^{T_2}} \rfloor, T_3)$ (see Table 3.1). With this formula, we also avoid any worries about division, since we can use a shift.

Table 3.1: BIKE's threshold parameters

	Level 1	Level 3	Level 5
T_0	58487	11306501	269987
T_1	113497866	32768023488	1199805825
T_2	23	31	26
T_3	36	52	69

Algorithm 3.21 T computing ($\mathsf{sec_T}$)

Input: $[\![S \in \mathbb{Z}_r]\!]$, T_0, T_1, T_2, T_3 fixed parameters of the scheme
Output: $[\![T = \mathsf{max}(\lfloor \frac{T_0 \cdot S + T_1}{2^{T_2}} \rfloor, T_3) \in \mathbb{Z}_r]\!]$
1: $[\![t]\!] \leftarrow \mathsf{SecMult}_{\text{partlymasked}}([\![S]\!], T_0)$ ▷ Algorithm 3.6
2: $[\![T]\!] \leftarrow \mathsf{sec}_{+\text{partlymasked}}([\![t]\!], T_1)$
3: $[\![T]\!] \leftarrow [\![T]\!] \gg T_2$
4: $[\![T]\!] \leftarrow \mathsf{sec}_{\mathsf{max}}([\![T]\!], [\![T_3]\!])$ ▷ Algorithm 3.2
5: **return** $[\![T]\!]$

Algorithm 3.22 SecThreshold

Input: $[\![\mathbf{s} \in \mathbb{F}_2^r]\!]$
Output: $[\![T \in \mathbb{Z}_r]\!]$ the threshold calculated from the syndrome
1: $[\![S]\!] \leftarrow \mathsf{sec}_{\mathsf{hw}}([\![\mathbf{s}]\!])$ ▷ Algorithm 3.12
2: $[\![T]\!] \leftarrow \mathsf{sec_T}([\![S]\!])$ ▷ Algorithm 3.21
3: **return** $[\![T]\!]$

Since we perform a sequence of operations that are d-NI themselves, we can establish the following theorem.

Theorem 3.20. *The algorithm computing the threshold 3.22 is* d-NI.

3.3.3.3 Computing the counters

During decoding, it is necessary to compute the number of unsatisfied parity-check equations for a given position. We present a masked version of this routine in Algorithm 3.23.

Let us denote by $\mathbf{C} \in (\mathbb{F}_2^{r\times(\lfloor\log_2(\frac{w}{2})\rfloor+1)} \times \mathbb{F}_2^{r\times(\lfloor\log_2(\frac{w}{2})\rfloor+1)})$ the matrix containing the binary representations of the counters of each coefficient. We manipulate this matrix as two double dimensional matrices, $[\![\mathbf{C}_0]\!]$ and $[\![\mathbf{C}_1]\!]$. This algorithm uses a gadget that consists in filling a matrix with a value, which was previously introduced in Algorithm 3.3.

Algorithm 3.23 Counter computing (SecCounter)

Input: $[\![\mathbf{s} \in \mathbb{F}_2^r]\!]$, $[\![T \in \mathbb{Z}_r]\!]$, $[\![\mathbf{h}_0^\circ \in \mathbb{Z}_r^{\frac{w}{2}}]\!]$, $[\![\mathbf{h}_1^\circ \in \mathbb{Z}_r^{\frac{w}{2}}]\!]$

Output: $[\![\mathbf{C} \in (\mathbb{F}_2^{(\lfloor\log_2(\frac{w}{2})\rfloor+1)\times r} \times \mathbb{F}_2^{(\lfloor\log_2(\frac{w}{2})\rfloor+1)\times r})]\!]$ two masked matrices containing the binary representations of the counters for each coefficient

1: $[\![-T]\!] \leftarrow \mathsf{SecMult}_{\text{partlymasked}}([\![T]\!], -1)$ ▷ Algorithm 3.6
2: $[\![\mathbf{C}_0]\!] \leftarrow$ matrix_zero_masking()
3: $[\![\mathbf{C}_1]\!] \leftarrow$ matrix_zero_masking()
4: $[\![\mathbf{P}]\!] \leftarrow [[\![\mathbf{h}_0^\circ]\!], [\![\mathbf{h}_1^\circ]\!]]$
5: **for** $i \leftarrow 0$ to 1 **do**
6: **for** $j \leftarrow 0$ to $\frac{w}{2} - 1$ **do**
7: $[\![\mathbf{s}]\!] \leftarrow \mathsf{refresh}([\![\mathbf{s}]\!])$
8: $[\![\mathbf{z}]\!] \leftarrow \mathsf{sec}_{\gg}([\![\mathbf{s}]\!], [\![\mathbf{P}_{i,j}]\!])$ ▷ Algorithm 3.13
9: $[\![\mathbf{C}_i]\!] \leftarrow \mathsf{SecHalf\_Bitslice}([\![\mathbf{C}_i]\!], [\![\mathbf{z}]\!])$ ▷ Algorithm 3.10
10: $[\![\mathbf{T}_0]\!] \leftarrow \mathsf{sec\_fill}([\![-T]\!])$ ▷ Algorithm 3.3
11: $[\![\mathbf{T}_1]\!] \leftarrow \mathsf{sec\_fill}(\mathsf{refresh}([\![-T]\!]))$ ▷ Algorithm 3.3
12: $[\![\mathbf{C}_0]\!] \leftarrow \mathsf{SecBitslice}([\![\mathbf{C}_0]\!], [\![\mathbf{T}_0]\!])$ ▷ Algorithm 3.11
13: $[\![\mathbf{C}_1]\!] \leftarrow \mathsf{SecBitslice}([\![\mathbf{C}_1]\!], [\![\mathbf{T}_1]\!])$ ▷ Algorithm 3.11
14: **return** $[\![\mathbf{C}]\!] = [[\![\mathbf{C}_0]\!], [\![\mathbf{C}_1]\!]]$

Theorem 3.21. *The counter computing algorithm is d-NI.*

Proof. The procedure in lines 7 to 9 of Algorithm 3.23 is depicted in fig. 3.2. One can see that all the loops are broken with a d-SNI refresh gadget. Thus lines 7 to 9 can be seen as a d-NI gadget.

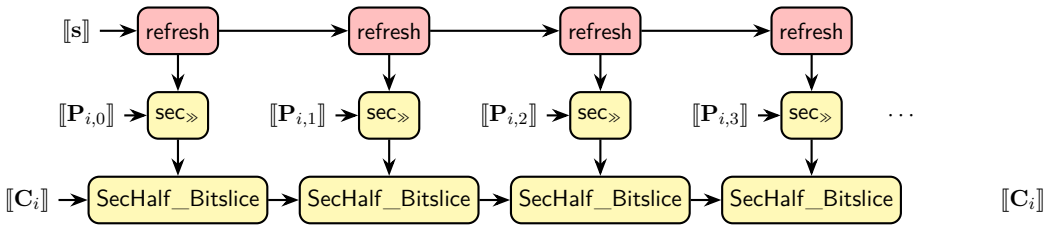

Figure 3.2: Sub-structure of the counter algorithm

The rest of the algorithm is a sequence of d-NI gadgets ($\mathsf{SecMult}_{\text{partlymasked}}$, $\mathsf{sec}_{\mathsf{fill}}$, SecBitslice), thus the full algorithm is d-NI. □

3.3.3.4 BGF decoder & grey zone

We now describe the most important part of the decapsulation: the masked BGF decoder. The unmasked version of the BGF decoder has been presented in Section 2.1.3, the grey zone is represented by lines 5 to 7 of Algorithm 2.5. The masked version of Algorithm 2.5 is detailed in Algorithm 3.25.

During decoding, we want to construct $[\![\mathbf{e}_0]\!]$ and $[\![\mathbf{e}_1]\!]$ such that $\mathbf{h}_0\mathbf{e}_0 + \mathbf{h}_1\mathbf{e}_1 = \mathbf{s}$. To achieve this, we carry out a fixed number of iterations, in which we calculate the syndrome, its Hamming weight, and the threshold. Once the matrix of counters minus the threshold has been obtained, we modify $[\![\mathbf{e}_0]\!]$ and $[\![\mathbf{e}_1]\!]$ according to the first row of the matrix, which corresponds to the sign bit of the counter. At the next iteration, the syndrome will be updated with the modifications made to $[\![\mathbf{e}_0]\!]$ and $[\![\mathbf{e}_1]\!]$.

