IACR Communications in Cryptology
ISSN 3006-5496, Vol. 1, No. 1, 37 pages.

https://doi.org/10.62056/aesgvua5v

A provably masked implementation of BIKE Key Encapsulation Mechanism

Loïc Demange[a, 1,2] and Mélissa Rossi[3]

[1] Thales Gennevilliers, France
[2] Inria Paris, France
[3] ANSSI, France

Abstract. BIKE is a post-quantum key encapsulation mechanism (KEM) selected for the 4th round of the NIST's standardization campaign. It relies on the hardness of the syndrome decoding problem for quasi-cyclic codes and on the indistinguishability of the public key from a random element, and provides the most competitive performance among round 4 candidates, which makes it relevant for future real-world use cases. Analyzing its side-channel resistance has been highly encouraged by the community and several works have already outlined various side-channel weaknesses and proposed ad-hoc countermeasures. However, in contrast to the well-documented research line on masking lattice-based algorithms, the possibility of generically protecting code-based algorithms by masking has only been marginally studied in a 2016 paper by Chen et al. in SAC 2015 . At this stage of the standardization campaign, it is important to assess the possibility of fully masking BIKE scheme and the resulting cost in terms of performances.

In this work, we provide the first high-order masked implementation of a code-based algorithm. We had to tackle many issues such as finding proper ways to handle large sparse polynomials, masking the key-generation algorithm or keeping the benefit of the bitslicing. In this paper, we present all the gadgets necessary to provide a fully masked implementation of BIKE, we discuss our different implementation choices and we propose a full proof of masking in the Ishai Sahai and Wagner (Crypto 2003) model.

More practically, we also provide an open C-code masked implementation of the key-generation, encapsulation and decapsulation algorithms with extensive benchmarks. While the obtained performance is slower than existing masked lattice-based algorithms, we show that masking at order 1, 2, 3, 4 and 5 implies a performance penalty of x5.8, x14.2, x24.4, x38 and x55.6 compared to order 0 (unmasked and unoptimized BIKE). This scaling is encouraging and no Boolean to Arithmetic conversion has been used.

Keywords: BIKE · PQC · Side-Channel countermeasure · Provable high-order masking · d-probing model

1 Introduction

In response to the potential quantum threat, the NIST has initiated a standardization campaign in 2017 for defining new post-quantum algorithms. Different families of mathematical problems have received a lot of attention. Particularly, lattices and error-correcting codes stood out as interesting building blocks for post-quantum schemes. Six years after the first round of the campaign, three lattice-based schemes have been selected as future

E-mail: ldemange-research@etik.com (Loïc Demange), melissa.rossi@ssi.gouv.fr (Mélissa Rossi)

[a] This work has been partially supported by the French Agence Nationale de la Recherche through the France 2030 program under grant agreement No. ANR-22-PETQ-0008 PQ-TLS.

Received: 2024-01-09 Accepted: 2024-03-05

NIST post-quantum standards. But in parallel, the standardization campaign continues for other key-encapsulation schemes like code-based ones, because it is important to be able to have diverse schemes based on other structures.

BIKE [ABB$^+$22], a round 4 candidate for the NIST standardization process, is still under analysis by the research community. It relies on the hardness of the syndrome decoding problem for quasi-cyclic codes and on the indistinguishability of the public key from a random element. It is the most efficient and offers the lowest key sizes of all round 4 candidates. BIKE belongs to a line of research on Quasi-Cyclic Moderate Density Parity-Check (QC-MDPC) code-based schemes started by [MTSB13]. The advantage of QC-MDPC-based algorithms is the sparse structure of the underlying variables. Such codes allow for the use of iterative bit-flipping decoding algorithms (detailed later in the paper) as part of the decapsulation. The first implementations of QC-MDPC codes [vMG14, MOG15, vMHG16] were not constant time and vulnerable to time attacks. In 2016, Chou proposed a portable constant-time C implementation [Cho16]. Some implementation proposals have been made over the years [DG19, GAB19, BOG20], and another constant-time C implementation was introduced in 2020 [DGK20], 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 [CGKT22].

Existing side-channel attacks Timing vulnerabilities have been handled with priority in the previously stated implementations. However, the authors of [GHJ$^+$22] have highlighted the possibility of using timing information of the constant weight word sampler in the decapsulation in order to apply [GJS16]'s 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 lower down 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 [RHHM17]. Later, an improvement of the previous attack and a single-trace analysis exploiting leakage in the syndrome computation were provided in [SKC$^+$19]. Very recently, [CARG23] 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. Next, they combine such knowledge with classical information set decoding techniques to recover the full key.

Existing generic side-channel protections The current implementations of code-based schemes are claimed to be protected against timing and cache attacks, but they are never fully masked, i.e. masked from key generation to decapsulation. Masking is known as the most deployed countermeasure against physical attackers 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 $(\mathbf{x}_i)_{0\leq i\leq d}$ called "shares". The integer d is referred to as the masking order. In this paper, the only necessary type of masking is *Boolean masking*. In other words, a sensitive variable $\mathbf{x}$ is shared in $(\mathbf{x}_i)_{0\leq i\leq d}$ such that

$$\mathbf{x} = \mathbf{x}_0 \oplus \cdots \oplus \mathbf{x}_d. \tag{1}$$

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, CPRR14, BBD$^+$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, [BBE$^+$18] introduced a new security notion to justify unmasking certain intermediate steps. In [GR20], the authors proposed a masked implementation of the qTesla signature scheme [BAA$^+$19]. In [KDVB$^+$22], a masked Fuijsaki-Okamoto transform is introduced for a fully masked Saber KEM implementation [DKR$^+$20]. The NIST post-quantum finalists Crystals-Dilithium [LDK$^+$22] and Crystals-Kyber [SAB$^+$22] have also been masked in [ABC$^+$23] and [BGR$^+$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 [KLRBG22] propose a first-order masked inversion in multiplicative masking. Another recent work [KLRBG23] presents a way to mask BIKE's key generation with a fixed weight polynomial sampling technique and arithmetic to Boolean conversions.

Our contribution In this paper, 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 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, CGTV15], 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 [KLRBG23] 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 the future work 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 in this paper all elementary gadgets that are necessary to mask BIKE even if their design did not pose any particular issue. 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 [BBE+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 can be a first building block towards masked code-based cryptography and could lead to future analysis and new optimization.

Organization of the paper In Section 2, we introduce all the necessary background on masking, QC-MDPC codes and BIKE. In Section 3, we present our general masked construction along with its composition security proof. In Section 4, we detail the gadgets. For brevity, we only detail a few main gadgets and refer to Appendix A for the description of the remaining gadgets. Finally, in Section 5, we present our implementation and its benchmarks. We conclude with the future work in Section 6.

2 Preliminaries

2.1 Masking

A shared variable $(\mathbf{x}_i)_{0 \leq i \leq d}$ according to Eq. (1) will be denoted by $[\![\mathbf{x}]\!]$ for readability. Note that for a masking order d, there are $d + 1$ shares.

Definition 1 (d-probing Security or ISW security [ISW03])**.** An algorithm is d-probing secure iff 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, [CJRR99] 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 1, other intermediate security properties were introduced to ease the security proofs [RP10, CPRR14, BBD+16]. The focus can be placed on proving these properties on small parts of the algorithms, denoted gadgets.

Definition 2 (Gadget)**.** A gadget is a probabilistic algorithm that takes shared and unshared inputs values and returns shared and un-shared values.

These new security properties open the door for securely composing gadgets.

Definition 3 (Non interference [BBD+16])**.** A gadget is:

- d-non-interfering (d-NI) iff any set of at most d observations can be perfectly simulated from at most d shares of each input.
- d-strong non-interfering (d-SNI) iff 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.

Note that any linear gadget for $\oplus$ is immediately d-NI. Besides, one can check that d-SNI implies d-NI, which itself implies d-probing security. Hence, it suffices to reach d-NI for the key generation, encryption and decryption algorithms to achieve d-probing security.

In the same paper [BBD+16], a proposition was made to construct and compose with d-NI and d-SNI gadgets.

Proposition 1 ([BBD+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 $1 \leq i \leq n$, and $[\![a_i]\!]$ is not used anywhere else in the algorithm*

We are going to rely on these definitions and this proposition to create our gadgets.

2.2 Codes

In this paper, we will only introduce the relevant information for masking BIKE. Not many aspects of coding theory are needed for understanding our work.

Definition 4 (Binary linear codes)**.** A binary linear code $\mathcal{C}$ of length n and dimension r is a r-dimensional vector subspace of $\mathbb{F}_2^n$. Is it possible to represent it in two equivalent ways:

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

Definition 5 (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 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 6 (Quasi-circulant matrix)**.** A matrix is quasi-circulant if it is composed of circulant square blocks of size greater than 2.

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. Similarly to Remark 1, it is possible to represent quasi-circulant matrices as sets of polynomials.

Definition 7 (Quasi-cyclic code)**.** A binary code $\mathcal{C}$ is quasi-cyclic iff it admits a quasi-circulant generating matrix.

Definition 8 (QC-MDPC code)**.** Let n, r, w be integer parameters for length, dimension and minimum code weight. 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 constant row weight $w \approx \sqrt{n}$.

2.3 BIKE scheme

BIKE (*Bit Flipping Key Encapsulation*) [ABB$^+$22] is a key encapsulation scheme based on QC-MDPC (Quasi-Cyclic Moderate Density Parity-Check) codes as introduced in Definition 8.

More precisely, let r and w be integer parameters. BIKE relies on $[2r, r, w]$ QC-MDPC codes. 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 [Ale03, BMvT78]. We refer to [ABB$^+$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 2.4. 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

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

Table 1: BIKE's proposed parameters [ABB+22]

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

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 denote $a \overset{\$}{\leftarrow} \mathcal{B}$ when a is sampled uniformly at random from $\mathcal{B}$, and $\leftarrow$ is an assigment of value.

Algorithm 1 Keygen

```
Ensure: $((h_0, h_1), \sigma) \in \mathcal{H}_w \times \mathcal{M}$, $h \in \mathcal{R}$
$(h_0, h_1) \overset{\$}{\leftarrow} \mathcal{H}_w$ ▷ slight bias, see Section 3.2
$h \leftarrow h_1 h_0^{-1}$
$\sigma \overset{\$}{\leftarrow} \mathcal{M}$
return $((h_0, h_1, \sigma), h)$
```

Algorithm 2 Encaps

```
Require: $h \in \mathcal{R}$
Ensure: $K \in \mathcal{K}, c \in \mathcal{R} \times \mathcal{M}$
$m \overset{\$}{\leftarrow} \mathcal{M}$
$(e_0, e_1) \leftarrow \mathbf{H}(m)$
$c \leftarrow (e_0 + e_1 h, m \oplus \mathbf{L}(e_0, e_1))$
$K \leftarrow \mathbf{K}(m, c)$
return $(K, c)$
```

Algorithm 3 Decaps

```
Require: $(h_0, h_1, \sigma) \in \mathcal{H}_w \times \mathcal{M}$, $c = (c_0, c_1) \in \mathcal{R} \times \mathcal{M}$
Ensure: $K \in \mathcal{K}$
$e' \leftarrow \mathsf{decoder}(c_0 h_0, h_0, h_1)$                ▷ see Section 2.4
$m' \leftarrow c_1 \oplus \mathbf{L}(e')$
if $e' = \mathbf{H}(m')$ then
    $K \leftarrow \mathbf{K}(m', c)$
else
    $K \leftarrow \mathbf{K}(\sigma, c)$
end if
return $K$
```

Parameter setting As defined in the specifications, the parameters should satisfy several constraints. The block length r should be a prime number, and 2 should be primitive modulo r. The parameter w should be such that $w = 2d \approx \sqrt{n}$ with d being odd. In addition, the error weight should be such that $t \approx \sqrt{n}$. We present the instantiated parameters in Table 1.

In the following, we will not discuss hash functions any further (see Section 3.3 for more details on implementation). We will use the **H** notation to represent the code's parity matrix, where h_0 and h_1 are the polynomials describing its two circulating blocks.

2.4 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 matrices, decoding techniques usually rely on Bit-Flipping techniques originally introduced in [Gal62] for low density parity-check matrices.

Technically, the Bit Flipping algorithm is presented in Algorithm 4 and works as follows: over several iterations, we compute the syndrome $c\mathbf{H}^T$ where c is the ciphertext 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 pre-computed threshold (on the fly according to the weight of the syndrome), the position is flipped and the syndrome is recomputed. Let syndrome be the syndrome computation, counter the counter computation, and threshold the threshold computation function. We refer to [ABB+22] for details.

Algorithm 4 Bit Flipping algorithm

```
Require: $\mathbf{H}^T$ the sparse parity matrix of a $[2r, r, w]$ MDPC code
    $c \in \mathbb{F}_2^n$ a noisy codeword
Ensure: A codeword $c$, $c\mathbf{H}^T = 0$
$s \leftarrow \mathsf{syndrome}(c, \mathbf{H})$
while $|s| \neq 0$ do
    $T \leftarrow \mathsf{threshold}(|s|)$
    for $j \in \{1, ..., n\}$ do
        if $\mathsf{counter}(s, j, \mathbf{H}) \geq T$ then
            $c_j \leftarrow c_j \oplus 1$
        end if
    end for
    $s \leftarrow \mathsf{syndrome}(c, \mathbf{H})$
end while
return $c$
```

The authors of BIKE chose a refined Black-Gray-Flip (BGF) technique introduced in [DGK20]. 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 5. This decoder also has a fixed number of iterations (set at 5), to avoid timing attacks.

