High-Order Masking of BIKE

Matthias Trannoy[1,2]

[1] University of Luxembourg
[2] IDEMIA, Cryptography & Security Labs, Courbevoie, France
matthias.trannoy@idemia.com

Abstract. Every cryptographic implementation on embedded device is vulnerable to side-channel attacks. To prevent these attacks, the main countermeasure consists in splitting each sensitive variable in shares and processing them independently. With the upcoming of new algorithms designed to resist quantum computers and the complexity of their operations, this protection represents a real challenge. In this article, we present an attack on an earlier attempt to protect the decoder of BIKE cryptosystem against first-order attack. Additionally, we introduce a new procedure for the high-order masking of the decoder, up-to-date with its latest improvement. We also present the first fully masked implementation of the whole cryptosystem, including the key generation and the encapsulation. Eventually, to assess the correctness of our countermeasures and initiate further comparison, we implemented our countermeasures in `C` and provide benchmarks of their performance.

1 Introduction

Post-Quantum Cryptography. The emergence of quantum computers poses a significant security risk to our existing methods of communication. Currently, the security of our main asymmetric cryptographic systems, relying on RSA and Elliptic Curve cryptography, is at stake. This concern arises from Shor's development of efficient algorithms capable of solving the factorization and discrete logarithm problems on a scalable quantum computer. To address this impending threat, the National Institute of Standards and Technology (NIST) initiated a standardization process in 2016 to identify new standards for asymmetric cryptography encompassing encryption and signatures. This competition eventually reached an important milestone in 2022 with the adoption of Kyber as the standard for Key Encapsulation Mechanism (KEM).

Moreover, NIST has initiated a fourth round with the objective of selecting a KEM based on Error-correcting codes in addition to the lattice-based Kyber scheme. This supplementary round aims to further enhance the security and resilience of the chosen algorithms against potential quantum attacks.

BIKE Key Encapsulation Mechanism. BIKE, BIt-flipping KEy, is an improvement of the original McEliece scheme. Its security relies on the hardness of decoding erroneous codewords in an arbitrary error correcting code. Additionally, BIKE employs more structured code than McEliece, which allows optimization of the computation involved, along with much more compact keys and ciphertext sizes. The main idea of the algorithm is to use two representations of an error correcting code: a public one for which correcting errors is hard, and a secret one that allows efficient error correction.

Side-channel Vulnerabilities. When implementing cryptography on embedded devices, it is well-known that implementations can suffer from attacks that are not considered in the security model in which BIKE security is proven. Namely, those peculiar attacks make use of physical leakage happening during the execution of the algorithm to retrieve sensitive intermediate variables that a classic attacker could not access. With the emergence of more and more powerful attacks using template attacks and deep-learning assistance, those attacks aim to extend their range of targets and attack CPU implementations.

Masking Countermeasures. To prevent these attacks, the generic countermeasure is masking. Given a sensitive variable x, the idea of masking is to blind x into $x \oplus r$ with a uniform random mask r and perform operations on $x \oplus r$ and r separately. Therefore, any leakage on one of the shares $x \oplus r$ or r is decorrelated from the sensitive data x. Formally, we model the adversary to be able to probe up to t

intermediate variables during the execution of the algorithm. When $t \geq 2$, masking with 2 shares is not sufficient. To ensure t-probing security, we need to split sensitive data into at least $t+1$ shares, with t uniform random masks. In 2003, a generic compiler was described in [ISW03]. In their work, they showed how to transform any Boolean circuit into t-probing secure with $2t+1$ masks and with a factor of $\mathcal{O}(t^2)$ on the number of gates. Later, with the introduction of the Non-Interference framework [BBD+16], the authors showed how to reduce the number of masks to t. They developed a framework in which the security of a whole circuit can be broken down into the security of individual parts by introducing stricter notions of security, the Non-Interference (NI) and Strong Non-Interference (SNI), along with composability theorems. While theoretically, any circuit can be compiled into an equivalent t-probing secure one, the conversion is in practice quite inefficient. Particularly in the case of post-quantum cryptography, it represents a challenge due to the complexity of the operations involved.