During the first iteration, additional operations are performed: this is the grey zone. After initial bitflipping, we will recover the positions that would have been in the black zone (those which were flipped, i.e. which exceeded the threshold) and those which were in the grey zone (those which were not flipped, but which where at τ units below the threshold e.g. $\tau = 3$). We will recalculate the counters, but with the threshold set at $\frac{\frac{w}{2}+1}{2}$. If we see that some positions exceed this threshold even though they were previously flipped, we reflip them in the other direction. Once done, we perform a final iteration, this time recalculating the counters and flipping positions that exceed the threshold $\frac{\frac{w}{2}+1}{2}$ but were in the grey zone (i.e. never flipped before).

Let $\mathbf{C} = (\mathbf{C}_0, \mathbf{C}_1) \in (\mathbb{F}_2^{r\times(\lfloor\log_2(\frac{w}{2})\rfloor+1)} \times \mathbb{F}_2^{r\times(\lfloor\log_2(\frac{w}{2})\rfloor+1)})$ be the same pair of matrices presented in section 3.3.3.3. The notation $\mathbf{C}_{0,\lfloor\log_2(\frac{w}{2})\rfloor,*}$ represents the entire row of height $\lfloor\log_2(\frac{w}{2})\rfloor$.

The grey zone gadget takes as input the black zone $\mathbf{T}_0$, which is the matrix containing the counters minus the threshold. With, we can calculate the grey zone $\mathbf{T}_1$, which contains the counters minus the threshold plus τ.

Algorithm 3.24 SecGreyZone

Input: $sk = \left(\llbracket \mathbf{h}_0 \in \mathbb{F}_2^r \rrbracket, \llbracket \mathbf{h}_1 \in \mathbb{F}_2^r \rrbracket, \llbracket \mathbf{h}_0^\circ \in \mathbb{Z}_r^{\frac{w}{2}} \rrbracket, \llbracket \mathbf{h}_1^\circ \in \mathbb{Z}_r^{\frac{w}{2}} \rrbracket\right)$, $\llbracket \mathbf{T}_0 \in (\mathbb{F}_2^{r\times(\lfloor \log_2(\frac{w}{2})\rfloor+1)} \times \mathbb{F}_2^{r\times(\lfloor \log_2(\frac{w}{2})\rfloor+1)})\rrbracket$, $\llbracket \mathbf{e}_0 \in \mathbb{F}_2^r \rrbracket$, $\llbracket \mathbf{e}_1 \in \mathbb{F}_2^r \rrbracket$, $\llbracket \mathbf{s} \in \mathbb{F}_2^r \rrbracket$

Output: $\llbracket \mathbf{e}_0 \in \mathbb{F}_2^r \rrbracket$, $\llbracket \mathbf{e}_1 \in \mathbb{F}_2^r \rrbracket$

1: Initialize $\llbracket \tau \rrbracket$ as a Boolean sharing of 3 ▷ 3 is a fixed parameter
2: $\llbracket \mathbf{V} \rrbracket \leftarrow \mathsf{sec}_{\mathsf{fill}}(\llbracket \tau \rrbracket)$ ▷ Algorithm 3.3
3: $\llbracket \mathbf{T}_{1,0} \rrbracket \leftarrow \mathsf{SecBitslice}(\llbracket \mathbf{T}_{0,0} \rrbracket, \llbracket \mathbf{V} \rrbracket)$ ▷ Algorithm 3.11
4: $\llbracket \mathbf{V} \rrbracket \leftarrow \mathsf{refresh}(\llbracket \mathbf{V} \rrbracket)$
5: $\llbracket \mathbf{T}_{1,1} \rrbracket \leftarrow \mathsf{SecBitslice}(\llbracket \mathbf{T}_{0,1} \rrbracket, \llbracket \mathbf{V} \rrbracket)$ ▷ Algorithm 3.11
6: $\llbracket \mathbf{T}_0 \rrbracket \leftarrow \mathsf{refresh}(\llbracket \mathbf{T}_0 \rrbracket)$
7: $\llbracket \mathbf{T}_{1,0,\lfloor \log_2(\frac{w}{2})\rfloor,*} \rrbracket \leftarrow \llbracket \mathbf{T}_{0,0,\lfloor \log_2(\frac{w}{2})\rfloor,*} \rrbracket \oplus \llbracket \mathbf{T}_{1,0,\lfloor \log_2(\frac{w}{2})\rfloor,*} \rrbracket$ ▷ Coefficient-wise XOR
8: $\llbracket \mathbf{T}_{1,1,\lfloor \log_2(\frac{w}{2})\rfloor,*} \rrbracket \leftarrow \llbracket \mathbf{T}_{0,1,\lfloor \log_2(\frac{w}{2})\rfloor,*} \rrbracket \oplus \llbracket \mathbf{T}_{1,1,\lfloor \log_2(\frac{w}{2})\rfloor,*} \rrbracket$ ▷ Coefficient-wise XOR
9: **for** $l \leftarrow 0$ to 1 **do**
10: $\quad \llbracket sk \rrbracket \leftarrow \mathsf{refresh}(\llbracket sk \rrbracket)$
11: $\quad \llbracket \mathbf{s}1 \rrbracket \leftarrow \mathsf{SecSyndrome}(\llbracket \mathbf{h}_0 \rrbracket, \llbracket \mathbf{h}_1 \rrbracket, \llbracket \mathbf{e}_0 \rrbracket, \llbracket \mathbf{e}_1 \rrbracket, \llbracket \mathbf{s} \rrbracket)$ ▷ Algorithm 3.20
12: $\quad \llbracket \mathbf{C} \rrbracket \leftarrow \mathsf{SecCounter}(\llbracket \mathbf{s}1 \rrbracket, \frac{\frac{w}{2}+1}{2}, \llbracket \mathbf{h}_0^\circ \rrbracket, \llbracket \mathbf{h}_1^\circ \rrbracket)$ ▷ Algorithm 3.23
13: $\quad \llbracket \mathbf{v}_0 \rrbracket \leftarrow \mathsf{sec}_{\&}(\neg \llbracket \mathbf{C}_{0,\lfloor \log_2(\frac{w}{2})\rfloor,*} \rrbracket, \llbracket \mathbf{T}_{l,0,\lfloor \log_2(\frac{w}{2})\rfloor,*} \rrbracket)$ ▷ Coefficient-wise $\mathsf{sec}_{\&}$
14: $\quad \llbracket \mathbf{v}_1 \rrbracket \leftarrow \mathsf{sec}_{\&}(\neg \llbracket \mathbf{C}_{1,\lfloor \log_2(\frac{w}{2})\rfloor,*} \rrbracket, \llbracket \mathbf{T}_{l,1,\lfloor \log_2(\frac{w}{2})\rfloor,*} \rrbracket)$ ▷ Coefficient-wise $\mathsf{sec}_{\&}$
15: $\quad \llbracket \mathbf{e}_0 \rrbracket, \llbracket \mathbf{e}_1 \rrbracket \leftarrow \mathsf{refresh}(\llbracket \mathbf{e}_0 \rrbracket, \llbracket \mathbf{e}_1 \rrbracket)$
16: $\quad \llbracket \mathbf{e}_0 \rrbracket \leftarrow \llbracket \mathbf{e}_0 \rrbracket \oplus \llbracket \mathbf{v}_0 \rrbracket$ ▷ Coefficient-wise XOR
17: $\quad \llbracket \mathbf{e}_1 \rrbracket \leftarrow \llbracket \mathbf{e}_1 \rrbracket \oplus \llbracket \mathbf{v}_1 \rrbracket$ ▷ Coefficient-wise XOR
18: **return** $\llbracket \mathbf{e}_0 \rrbracket, \llbracket \mathbf{e}_1 \rrbracket$

Theorem 3.22. *The grey zone algorithm 3.24 is* d-NI.

Proof. We define a particular gadget called "block" for lines 11 to 17.