Algorithm 5 Black – Gray – Flip (BGF)

Parameters: r, w, t, $d = w/2$, $n = 2r$; Nbr_Iter, τ, `threshold` (see text for details)

```
Require: $s \in \mathbb{F}_2^r$, $\mathbf{H} \in \mathbb{F}_2^{r \times n}$
$e \leftarrow 0^n$
for $i = 1, \ldots,$Nbr_Iter do
    $T \leftarrow \mathsf{threshold}(|s + e\mathbf{H}^T|, i)$
    $e, black, grey \leftarrow \mathsf{BFIter}(s + e\mathbf{H}^T, e, T, \mathbf{H})$
    if $i = 1$ then
        $e \leftarrow \mathsf{BFMaskedIter}(s + e\mathbf{H}^T, e, black, (d+1)/2 + 1, \mathbf{H})$
        $e \leftarrow \mathsf{BFMaskedIter}(s + e\mathbf{H}^T, e, grey, (d+1)/2 + 1, \mathbf{H})$
    end if
end for
if $s = e\mathbf{H}^T$ then
    return $e$
else
    return $\perp$
end if

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

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

3 Masked BIKE

We present here the core contribution of this paper: a fully masked encapsulation, decapsulation and key generation for BIKE. While the encapsulation uses mostly public data, most of it had to be masked anyway as part of the decapsulation process due to the IND-CCA transform. Thus, for a perfectly complete masked design, the masked encapsulation is also included in our code. The masked decapsulation is obviously the most important part as it is the primary target of side-channel attacks. A masked key generation can also be relevant to prevent single-trace key recovery attacks when the private key is generated. A masked encapsulation might be relevant in advanced attack models to prevent single-trace message-recovery attacks.

In this section, we present the salient ideas of our masking design. Details on some selected underlying gadgets will be presented later in Section 4. Some gadgets were already

Table 2: Security properties of the known gadgets.

Gadget	Security Property	Reference
$\mathsf{sec}_{\&}$	$d-\mathsf{SNI}$	[CGTV15, BBE+18].
refresh	$d-\mathsf{SNI}$	[Cor14]
sec_{+}	$d-\mathsf{NI}$	[Cor14]

introduced in the literature but many new gadgets have been introduced to achieve our design. The complete list of gadgets is summed-up in Tables 2 and 3.

3.1 Sparse and dense notation

BIKE's private key $\mathbf{H}$ is a sparse polynomial (see Remark 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 masked private key will then be denoted by $[\![\mathbf{h}_0]\!]^\circ$, $[\![\mathbf{h}_1]\!]^\circ$ when it is masked 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 the full polynomial is masked. The same convention is applied for other intermediate variables that can be masked in dense or sparse form. For simplicity of reading, we define masked elements by omitting the order of masking. Thus, when we define $[\![\mathbf{h}_0]\!] \in \mathbb{F}_2^r$, $[\![\mathbf{h}_0]\!]$ actually has dimension $(\mathbb{F}_2^r)^{d+1}$.

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

3.2 Key generation

The masked key generation is introduced below in Algorithm 6. We use a masked version of the Fisher-Yates algorithm. It consists in drawing a vector of n random elements, where each position i 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 will allow us to generate our private keys $\mathbf{h}_0$ and $\mathbf{h}_1$, to then compute the public key $\mathbf{h}$. 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 1. *The masked key generation algorithm is $d-\mathsf{NI}$.*

Remark 3. In practice, we tend to make this algorithm $d-\mathsf{SNI}$ by adding a refresh on $\mathbf{h}_0$ and $\mathbf{h}_1$ before returning them, allowing us to use the freshly created keys without having to renew the randomness.
Also, we can see that the public key is returned in masked form. We have chosen to leave

Table 3: Security properties of the introduced gadgets.

Non-Specific Gadgets

Gadget	d-NI Theorem	Function
SecAdder (Alg. 23 & 24)	Th. 16 & 17	Addition in $\mathbb{Z}$
$\mathsf{sec}_{=}$ (Alg. 25)	Th. 18	Equality check
SecBitslice (Alg. 11 & 12)	Th. 6	Bitsliced addition
$\mathsf{SecMult}_{\text{partlymasked}}$ (Alg. 26)	Th. 19	Multiplication between masked and unmasked
$\mathsf{sec}_{\mathsf{rand}}$ (Alg. 27)	Th. 20	Modular random number
SecPolymul (Alg. 28)	Th. 21	Polynomial multiplication
SecKaratsuba (Alg.13)	Th. 7	Karatsuba multiplication
$\mathsf{sec}_{\gg}$ (Alg. 29)	Th. 22	Cyclic shift
$\mathsf{SecMult}_{\text{sparsedense}}$ (Alg. 14)	Th. 8	Multiplication between sparse and dense
$\mathsf{sec}_{\mathsf{hw}}$ (Alg. 15)	Th. 9	Hamming weight computation
$\mathsf{sec}_{\mathsf{if}}$ (Alg. 30)	Th. 23	Conditional if
$\mathsf{sec}_{\mathsf{max}}$ (Alg. 31)	Th. 24	Maximum function
$\mathsf{sec}_{\mathsf{fill}}$ (Alg. 32)	Th. 25	Matrix filling

BIKE-adapted Gadgets

Gadget	$d-$NI Theorem	Function
SecBGF (Alg. 9)	Th. 4	BGF Decoder (Alg. 5)
SecKeyGen (Alg. 6)	Th. 1	Key Generation (Alg. 1)
SecErrorGen (Alg. 7)	Th. 2	Generation of (e_0, e_1) in Decaps
SecGreyZone (Alg. 22)	Th. 15	Grey Zone technique ([DGK20])
SecFisherYates (Alg. 17)	Th. 11	Fisher-Yates generation of sparse polynomials ([Sen21])
SecInversion (Alg. 18)	Th. 12	Polynomial inversion
SecSyndrome (Alg. 16)	Th. 10	Syndrome computation
SecThreshold (Alg. 19)	Th. 13	BIKE's Threshold computation
SecCounter (Alg. 21)	Th. 14	BIKE's counter computation

Algorithm 6 Masked key generation

```
Ensure: $[\![\mathbf{h}_0]\!]^\circ \in \mathbb{Z}_r^{\frac{w}{2}}$, $[\![\mathbf{h}_1]\!]^\circ \in \mathbb{Z}_r^{\frac{w}{2}}$, $[\![\mathbf{h}_0]\!] \in \mathbb{F}_2^r$, $[\![\mathbf{h}_1]\!] \in \mathbb{F}_2^r$, $[\![\mathbf{h}]\!] \in \mathbb{F}_2^r$, $[\![\vec{\sigma}]\!] \in \mathbb{F}_2^\ell$
$[\![\mathbf{h}_0]\!]^\circ \leftarrow \mathsf{SecFisherYates}(\frac{w}{2}, r)$        ▷ Algorithm 17
$[\![\mathbf{h}_1]\!]^\circ \leftarrow \mathsf{SecFisherYates}(\frac{w}{2}, r)$
$[\![\mathbf{h}_0]\!] \leftarrow \mathsf{sparse\_to\_dense}([\![\mathbf{h}_0]\!]^\circ)$        ▷ see Section 3.1
$[\![\mathbf{h}_1]\!] \leftarrow \mathsf{sparse\_to\_dense}([\![\mathbf{h}_1]\!]^\circ)$
$[\![\mathbf{h}_0^{-1}]\!] \leftarrow \mathsf{SecInversion}([\![\mathbf{h}_0]\!], r)$        ▷ Algorithm 18
$[\![\mathbf{h}]\!] \leftarrow \mathsf{SecKaratsuba}([\![\mathbf{h}_0^{-1}]\!], [\![\mathbf{h}_1]\!])$        ▷ Algorithm 13
$[\![\vec{\sigma}]\!] \xleftarrow{\$} \mathbb{F}_{2^\ell}$        ▷ Draw $\ell$ bits on each share
return $\mathsf{sk} = ([\![\mathbf{h}_0]\!]^\circ, [\![\mathbf{h}_1]\!]^\circ, [\![\mathbf{h}_0]\!], [\![\mathbf{h}_1]\!])$, $\mathsf{pk} = [\![\mathbf{h}]\!]$, $[\![\vec{\sigma}]\!]$
```

it masked because it allows a simple syndrome computation, in a practical implementation we would unmask it directly after computing it in order to transmit less data.
Note that the public parameters (public keys etc.) can always be used for simulating the probes and that in BIKE 's case, the joint distribution of the public key and any set of at most d internal shares is never correlated to the secret key..

3.3 Encapsulation

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 [ABB+22], the $\mathbf{K}$, $\mathbf{L}$ and $\mathbf{H}$ hash functions (see Algorithm 3) are instantiated with SHAKE256 and SHA384. These functions have already been protected in the masked implementation of Saber (see [DKR+20] for more information about Saber) in [KDVB+22]. This framework is easily adaptable for BIKE without major modification. Masking is done in a similar way, keeping the same masking order.

3.3.1 Error generation

The error generation algorithm is necessary for both encapsulation and decapsulation. Its masked version is introduced below in Algorithm 7. It consists in generating a masked error vector $[\![\mathbf{e}_0]\!], [\![\mathbf{e}_1]\!]$.
It uses two d-NI sub-gadgets:

- sec_{+} corresponds to the logical addition of two integers [Cor14]. 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).
- $\mathsf{sec}_{\mathsf{if}}$ represents a conditional branch, it outputs either the first input or the second one depending on the Boolean value of the last input. It is detailed in Algorithm 30 in Appendix A.6.

The error cannot be represented in sparse representation, as the weights of $[\![\mathbf{e}_0]\!], [\![\mathbf{e}_1]\!]$ are not constant. This would leak sensitive information.

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 2. *The error generation algorithm is $d - \mathsf{NI}$.*

Remark 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) which is then processed in the same way as Fisher-Yates. Since we have defined Fisher-Yates (Algorithm 17) 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.

3.3.2 Encapsulation algorithm

All the functions used are $d - \mathsf{NI}$.
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 $d - \mathsf{NI}$.

Theorem 3. *The encapsulation algorithm is $d - \mathsf{NI}$.*

Algorithm 7 Masked Error generation SecErrorGen

```
Require: $\llbracket s \rrbracket \in \mathbb{F}_2^{\ell}$ the seed for SecFisherYates
Ensure: $\llbracket \mathbf{e}_0 \rrbracket \in \mathbb{F}_2^{r}$, $\llbracket \mathbf{e}_1 \rrbracket \in \mathbb{F}_2^{r}$
$\llbracket \mathbf{e}_0 \rrbracket \leftarrow \mathsf{vector\_zero\_masking}()$
$\llbracket \mathbf{e}_1 \rrbracket \leftarrow \mathsf{vector\_zero\_masking}()$
$\llbracket \mathbf{e} \rrbracket^{\circ} \leftarrow \mathsf{SecFisherYates}(t, 2 \times r)$    ▷ Algorithm 17
for $i \leftarrow 0$ to $t-1$ do
    $\llbracket v \rrbracket \leftarrow \mathsf{sec}_{+\mathrm{partlymasked}}(\llbracket \mathbf{e} \rrbracket_i^{\circ}, -r)$    ▷ see Section 3.3.1
    $\llbracket \mathbf{e} \rrbracket_i^{\circ} \leftarrow \mathsf{refresh}(\llbracket \mathbf{e} \rrbracket_i^{\circ})$
    $\llbracket \mathbf{t} \rrbracket^{\circ} \leftarrow \mathsf{sec}_{\mathsf{if}}(\llbracket \mathbf{e} \rrbracket_i^{\circ}, \llbracket v \rrbracket, \mathsf{sign\_bit}(\mathsf{refresh}(\llbracket v \rrbracket)))$    ▷ see Section 3.3.1
    $\llbracket \mathbf{t} \rrbracket \leftarrow \mathsf{sparse\_to\_dense}(\llbracket \mathbf{t} \rrbracket^{\circ})$    ▷ see Section 3.1, polynomial with only one coefficient
    $\llbracket \mathbf{e}_0 \rrbracket \leftarrow \llbracket \mathbf{e}_0 \rrbracket \oplus \mathsf{sec}_{\mathsf{if}}(\llbracket \mathbf{t} \rrbracket, \mathsf{vector\_zero\_masking}(), \mathsf{sign\_bit}(\mathsf{refresh}(\llbracket v \rrbracket)))$    ▷ Coefficient-wise $\mathsf{sec}_{\mathsf{if}}$ and XOR
    $\llbracket \mathbf{e}_1 \rrbracket \leftarrow \llbracket \mathbf{e}_1 \rrbracket \oplus \mathsf{sec}_{\mathsf{if}}(\mathsf{vector\_zero\_masking}(), \llbracket \mathbf{t} \rrbracket, \mathsf{sign\_bit}(\mathsf{refresh}(\llbracket v \rrbracket)))$
end for
return $\llbracket \mathbf{e}_0 \rrbracket, \llbracket \mathbf{e}_1 \rrbracket$
```

Algorithm 8 Encapsulation