State of the Art. To the best of our knowledge, the only article dealing with the masking of the BIKE scheme is [CEvMS16]. This article initiated the research on masking BIKE. However, we note the following points. On the one hand, the article only deals with the masking of the decoding part of the algorithm and not the complete scheme. This motivated our work since the decoder has been tweaked in the past 7 years and the countermeasure does not directly apply. Additionally, in the case of Key Encapsulation, we also need to mask the encryption, and as recently shown, new template attacks can target the key generation directly. This shows the necessity of masking the whole scheme. On the other hand, this article's countermeasure only claims to achieve first-order security, and as we will show in our article, their implementation actually suffers from a 1st-order attack.

Concurrent Work. Concurrently to our paper a recent e-print dealing with BIKE high-order masking was published [DR24]. While our article shares many similarities with the work of [DR24], we diverge by proposing an alternative approach to sparse representation, leveraging the inherent structure of the ring $\mathbb{Z}_2[X]/(X^r-1)$. This alternate representation facilitates a significantly more efficient conversion between sparse and dense representations of masked polynomials. Furthermore, alongside our proposed countermeasures, we unveil an attack targeting a previous endeavor to protect BIKE cryptosystem against side-channel attacks, providing a comprehensive examination of security measures and vulnerabilities.

Our Contributions. In this paper we present a high-order description of BIKE cryptosystem, a NIST fourth round candidate. More precisely, we provide high-order algorithm of each step of the cryptosystem, including the key generation, the encapsulation and the decapsulation. Our countermeasures are proven secure against t-probing attacker with the methodology introduced by [BBD+16]. The main challenge is the high-order masking of the decoder of BIKE error correcting codes. We review an earlier attempt to secure the decoder against first order attack with threshold implementation methodology [CEvMS16] against which we present a first order attack using the unmasking of an intermediate variable, the UPC. Additionally, our countermeasure focuses on the high-order security (not only first-order security) and is up-to-date to the latest refinement of the decoder. Regarding the key generation and the encapsulation, we present a high-order countermeasure for the fixed-weight sampler of BIKE. In particular, we introduce a new representation of masked polynomial of fixed-weight altogether with this sparse representation, we also describe a procedure to convert this new representation into the usual arithmetic masked representation of polynomials.

Eventually, we provide a performance analysis of our countermeasure with a proof-of-concept implementation made in `C` and run on an `AMD Ryzen` CPU.

2 Definitions and Notations

2.1 Notations

In this section we detail the notations and parameters we will use across the article.

1. System Parameters:
   - r (block length): a prime number such that 2 is of maximal order in $\mathbb{Z}_r^\times$.
   - w (row weight): an even positive integer such that $w/2$ is odd.
   - t (error weight): a positive integer.
   - ℓ (shared secret size): a positive integer.
2. Space sets:
   - $\mathcal{M} = \{0,1\}^\ell$: the message space.
   - $\mathcal{K} = \{0,1\}^\ell$: the shared secret space.
   - $\mathcal{R} = \mathbb{F}_2[X]/(X^r - 1)$: cyclic polynomial ring.
   - $\mathcal{H}_w = \{(h_0, h_1) \in \mathcal{R}^2 \mid |h_0| = |h_1| = w/2\}$: the private key space.
   - $\mathcal{E}_t = \{(e_0, e_1) \in \mathcal{R}^2 \mid |e_0| + |e_1| = t\}$: the error space.
3. Hash Functions:
   - $\mathbf{H} : \mathcal{M} \mapsto \mathcal{E}_t$: Instantiated as the `EOF SHAKE256`
   - $\mathbf{K} : \mathcal{M} \times \mathcal{R} \times \mathcal{M} \mapsto \mathcal{K}$: Instantiated as $\texttt{SHA3} - \texttt{384}$
   - $\mathbf{L} : \mathcal{R}^2 \mapsto \mathcal{M}$: Instantiated as $\texttt{SHA3} - \texttt{384}$

2.2 Code Definitions

The BIKE scheme is built upon Quasi-Cyclic Moderate Density Parity Check McEliece, QC-MDPC.