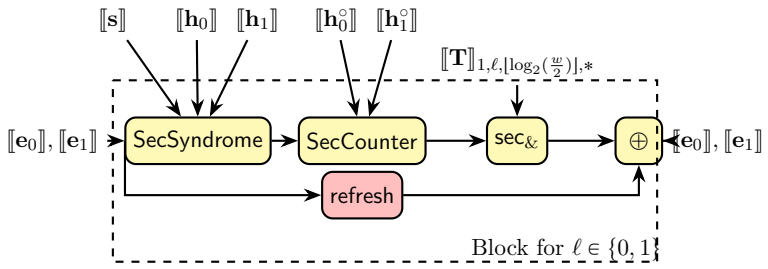

Figure 3.3: Structure of one block

The overall details of the dependencies are presented in Figures 3.3 and 3.4. As illustrated in Figure 3.3, the block gadget is d-NI. Indeed, the only dependency loop is broken by a d-SNI refresh algorithm. Let us consider the full algorithm. Let us assume that an attacker has access to $\delta \leqslant d$ observations on this gadget. Then, we want to

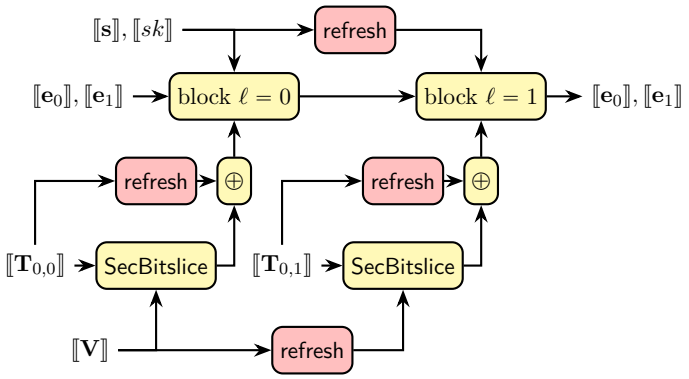

Figure 3.4: Structure of the grey zone gadget

prove that all these δ observations can be perfectly simulated with at most δ shares of $[\![sk]\!], [\![\mathbf{T}_0]\!], [\![\mathbf{e}_0]\!], [\![\mathbf{e}_1]\!], [\![\mathbf{s}]\!]$ and $[\![\mathbf{V}]\!]$ (note that the last one can be omitted as it is derived from a public parameter). To fix notations, let us consider the following distribution of the attacker's δ observations:

- δ_1 during the bitslice of Line 3
- δ_2 during the refreshing of $\mathbf{V}$
- δ_3 during the bitslice of Line 5
- δ_4 during the refresh of $\mathbf{T}_0$ (splitted in two sub-gadgets in the figure)
- δ_5 during the $\oplus$ in Line 8,
- δ_6 during the $\oplus$ in Line 7,
- δ_7 during the refreshing of sk and $\mathbf{s}$,
- δ_8 in the block with $\ell = 0$,
- δ_9 in the block with $\ell = 1$

By definition of the d-probing model, we have $\sum_{j=1}^{9} \delta_i \leqslant \delta \leqslant d$. All the gadgets are proved d-NI and the refresh gadgets are d-SNI. We skip the progressive part of the proof and directly claim that all the observations that are made during the execution of the gadget can be perfectly simulated with

- $\delta_2 + \delta_9 + \delta_8 + \delta_5 + \delta_1 + \delta_7$ shares of $[\![\mathbf{V}]\!]$
- $\delta_9 + \delta_8 + \delta_7$ shares of $[\![sk]\!]$ and $[\![\mathbf{s}]\!]$ (each)
- $\delta_9 + \delta_8 + \delta_5 + \delta_4 + \delta_1$ shares of $\mathbf{T}_{0,0}$,
- $\delta_9 + \delta_6 + \delta_3 + \delta_4$ shares of $\mathbf{T}_{0,1}$,
- $\delta_9 + \delta_8$ shares of $[\![\mathbf{e}_0]\!]$ and $[\![\mathbf{e}_1]\!]$ (each).

This can be verified with the help of the figure. All these numbers of shares are smaller to $\delta \leqslant d$ which concludes the proof. □

Algorithm 3.25 BGF decoder

Input: $sk = \left(\llbracket \mathbf{h}_0 \in \mathbb{F}_2^r \rrbracket, \llbracket \mathbf{h}_1 \in \mathbb{F}_2^r \rrbracket, \llbracket \mathbf{h}_0^\circ \in \mathbb{Z}_r^{\frac{w}{2}} \rrbracket, \llbracket \mathbf{h}_1^\circ \in \mathbb{Z}_r^{\frac{w}{2}} \rrbracket\right)$, $\llbracket \mathbf{c}_0 \in \mathbb{F}_2^r \rrbracket$

Output: $\llbracket \mathbf{e}_0 \in \mathbb{F}_2^r \rrbracket$, $\llbracket \mathbf{e}_1 \in \mathbb{F}_2^r \rrbracket$ such that $(\mathbf{c}_0 + \mathbf{e}_0) \cdot \mathbf{h}_0 = 0$

1: $\llbracket \mathbf{e}_0 \rrbracket \leftarrow$ vector_zero_masking()
2: $\llbracket \mathbf{e}_1 \rrbracket \leftarrow$ vector_zero_masking()
3: $\llbracket \mathbf{s} \rrbracket \leftarrow$ SecKaratsuba($\llbracket \mathbf{c}_0 \rrbracket, \llbracket \mathbf{h}_0 \rrbracket$) ▷ Algorithm 3.9
4: $\llbracket \mathbf{h}_0 \rrbracket \leftarrow$ refresh($\llbracket \mathbf{h}_0 \rrbracket$)
5: **for** $i \leftarrow 0$ to Nbr_Iter -1 **do**
6: $\quad \llbracket \mathbf{s1} \rrbracket \leftarrow$ SecSyndrome($\llbracket \mathbf{h}_0 \rrbracket, \llbracket \mathbf{h}_1 \rrbracket, \llbracket \mathbf{e}_0 \rrbracket, \llbracket \mathbf{e}_1 \rrbracket, \llbracket \mathbf{s} \rrbracket$) ▷ Algorithm 3.20
7: $\quad \llbracket T \rrbracket \leftarrow$ SecThreshold($\llbracket \mathbf{s1} \rrbracket$) ▷ Algorithm 3.22
8: $\quad \llbracket \mathbf{s1} \rrbracket \leftarrow$ refresh($\llbracket \mathbf{s1} \rrbracket$)
9: $\quad \llbracket \mathbf{C} \rrbracket \leftarrow$ SecCounter($\llbracket \mathbf{s1} \rrbracket, \llbracket T \rrbracket, \llbracket \mathbf{h}_0^\circ \rrbracket, \llbracket \mathbf{h}_1^\circ \rrbracket$) ▷ Algorithm 3.23
10: $\quad \llbracket \mathbf{e}_0 \rrbracket \leftarrow$ refresh($\llbracket \mathbf{e}_0 \rrbracket$); $\llbracket \mathbf{e}_1 \rrbracket \leftarrow$ refresh($\llbracket \mathbf{e}_1 \rrbracket$)
11: $\quad$ ▷ $\llbracket \mathbf{C}_{0,\lfloor \log_2(\frac{w}{2}) \rfloor,*} \rrbracket$ and $\llbracket \mathbf{C}_{1,\lfloor \log_2(\frac{w}{2}) \rfloor,*} \rrbracket$ are the sign bit of the counters minus the threshold
12: $\quad \llbracket \mathbf{e}_0 \rrbracket \leftarrow \neg((\llbracket \mathbf{e}_0 \rrbracket) \oplus (\llbracket \mathbf{C}_{0,\lfloor \log_2(\frac{w}{2}) \rfloor,*} \rrbracket))$ ▷ Coefficient-wise XOR
13: $\quad \llbracket \mathbf{e}_1 \rrbracket \leftarrow \neg((\llbracket \mathbf{e}_1 \rrbracket) \oplus (\llbracket \mathbf{C}_{1,\lfloor \log_2(\frac{w}{2}) \rfloor,*} \rrbracket))$ ▷ Coefficient-wise XOR
14: $\quad$ **if** $i = 0$ **then**
15: $\quad\quad \llbracket \mathbf{s} \rrbracket \leftarrow$ refresh($\llbracket \mathbf{s} \rrbracket$)
16: $\quad\quad \llbracket \mathbf{e}_0 \rrbracket, \llbracket \mathbf{e}_1 \rrbracket \leftarrow$ SecGreyZone($\llbracket \mathbf{C} \rrbracket, \llbracket \mathbf{h}_0 \rrbracket, \llbracket \mathbf{h}_1 \rrbracket, \llbracket \mathbf{h}_0^\circ \rrbracket, \llbracket \mathbf{h}_1^\circ \rrbracket, \llbracket \mathbf{e}_0 \rrbracket, \llbracket \mathbf{e}_1 \rrbracket, \llbracket \mathbf{s} \rrbracket$) ▷ Algorithm 3.24
17: $\quad \llbracket sk \rrbracket \leftarrow$ refresh($\llbracket sk \rrbracket$)
18: $\quad \llbracket \mathbf{s} \rrbracket \leftarrow$ refresh($\llbracket \mathbf{s} \rrbracket$)
19: **return** $(\llbracket \mathbf{e}_0 \rrbracket, \llbracket \mathbf{e}_1 \rrbracket)$

Theorem 3.23. *The BGF decoder algorithm 3.25 is d-NI.*

Proof. We represent the whole decoding algorithm in Figures 3.5 and 3.6. To avoid complex graphs, the content of an iteration for $i \neq 0$ can be proved separately (if $i \neq 0$, there is no application of the SecGreyZone algorithm, in Lines 14 to 16).