```
Require: $\llbracket \mathbf{h} \rrbracket \in \mathbb{F}_2^{r}$
Ensure: $\llbracket \mathbf{c} \rrbracket \in \mathbb{F}_2^{r+\ell}$
$\llbracket \mathbf{m} \rrbracket \overset{\$}{\leftarrow} \mathbb{F}_2^{\ell}$
$\llbracket \mathbf{e} \rrbracket = (\llbracket \mathbf{e}_0 \rrbracket, \llbracket \mathbf{e}_1 \rrbracket) \leftarrow \mathsf{SecErrorGen}(\llbracket \mathbf{m} \rrbracket)$    ▷ Algorithm 7
$\llbracket \mathbf{m} \rrbracket \leftarrow \mathsf{refresh}(\llbracket \mathbf{m} \rrbracket)$
$\llbracket \mathbf{c}_0 \rrbracket \leftarrow \mathsf{SecKaratsuba}(\llbracket \mathbf{e}_1 \rrbracket, \llbracket \mathbf{h} \rrbracket)$
$\llbracket \mathbf{c}_0 \rrbracket \leftarrow \llbracket \mathbf{c}_0 \rrbracket \oplus \llbracket \mathbf{e}_0 \rrbracket$    ▷ Coefficient-wise XOR
$\llbracket \mathbf{e} \rrbracket \leftarrow \mathsf{refresh}(\llbracket \mathbf{e} \rrbracket)$
$\llbracket \mathbf{c}_1 \rrbracket \leftarrow \mathsf{L}(\llbracket \mathbf{e} \rrbracket) \oplus \llbracket \mathbf{m} \rrbracket$    ▷ see Section 3.3, Coefficient-wise XOR
return $\llbracket \mathbf{c} \rrbracket = (\llbracket \mathbf{c}_0 \rrbracket, \llbracket \mathbf{c}_1 \rrbracket)$
```

3.4 Decapsulation

Decapsulation consists of first decoding the ciphertext and secondly checking that it is correct. While the most challenging masking work was on the decoding algorithm, we propose below a fully masked version of the decapsulation for completeness in Algorithm 10.

3.4.1 BGF decoder

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 2.4. The masked version of Algorithm 5 is detailed in Algorithm 9. Recall that all the sub-gadgets are detailed in Section 4 and Appendix A (see Tables 2 and 3).

Where SecThreshold and SecSyndrome are fairly simple gadgets (based on additions, multiplications or shifts), SecCounter was quite challenging to mask as it relies on bitslicing. The SecGreyZone optimization allows for a much performant decoder but it also adds a layer of complexity in the decoding. This complexity is also transferred when masking is involved as several sensitive data are used inside the computations.

We denote by vector_zero_masking a subroutine that initializes a d sharing of an r-dimensional zero vector. Let $\mathbf{C}$ be a pair of matrices represented as table of dimension $2 \times r \times (|\frac{w}{2}| + 1)$, that can be decomposed into two matrices $\mathbf{C}_0$ and $\mathbf{C}_1$ of dimension $r \times |(\frac{w}{2}| + 1)$. The notation $\mathbf{C}_{0,*,|\frac{w}{2}|}$ represents the entire row of height $|\frac{w}{2}|$.

Algorithm 9 BGF decoder

```
Require: sk = $\left(\llbracket \mathbf{h}_0 \rrbracket \in \mathbb{F}_2^r, \llbracket \mathbf{h}_1 \rrbracket \in \mathbb{F}_2^r, \llbracket \mathbf{h}_0 \rrbracket^\circ \in \mathbb{Z}_r^{\frac{w}{2}}, \llbracket \mathbf{h}_1 \rrbracket^\circ \in \mathbb{Z}_r^{\frac{w}{2}}\right)$, $\llbracket \mathbf{c}_0 \rrbracket \in \mathbb{F}_2^r$
Ensure: $\llbracket \mathbf{e}_0 \rrbracket \in \mathbb{F}_2^r$, $\llbracket \mathbf{e}_1 \rrbracket \in \mathbb{F}_2^r$ such that $(\mathbf{c}_0 + \mathbf{e}_0) \cdot \mathbf{h}_0 = 0$
$\llbracket \mathbf{e}_0 \rrbracket \leftarrow$ vector_zero_masking()
$\llbracket \mathbf{e}_1 \rrbracket \leftarrow$ vector_zero_masking()
$\llbracket \mathbf{s} \rrbracket \leftarrow$ SecKaratsuba($\llbracket \mathbf{c}_0 \rrbracket, \llbracket \mathbf{h}_0 \rrbracket$)    ▷ Algorithm 13
$\llbracket \mathbf{h}_0 \rrbracket \leftarrow$ refresh($\llbracket \mathbf{h}_0 \rrbracket$)
for $i \leftarrow 0$ to Nbr_Iter $-1$ do
    $\llbracket \mathbf{s1} \rrbracket \leftarrow$ 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 16
    $\llbracket T \rrbracket \leftarrow$ SecThreshold($\llbracket \mathbf{s1} \rrbracket$)    ▷ Algorithm 19
    $\llbracket \mathbf{s1} \rrbracket \leftarrow$ refresh($\llbracket \mathbf{s1} \rrbracket$)
    $\llbracket \mathbf{C} \rrbracket \leftarrow$ SecCounter($\llbracket \mathbf{s1} \rrbracket, \llbracket T \rrbracket, \llbracket \mathbf{h}_0 \rrbracket^\circ, \llbracket \mathbf{h}_1 \rrbracket^\circ$)    ▷ Algorithm 21
    $\llbracket \mathbf{e}_0 \rrbracket \leftarrow$ refresh($\llbracket \mathbf{e}_0 \rrbracket$); $\llbracket \mathbf{e}_1 \rrbracket \leftarrow$ refresh($\llbracket \mathbf{e}_1 \rrbracket$)
    ▷ $\llbracket \mathbf{C}_{0,*,|\frac{w}{2}|} \rrbracket$ and $\llbracket \mathbf{C}_{1,*,|\frac{w}{2}|} \rrbracket$ are the sign bit of the counters minus the threshold
    $\llbracket \mathbf{e}_0 \rrbracket \leftarrow \neg((\llbracket \mathbf{e}_0 \rrbracket) \oplus (\llbracket \mathbf{C}_{0,*,|\frac{w}{2}|} \rrbracket))$    ▷ Coefficient-wise XOR
    $\llbracket \mathbf{e}_1 \rrbracket \leftarrow \neg((\llbracket \mathbf{e}_1 \rrbracket) \oplus (\llbracket \mathbf{C}_{1,*,|\frac{w}{2}|} \rrbracket))$    ▷ Coefficient-wise XOR
    if $i = 0$ then
        $\llbracket \mathbf{s} \rrbracket \leftarrow$ refresh($\llbracket \mathbf{s} \rrbracket$)
        $\llbracket \mathbf{e}_0 \rrbracket, \llbracket \mathbf{e}_1 \rrbracket \leftarrow$ SecGreyZone($\llbracket \mathbf{C} \rrbracket, \llbracket \mathbf{h}_0 \rrbracket, \llbracket \mathbf{h}_1 \rrbracket, \llbracket \mathbf{h}_0 \rrbracket^\circ, \llbracket \mathbf{h}_1 \rrbracket^\circ, \llbracket \mathbf{e}_0 \rrbracket, \llbracket \mathbf{e}_1 \rrbracket, \llbracket \mathbf{s} \rrbracket$)    ▷ Algorithm 22
    end if
    $\llbracket$sk$\rrbracket \leftarrow$ refresh($\llbracket$sk$\rrbracket$)
    $\llbracket \mathbf{s} \rrbracket \leftarrow$ refresh($\llbracket \mathbf{s} \rrbracket$)
end for
return $(\llbracket \mathbf{e}_0 \rrbracket, \llbracket \mathbf{e}_1 \rrbracket)$
```

Theorem 4. *The BGF decoder algorithm is $d-$*NI.

Proof. We represent the whole decoding algorithm in Figs. 1 and 2. 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).

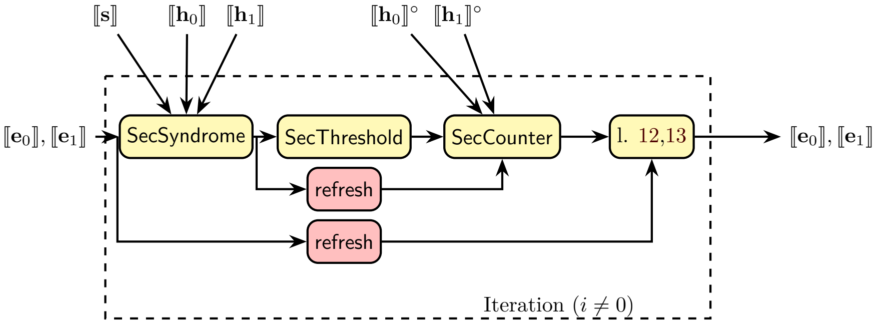
$\llbracket \mathbf{s} \rrbracket$
$\llbracket \mathbf{h}_0 \rrbracket$
$\llbracket \mathbf{h}_1 \rrbracket$
$\llbracket \mathbf{h}_0 \rrbracket^\circ$
$\llbracket \mathbf{h}_1 \rrbracket^\circ$
$\llbracket \mathbf{e}_0 \rrbracket, \llbracket \mathbf{e}_1 \rrbracket$
SecSyndrome
SecThreshold
SecCounter
l. 12,13
$\llbracket \mathbf{e}_0 \rrbracket, \llbracket \mathbf{e}_1 \rrbracket$
refresh
refresh
Iteration ($i \neq 0$)

Figure 1: Structure of an iteration

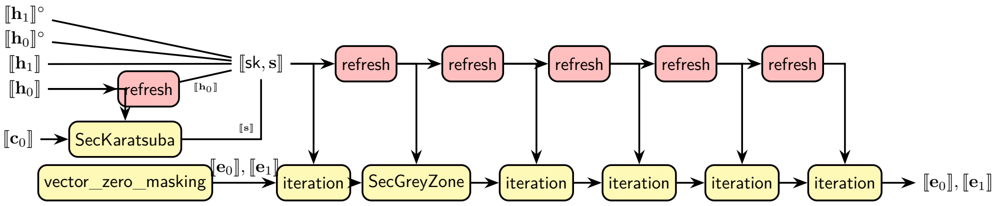
$\llbracket \mathbf{h}_1 \rrbracket^\circ$
$\llbracket \mathbf{h}_0 \rrbracket^\circ$
$\llbracket \mathbf{h}_1 \rrbracket$
$\llbracket \mathbf{h}_0 \rrbracket$
$\llbracket \mathbf{c}_0 \rrbracket$
$\llbracket \mathsf{sk}, \mathbf{s} \rrbracket$
refresh
$\llbracket \mathbf{h}_0 \rrbracket$
SecKaratsuba
$\llbracket \mathbf{s} \rrbracket$
refresh
refresh
refresh
refresh
refresh
vector_zero_masking
$\llbracket \mathbf{e}_0 \rrbracket, \llbracket \mathbf{e}_1 \rrbracket$
iteration
SecGreyZone
iteration
iteration
iteration
iteration
$\llbracket \mathbf{e}_0 \rrbracket, \llbracket \mathbf{e}_1 \rrbracket$

Figure 2: 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 $\llbracket \mathbf{e}_0 \rrbracket$, $\llbracket \mathbf{e}_1 \rrbracket$, $\llbracket \mathsf{sk} \rrbracket$ and $\llbracket \mathbf{s} \rrbracket$. And we assume that this iteration's output is a modified version of $\llbracket \mathbf{e}_0 \rrbracket, \llbracket \mathbf{e}_1 \rrbracket$. Let us assume that an attacker has access to $\delta \leq d$ 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 $\llbracket \mathsf{sk} \rrbracket, \llbracket \mathbf{s} \rrbracket, \llbracket \mathbf{e}_0 \rrbracket$ and $\llbracket \mathbf{e}_1 \rrbracket$. 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 $\llbracket \mathbf{s1} \rrbracket$ si refreshed,
- δ_1 when $\llbracket \mathbf{e}_0 \rrbracket$ and $\llbracket \mathbf{e}_1 \rrbracket$ are refreshed.

By definition of the d-probing model, we have $\sum_{j=1}^{6} \delta_i \leq \delta \leq d$.
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 specified in Table 3. The proofs will be provided later in the paper. Finally, all the observations performed during this iteration can be perfectly simulated with at most $\sum_{j=1}^{6} \delta_i$ shares of $\llbracket \mathbf{e}_0 \rrbracket$, the same amount for $\llbracket \mathbf{e}_1 \rrbracket$, $\delta_6 + \delta_5$ shares of $\llbracket \mathbf{h}_0 \rrbracket^\circ$, the same for $\llbracket \mathbf{h}_0 \rrbracket^\circ$, $\sum_{j=2}^{6} \delta_i$ shares of $\llbracket \mathbf{h}_0 \rrbracket$ and finally the same for $\llbracket \mathbf{h}_1 \rrbracket$.

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

Now let us analyze the complete construction in Fig. 2. The same reasoning applies. Let us assume that an attacker has access to $\delta \leq d$ observations on this algorithm. We consider the following distribution of the attacker's δ observations:

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

By definition, $\sum_{i=0}^{Nbr\_Iter-1}(\delta_{\text{iter,i}}+\delta_{\text{ref,i}})+\delta_{\mathsf{SecGreyZone}}+\delta_{\mathsf{vector\_zero\_masking}}+\delta_{\mathsf{SecKaratsuba}}+\delta_{\text{ref}}\leq\delta\leq d$.
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 $\llbracket\mathsf{sk}\rrbracket,\llbracket\mathbf{s}\rrbracket,\llbracket\mathbf{e}_0\rrbracket$ and $\llbracket\mathbf{e}_1\rrbracket$. 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 $\llbracket\mathbf{c}_0\rrbracket$, the same for $\llbracket\mathbf{h}_0\rrbracket$ and $\sum_{i=0}^{Nbr\_Iter-1}(\delta_{\text{iter,i}}+\delta_{\text{ref,i}})+\delta_{\mathsf{SecGreyZone}}$ for the rest of the secret key. All these numbers are smaller than to $\delta\leq d$ which concludes the proof.

□

3.4.2 Decapsulation algorithm

For the needs of the decapsulation algorithm, we will introduce subvector function, an algorithm which returns the subvector starting and ending with the bounds given as parameters.

Algorithm 10 Decapsulation