In the BIKE scheme, error correcting codes are characterized by their parity-check matrices. For a parity check matrix $H \in \mathbb{F}_2^{n_1 \times n_2}$, a codeword $c \in \mathbb{F}_2^{n_2}$ belongs to the code if and only if its syndrome $s = H \cdot c \in \mathbb{F}_2^{n_1}$ is zero.

The codes used in BIKE are more structured than the initial McEliece, notably being Quasi-Cyclic.

Definition 1 (Quasi-Cyclic Code). *An error correcting code is termed Quasi-Cyclic if there exists an integer n_0 such that for all codewords c, the cyclic shift $\mathsf{shift}_{n_0}(c)$ by n_0 places is also a code word.*

This definition implies, in particular, that the Parity-Check Matrix H can be expressed in blocks as $H = [H_0 \mid \cdots \mid H_{n_0-1}]$, where each sub-matrix $H_i \in \mathbb{F}_2^{n_1 \times n_1}$ is a square circulant. It also implies that $n_2 = n_0 \times n_1$.

In the case of BIKE, $n_0 = 2$, signifying that the parity-check matrix can be expressed as $H = [H_0 \mid H_1] \in \mathbb{F}_2^{r \times 2r}$, where r is the block length parameter of the scheme.

Definition 2 (Circulant Matrix). *A matrix $H \in \mathbb{F}_2^{r \times r}$ is deemed circulant if and only if it can be expressed as:*

$$H = \begin{pmatrix} a_1 & a_2 & \cdots & a_n \\ a_n & a_1 & \cdots & a_{n-1} \\ \vdots & \ddots & \ddots & \vdots \\ a_2 & a_3 & \cdots & a_1 \end{pmatrix}$$

Circulant matrices can be succinctly represented by their first row $a_1, \ldots, a_n$. Additionally, the sets of circulant matrices are isomorphic to the set of polynomials $\mathbb{F}_2[X]/(X^r - 1) = \mathcal{R}$. Hence, the parity-check matrix H in the BIKE scheme can be represented as a pair of polynomials (h_0, h_1).

Finally, the secret matrix H exhibits moderate density, enabling a more compact representation of the secret key by storing only the positions of the non-zero coefficients. This density is also a prerequisite for the decoding algorithm described in Section 3.1 to function.

Definition 3 (Moderate Density Parity Check Code). *A quasi-cyclic error correcting code is deemed as moderate density parity check if and only if it the row weight of the parity-check matrix is $\mathcal{O}(\sqrt{r})$.*

In the case of BIKE, the secret key (h_0, h_1) has a row weight of w. Specifically, each polynomial h_0 and h_1 has the same weight $w/2$. Since 2 is of maximal order in $\mathbb{Z}_r^\times$ each polynomial h_0 and h_1 is invertible in $\mathbb{Z}_2[X]/(\Phi_r)$, where Φ_r is the r-th cyclotomic polynomial. Additionally, $w/2$ is odd, it ensures that both h_0 and h_1 are invertible in $\mathbb{Z}_2[X]/(X+1)$ polynomials. From the Chinese Remainders Theorems, the choice of parameters garantees that both h_0 and h_1 are invertible polynomials in $\mathbb{Z}_2[X]/(X^r+1)$

Throughout the remainder of the article, we will favor the polynomial notation due to its more compact form.

2.3 Non-Interference Security Notions

To formalize the security properties of our masking countermeasures, we recall two key notions introduced in [BBD+16]: Non-Interference (t-NI) security and Strong Non-Interference (t-SNI) security.

Definition 4 (t-NI Security). *Consider a gadget G taking n input shares $(a_1, \ldots, a_n)$ and producing n output shares $(b_1, \ldots, b_n)$. The gadget G is t-NI secure if, for any set of t_1 intermediate variables and any subset $O \subset [1, n]$ of output indices such that $t_1 + |O| \leq t$, there exists a subset of input indices $I \subset [1, n]$ such that the t_1 intermediate variables and the outputs $b_{|O}$ can be perfectly simulated from $a_{|I}$, with $|I| \leq t_1$.*

Definition 5 (t-SNI Security). *Let G be a gadget taking n input shares $(a_1, \ldots, a_n)$ and outputting n shares $(b_1, \ldots, b_n)$. The gadget G is t-SNI secure if, for any set of t_1 intermediate variables and any subset $O \subset [1, n]$ of output indices such that $t_1 + |O| \leq t$, there exists a subset of input indices $I \subset [1, n]$ such that the t_1 intermediate variables and the outputs $b_{|O}$ can be perfectly simulated from $a_{|I}$, with $|I| \leq t_1$.*

These security notions imply the t-probing security of [ISW03]. Therefore, it ensures that an adversary capable of probing up to t variable cannot retrieve any sensitive data. Moreover, the authors of [BBD+16] additionally proved composability theorems that allows to break down the security of an algorithm by proving the security of its individual parts. The two security notions differs in the following sense.