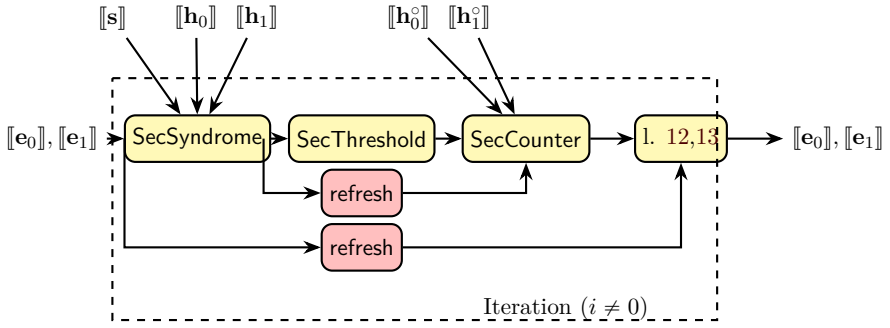

Figure 3.5: Structure of an iteration

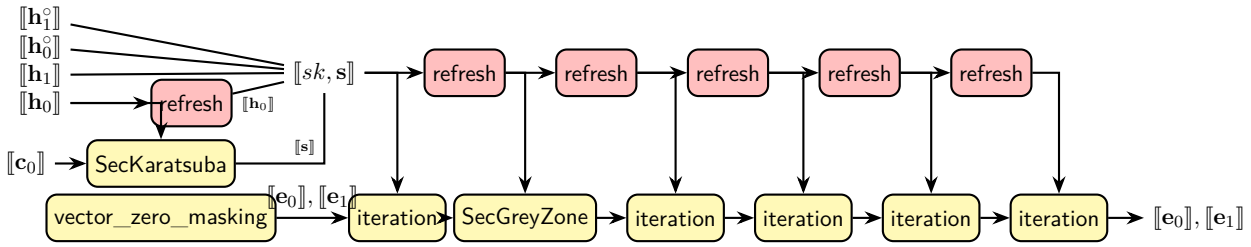

Figure 3.6: Structure of the BGF decoder

Let us first look at one iteration with $i \neq 0$. Let us assume that it is a gadget with inputs $[\![\mathbf{e}_0]\!], [\![\mathbf{e}_1]\!]$, $[\![sk]\!]$ and $[\![\mathbf{s}]\!]$. And we assume that this iteration's output is a modified version of $[\![\mathbf{e}_0]\!], [\![\mathbf{e}_1]\!]$. Let us assume that an attacker has access to $\delta \leqslant d$ observations on this sub-gadget. Thus, we want to prove that all these δ observations can be perfectly simulated with at most δ shares of $[\![sk]\!], [\![\mathbf{s}]\!], [\![\mathbf{e}_0]\!]$ and $[\![\mathbf{e}_1]\!]$. To fix notations, let us consider the following distribution of the attacker's δ observations:

- δ_6 on Lines 12 and 13,
- δ_5 during the SecCounter computation,
- δ_4 during the SecThreshold computation,
- δ_3 during the SecSyndrome computation,
- δ_2 when $[\![\mathbf{s}1]\!]$ si refreshed,
- δ_1 when $[\![\mathbf{e}_0]\!]$ and $[\![\mathbf{e}_1]\!]$ are refreshed.

By definition of the d-probing model, we have $\sum_{j=1}^{6} \delta_i \leqslant \delta \leqslant d$.
Since Lines 12 and 13 are $\mathbb{F}_2$-linear operations performed share by share, this computation verifies the d-NI property. In addition, all the gadgets are either d-NI or d-SNI as developed earlier. Finally, all the observations performed during this iteration can be perfectly simulated with at most $\sum_{j=1}^{6} \delta_i$ shares of $[\![\mathbf{e}_0]\!]$, the same amount for $[\![\mathbf{e}_1]\!]$, $\delta_6 + \delta_5$ shares of $\mathbf{h}_0^\circ$, the same for $\mathbf{h}_0^\circ$, $\sum_{j=2}^{6} \delta_i$ shares of $\mathbf{h}_0$ and finally the same for $\mathbf{h}_1$.

In the end, we have proved that all the probes can be perfectly simulated with at most $\delta \leqslant d$ shares of $[\![sk]\!], [\![\mathbf{s}]\!], [\![\mathbf{e}_0]\!]$ and $[\![\mathbf{e}_1]\!]$.

Now let us analyze the complete construction in Figure 3.6. The same reasoning applies. Let us assume that an attacker has access to $\delta \leqslant d$ observations on this algorithm. We consider the following distribution of the attacker's δ observations:

- $\delta_{\text{iter,i}}$ on each $i-th$ iteration,
- $\delta_{\text{SecGreyZone}}$ on the SecGreyZone computation,
- $\delta_{\text{ref,i}}$ on the $i-th$ refresh of the secret key and the syndrome,
- $\delta_{\text{vector\_zero\_masking}}$ on the vector_zero_masking computation,
- $\delta_{\text{SecKaratsuba}}$ on the computation of the syndrome,
- δ_{ref} on the very first refresh.

By definition, $\sum_{i=0}^{Nbr\_Iter-1}(\delta_{\text{iter,i}}+\delta_{\text{ref,i}})+\delta_{\mathsf{SecGreyZone}}+\delta_{\mathsf{vector\_zero\_masking}}+\delta_{\mathsf{SecKaratsuba}}+\delta_{\text{ref}} \leqslant \delta \leqslant d$.
All the gadgets are proved d-NI and the refresh gadgets are d-SNI. All the probes performed after the first iteration (including the grey zone, the key refresh and the other following iterations), can be perfectly simulated with at most $\sum_{i=0}^{Nbr\_Iter-1}(\delta_{\text{iter,i}}+\delta_{\text{ref,i}})+\delta_{\mathsf{SecGreyZone}}$ shares of $[\![sk]\!], [\![\mathbf{s}]\!], [\![\mathbf{e}_0]\!]$ and $[\![\mathbf{e}_1]\!]$. Next, we use the probing security of the refresh, SecKaratsuba and vector_zero_masking. All the probes performed during the full decoding algorithm can be perfectly simulated with at most $\sum_{i=0}^{Nbr\_Iter-1}(\delta_{\text{iter,i}}+\delta_{\text{ref,i}})+\delta_{\mathsf{SecGreyZone}}+\delta_{\mathsf{SecKaratsuba}}+\delta_{\text{ref}}$ shares of $[\![\mathbf{c}_0]\!]$, the same for $[\![\mathbf{h}_0]\!]$ and $\sum_{i=0}^{Nbr\_Iter-1}(\delta_{\text{iter,i}}+\delta_{ref,i})+\delta_{\mathsf{SecGreyZone}}$ for the rest of the secret key. All these numbers are smaller than to $\delta \leqslant d$ which concludes the proof. □

3.3.3.5 Decapsulation algorithm

Decapsulation consists of first decoding the ciphertext and second checking that it is correct. We propose below a fully masked version of the decapsulation for completeness in Algorithm 3.26.

Algorithm 3.26 Decapsulation

Input: $[\![\mathbf{c} \in \mathbb{F}_2^{r+\ell}]\!]$, $[\![\vec{\sigma} \in \mathbb{F}_2^{\ell}]\!]$
Output: $[\![\mathbf{k} \in \mathbb{F}_2^{\ell}]\!]$