```
Require: $\mathsf{sk}=\left(\llbracket\mathbf{h}_0\rrbracket\in\mathbb{F}_2^r,\llbracket\mathbf{h}_1\rrbracket\in\mathbb{F}_2^r,\llbracket\mathbf{h}_0\rrbracket^\circ\in\mathbb{Z}_r^{\frac{w}{2}},\llbracket\mathbf{h}_1\rrbracket^\circ\in\mathbb{Z}_r^{\frac{w}{2}}\right)$,$\llbracket\mathbf{c}\rrbracket\in\mathbb{F}_2^{r+\ell}$, $\llbracket\vec{\sigma}\rrbracket\in\mathbb{F}_2^\ell$
Ensure: $\llbracket\mathbf{k}\rrbracket\in\mathbb{F}_2^\ell$
$\llbracket\mathbf{e}'\rrbracket\leftarrow\mathsf{SecBGF}(\llbracket\mathbf{h}_0\rrbracket,\llbracket\mathbf{h}_1\rrbracket,\llbracket\mathbf{h}_0\rrbracket^\circ,\llbracket\mathbf{h}_1\rrbracket^\circ,\mathsf{subvector}(\llbracket\mathbf{c}\rrbracket,0,r-1))$    ▷ Algorithm 9
$\llbracket\mathbf{m}'\rrbracket\leftarrow\mathsf{L}(\llbracket\mathbf{e}'\rrbracket)$    ▷ see Section 3.3
$\llbracket\mathbf{m}'\rrbracket\leftarrow\llbracket\mathbf{m}'\rrbracket\oplus\mathsf{subvector}(\llbracket\mathbf{c}\rrbracket,r,r+\ell)$    ▷ see Section 3.4.2, coefficient-wise XOR
$(\llbracket\mathbf{e}_0\rrbracket,\llbracket\mathbf{e}_1\rrbracket)\leftarrow\mathsf{SecErrorGen}(\llbracket\mathbf{m}'\rrbracket)$    ▷ Algorithm 7
$\llbracket\mathbf{m}'\rrbracket\leftarrow\mathsf{refresh}(\llbracket\mathbf{m}'\rrbracket)$
$\llbracket v\rrbracket\leftarrow 1$    ▷ Masked value of 1
for $i\leftarrow 0$ to 1 do
    for $j\leftarrow 0$ to $r-1$ do
        $\llbracket t\rrbracket\leftarrow\mathsf{sec}_{=}(\llbracket\mathbf{e}_{i,j}\rrbracket,\llbracket\mathbf{e}'_{i,j}\rrbracket)$    ▷ see Appendix A.2
        $\llbracket t\rrbracket_0\leftarrow\llbracket t\rrbracket_0\oplus 1$
        $\llbracket v\rrbracket\leftarrow\mathsf{sec}_{\&}(\llbracket v\rrbracket,\llbracket t\rrbracket)$
    end for
end for
$\llbracket\mathbf{t}\rrbracket\leftarrow\mathsf{K}(\llbracket\mathbf{m}'\rrbracket,\llbracket\mathbf{c}\rrbracket)$    ▷ Section 3.4.2
$\llbracket\mathbf{t1}\rrbracket\leftarrow\mathsf{K}(\llbracket\vec{\sigma}\rrbracket,\llbracket\mathbf{c}\rrbracket)$
$\llbracket\mathbf{k}\rrbracket\leftarrow\mathsf{sec}_{\mathsf{if}}(\llbracket\mathbf{t}\rrbracket,\llbracket\mathbf{t1}\rrbracket,\llbracket v\rrbracket)$    ▷ Coefficient-wise $\mathsf{sec}_{\mathsf{if}}$
return $\llbracket\mathbf{k}\rrbracket$
```

Algorithm 10 uses $d-\mathsf{NI}$ gadgets, and the only variable that is used twice without modification is $\mathbf{m}'$. However, the dependancy loop is broken by the $d-\mathsf{SNI}$ refresh. Thus, we introduce the following theorem.

Theorem 5. *The decapsulation algorithm is $d-\mathsf{NI}$.*

For reasons similar to encapsulation, we may want to make this algorithm $d - \mathsf{SNI}$ if we wish to reuse the private key several times. However, it becomes less relevant if BIKE is used with an ephemeral key.

4 Details on selected gadgets

In this section, we provide some details about selected gadgets.

4.1 Bitslicing

Bitslicing was introduced in [Cho16] for the QcBits implementation, with many similarities to BIKE. These techniques allow computations to be performed very efficiently and in constant time by focusing on the binary representation. In Algorithms 11 and 12, we present two versions of this BitSlice procedure depending on the type of the input. Both versions will be used in our implementation.

In Algorithms 11 and 12, we denote by SecHalf_Adder the procedure that computes the addition in $\mathbb{Z}$ of the inputs while outputing the carry as a second output. The SecAdder performs the same operation but is given an extra carry. These simple gadgets are detailed and proved d-NI in Appendix A.1 for completeness. We also denote by zero_masking an initialization of a d-sharing of zero.

Algorithm 11 SecHalf_Bitslice

```
Require: $\llbracket \mathbf{X} := (\mathbf{X}_0, \cdots, \mathbf{X}_\ell) \rrbracket \in \mathbb{Z}_2^{k \times \ell}$, $\llbracket \mathbf{y} \rrbracket \in \mathbb{Z}_2^{\ell}$
Ensure: $\llbracket \mathbf{X} \rrbracket \in \mathbb{Z}_2^{\ell \times k}$ the result of the bitsliced addition between $\llbracket \mathbf{X} \rrbracket$ and $\mathbf{y}$
for $i := 0$ to $\ell - 1$ do
    $\llbracket r \rrbracket := \llbracket \mathbf{y}_i \rrbracket$
    for $j := 0$ to $k - 1$ do
        $(\llbracket \mathbf{X}_{ij} \rrbracket, \llbracket r \rrbracket) \leftarrow \mathsf{SecHalf\_Adder}(\llbracket \mathbf{X}_{ij} \rrbracket, \llbracket r \rrbracket)$
    end for
end for
return $\llbracket \mathbf{X} \rrbracket$
```

Algorithm 12 SecBitslice

```
Require: $\llbracket \mathbf{X} := (\mathbf{X}_0, \cdots, \mathbf{X}_k) \rrbracket \in \mathbb{Z}_2^{\ell \times k}$, $\llbracket \mathbf{Y} := (\mathbf{Y}_0, \cdots, \mathbf{Y}_k) \rrbracket \in \mathbb{Z}_2^{\ell \times k}$
Ensure: $\llbracket \mathbf{X} \rrbracket \in \mathbb{Z}_2^{\ell \times k}$ the result of the bitsliced addition between $\llbracket \mathbf{X} \rrbracket$ and $\llbracket \mathbf{Y} \rrbracket$
for $i := 0$ to $\ell - 1$ do
    $\llbracket r \rrbracket \leftarrow \mathsf{zero\_masking}()$
    for $j := 0$ to $k - 1$ do
        $(\llbracket \mathbf{X}_{ij} \rrbracket, \llbracket r \rrbracket) \leftarrow \mathsf{SecAdder}(\llbracket \mathbf{X}_{ij} \rrbracket, \llbracket \mathbf{Y}_{ij} \rrbracket, \llbracket r \rrbracket)$
    end for
end for
return $\llbracket \mathbf{X} \rrbracket$
```

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 6. *The* SecHalf_Bitslice *and* SecBitslice *algorithms are* $d - \mathsf{NI}$.

4.2 Multiplications

Several multiplication algorithms are necessary for masking BIKE. Indeed, 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. Thus, one valid solution is to fully mask the classical Karatsuba algorithm, as presented below. We denote by SecPolymul the naive schoolbook polynomial multiplication (detailed in Algorithm 28 in Appendix A.3 for completeness). 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 size of the inputs. Let split be a subroutine that splits the $s/2$ high order and $s/2$ low order bits into two variables.

Algorithm 13 Karatsuba multiplication on vectors

```
Require: $[\![\mathbf{p1}]\!] \in \mathbb{F}_2^s$, $[\![\mathbf{p2}]\!] \in \mathbb{F}_2^s$
Ensure: $[\![\mathbf{z}]\!] = [\![\mathbf{p1}]\!] \cdot [\![\mathbf{p2}]\!] \in \mathbb{F}_2^{2s}$
if $s = B$ then
    return SecPolymul($[\![\mathbf{p1}]\!], [\![\mathbf{p2}]\!]$)    ▷ Naive polynomial multiplication, see Algorithm 28 in Appendix A.3
end if
($[\![\mathsf{left1}]\!], [\![\mathsf{right1}]\!]$) ← split($[\![\mathbf{p1}]\!]$)  ▷ Splitting the $s/2$ high order and $s/2$ low order bits
($[\![\mathsf{left2}]\!], [\![\mathsf{right2}]\!]$) ← split($[\![\mathbf{p2}]\!]$)  ▷ Splitting the $s/2$ high order and $s/2$ low order bits
$[\![\mathbf{z1}]\!]$ ← SecKaratsuba($[\![\mathsf{right1}]\!], [\![\mathsf{right2}]\!]$)
$[\![\mathbf{z2}]\!]$ ← SecKaratsuba($[\![\mathsf{left1}]\!], [\![\mathsf{left2}]\!]$)
$[\![\mathsf{left1}]\!]$ ← refresh($[\![\mathsf{left1}]\!]$)
$[\![\mathsf{right1}]\!]$ ← refresh($[\![\mathsf{right1}]\!]$)
$[\![\mathsf{left2}]\!]$ ← refresh($[\![\mathsf{left2}]\!]$)
$[\![\mathsf{right2}]\!]$ ← refresh($[\![\mathsf{right2}]\!]$)
$[\![\mathbf{t1}]\!]$ ← $[\![\mathsf{left1}]\!] \oplus [\![\mathsf{right1}]\!]$    ▷ Coefficient-wise XOR
$[\![\mathbf{t2}]\!]$ ← $[\![\mathsf{left2}]\!] \oplus [\![\mathsf{right2}]\!]$    ▷ Coefficient-wise XOR
$[\![\mathbf{z3}]\!]$ ← SecKaratsuba($[\![\mathbf{t1}]\!], [\![\mathbf{t2}]\!]$)
return $[\![\mathbf{z}]\!] \leftarrow [\![\mathbf{z1}]\!] \oplus ([\![\mathbf{z2}]\!] \ll s/4) \oplus ([\![\mathbf{z3}]\!] \ll s/2)$    ▷ Coefficient-wise
```

Theorem 7. *The Karatsuba algorithm is $d - \mathsf{NI}$ for any power of two s and any bound $B \leq s$.*

Proof. Let us prove this theorem by induction on the parameter s. If $s \leq B$, the $d - \mathsf{NI}$ property is directly inherited from the $d - \mathsf{NI}$ property of SecPolymul (Theorem 21 in Appendix A.3). Let us assume that the Karatsuba algorithm is $d - \mathsf{NI}$ for $s > B$ and let us sketch a proof that is it $d - \mathsf{NI}$ for the next power of two: $2 \cdot s$. The algorithm first computes $[\![\mathbf{z}_1]\!], [\![\mathbf{z}_2]\!]$ with $d - \mathsf{NI}$ 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 $d - \mathsf{NI}$ for $2 \cdot s$ which concludes the proof.

□

Remark 5 (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. In this paper, we use the same notation "SecKaratsuba" even when the multiplication is applied in the context of polynomials.

In parallel, we also introduce a multiplication algorithm that takes only one masked input, the other input being a public value, as this algorithm is also necessary for our design. We denote it $\mathsf{SecMult}_{\text{partlymasked}}$ and its design is detailed in Appendix A.3. It is directly inspired from the Montgomery ladder technique.

Leveraging sparse polynomials In BIKE, it is often possible to leverage the fact that some masked polynomials are stored in sparse notation. We then introduce an extra gadget that takes one masked dense input and one masked sparse input. The multiplication technique uses a cyclic shift, denoted $\mathsf{sec}_{\gg}$. The idea is to shift a masked dense polynomial by a masked value. It is described and proved in Appendix A.5.

Algorithm 14 Sparse-dense multiplication ($\mathsf{SecMult}_{\text{sparsedense}}$)

```
Require: $[\![\mathbf{x}]\!] \in \mathbb{F}_2^n$, $[\![\mathbf{y}]\!] \in \mathbb{F}_2^c$
Ensure: $[\![\mathbf{z}]\!] = [\![\mathbf{x}]\!] \cdot [\![\mathbf{y}]\!] \in \mathbb{F}_2^n$
for $i \leftarrow 0$ to $c-1$ do
    $[\![\mathbf{t}]\!] \leftarrow \mathsf{sec}_{\gg}([\![\mathbf{x}]\!], [\![\mathbf{y}_i]\!])$        ▷ See Algorithm 29 in Appendix A.5
    $[\![\mathbf{x}]\!] \leftarrow \mathsf{refresh}([\![\mathbf{x}]\!])$
    $[\![\mathbf{z}]\!] \leftarrow [\![\mathbf{z}]\!] \oplus [\![\mathbf{t}]\!]$        ▷ Coefficient-wise XOR
end for
return $[\![\mathbf{z}]\!]$
```

Theorem 8. *The* $\mathsf{SecMult}_{\text{sparsedense}}$ *algorithm is* $d - \mathsf{NI}$.

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

4.3 Hamming weight

We introduce a masked Hamming weight computation. It has been optimized and involves the masked bitslice algorithm presented in Algorithm 12. 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 $\mathsf{left}([\![\mathbf{T}]\!]) \in \mathbb{F}_2^{\frac{l}{2} \times k}$.
$[\![\mathbf{T}]\!]$ 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 15 Hamming weight ($\mathsf{sec}_{\mathsf{hw}}$)