1: $[\![\mathbf{e}']\!] \leftarrow \mathsf{SecBGF}([\![\mathbf{h}_0]\!], [\![\mathbf{h}_1]\!], [\![\mathbf{h}_0^{\circ}]\!], [\![\mathbf{h}_1^{\circ}]\!], \mathsf{subvector}([\![\mathbf{c}]\!], 0, r-1))$ ▷ Algorithm 3.25
2: $[\![\mathbf{m}']\!] \leftarrow \mathsf{L}([\![\mathbf{e}']\!])$ ▷ see Section 3.3.2
3: $[\![\mathbf{m}']\!] \leftarrow [\![\mathbf{m}']\!] \oplus \mathsf{subvector}([\![\mathbf{c}]\!], r, r+\ell)$ ▷ see Section 3.1, coefficient-wise XOR
4: $([\![\mathbf{e}_0]\!], [\![\mathbf{e}_1]\!]) \leftarrow \mathsf{SecErrorGen}([\![\mathbf{m}']\!])$ ▷ Algorithm 3.18
5: $[\![\mathbf{m}']\!] \leftarrow \mathsf{refresh}([\![\mathbf{m}']\!])$
6: $[\![v]\!] \leftarrow 1$ ▷ Masked value of 1
7: **for** $i \leftarrow 0$ to 1 **do**
8: **for** $j \leftarrow 0$ to $r-1$ **do**
9: $[\![t]\!] \leftarrow \mathsf{sec}_{=}([\![\mathbf{e}_{i,j}]\!], [\![\mathbf{e}'_{i,j}]\!])$ ▷ see Section 3.2.1.1
10: $[\![t]\!]_0 \leftarrow [\![t]\!]_0 \oplus 1$
11: $[\![v]\!] \leftarrow \mathsf{sec}_{\&}([\![v]\!], [\![t]\!])$
12: $[\![\mathbf{t}]\!] \leftarrow \mathsf{K}([\![\mathbf{m}']\!], [\![\mathbf{c}]\!])$ ▷ Section 3.3.3.5
13: $[\![\mathbf{t}1]\!] \leftarrow \mathsf{K}([\![\vec{\sigma}]\!], [\![\mathbf{c}]\!])$
14: $[\![\mathbf{k}]\!] \leftarrow \mathsf{sec}_{\mathsf{if}}([\![\mathbf{t}]\!], [\![\mathbf{t}1]\!], [\![v]\!])$ ▷ Coefficient-wise $\mathsf{sec}_{\mathsf{if}}$
15: **return** $[\![\mathbf{k}]\!]$

Algorithm 3.26 uses d-NI gadgets, and the only variable that is used twice without modification is $\mathbf{m}'$. However, the dependancy loop is broken by the d-SNI refresh. Thus, we introduce the following theorem.

Theorem 3.24. *The decapsulation algorithm 3.26 is d-NI.*

3.4 Benchmark & pratical testing

All the gadgets introduced have been implemented in complete C-code. Side-channel attacks are highly dependent on the platform on which the code is executed and it is true that assembly codes are always the best pratical solution. However, C-code seems the best option to provide a multi-platform proof of concept. This code could be reused for future analysis and optimizations. You can find the full code, publicly available for code-checking and reproducibility, on Github[1].

Sparse vs dense representation Since most of the computations are polynomial operations performed on sparse objects, let us recall that we had two available options: the fully-dense implementation and the hybrid-sparse-dense one (see Chapter 3). In the first case, we see the polynomials as dense (with a conversion of the keys during the SecKeyGen) and we use Karatsuba for the majority of the calculations. In the other case, since we can represent a number of polynomials in sparse representation, we use $\mathsf{SecMult}_{\text{sparsedense}}$ as much as possible. In the algorithms presented, we have focused on the "fully-dense implementation", but we have also implemented the "hybrid sparse-dense" version, which consists of using sparse representations and $\mathsf{SecMult}_{\text{sparsedense}}$ instead of Karatsuba where possible. You will find this in the Table 3.2, under the "sparse" and "dense" labels.

As presented in Figure 3.7, our benchmarks show that while both approaches seem similar for one or two shares, a fully dense approach is indeed more relevant for higher orders.

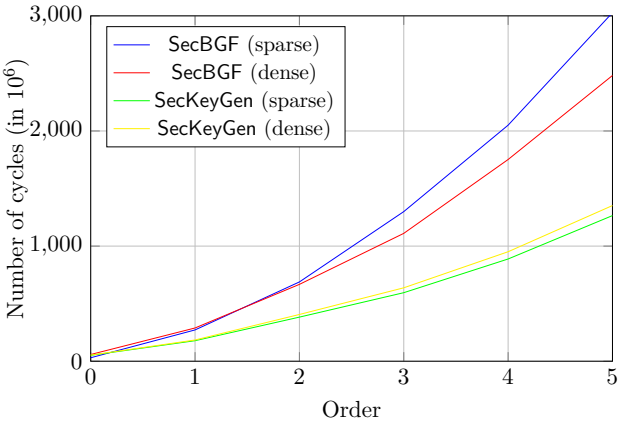

Figure 3.7: Sparse vs dense, SecBGF and SecKeyGen

One can conclude from our work that for the moment (except with potentially upcoming new optimizations), the dense representation seems more relevant. We will

[1] https://github.com/loicdemange/masked_BIKE_code

therefore keep the dense representation for the rest of the benchmarks, as it scales better when the order exceeds or equals 2.

3.4.1 Detailed benchmarks

The code was benchmarked on an i7-4710MQ running at 2.5Ghz, 8GB of RAM, and compiling with gcc 12.2.0 -O3 flag. The given performances are obtained for NIST security level 1 ($r = 12323$). Identical experiments can provide data for the other security levels. Multiple benchmarks were performed and the results are listed in Table 3.2. We can notice that the performance of the gadgets depends on the performance of the multiplicative gadgets.

Table 3.2: Scaling benchmarks on particular gadgets, i7-4710MQ 2.5Ghz gcc 12.2.0 -03, NIST Level 1, median results on 200 executions

	Order 0	1	2	3	4	5
SecBGF(Alg. 3.25) (sparse)	1	×9	×22.8	×43	×67.9	×100.2
SecBGF(Alg. 3.25) (dense)	1	×4.9	×11.4	×19	×30	×42.4
SecKeyGen (Alg. 3.17) (dense)	1	×3.5	×7.7	×12.1	×18	×25.6
SecErrorGen (Alg. 3.18)	1	×8.6	×22.3	×40.8	×66	×96.2
SecGreyZone(Alg. 3.24) (sparse)	1	×9	×23.9	×41.9	×67.4	×98.3
SecGreyZone(Alg. 3.24) (dense)	1	×4.8	×11.2	×18.8	×29.2	×42
SecFisherYates(Alg. 3.14)	1	×9.5	×18.5	×29.7	×45.7	×66
SecInversion(Alg. 3.16)	1	×3.5	×7.2	×11	×16.4	×23.6
SecSyndrome(Alg. 3.20) (sparse)	1	×8	×21.3	×42.3	×63.7	×93
SecSyndrome(Alg. 3.20) (dense)	1	×3.3	×7.3	×10.8	×15.7	×22.2
SecThreshold(Alg. 3.22)	1	×8.6	×12.9	×19.1	×30.5	×43.1
SecCounter(Alg. 3.23)	1	×9.4	×23.1	×42.1	×67.3	×97
SecKaratsuba(Alg. 3.9)	1	×3.4	×7.4	×11.1	×16.7	×23.4
$\text{SecMult}_{\text{sparsedense}}$(Alg. 3.15)	1	×8.2	×21.3	×40	×64.8	×95.6

Bottlenecks The sparse-dense multiplication seems to be the biggest bottleneck of our implementation. An optimization of this gadget could lead to big improvements of the complete scheme's performance. There is also room for optimization in Karatsuba, which, although its scaling looks good, is called a large number of times in most BIKE sections. One idea to improve these gadgets could be to optimize the last recursion calculation of Karatsuba. In the unmasked implementation, specific instructions are used, while in our masked implementation, only a naive multiplication is applied. The problem is that most known optimized techniques require arithmetic operations, thus, a masked form would require a mask conversion. Given the complexity of such conversions, this approach may end up to be equivalent to our original naive technique. In the end,

future work is still necessary to innovate and find new optimizations on this instruction.

Similarly, the cyclic shift is performed here directly, while the reference implementation stores the polynomials in duplicate (contiguously) and just has to change its "window" to perform the shift. We could not see any way to keep this advantage in a masked form. This also explains why there is such a difference in performance between the reference implementation and this implementation when the order equals 0.

General performance for masked BIKE (fully-dense) The performances and scaling for the scheme are detailed in Table 3.3 and Figure 3.8.

Table 3.3: Scaling benchmarks on BIKE, i7-4710MQ 2.5Ghz gcc 12.2.0 -03, NIST Level 1, median results on 100 executions, in million of cycles

	Order 0	1	2	3	4	5
SecKeyGen (RNG off)	55	162	351	477	640	853
SecKeyGen (RNG on)	55	188	409	635	980	1 330
Scaling SecKeyGen (RNG off)	**1**	**×3**	**×6.4**	**×8.7**	**×11.7**	**×15.5**
Scaling SecKeyGen (RNG on)	**1**	**×3.4**	**×7.4**	**×11.5**	**×17.9**	**×24.2**
Encaps (RNG off)	5	24	53	84	120	170
Encaps (RNG on)	5	29	71	122	190	278
Scaling Encaps (RNG off)	**1**	**×4.8**	**×10.6**	**×16.8**	**×24**	**×34**
Scaling Encaps (RNG on)	**1**	**×5.8**	**×14.2**	**×24.4**	**×38**	**×55.6**
Decaps (RNG off)	63	262	559	842	1 220	1 652
Decaps (RNG on)	63	329	723	1 211	1 873	2 693
Scaling Decaps (RNG off)	**1**	**×4.1**	**×8.9**	**×13.4**	**×19.4**	**×26.2**
Scaling Decaps (RNG on)	**1**	**×5.2**	**×11.5**	**×19.2**	**×29.7**	**×42.7**

Remark 3.5. RNG off refers to returning 0 instead of drawing a random integer. This allows to measure the cost of the number of calls to the RNG, relative to the performance of the implementation.

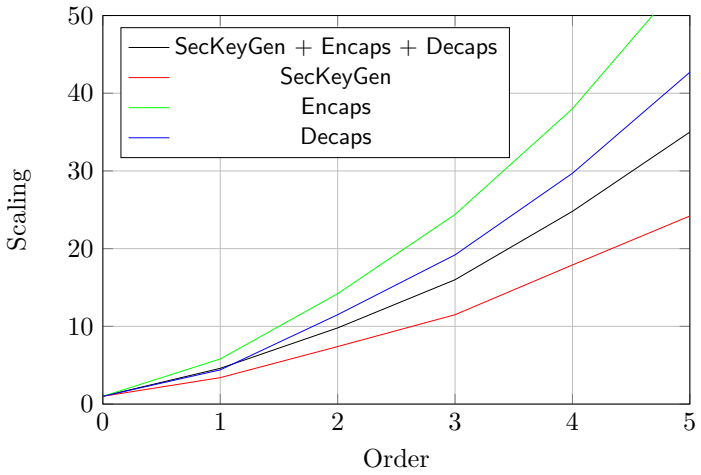

Figure 3.8: The scaling of masked BIKE (with RNG on)

We can see that the performance of masked BIKE as a function of the order is slightly above quadratic. This unoptimized implementation is still encouraging as it leaves the door open for many possible scaling improvements.
In fact, there are still a lot of possible optimizations, in particular on the cyclic shift and on the naive polynomial multiplication. Once optimized, the scaling will probably be improved, especially since there is no Boolean arithmetic conversion within the masked scheme.

Chapter 4
Conclusion & prospect

Boolean masking was initially developed for symmetric schemes. Recent research extended these techniques for asymmetric (especially lattice-based) schemes with arithmetic masking. Here, we propose BIKE masked from end-to-end. We have adapted, optimized, created, assembled and proven a large number of gadgets to perform the full range of operations required by the scheme, despite the scant literature on the subject. We chose Boolean masking, as BIKE manipulates binary objects, and explored two ways of performing the computations, based on the fact that polynomials in BIKE are sparse: fully-dense and hybrid-sparse-dense. This has led to a full implementation of the scheme in C, benchmarking the most essential gadgets as well as the three main steps of the scheme (key generation, encapsulation and decapsulation).

For future work, there are several possible directions. Firstly, we did not have the opportunity to formally verify the absence of leaks, with tools such as Test Vector Leakage Assessment (TVLA). For this, we need a set of representative consumption traces and to apply statistical tools to them. However, the lack of time and the number of cycles of the algorithm made it difficult to implement. We can therefore envisage submitting this implementation to attacks to confirm its resistance.

Secondly, we can look further on the performance aspect. In fact, it is possible to highly optimize the performance of our implementation by simply optimizing two important basic gadgets: the naive multiplication (in the last level of the Karatsuba recursion) and the cyclic shift. As outlined above, these gadgets are the bottleneck of our implementation. Thus, the impact on the performance to be very high. The relevance of avoiding mask conversions may also be questioned if such conversions help to gain orders of magnitude in the performance; even though we do not currently believe that conversions would significantly help here. In addition, we think that further optimization could impact the difference between the sparse version and the dense version.
Generally speaking, since the algorithm is broken down into gadgets, it is perfectly possible to swap some of them with more efficient alternatives, as long as you keep the same input-output and maintain a proof of composition. For example, the refresh gadget can be replaced by its quasi-linear alternative [Pin+23].

Also, attacking unprotected implementations with side-channel measurements is often not

the best choice to evaluate practical security. But, until now, no masked implementation of BIKE and other code-based schemes were available. This masked implementation is openly accessible and can serve as target for elaborate high order side-channel attacks.

Finally, as BIKE is the first QC-MDPC-based cryptographic scheme to be end-to-end masked, this work and the gadgets can be reused to mask other schemes, such as HQC [Agu+22].

Bibliography

[77] *Data Encryption Standard*. National Bureau of Standards, NBS FIPS PUB 46, U.S. Department of Commerce. Jan. 1977 (cit. on p. 2).

[Agu+22] Carlos Aguilar Melchor, Nicolas Aragon, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Edoardo Persichetti, Gilles Zémor, Jurjen Bos, Arnaud Dion, Jerome Lacan, Jean-Marc Robert, and Pascal Veron. *HQC*. Tech. rep. available at https://csrc.nist.gov/Projects/post-quantum-cryptography/round-4-submissions. National Institute of Standards and Technology, 2022 (cit. on p. 54).

[Ale03] Michael Alekhnovich. "More on Average Case vs Approximation Complexity". In: *44th FOCS*. IEEE Computer Society Press, Oct. 2003, pp. 298–307 (cit. on p. 9).

[And97] Ross Anderson. "Advanced Encryption Standard (Discussion)". In: *FSE'97*. Ed. by Eli Biham. Vol. 1267. LNCS. Springer, Heidelberg, Jan. 1997, pp. 83–87 (cit. on p. 2).

[Ara+22] Nicolas Aragon, Paulo Barreto, Slim Bettaieb, Loic Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Phillipe Gaborit, Shay Gueron, Tim Guneysu, Carlos Aguilar Melchor, Rafael Misoczki, Edoardo Persichetti, Nicolas Sendrier, Jean-Pierre Tillich, Gilles Zémor, Valentin Vasseur, Santosh Ghosh, and Jan Richter-Brokmann. *BIKE*. Tech. rep. available at https://csrc.nist.gov/Projects/post-quantum-cryptography/round-4-submissions. National Institute of Standards and Technology, 2022 (cit. on pp. 9–11, 38).

[Azo+23] Melissa Azouaoui, Olivier Bronchain, Gaëtan Cassiers, Clément Hoffmann, Yulia Kuzovkova, Joost Renes, Tobias Schneider, Markus Schönauer, François-Xavier Standaert, and Christine van Vredendaal. "Protecting Dilithium against Leakage: Revisited Sensitivity Analysis and Improved Implementations". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2023.4 (Aug. 2023), pp. 58–79 (cit. on p. 14).

[Bar+16] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, Pierre-Yves Strub, and Rébecca Zucchini. "Strong Non-Interference and Type-Directed Higher-Order Masking". In: *ACM CCS 2016*. Ed. by Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel,

Andrew C. Myers, and Shai Halevi. ACM Press, Oct. 2016, pp. 116–129 (cit. on pp. 14, 15, 17).

[Bar+18] Gilles Barthe, Sonia Belaïd, Thomas Espitau, Pierre-Alain Fouque, Benjamin Grégoire, Mélissa Rossi, and Mehdi Tibouchi. "Masking the GLP Lattice-Based Signature Scheme at Any Order". In: *EUROCRYPT 2018, Part II*. Ed. by Jesper Buus Nielsen and Vincent Rijmen. Vol. 10821. LNCS. Springer, Heidelberg, Apr. 2018, pp. 354–384 (cit. on pp. 14, 19, 24).

[BC22] Olivier Bronchain and Gaëtan Cassiers. "Bitslicing Arithmetic/Boolean Masking Conversions for Fun and Profit: with Application to Lattice-Based KEMs". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2022.4 (Aug. 2022), pp. 553–588 (cit. on p. 19).

[Bin+19] Nina Bindel, Sedat Akleylek, Erdem Alkim, Paulo S. L. M. Barreto, Johannes Buchmann, Edward Eaton, Gus Gutoski, Juliane Kramer, Patrick Longa, Harun Polat, Jefferson E. Ricardini, and Gustavo Zanon. *qTESLA*. Tech. rep. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions. National Institute of Standards and Technology, 2019 (cit. on p. 14).

[BMT78] E. Berlekamp, R. McEliece, and H. van Tilborg. "On the inherent intractability of certain coding problems (Corresp.)" In: *IEEE Transactions on Information Theory* 24.3 (1978), pp. 384–386 (cit. on p. 9).

[BOG20] Mario Bischof, Tobias Oder, and Tim Güneysu. "Efficient Microcontroller Implementation of BIKE". In: *Innovative Security Solutions for Information Technology and Communications*. Ed. by Emil Simion and Rémi Géraud-Stewart. Cham: Springer International Publishing, 2020, pp. 34–49. ISBN: 978-3-030-41025-4 (cit. on p. 13).