```
Require: $[\![\mathbf{x}]\!] \in \mathbb{F}_2^n$
Ensure: $[\![y]\!] \in \mathbb{F}_2$ 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
$j \leftarrow 1$
for $i \leftarrow \frac{n}{2}$ to 1 step $\frac{i}{2}$ do
    $[\![\mathbf{T}]\!] \leftarrow \mathsf{SecBitslice}(\mathsf{left}([\![\mathbf{T}]\!]), \mathsf{right}([\![\mathbf{T}]\!]))$        ▷ Cut in length
    $j \leftarrow j + 1$
end for
$[\![y]\!] \leftarrow \mathsf{zero\_masking}()$
for $i \leftarrow 0$ to $j-1$ do
    $[\![y]\!] \leftarrow [\![y]\!] \oplus ([\![\mathbf{T}_{0,i}]\!] \ll i)$
end for
return $[\![y]\!]$
```

Theorem 9. *The hamming weight algorithm is* $d - \mathsf{NI}$.

Proof. Since as SecBitslice has been proved d-NI in Theorem 6 and all loops use updated variables, their composition leads to a d-NI algorithm. □

In this part we will introduce the main gadgets necessary for the realization of masked BIKE.

4.4 Computing the syndrome

Algorithm 16 compute_syndrome

```
Require: $\llbracket \mathbf{h}_0 \rrbracket \in \mathbb{F}_2^R$, $\llbracket \mathbf{h}_1 \rrbracket \in \mathbb{F}_2^R$, $\llbracket \mathbf{e}_0 \rrbracket \in \mathbb{F}_2^R$, $\llbracket \mathbf{e}_1 \rrbracket \in \mathbb{F}_2^R$, $\llbracket \mathbf{s} \rrbracket = c_0 h_0$
Ensure: $\llbracket \mathbf{s1} \rrbracket = c_0 h_0 + e_0 h_0 + e_1 h_1 \in \mathbb{F}_2^r$
  $\llbracket \mathbf{s2} \rrbracket \leftarrow$ SecKaratsuba($\llbracket \mathbf{e}_0 \rrbracket, \llbracket \mathbf{h}_0 \rrbracket$)
  $\llbracket \mathbf{s3} \rrbracket \leftarrow$ SecKaratsuba($\llbracket \mathbf{e}_1 \rrbracket, \llbracket \mathbf{h}_1 \rrbracket$)
  $\llbracket \mathbf{s1} \rrbracket \leftarrow \llbracket \mathbf{s} \rrbracket \oplus \llbracket \mathbf{s2} \rrbracket \oplus \llbracket \mathbf{s3} \rrbracket$        ▷ Coefficient-wise XOR
  return $\llbracket \mathbf{s1} \rrbracket$
```

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

Theorem 10. *The syndrome computing algorithm is $d-$ NI.*

4.5 Generation of random polynomials

The generation of sparse polynomials is performed using the Fisher-Yates technique. It was already introduced in Section 3.2. This procedure can be masked as presented in Algorithm 17. It uses $\mathsf{sec}_{\mathsf{rand}}$, presented in Appendix A.4.

Algorithm 17 Fisher-Yates (SecFisherYates)

```
Require: $s \in \mathbb{N}$, $n \in \mathbb{N}$
Ensure: $\llbracket \mathbf{r} \rrbracket \in \mathbb{Z}_n^s$ a randomly generated vector without repeated values
 1: for $i \leftarrow s-1$ to 0 do
 2:     $\llbracket \mathbf{r}_i \rrbracket \leftarrow \mathsf{sec}_{\mathsf{rand}}(n-i)$
 3:     Initialize $\llbracket i \rrbracket$ as a Boolean sharing of $i$
 4:     $\llbracket \mathbf{r}_i \rrbracket \leftarrow \mathsf{sec}_{+\text{partlymasked}}(\llbracket \mathbf{r}_i \rrbracket, i)$
 5:     for $j \leftarrow i+1$ to $s-1$ do
 6:         $\llbracket \mathbf{r}_j \rrbracket \leftarrow$ refresh($\llbracket \mathbf{r}_j \rrbracket$)
 7:         $\llbracket b \rrbracket \leftarrow \mathsf{sec}_{=}(\llbracket \mathbf{r}_i \rrbracket, \llbracket \mathbf{r}_j \rrbracket)$
 8:         $\llbracket \mathbf{r}_i \rrbracket \leftarrow$ refresh($\llbracket \mathbf{r}_i \rrbracket$)
 9:         $\llbracket i \rrbracket \leftarrow$ refresh($\llbracket i \rrbracket$)
10:         $\llbracket \mathbf{r}_i \rrbracket \leftarrow \mathsf{sec}_{\mathsf{if}}(\llbracket i \rrbracket, \llbracket \mathbf{r}_i \rrbracket, \llbracket b \rrbracket))$
11:     end for
12: end for
13: return $\llbracket \mathbf{r} \rrbracket$
```

Theorem 11. *The Fisher-Yates algorithm is $d-$ NI.*

Proof. The Fisher-Yates algorithm involves many dependency loops. Indeed, each random $\llbracket \mathbf{r}_i \rrbracket$ 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 $\llbracket \mathbf{r}_i \rrbracket$. 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. □

4.6 Masked polynomial inversion

A masked polynomial inversion is needed for inverting $\mathbf{h}_0$ inside the key generation. The masked polynomial inversion is presented in Algorithm 18.

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 18 SecInversion

```
Require: $[\![\mathbf{x}]\!] \in \mathbb{F}_2^n$
Ensure: $[\![\mathbf{y}]\!] = [\![\mathbf{x}]\!]^{-1} \in \mathbb{F}_2^n$
$[\![\mathbf{f}]\!] \leftarrow [\![\mathbf{x}]\!]$
$[\![\mathbf{y}]\!] \leftarrow [\![\mathbf{x}]\!]$
$[\![\mathbf{y}]\!] \leftarrow \mathsf{refresh}([\![\mathbf{y}]\!])$
for $i \leftarrow 0$ to $|n| - 1$ do
    $[\![\mathbf{g}]\!] \leftarrow \mathsf{sec}_{\mathsf{pow}}([\![\mathbf{f}]\!], 2^{2^i})$
    $[\![\mathbf{f}]\!] \leftarrow \mathsf{refresh}([\![\mathbf{f}]\!])$
    $[\![\mathbf{f}]\!] \leftarrow \mathsf{SecKaratsuba}([\![\mathbf{f}]\!], [\![\mathbf{g}]\!])$
    if the $(i+1)^{\text{th}}$ bit of $n-2$ is 1 then
        $[\![\mathbf{t}]\!] \leftarrow \mathsf{sec}_{\mathsf{pow}}([\![\mathbf{f}]\!], 2^{(n-2) \pmod{2^{(i+1)}}})$
        $[\![\mathbf{y}]\!] \leftarrow \mathsf{SecKaratsuba}([\![\mathbf{y}]\!], [\![\mathbf{t}]\!])$
    end if
end for
$[\![\mathbf{y}]\!] \leftarrow \mathsf{sec}_{\mathsf{pow}}([\![\mathbf{y}]\!], 2)$
return $[\![\mathbf{y}]\!]$
```

Theorem 12. *The masked inversion algorithm is $d - \mathsf{NI}$.*

Proof. The first iteration of the algorithm is presented in Fig. 3. 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.

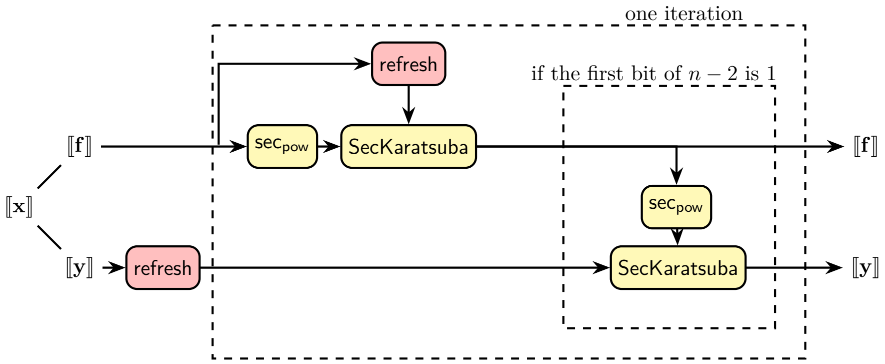

Figure 3: Sub-structure of the polynomial inversion algorithm

□

4.7 Threshold and counters

The decoder needs the computation of a threshold and counters, as presented in Algorithms 19 and 20. The threshold is an integer value that needs to be recomputed several times during decoding. 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 $\mathsf{max}(\lfloor \frac{T_0 \cdot S + T_1}{2^{T_2}} \rfloor, T_3)$.
The procedure to mask it involves gadgets previously introduced apart from $\mathsf{sec}_{\mathsf{max}}$, a gadget that, given two masked values, computes the greatest. The $\mathsf{sec}_{\mathsf{max}}$ gadget is detailed in Algorithm 31 in Appendix A.7.

Algorithm 19 SecThreshold

```
Require: $[\![\mathbf{s}]\!] \in \mathbb{F}_2^r$
Ensure: $[\![T]\!] \in \mathbb{Z}_r$ the threshold calculated from the syndrome
  $[\![S]\!] \leftarrow \mathsf{sec}_{\mathsf{hw}}([\![\mathbf{s}]\!])$        ▷ Algorithm 15
  $[\![T]\!] \leftarrow \mathsf{sec}_{\mathsf{T}}([\![S]\!])$        ▷ Algorithm 20
  return $[\![T]\!]$
```

Algorithm 20 T computing ($\mathsf{sec}_{\mathsf{T}}$)

```
Require: $[\![S]\!] \in \mathbb{Z}_r$, $T_0, T_1, T_2, T_3$ fixed parameters of the scheme
Ensure: $[\![T]\!] = \mathsf{max}(\lfloor \frac{T_0 \cdot S + T_1}{2^{T_2}} \rfloor, T_3) \in \mathbb{Z}_r$
  $[\![t]\!] \leftarrow \mathsf{SecMult}_{\mathrm{partlymasked}}([\![S]\!], T_0)$    ▷ Algorithm 26
  $[\![T]\!] \leftarrow \mathsf{sec}_{+\mathrm{partlymasked}}([\![t]\!], T_1)$
  $[\![T]\!] \leftarrow [\![T]\!] \gg T_2$
  $[\![T]\!] \leftarrow \mathsf{sec}_{\mathsf{max}}([\![T]\!], [\![T_3]\!])$ ▷ Algorithm 31
  return $[\![T]\!]$
```

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

Theorem 13. *The computation of the threshold is* $d - \mathsf{NI}$.

During decoding, it is necessary to compute the number of unsatisfied parity check equations. We present in Algorithm 21 a masked version of this routine. Let denote by $[\![\mathbf{C}]\!] \in \mathbb{F}_2^{2 \times r \times (\lfloor \frac{w}{2} \rfloor + 1)}$ 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 $[\![\mathbf{C}_1]\!]$. Let $\mathsf{matrix\_zero\_masking}$ be the initialization of a d-sharing of a 2-dimensional zero matrix. This algorithm uses a gadget that consists in filling a matrix with a value. This technical gadget does not present any difficulties and is detailed in Algorithm 32 in Appendix A.8.

Theorem 14. *The counter computing algorithm is* $d - \mathsf{NI}$.

Proof. The procedure in lines 7 to 9 of Algorithm 21 is depicted in Fig. 4. 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.

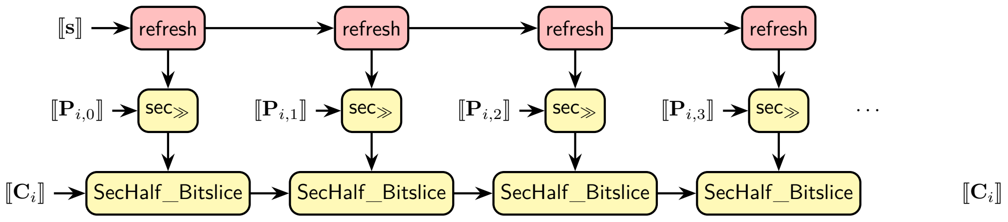

Figure 4: Sub-structure of the counter algorithm

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

Algorithm 21 Counter computing (SecCounter)