[Bos+21] Joppe W. Bos, Marc Gourjon, Joost Renes, Tobias Schneider, and Christine van Vredendaal. "Masking Kyber: First- and Higher-Order Implementations". In: *IACR TCHES* 2021.4 (2021). https://tches.iacr.org/index.php/TCHES/article/view/9064, pp. 173–214. ISSN: 2569-2925 (cit. on p. 14).

[Bou+20] Fabrice Boudot, Pierrick Gaudry, Aurore Guillevic, Nadia Heninger, Emmanuel Thomé, and Paul Zimmermann. "Nouveaux records de factorisation et de calcul de logarithme discret". In: *Techniques de l'Ingénieur* (Dec. 2020), p. 17 (cit. on p. 3).

[CCK21] Ming-Shing Chen, Tung Chou, and Markus Krausz. "Optimizing BIKE for the Intel Haswell and ARM Cortex-M4". In: *IACR TCHES* 2021.3 (2021). https://tches.iacr.org/index.php/TCHES/article/view/8969, pp. 97–124. ISSN: 2569-2925 (cit. on p. 13).

[CGV14] Jean-Sébastien Coron, Johann Großschädl, and Praveen Kumar Vadnala. "Secure Conversion between Boolean and Arithmetic Masking of Any Order". In: *CHES 2014*. Ed. by Lejla Batina and Matthew Robshaw. Vol. 8731. LNCS. Springer, Heidelberg, Sept. 2014, pp. 188–205 (cit. on p. 23).

[Cha+99] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. "Towards Sound Approaches to Counteract Power-Analysis Attacks". In: *CRYPTO'99*. Ed. by Michael J. Wiener. Vol. 1666. LNCS. Springer, Heidelberg, Aug. 1999, pp. 398–412 (cit. on p. 15).

[Che+16] Cong Chen, Thomas Eisenbarth, Ingo von Maurich, and Rainer Steinwandt. "Masking Large Keys in Hardware: A Masked Implementation of McEliece". In: *SAC 2015*. Ed. by Orr Dunkelman and Liam Keliher. Vol. 9566. LNCS. Springer, Heidelberg, Aug. 2016, pp. 293–309 (cit. on p. 6).

[Che+22] Ming-Shing Chen, Tim Güneysu, Markus Krausz, and Jan Philipp Thoma. "Carry-Less to BIKE Faster". In: *ACNS 22*. Ed. by Giuseppe Ateniese and Daniele Venturi. Vol. 13269. LNCS. Springer, Heidelberg, June 2022, pp. 833–852 (cit. on p. 13).

[Che+23] Agathe Cheriere, Nicolas Aragon, Tania Richmond, and Benoît Gérard. *BIKE Key-Recovery: Combining Power Consumption Analysis and Information-Set Decoding*. 2023 (cit. on p. 13).

[Cho16] Tung Chou. "QcBits: Constant-Time Small-Key Code-Based Cryptography". In: *CHES 2016*. Ed. by Benedikt Gierlichs and Axel Y. Poschmann. Vol. 9813. LNCS. Springer, Heidelberg, Aug. 2016, pp. 280–300 (cit. on pp. 13, 31).

[Cor+14] Jean-Sébastien Coron, Emmanuel Prouff, Matthieu Rivain, and Thomas Roche. "Higher-Order Side Channel Security and Mask Refreshing". In: *FSE 2013*. Ed. by Shiho Moriai. Vol. 8424. LNCS. Springer, Heidelberg, Mar. 2014, pp. 410–424 (cit. on pp. 14, 15).

[Cor+15] Jean-Sébastien Coron, Johann Großschädl, Mehdi Tibouchi, and Praveen Kumar Vadnala. "Conversion from Arithmetic to Boolean Masking with Logarithmic Complexity". In: *FSE 2015*. Ed. by Gregor Leander. Vol. 9054. LNCS. Springer, Heidelberg, Mar. 2015, pp. 130–149 (cit. on pp. 19, 23).

[Cor14] Jean-Sébastien Coron. "Higher Order Masking of Look-Up Tables". In: *EUROCRYPT 2014*. Ed. by Phong Q. Nguyen and Elisabeth Oswald. Vol. 8441. LNCS. Springer, Heidelberg, May 2014, pp. 441–458 (cit. on pp. 18, 19).

[Cor17] Jean-Sébastien Coron. "High-Order Conversion from Boolean to Arithmetic Masking". In: *CHES 2017*. Ed. by Wieland Fischer and Naofumi Homma. Vol. 10529. LNCS. Springer, Heidelberg, Sept. 2017, pp. 93–114 (cit. on p. 23).

[CT19] Rodolfo Canto-Torres and Jean-Pierre Tillich. "Speeding up decoding a code with a non-trivial automorphism group up to an exponential factor". In: *2019 IEEE International Symposium on Information Theory (ISIT)*. 2019, pp. 1927–1931 (cit. on p. 10).

[DAn+20] Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, Frederik Vercauteren, Jose Maria Bermudo Mera, Michiel Van Beirendonck, and Andrea Basso. *SABER*. Tech. rep. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions. National Institute of Standards and Technology, 2020 (cit. on pp. 14, 38).

[DAn+22] Jan-Pieter D'Anvers, Daniel Heinz, Peter Pessl, Michiel Van Beirendonck, and Ingrid Verbauwhede. "Higher-Order Masked Ciphertext Comparison for Lattice-Based Cryptography". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2022.2 (Feb. 2022), pp. 115–139 (cit. on p. 26).

[DDF14] Alexandre Duc, Stefan Dziembowski, and Sebastian Faust. "Unifying Leakage Models: From Probing Attacks to Noisy Leakage". In: *EUROCRYPT 2014*. Ed. by Phong Q. Nguyen and Elisabeth Oswald. Vol. 8441. LNCS. Springer, Heidelberg, May 2014, pp. 423–440 (cit. on p. 15).

[DG19] Nir Drucker and Shay Gueron. "A toolbox for software optimization of QC-MDPC code-based cryptosystems". In: *Journal of Cryptographic Engineering* 9.4 (Nov. 2019), pp. 341–357 (cit. on p. 13).

[DGK20a] Nir Drucker, Shay Gueron, and Dusan Kostic. "Fast Polynomial Inversion for Post Quantum QC-MDPC Cryptography". In: *Cyber Security Cryptography and Machine Learning*. Ed. by Shlomi Dolev, Vladimir Kolesnikov, Sachin Lodha, and Gera Weiss. Cham: Springer International Publishing, 2020, pp. 110–127. ISBN: 978-3-030-49785-9 (cit. on p. 36).

[DGK20b] Nir Drucker, Shay Gueron, and Dusan Kostic. "QC-MDPC Decoders with Several Shades of Gray". In: *Post-Quantum Cryptography - 11th International Conference, PQCrypto 2020*. Ed. by Jintai Ding and Jean-Pierre Tillich. Springer, Heidelberg, 2020, pp. 35–50 (cit. on pp. 12, 13).

[DH76] Whitfield Diffie and Martin E. Hellman. "New Directions in Cryptography". In: *IEEE Transactions on Information Theory* 22.6 (1976), pp. 644–654 (cit. on p. 3).

[Dob+21] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. "Ascon v1.2: Lightweight Authenticated Encryption and Hashing". In: *Journal of Cryptology* 34.3 (July 2021), p. 33 (cit. on p. 2).

[DR24] Loïc Demange and Mélissa Rossi. "A provably masked implementation of BIKE Key Encapsulation Mechanism". In: *IACR Communications in Cryptology* 1.1 (Apr. 9, 2024). ISSN: 3006-5496 (cit. on pp. 6, 24).

[Dur64] Richard Durstenfeld. "Algorithm 235: Random permutation". In: *Communications of the ACM* 7 (1964), p. 420 (cit. on p. 34).

[Eat+18] Edward Eaton, Matthieu Lequesne, Alex Parent, and Nicolas Sendrier. "QC-MDPC: A Timing Attack and a CCA2 KEM". In: *Post-Quantum Cryptography - 9th International Conference, PQCrypto 2018*. Ed. by Tanja Lange and Rainer Steinwandt. Springer, Heidelberg, 2018, pp. 47–76 (cit. on p. 13).

[FMI99] M.P.C. Fossorier, M. Mihaljevic, and H. Imai. "Reduced complexity iterative decoding of low-density parity check codes based on belief propagation". In: *IEEE Transactions on Communications* 47.5 (1999), pp. 673–680 (cit. on p. 11).

[FO99] Eiichiro Fujisaki and Tatsuaki Okamoto. "Secure Integration of Asymmetric and Symmetric Encryption Schemes". In: *CRYPTO'99*. Ed. by Michael J. Wiener. Vol. 1666. LNCS. Springer, Heidelberg, Aug. 1999, pp. 537–554 (cit. on p. 9).

[GAB19] Antonio Guimarães, Diego F. Aranha, and Edson Borin. "Optimized implementation of QC-MDPC code-based cryptography". In: *Concurrency and Computation: Practice and Experience* 31.18 (2019), e5089 (cit. on p. 13).

[Gal62] R. Gallager. "Low-density parity-check codes". In: *IRE Transactions on Information Theory* 8.1 (1962), pp. 21–28 (cit. on p. 11).

[GJS16] Qian Guo, Thomas Johansson, and Paul Stankovski. "A Key Recovery Attack on MDPC with CCA Security Using Decoding Errors". In: *ASIACRYPT 2016, Part I*. Ed. by Jung Hee Cheon and Tsuyoshi Takagi. Vol. 10031. LNCS. Springer, Heidelberg, Dec. 2016, pp. 789–815 (cit. on pp. 11, 13).

[GR20] François Gérard and Mélissa Rossi. "An Efficient and Provable Masked Implementation of qTESLA". In: *Smart Card Research and Advanced Applications*. Springer International Publishing, Mar. 2020, pp. 74–91. ISBN: 978-3-030-42067-3 (cit. on pp. 6, 14).

[Guo+22] Qian Guo, Clemens Hlauschek, Thomas Johansson, Norman Lahr, Alexander Nilsson, and Robin Leander Schröder. "Don't Reject This: Key-Recovery Timing Attacks Due to Rejection-Sampling in HQC and BIKE". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2022.3 (June 2022), pp. 223–263 (cit. on p. 13).

[HHK17] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. "A Modular Analysis of the Fujisaki-Okamoto Transformation". In: *TCC 2017, Part I*. Ed. by Yael Kalai and Leonid Reyzin. Vol. 10677. LNCS. Springer, Heidelberg, Nov. 2017, pp. 341–371 (cit. on p. 9).

[ISW03] Yuval Ishai, Amit Sahai, and David Wagner. "Private Circuits: Securing Hardware against Probing Attacks". In: *CRYPTO 2003*. Ed. by Dan Boneh. Vol. 2729. LNCS. Springer, Heidelberg, Aug. 2003, pp. 463–481 (cit. on pp. 5, 14, 15, 24).

[Kra+22] Markus Krausz, Georg Land, Jan Richter-Brockmann, and Tim Güneysu. "Efficiently Masking Polynomial Inversion at Arbitrary Order". In: *Post-Quantum Cryptography*. Ed. by Jung Hee Cheon and Thomas Johansson. Cham: Springer International Publishing, 2022, pp. 309–326. ISBN: 978-3-031-17234-2 (cit. on p. 14).

[Kra+23] Markus Krausz, Georg Land, Jan Richter-Brockmann, and Tim Güneysu. "A Holistic Approach Towards Side-Channel Secure Fixed-Weight Polynomial Sampling". In: *Public-Key Cryptography – PKC 2023*. Ed. by Alexandra Boldyreva and Vladimir Kolesnikov. Cham: Springer Nature Switzerland, 2023, pp. 94–124. ISBN: 978-3-031-31371-4 (cit. on pp. 14, 20, 23).

[Kun+22] Suparna Kundu, Jan-Pieter D'Anvers, Michiel Van Beirendonck, Angshuman Karmakar, and Ingrid Verbauwhede. "Higher-Order Masked Saber". In: *Security and Cryptography for Networks*. Ed. by Clemente Galdi and Stanislaw Jarecki. Cham: Springer International Publishing, 2022, pp. 93–116. ISBN: 978-3-031-14791-3 (cit. on pp. 14, 38).

[Lem19] Daniel Lemire. "Fast Random Integer Generation in an Interval". In: *ACM Transactions on Modeling and Computer Simulation* 29.1 (Jan. 2019), pp. 1–12 (cit. on p. 29).

[Lyu+22] Vadim Lyubashevsky, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Peter Schwabe, Gregor Seiler, Damien Stehlé, and Shi Bai. *CRYSTALS-DILITHIUM*. Tech. rep. available at https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022. National Institute of Standards and Technology, 2022 (cit. on p. 14).

[MG14] Ingo von Maurich and Tim Güneysu. "Towards Side-Channel Resistant Implementations of QC-MDPC McEliece Encryption on Constrained Devices". In: *Post-Quantum Cryptography - 6th International Workshop, PQCrypto 2014*. Ed. by Michele Mosca. Springer, Heidelberg, Oct. 2014, pp. 266–282 (cit. on p. 13).

[MHG16] Ingo von Maurich, Lukas Heberle, and Tim Güneysu. "IND-CCA Secure Hybrid Encryption from QC-MDPC Niederreiter". In: *Post-Quantum Cryptography - 7th International Workshop, PQCrypto 2016*. Ed. by Tsuyoshi Takagi. Springer, Heidelberg, 2016, pp. 1–17 (cit. on p. 13).

[MOG15] Ingo Von Maurich, Tobias Oder, and Tim Güneysu. "Implementing QC-MDPC McEliece Encryption". In: *ACM Trans. Embed. Comput. Syst.* 14.3 (Apr. 2015). ISSN: 1539-9087 (cit. on p. 13).

[MRS00] Chris Monico, Joachim Rosenthal, and Amin Shokrollahi A. "Using low density parity check codes in the McEliece cryptosystem". In: Sorrento, Italy, 2000, p. 215 (cit. on p. 8).

[Nie86] H. Niederreiter. "Knapsack-Type Cryptosystems and Algebraic Coding Theory". In: *Problems of Control and Information Theory* 15.2 (1986), pp. 159–166 (cit. on p. 9).

[Nil+21] Alexander Nilsson, Irina E. Bocharova, Boris D. Kudryashov, and Thomas Johansson. "A Weighted Bit Flipping Decoder for QC-MDPC-based Cryptosystems". In: *2021 IEEE International Symposium on Information Theory (ISIT)*. 2021, pp. 1266–1271 (cit. on p. 12).

[Pin+23] Rafaël del Pino, Thomas Prest, Mélissa Rossi, and Markku-Juhani O. Saarinen. "High-Order Masking of Lattice Signatures in Quasilinear Time". In: *2023 IEEE Symposium on Security and Privacy (SP)*. 2023, pp. 1168–1185 (cit. on pp. 6, 18, 53).

[Ros+17] Melissa Rossi, Mike Hamburg, Michael Hutter, and Mark E. Marson. "A Side-Channel Assisted Cryptanalytic Attack Against QcBits". In: *CHES 2017*. Ed. by Wieland Fischer and Naofumi Homma. Vol. 10529. LNCS. Springer, Heidelberg, Sept. 2017, pp. 3–23 (cit. on p. 13).

[RP10] Matthieu Rivain and Emmanuel Prouff. "Provably Secure Higher-Order Masking of AES". In: *CHES 2010*. Ed. by Stefan Mangard and François-Xavier Standaert. Vol. 6225. LNCS. Springer, Heidelberg, Aug. 2010, pp. 413–427 (cit. on pp. 14, 15).

[RSA78] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems". In: *Communications of the Association for Computing Machinery* 21.2 (1978), pp. 120–126 (cit. on p. 3).

[Sch+22] Peter Schwabe, Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Gregor Seiler, Damien Stehlé, and Jintai Ding. *CRYSTALS-KYBER*. Tech. rep. available at https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022. National Institute of Standards and Technology, 2022 (cit. on p. 14).

[Sen21] Nicolas Sendrier. *Secure Sampling of Constant-Weight Words ? Application to BIKE*. Cryptology ePrint Archive, Report 2021/1631. https://eprint.iacr.org/2021/1631. 2021 (cit. on pp. 13, 34).

[Sho94] Peter W. Shor. "Algorithms for Quantum Computation: Discrete Logarithms and Factoring". In: *35th FOCS*. IEEE Computer Society Press, Nov. 1994, pp. 124–134 (cit. on p. 3).

[Sim+19] Bo-Yeon Sim, Jihoon Kwon, Kyu Young Choi, Jihoon Cho, Aesun Park, and Dong-Guk Han. "Novel Side-Channel Attacks on Quasi-Cyclic Code-Based Cryptography". In: *IACR TCHES* 2019.4 (2019). https://tches.iacr.org/index.php/TCHES/article/view/8349, pp. 180–212. ISSN: 2569-2925 (cit. on p. 13).

[SV] Nicolas Sendrier and Valentin Vasseur. *Backflip: An improved qc-mdpc bitflipping decoder. CBC 2019, 2019* (cit. on p. 12).