```
Require: $[\![\mathbf{s}]\!] \in \mathbb{F}_2^r$, $[\![T]\!] \in \mathbb{Z}_r$, $[\![\mathbf{h}_0]\!]^\circ \in \mathbb{Z}_r^{\frac{w}{2}}$, $[\![\mathbf{h}_1]\!]^\circ \in \mathbb{Z}_r^{\frac{w}{2}}$
Ensure: $[\![\mathbf{C}]\!] \in \mathbb{F}_2^{2\times r\times(\lfloor\frac{w}{2}\rfloor+1)}$ two matrices containing the binary representations of the
    counters for each coefficient
$[\![-T]\!] \leftarrow \mathsf{SecMult}_{\text{partlymasked}}([\![T]\!], -1)$          ▷ Algorithm 26
$[\![\mathbf{C}_0]\!] \leftarrow \mathsf{matrix\_zero\_masking}()$
$[\![\mathbf{C}_1]\!] \leftarrow \mathsf{matrix\_zero\_masking}()$
$[\![\mathbf{P}]\!] \leftarrow [[\![\mathbf{h}_0]\!]^\circ, [\![\mathbf{h}_1]\!]^\circ]$
for $i \leftarrow 0$ to $1$ do
    for $j \leftarrow 0$ to $\frac{w}{2} - 1$ do
        $[\![\mathbf{s}]\!] \leftarrow \mathsf{refresh}([\![\mathbf{s}]\!])$
        $[\![\mathbf{z}]\!] \leftarrow \mathsf{sec}_{\gg}([\![\mathbf{s}]\!], [\![\mathbf{P}_{i,j}]\!])$          ▷ Algorithm 29
        $[\![\mathbf{C}_i]\!] \leftarrow \mathsf{SecHalf\_Bitslice}([\![\mathbf{C}_i]\!], [\![\mathbf{z}]\!])$          ▷ Algorithm 11
    end for
end for
$[\![\mathbf{T}_0]\!] \leftarrow \mathsf{sec\_fill}([\![-T]\!])$          ▷ Algorithm 32
$[\![\mathbf{T}_1]\!] \leftarrow \mathsf{sec\_fill}(\mathsf{refresh}([\![-T]\!]))$          ▷ Algorithm 32
$[\![\mathbf{C}_0]\!] \leftarrow \mathsf{SecBitslice}([\![\mathbf{C}_0]\!], [\![\mathbf{T}_0]\!])$          ▷ Algorithm 12
$[\![\mathbf{C}_1]\!] \leftarrow \mathsf{SecBitslice}([\![\mathbf{C}_1]\!], [\![\mathbf{T}_1]\!])$          ▷ Algorithm 12
return$[\![\mathbf{C}]\!] = [[\![\mathbf{C}_0]\!], [\![\mathbf{C}_1]\!]]$
```

Remark 6. Note that $[\![-T]\!]$ is manipulated. To calculate the negative of a masked value, we use its two's complement: we XOR with a binary of only 1, then add 1 with sec_+.

4.8 Grey Zone

The grey zone is an additional iteration of the decoder that is only realized at the first loop of the decoder.
We will carry out the same operations as the classic decoder, but with an additional step with another threshold in order to detect more false positions and to be able to catch the possible errors of some ambiguous positions.

It 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.

Theorem 15. *The grey zone algorithm is* $d - \mathsf{NI}$.

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

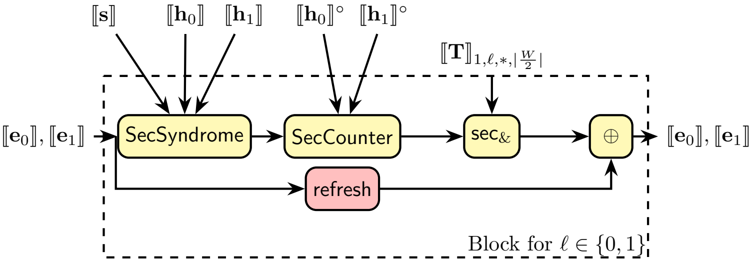
$[\![\mathbf{s}]\!]$
$[\![\mathbf{h}_0]\!]$
$[\![\mathbf{h}_1]\!]$
$[\![\mathbf{h}_0]\!]^\circ$
$[\![\mathbf{h}_1]\!]^\circ$
$[\![\mathbf{T}]\!]_{1,\ell,*,\lfloor\frac{W}{2}\rfloor}$
$[\![\mathbf{e}_0]\!], [\![\mathbf{e}_1]\!]$
SecSyndrome
SecCounter
$\mathsf{sec}_\&$
$\oplus$
$[\![\mathbf{e}_0]\!], [\![\mathbf{e}_1]\!]$
refresh
Block for $\ell \in \{0,1\}$

Figure 5: Structure of one block

Algorithm 22 SecGreyZone

```
Require: $\mathsf{sk} = \left(\llbracket \mathbf{h}_0 \rrbracket \in \mathbb{F}_2^r, \llbracket \mathbf{h}_1 \rrbracket \in \mathbb{F}_2^r, \llbracket \mathbf{h}_0 \rrbracket^\circ \in \mathbb{Z}_r^{\frac{w}{2}}, \llbracket \mathbf{h}_1 \rrbracket^\circ \in \mathbb{Z}_r^{\frac{w}{2}}\right)$, $\llbracket \mathbf{T}_0 \rrbracket \in \mathbb{F}_2^{2\times r\times(|\frac{w}{2}|+1)}$,
    $\llbracket \mathbf{e}_0 \rrbracket \in \mathbb{F}_2^r$, $\llbracket \mathbf{e}_1 \rrbracket \in \mathbb{F}_2^r$, $\llbracket \mathbf{s} \rrbracket \in \mathbb{F}_2^r$
Ensure: $\llbracket \mathbf{e}_0 \rrbracket \in \mathbb{F}_2^r$, $\llbracket \mathbf{e}_1 \rrbracket \in \mathbb{F}_2^r$
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 32
3: $\llbracket \mathbf{T}_{1,0} \rrbracket \leftarrow \mathsf{SecBitslice}(\llbracket \mathbf{T}_{0,0} \rrbracket, \llbracket \mathbf{V} \rrbracket)$          ▷ Algorithm 12
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 12
6: $\llbracket \mathbf{T}_0 \rrbracket \leftarrow \mathsf{refresh}(\llbracket \mathbf{T}_0 \rrbracket)$
7: $\llbracket \mathbf{T}_{1,0,*,|\frac{w}{2}|} \rrbracket \leftarrow \llbracket \mathbf{T}_{0,0,*,|\frac{w}{2}|} \rrbracket \oplus \llbracket \mathbf{T}_{1,0,*,|\frac{w}{2}|} \rrbracket$          ▷ Coefficient-wise XOR
8: $\llbracket \mathbf{T}_{1,1,*,|\frac{w}{2}|} \rrbracket \leftarrow \llbracket \mathbf{T}_{0,1,*,|\frac{w}{2}|} \rrbracket \oplus \llbracket \mathbf{T}_{1,1,*,|\frac{w}{2}|} \rrbracket$          ▷ Coefficient-wise XOR
9: for $l \leftarrow 0$ to 1 do
10:     $\llbracket \mathsf{sk} \rrbracket \leftarrow \mathsf{refresh}(\llbracket \mathsf{sk} \rrbracket)$
11:     $\llbracket \mathbf{s1} \rrbracket \leftarrow \mathsf{SecSyndrome}(\llbracket \mathbf{h}_0 \rrbracket, \llbracket \mathbf{h}_1 \rrbracket, \llbracket \mathbf{e}_0 \rrbracket, \llbracket \mathbf{e}_1 \rrbracket, \llbracket \mathbf{s} \rrbracket)$          ▷ Algorithm 16
12:     $\llbracket \mathbf{C} \rrbracket \leftarrow \mathsf{SecCounter}(\llbracket \mathbf{s1} \rrbracket, \frac{\frac{w}{2}+1}{2}, \llbracket \mathbf{h}_0 \rrbracket^\circ, \llbracket \mathbf{h}_1 \rrbracket^\circ)$          ▷ Algorithm 21
13:     $\llbracket \mathbf{v}_0 \rrbracket \leftarrow \mathsf{sec}_\&(\neg \llbracket \mathbf{C}_{0,*,|\frac{w}{2}|} \rrbracket, \llbracket \mathbf{T}_{l,0,*,|\frac{w}{2}|} \rrbracket)$          ▷ Coefficient-wise $\mathsf{sec}_\&$
14:     $\llbracket \mathbf{v}_1 \rrbracket \leftarrow \mathsf{sec}_\&(\neg \llbracket \mathbf{C}_{1,*,|\frac{w}{2}|} \rrbracket, \llbracket \mathbf{T}_{l,1,*,|\frac{w}{2}|} \rrbracket)$          ▷ Coefficient-wise $\mathsf{sec}_\&$
15:     $\llbracket \mathbf{e}_0 \rrbracket, \llbracket \mathbf{e}_1 \rrbracket \leftarrow \mathsf{refresh}(\llbracket \mathbf{e}_0 \rrbracket, \llbracket \mathbf{e}_1 \rrbracket)$
16:     $\llbracket \mathbf{e}_0 \rrbracket \leftarrow \llbracket \mathbf{e}_0 \rrbracket \oplus \llbracket \mathbf{v}_0 \rrbracket$          ▷ Coefficient-wise XOR
17:     $\llbracket \mathbf{e}_1 \rrbracket \leftarrow \llbracket \mathbf{e}_1 \rrbracket \oplus \llbracket \mathbf{v}_1 \rrbracket$          ▷ Coefficient-wise XOR
18: end for
19: return $\llbracket \mathbf{e}_0 \rrbracket, \llbracket \mathbf{e}_1 \rrbracket$
```

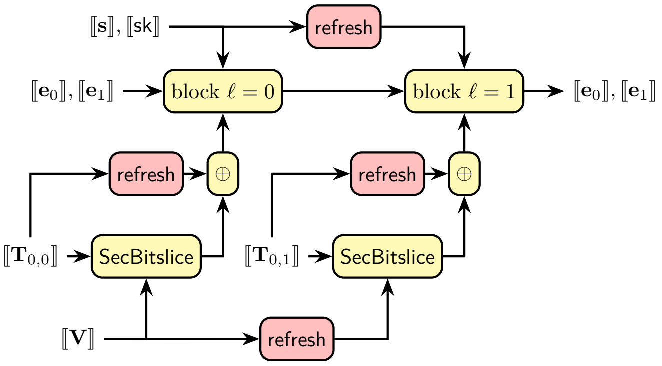
$\llbracket \mathbf{s} \rrbracket, \llbracket \mathsf{sk} \rrbracket$
refresh
$\llbracket \mathbf{e}_0 \rrbracket, \llbracket \mathbf{e}_1 \rrbracket$
block $\ell = 0$
block $\ell = 1$
$\llbracket \mathbf{e}_0 \rrbracket, \llbracket \mathbf{e}_1 \rrbracket$
refresh
$\oplus$
refresh
$\oplus$
$\llbracket \mathbf{T}_{0,0} \rrbracket$
SecBitslice
$\llbracket \mathbf{T}_{0,1} \rrbracket$
SecBitslice
$\llbracket \mathbf{V} \rrbracket$
refresh

Figure 6: Structure of the grey zone gadget

The overall details of the dependencies are presented in Figs. 5 and 6. As illustrated in Fig. 5, 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 $\delta \leq d$ observations on this gadget. Then, we want to prove that all these δ observations can be perfectly simulated with at most δ shares of $\llbracket \mathsf{sk} \rrbracket, \llbracket \mathbf{T}_0 \rrbracket, \llbracket \mathbf{e}_0 \rrbracket, \llbracket \mathbf{e}_1 \rrbracket, \llbracket \mathbf{s} \rrbracket$ and $\llbracket \mathbf{V} \rrbracket$ (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 $\sum_{j=1}^{9} \delta_i \leq \delta \leq d$. 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 $[\![\mathsf{sk}]\!]$ 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 $\delta \leq d$ which concludes the proof. □

5 Performance and experiments

5.1 Implementation

All the gadgets introduced in this paper have been implemented in large and complete C-code. Side-channel attacks are highly dependent on the chip on which the algorithm is executed and it is true that assembly codes are always the best practical 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. The full code will be publicly available for code-checking and reproducibility. You can find it in the supplementary archive.

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. 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.
As presented in Fig. 7, our benchmarks show that while both approaches seem equivalent for one or two shares, a fully dense approach is indeed more relevant for higher orders.

One can conclude from our work that for the moment (except with potentially upcoming new optimizations), the dense representation seems more relevant. We will therefore keep the dense representation for the rest of the benchmarks, as it scales better when the order exceeds or equals 2.

5.2 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, although according to our tests the scaling is the same. Multiple benchmarks were performed and the results are listed in Table 4. We can notice that the performance of the gadgets depends on the performance of the multiplicative gadgets.

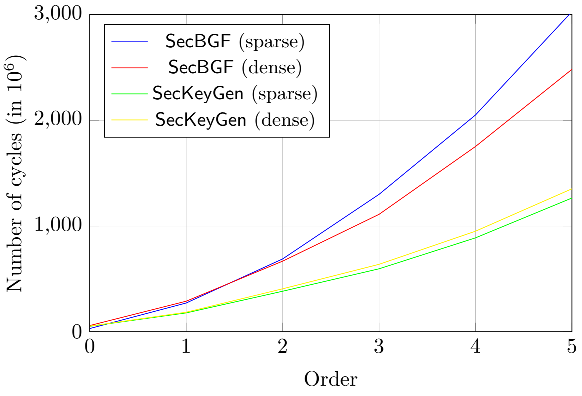

Figure 7: Sparse vs dense, SecBGF and SecKeyGen

Table 4: 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. 9) (sparse)	1	×9	×22.8	×43	×67.9	×100.2
SecBGF(Alg. 9) (dense)	1	×4.9	×11.4	×19	×30	×42.4
SecKeyGen (Alg. 6) (dense)	1	×3.5	×7.7	×12.1	×18	×25.6
SecErrorGen (Alg. 7)	1	×8.6	×22.3	×40.8	×66	×96.2
SecGreyZone(Alg. 22) (sparse)	1	×9	×23.9	×41.9	×67.4	×98.3
SecGreyZone(Alg. 22) (dense)	1	×4.8	×11.2	×18.8	×29.2	×42
SecFisherYates(Alg. 17)	1	×9.5	×18.5	×29.7	×45.7	×66
SecInversion(Alg. 18)	1	×3.5	×7.2	×11	×16.4	×23.6
SecSyndrome(Alg. 16) (sparse)	1	×8	×21.3	×42.3	×63.7	×93
SecSyndrome(Alg. 16) (dense)	1	×3.3	×7.3	×10.8	×15.7	×22.2
SecThreshold(Alg. 19)	1	×8.6	×12.9	×19.1	×30.5	×43.1
SecCounter(Alg. 21)	1	×9.4	×23.1	×42.1	×67.3	×97
SecKaratsuba(Alg. 13)	1	×3.4	×7.4	×11.1	×16.7	×23.4
$\mathsf{SecMult}_{\text{sparsedense}}$(Alg. 14)	1	×8.2	×21.3	×40	×64.8	×95.6

Table 5: 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**

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 performances for masked BIKE (fully-dense) The performances and scaling for the scheme are detailed in Table 5 and Fig. 8.

Remark 7. 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.

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.

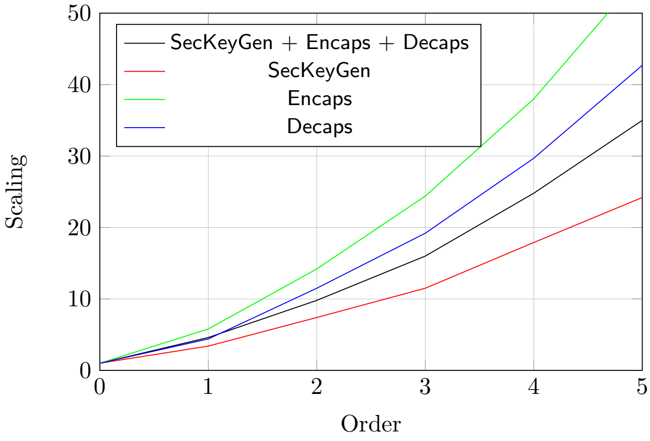

Figure 8: The scaling of masked BIKE (with RNG on)

6 Future Work

TVLA The next step would be to use TVLA verification techniques on our code to check that there are no apparent leaks.

More optimizations 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.

High-order attacks 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.

References

[ABB+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. Technical report, National Institute of Standards and Technology, 2022. available at https://csrc.nist.gov/Projects/post-quantum-cryptography/round-4-submissions.

[ABC+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. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2023(4):58–79, Aug. 2023. URL: https://tches.iacr.org/index.php/TCHES/article/view/11158, doi:10.46586/tches.v2023.i4.58-79.

[Ale03] Michael Alekhnovich. More on average case vs approximation complexity. In *44th FOCS*, pages 298–307. IEEE Computer Society Press, October 2003. doi:10.1109/SFCS.2003.1238204.

[BAA+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. Technical report, National Institute of Standards and Technology, 2019. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions.

[BBD+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 Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 116–129. ACM Press, October 2016. doi:10.1145/2976749.2978427.

[BBE+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 Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part II*, volume 10821 of *LNCS*, pages 354–384. Springer, Heidelberg, April / May 2018. doi:10.1007/978-3-319-78375-8_12.

[BGR+21] Joppe W. Bos, Marc Gourjon, Joost Renes, Tobias Schneider, and Christine van Vredendaal. Masking kyber: First- and higher-order implementations. *IACR TCHES*, 2021(4):173–214, 2021. https://tches.iacr.org/index.php/TCHES/article/view/9064. doi:10.46586/tches.v2021.i4.173-214.

[BMvT78] E. Berlekamp, R. McEliece, and H. van Tilborg. On the inherent intractability of certain coding problems (corresp.). *IEEE Transactions on Information Theory*, 24(3):384–386, 1978. doi:10.1109/TIT.1978.1055873.

[BOG20] Mario Bischof, Tobias Oder, and Tim Güneysu. Efficient Microcontroller Implementation of BIKE. In Emil Simion and Rémi Géraud-Stewart, editors, *Innovative Security Solutions for Information Technology and Communications*, pages 34–49, Cham, 2020. Springer International Publishing. doi:10.1007/978-3-030-41025-4_3.

[CARG23] Agathe Cheriere, Nicolas Aragon, Tania Richmond, and Benoît Gérard. Bike key-recovery: Combining power consumption analysis and information-set decoding, 2023.

[CCK21] Ming-Shing Chen, Tung Chou, and Markus Krausz. Optimizing BIKE for the intel haswell and ARM cortex-M4. *IACR TCHES*, 2021(3):97–124, 2021. https://tches.iacr.org/index.php/TCHES/article/view/8969. doi:10.46586/tches.v2021.i3.97-124.

[CEvMS16] Cong Chen, Thomas Eisenbarth, Ingo von Maurich, and Rainer Steinwandt. Masking large keys in hardware: A masked implementation of McEliece. In

Orr Dunkelman and Liam Keliher, editors, *SAC 2015*, volume 9566 of *LNCS*, pages 293–309. Springer, Heidelberg, August 2016. doi:10.1007/978-3-319-31301-6_18.

[CGKT22] Ming-Shing Chen, Tim Güneysu, Markus Krausz, and Jan Philipp Thoma. Carry-less to BIKE faster. In Giuseppe Ateniese and Daniele Venturi, editors, *ACNS 22*, volume 13269 of *LNCS*, pages 833–852. Springer, Heidelberg, June 2022. doi:10.1007/978-3-031-09234-3_41.

[CGTV15] Jean-Sébastien Coron, Johann Großschädl, Mehdi Tibouchi, and Praveen Kumar Vadnala. Conversion from arithmetic to Boolean masking with logarithmic complexity. In Gregor Leander, editor, *FSE 2015*, volume 9054 of *LNCS*, pages 130–149. Springer, Heidelberg, March 2015. doi:10.1007/978-3-662-48116-5_7.

[CGV14] Jean-Sébastien Coron, Johann Großschädl, and Praveen Kumar Vadnala. Secure conversion between Boolean and arithmetic masking of any order. In Lejla Batina and Matthew Robshaw, editors, *CHES 2014*, volume 8731 of *LNCS*, pages 188–205. Springer, Heidelberg, September 2014. doi:10.1007/978-3-662-44709-3_11.

[Cho16] Tung Chou. QcBits: Constant-time small-key code-based cryptography. In Benedikt Gierlichs and Axel Y. Poschmann, editors, *CHES 2016*, volume 9813 of *LNCS*, pages 280–300. Springer, Heidelberg, August 2016. doi:10.1007/978-3-662-53140-2_14.

[CJRR99] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In Michael J. Wiener, editor, *CRYPTO'99*, volume 1666 of *LNCS*, pages 398–412. Springer, Heidelberg, August 1999. doi:10.1007/3-540-48405-1_26.

[Cor14] Jean-Sébastien Coron. Higher order masking of look-up tables. In Phong Q. Nguyen and Elisabeth Oswald, editors, *EUROCRYPT 2014*, volume 8441 of *LNCS*, pages 441–458. Springer, Heidelberg, May 2014. doi:10.1007/978-3-642-55220-5_25.

[Cor17] Jean-Sébastien Coron. High-order conversion from Boolean to arithmetic masking. In Wieland Fischer and Naofumi Homma, editors, *CHES 2017*, volume 10529 of *LNCS*, pages 93–114. Springer, Heidelberg, September 2017. doi:10.1007/978-3-319-66787-4_5.

[CPRR14] Jean-Sébastien Coron, Emmanuel Prouff, Matthieu Rivain, and Thomas Roche. Higher-order side channel security and mask refreshing. In Shiho Moriai, editor, *FSE 2013*, volume 8424 of *LNCS*, pages 410–424. Springer, Heidelberg, March 2014. doi:10.1007/978-3-662-43933-3_21.

[DDF14] Alexandre Duc, Stefan Dziembowski, and Sebastian Faust. Unifying leakage models: From probing attacks to noisy leakage. In Phong Q. Nguyen and Elisabeth Oswald, editors, *EUROCRYPT 2014*, volume 8441 of *LNCS*, pages 423–440. Springer, Heidelberg, May 2014. doi:10.1007/978-3-642-55220-5_24.

[DG19] Nir Drucker and Shay Gueron. A toolbox for software optimization of QC-MDPC code-based cryptosystems. *Journal of Cryptographic Engineering*, 9(4):341–357, November 2019. doi:10.1007/s13389-018-00200-4.

[DGK20] Nir Drucker, Shay Gueron, and Dusan Kostic. QC-MDPC decoders with several shades of gray. In Jintai Ding and Jean-Pierre Tillich, editors, *Post-Quantum Cryptography - 11th International Conference, PQCrypto 2020*, pages 35–50. Springer, Heidelberg, 2020. doi:10.1007/978-3-030-44223-1_3.

[DHP+22] Jan-Pieter D'Anvers, Daniel Heinz, Peter Pessl, Michiel Van Beirendonck, and Ingrid Verbauwhede. Higher-order masked ciphertext comparison for lattice-based cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(2):115–139, Feb. 2022. URL: https://tches.iacr.org/index.php/TCHES/article/view/9483, doi:10.46586/tches.v2022.i2.115-139.

[DKR+20] Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, Frederik Vercauteren, Jose Maria Bermudo Mera, Michiel Van Beirendonck, and Andrea Basso. SABER. Technical report, National Institute of Standards and Technology, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.

[FO99] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In Michael J. Wiener, editor, *CRYPTO'99*, volume 1666 of *LNCS*, pages 537–554. Springer, Heidelberg, August 1999. doi:10.1007/3-540-48405-1_34.

[GAB19] Antonio Guimarães, Diego F. Aranha, and Edson Borin. Optimized implementation of QC-MDPC code-based cryptography. *Concurrency and Computation: Practice and Experience*, 31(18):e5089, 2019. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.5089, doi:10.1002/cpe.5089.

[Gal62] R. Gallager. Low-density parity-check codes. *IRE Transactions on Information Theory*, 8(1):21–28, 1962. doi:10.1109/TIT.1962.1057683.

[GHJ+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. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(3):223–263, Jun. 2022. URL: https://tches.iacr.org/index.php/TCHES/article/view/9700, doi:10.46586/tches.v2022.i3.223-263.

[GJS16] Qian Guo, Thomas Johansson, and Paul Stankovski. A key recovery attack on MDPC with CCA security using decoding errors. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *ASIACRYPT 2016, Part I*, volume 10031 of *LNCS*, pages 789–815. Springer, Heidelberg, December 2016. doi:10.1007/978-3-662-53887-6_29.

[GR20] François Gérard and Mélissa Rossi. *An Efficient and Provable Masked Implementation of qTESLA*, pages 74–91. Springer International Publishing, 03 2020. doi:10.1007/978-3-030-42068-0_5.

[HHK17] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. A modular analysis of the Fujisaki-Okamoto transformation. In Yael Kalai and Leonid Reyzin, editors, *TCC 2017, Part I*, volume 10677 of *LNCS*, pages 341–371. Springer, Heidelberg, November 2017. doi:10.1007/978-3-319-70500-2_12.

[ISW03] Yuval Ishai, Amit Sahai, and David Wagner. Private circuits: Securing hardware against probing attacks. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 463–481. Springer, Heidelberg, August 2003. doi:10.1007/978-3-540-45146-4_27.

[KDVB$^+$22] Suparna Kundu, Jan-Pieter D'Anvers, Michiel Van Beirendonck, Angshuman Karmakar, and Ingrid Verbauwhede. Higher-order masked saber. In Clemente Galdi and Stanislaw Jarecki, editors, *Security and Cryptography for Networks*, pages 93–116, Cham, 2022. Springer International Publishing.

[KLRBG22] Markus Krausz, Georg Land, Jan Richter-Brockmann, and Tim Güneysu. Efficiently masking polynomial inversion at arbitrary order. In Jung Hee Cheon and Thomas Johansson, editors, *Post-Quantum Cryptography*, pages 309–326, Cham, 2022. Springer International Publishing.

[KLRBG23] Markus Krausz, Georg Land, Jan Richter-Brockmann, and Tim Güneysu. A holistic approach towards side-channel secure fixed-weight polynomial sampling. In Alexandra Boldyreva and Vladimir Kolesnikov, editors, *Public-Key Cryptography – PKC 2023*, pages 94–124, Cham, 2023. Springer Nature Switzerland.

[LDK$^+$22] Vadim Lyubashevsky, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Peter Schwabe, Gregor Seiler, Damien Stehlé, and Shi Bai. CRYSTALS-DILITHIUM. Technical report, National Institute of Standards and Technology, 2022. available at https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022.

[Lem19] Daniel Lemire. Fast random integer generation in an interval. *ACM Transactions on Modeling and Computer Simulation*, 29(1):1–12, jan 2019. URL: https://doi.org/10.1145%2F3230636, doi:10.1145/3230636.

[MOG15] Ingo Von Maurich, Tobias Oder, and Tim Güneysu. Implementing QC-MDPC McEliece Encryption. *ACM Trans. Embed. Comput. Syst.*, 14(3), April 2015. doi:10.1145/2700102.

[MTSB13] Rafael Misoczki, Jean-Pierre Tillich, Nicolas Sendrier, and Paulo S. L. M. Barreto. Mdpc-mceliece: New mceliece variants from moderate density parity-check codes. In *2013 IEEE International Symposium on Information Theory*, pages 2069–2073, 2013. doi:10.1109/ISIT.2013.6620590.

[Nie86] H. Niederreiter. Knapsack-type cryptosystems and algebraic coding theory. *Problems of Control and Information Theory*, 15(2):159–166, 1986.

[RHHM17] Melissa Rossi, Mike Hamburg, Michael Hutter, and Mark E. Marson. A side-channel assisted cryptanalytic attack against QcBits. In Wieland Fischer and Naofumi Homma, editors, *CHES 2017*, volume 10529 of *LNCS*, pages 3–23. Springer, Heidelberg, September 2017. doi:10.1007/978-3-319-66787-4_1.

[RP10] Matthieu Rivain and Emmanuel Prouff. Provably secure higher-order masking of AES. In Stefan Mangard and François-Xavier Standaert, editors, *CHES 2010*, volume 6225 of *LNCS*, pages 413–427. Springer, Heidelberg, August 2010. doi:10.1007/978-3-642-15031-9_28.

[SAB$^+$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. Technical report, National Institute of Standards and Technology, 2022. available at https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022.

[Sen21] Nicolas Sendrier. Secure sampling of constant-weight words ? application to bike. Cryptology ePrint Archive, Report 2021/1631, 2021. https://eprint.iacr.org/2021/1631.

[SKC+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. *IACR TCHES*, 2019(4):180–212, 2019. https://tches.iacr.org/index.php/TCHES/article/view/8349. doi:10.13154/tches.v2019.i4.180-212.

[vMG14] Ingo von Maurich and Tim Güneysu. Towards side-channel resistant implementations of QC-MDPC McEliece encryption on constrained devices. In Michele Mosca, editor, *Post-Quantum Cryptography - 6th International Workshop, PQCrypto 2014*, pages 266–282. Springer, Heidelberg, October 2014. doi:10.1007/978-3-319-11659-4_16.

[vMHG16] Ingo von Maurich, Lukas Heberle, and Tim Güneysu. IND-CCA secure hybrid encryption from QC-MDPC niederreiter. In Tsuyoshi Takagi, editor, *Post-Quantum Cryptography - 7th International Workshop, PQCrypto 2016*, pages 1–17. Springer, Heidelberg, 2016. doi:10.1007/978-3-319-29360-8_1.

A Extra small gadgets and proofs

This part is about extra small gadgets and their proofs.

A.1 Adders and carries

Algorithm 23 Half Adder

```
Require: $[\![x]\!] \in \mathbb{F}_2$, $[\![y]\!] \in \mathbb{F}_2$
Ensure: $[\![z]\!] = [\![x]\!] + [\![y]\!] \in \mathbb{F}_2$, $[\![c]\!] \in \mathbb{F}_2$ the carry
$[\![z]\!] \leftarrow [\![x]\!] \oplus [\![y]\!]$
$[\![c]\!] \leftarrow \mathsf{sec}_{\&}([\![x]\!], [\![y]\!])$
return $[\![z]\!], [\![c]\!]$
```

Algorithm 24 Adder

```
Require: $[\![x]\!] \in \mathbb{F}_2$, $[\![y]\!] \in \mathbb{F}_2$, $[\![c_0]\!] \in \mathbb{F}_2$
Ensure: $[\![z]\!] = [\![x]\!] + [\![y]\!] + [\![c_0]\!] \in \mathbb{F}_2$, $[\![c]\!] \in \mathbb{F}_2$ the carry
$([\![t]\!], [\![s]\!]) \leftarrow \mathsf{SecHalf\_Adder}([\![x]\!], [\![y]\!])$
$([\![z]\!], [\![u]\!]) \leftarrow \mathsf{SecHalf\_Adder}([\![t]\!], [\![c_0]\!])$
$[\![c]\!] \leftarrow [\![s]\!] \oplus [\![u]\!]$
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 16. *The half adder algorithm is $d - \mathsf{NI}$.*

Since the adder uses only two calls to $\mathsf{SecHalf\_Adder}$ (itself $d - \mathsf{NI}$), handling different variables, we can infer the following.

Theorem 17. *The adder algorithm is $d - \mathsf{NI}$.*

A.2 Equality

The gadget $\mathrm{sec}_=$ 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 $\mathsf{sec}_{\&}$ between the negation of each obtained bit. Such a procedure has been outlined in the literature e.g. in [DHP+22].

We decided to optimize it in such a way as to dichotomize the operations about the word binary, and thus achieve better performance. Given that the only operation manipulating the data is $\mathsf{sec}_{\&}$, and that it is a d-SNI function, we can deduce that the algorithm is d-NI.

Algorithm 25 Masked equality ($\mathrm{sec}_{=}$)

```
Require: $[\![\mathbf{x}]\!] \in \mathbb{F}_2^n, [\![\mathbf{y}]\!] \in \mathbb{F}_2^n, n$ a power of 2
Ensure: $[\![z]\!] \in \mathbb{F}_2$ equals 0 if $\mathbf{x} = \mathbf{y}$ and 1 if not
$[\![\mathbf{z}]\!] \leftarrow [\![\mathbf{x}]\!] \oplus [\![\mathbf{y}]\!]$
for $i \leftarrow \frac{n}{2}$ to 1 step $-\frac{i}{2}$ do
    $[\![\mathbf{a}]\!] \leftarrow \mathsf{left}([\![\mathbf{z}]\!])$                ▷ Cut in length
    $[\![\mathbf{b}]\!] \leftarrow \mathsf{right}([\![\mathbf{z}]\!])$               ▷ Cut in length
    $[\![\mathbf{a}]\!]_0 \leftarrow \neg [\![\mathbf{a}]\!]_0$                       ▷ Coefficient-wise not
    $[\![\mathbf{b}]\!]_0 \leftarrow \neg [\![\mathbf{b}]\!]_0$                       ▷ Coefficient-wise not
    $[\![\mathbf{z}]\!] \leftarrow \mathsf{sec}_{\&}([\![\mathbf{a}]\!], [\![\mathbf{b}]\!])$   ▷ Coefficient-wise $\mathsf{sec}_{\&}$
    $[\![\mathbf{z}]\!]_0 \leftarrow \neg [\![\mathbf{z}]\!]_0$                       ▷ Coefficient-wise not
end for
return $[\![\mathbf{z}_0]\!]$
```

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 18. *The equality algorithm is d-NI.*

A.3 Multiplications

Algorithm 26 Partly masked multiplication ($\mathsf{SecMult}_{\mathrm{partlymasked}}$)

```
Require: $[\![x]\!] \in \mathbb{Z}_n$, $y \in \mathbb{Z}$
Ensure: $[\![z]\!] = [\![x]\!] \cdot y \in \mathbb{Z}_n$
$[\![z]\!] \leftarrow \mathsf{zero\_masking}()$
$[\![t]\!] \leftarrow [\![x]\!]$
for $i \leftarrow 0$ to $\lfloor \log_2(y) \rfloor$ do
    if $y[i] = 1$ then                                        ▷ $y[i] = (y \gg i)\ \&\ 1$
        $[\![z]\!] \leftarrow \mathsf{sec}_{+}([\![z]\!], [\![t]\!])$
        $[\![t]\!] \leftarrow \mathsf{refresh}([\![t]\!])$
    end if
    $[\![t]\!] \leftarrow [\![t]\!] \ll 1$
end for
return $[\![z]\!]$
```

Theorem 19. *The partly masked multiplication algorithm is $d - \mathsf{NI}$.*

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 □

A.4 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], which allows us to draw an

integer between 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 26) and the shift, which is a linear operation.

Remark 8. 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 27 Modular random number ($\mathsf{sec}_{\mathsf{rand}}$)

```
Require: $n \in \mathbb{N}^*, p \in \mathbb{N}^*, 2^p \geq n$
Ensure: $[\![r]\!] \xleftarrow{\$} \mathbb{Z}_n$
$[\![r]\!] \xleftarrow{\$} \mathbb{F}_{2^p}$                                ▷ Draw $p$ bits on each share
$[\![r]\!] \leftarrow \mathsf{SecMult}_{\text{partlymasked}}([\![r]\!], n)$
$[\![r]\!] \leftarrow [\![r]\!] \gg p$                                  ▷ Shift on each share
return $[\![r]\!]$
```

Theorem 20. *The modular random number Algorithm 27 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. □

Algorithm 28 $\mathsf{SecPolymul}$: Naive Polynomial multiplication (parameterized by B, the size of its inputs)

```
Require: $[\![\mathbf{x}]\!] \in \mathbb{F}_2^B$, $[\![\mathbf{y}]\!] \in \mathbb{F}_2^B$
Ensure: $[\![\mathbf{z}]\!] = [\![\mathbf{x}]\!] \cdot [\![\mathbf{y}]\!] \in \mathbb{F}_2^{2B}$
for $i \leftarrow 0$ to $B-1$ do
    for $j \leftarrow 0$ to $B-1$ do
        $[\![u]\!] \leftarrow \mathsf{sec}_{\&}([\![\mathbf{x}_i]\!], [\![\mathbf{y}_j]\!])$
        $[\![\mathbf{z}_{(i+j)}]\!] \leftarrow [\![\mathbf{z}_{(i+j)}]\!] \oplus [\![u]\!]$
    end for
end for
return $[\![\mathbf{z}]\!]$
```

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

Theorem 21. *The polynomial multiplication* $\mathsf{SecPolymul}$ *parametered with B algorithm is* $d-\mathsf{NI}$.

A.5 Cyclic shift

This is a masked version of the barrel shifter algorithm.
We define $\mathsf{SecCyclic\_Shift}$ the function that allows to shift a masked polynomial with a public value. As it is only a linear operation, it is safe and not a concern.

Theorem 22. *The secure cyclic shift algorithm is* $d-\mathsf{NI}$.

Algorithm 29 Secure masked cyclic shift ($\mathsf{sec}_{\gg}$)

```
Require: $[\![\mathbf{x}]\!] \in \mathbb{F}_2^n$, $[\![s]\!] \in \mathbb{N}$
Ensure: $[\![\mathbf{y}]\!] = [\![\mathbf{x}]\!] \gg [\![s]\!] \in \mathbb{F}_2^n$
$[\![\mathbf{y}]\!] \leftarrow [\![\mathbf{x}]\!]$
for $i \leftarrow 0$ to $|n|$ do                                  ▷ $|n| = \lfloor \log_2(n) \rfloor$
    $[\![v]\!] \leftarrow [\![s]\!][i]$                          ▷ $[\![s]\!][i] = ([\![s]\!] \gg i)$ & 1
    $[\![\mathbf{t}]\!] \leftarrow \mathsf{SecCyclic\_Shift}([\![\mathbf{y}]\!], 2^i)$
    for $j \leftarrow 0$ to $n-1$ do
        $[\![s1]\!] \leftarrow \mathsf{sec}_{\&}([\![\mathbf{t}_j]\!], [\![v]\!])$
        $[\![s2]\!] \leftarrow \mathsf{sec}_{\&}([\![\mathbf{y}_j]\!], \neg[\![v]\!])$
        $[\![\mathbf{y}_j]\!] \leftarrow [\![s1]\!] \oplus [\![s2]\!]$
    end for
end for
return $[\![\mathbf{y}]\!]$
```

Proof. In the most imbricated 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 j loop, all of this is d-NI.
So their sequential combination leads to a d-NI algorithm. □

A.6 Masked conditional branch

Algorithm 30 Choose value ($\mathsf{sec}_{\mathsf{if}}$)

```
Require: $[\![a]\!] \in \mathbb{Z}_n$, $[\![b]\!] \in \mathbb{Z}_n$, $[\![t]\!] \in \mathbb{F}_2$
Ensure: $[\![a]\!]$ if $[\![t]\!] = 1$, $[\![b]\!]$ otherwise
$[\![c]\!] \leftarrow \mathsf{sec}_{\&}^{\mathbf{bitwise}}([\![a]\!], [\![t]\!])$   ▷ Bitwise $\mathsf{sec}_{\&}$ between all bits of $a$ and the single bit of $t$
$[\![t]\!]_0 \leftarrow \neg[\![t]\!]_0$
$[\![d]\!] \leftarrow \mathsf{sec}_{\&}^{\mathbf{bitwise}}([\![b]\!], [\![t]\!])$   ▷ Bitwise $\mathsf{sec}_{\&}$ between all bits of $b$ and the single bit of $t$
return $[\![c]\!] \oplus [\![d]\!]$                              ▷ Coefficient-wise XOR
```

Since $\mathsf{sec}_{\&}^{\mathbf{bitwise}}$ is just a succession of $\mathsf{sec}_{\&}$ which is d-SNI, $\mathsf{sec}_{\&}^{\mathbf{bitwise}}$ is also d-SNI property. Since the last $\oplus$ is d-NI, we deduce the theorem below.

Theorem 23. *The choose value algorithm is $d-$ NI.*

A.7 Masked maximum computation

Algorithm 31 Max ($\mathsf{sec}_{\mathsf{max}}$)

```
Require: $[\![a]\!] \in \mathbb{Z}_n$, $[\![b]\!] \in \mathbb{Z}_n$
Ensure: $[\![c]\!] = \mathsf{sec}_{\mathsf{max}}([\![a]\!], [\![b]\!]) \in \mathbb{Z}_n$
$[\![t]\!] \leftarrow \mathsf{sec}_{+}([\![a]\!], [\![-b]\!])$
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 24. *The max algorithm is $d-$ NI.*

A.8 Filling a matrix in masked form

Algorithm 32 Fill matrix ($\mathsf{sec}_{\mathsf{fill}}$)

```
Require: $[\![v]\!] \in \mathbb{Z}_n$
Ensure: $[\![\mathbf{X}]\!] \in \mathbb{F}_2^{k\times(|n|+1)}$ a matrix filled with the binary representation of $[\![v]\!]$
for $i \leftarrow 0$ to $k-1$ do
    for $j \leftarrow 0$ to $|n|$ do
        $[\![\mathbf{X}_{i,j}]\!] \leftarrow [\![v]\!][j]$
    end for
    $[\![v]\!] \leftarrow \mathsf{refresh}([\![v]\!])$
end for
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 25. *The fill algorithm is $d-\mathsf{NI}$.*