The t-NI security ensures that the total number of input shares required for simulation is bounded by the total number of probes and outputs combined.

On the other hand, the Strong Non-Interference (t-SNI) security modifies this requirement, limiting the total number of input shares by the number of internal probes only.

These security notions were extended in work by [BBE+18], who introduced the Non-Interference with public output (t-NIo) security.

Definition 6 (t-NIo Security). *Let G be a gadget taking input $(x_i)_{1 \leq i \leq n}$ and outputting b. The gadget G is t-NIo secure if, for any set of $t_1 \leq t$ intermediate variables, there exists a subset I of input indices with $|I| \leq t_1$, such that the t_1 intermediate variables can be perfectly simulated from $x_{|I}$ and b.*

The t-NIo security is particularly relevant in scenarios like encryption and encapsulation, where the algorithm processes sensitive data (e.g., message and session key), but part of the output (e.g., ciphertext) needs to be unmasked.

3 Description of BIKE: Bit Flipping Key

This section provides an overview of the BIKE Key Encapsulation Mechanism, an enhancement of the original Code-Based McEliece. The BIKE scheme leverages structured codes to achieve improved performance with significantly reduced key sizes. Specifically, it capitalizes on the Quasi-Cyclic (QC) code structure to compress matrices, benefiting from faster computations due to the isomorphism to polynomial quotient rings.

The core idea of the BIKE scheme revolves around the difficulty of decoding errors in arbitrary quasi-cyclic codes. The secret key is a low-weight parity-check matrix $sk = (h_0, h_1) \in (\mathbb{F}_2[X]/X^r - 1)^\star$. The public key $pk = (1, h)$ is derived from the secret key, where $h = h_1 \cdot h_0^{-1}$. Encoding a message involves introducing

an error (e_0, e_1), with syndromes computed in both the public code ($s_{pub} = 1 \cdot e_0 + h \cdot e_1$) and the secret code ($s_{priv} = h_0 \cdot e_0 + h_1 \cdot e_1$). Notably, the private syndrome can be deduced from the public one using the private row h_0, as shown by the equation:

$$h_0 \cdot s_{pub} = h_0 e_0 + h_0 h e_1 = h_0 e_0 + h_1 e_1 = s_{priv}$$

The ciphertext then corresponds to the tuple $(c_0, c_1) = (s_{pub}, m \oplus \mathbf{L}(e_0, e_1))$, where c_0 is the public syndrome s_{pub} and c_1 the message blinded by a hash of the error, $m \oplus \mathbf{L}(e_0, e_1)$.

The security of BIKE then relies on the hardness of retrieving low-weight h_0 and h_1 from h and the difficulty of decoding the error given the public code representation and the public syndrome s_{pub}.

In order to decrypt a ciphertext, one needs to first decode the error vector from the public syndrome c_0 and then recompute the mask of the message in c_1. Decoding the error (e_0, e_1) from the private syndrome and the private key (h_0, h_1) is achieved using the Bit-Flipping decoding algorithm introduced by Galager in [Gal62]. This algorithm's efficiency and correctness depend on the quasi-cyclic structure of the code and the low-weight of the parity-check matrix. To address potential decoding failures, the algorithm was refined in [DGK19] to minimize the Decoding Failure Rate, upon which the CCA security of BIKE relies (refer to Section 3.1 for detailed explanation of the decoder).

The following sections provide a detailed explanation of each part of the BIKE scheme.

Key Generation The key generation process, illustrated in Alg. 1, proceeds as follows: First, it generates two rows h_0 and h_1 of odd weight $w/2$. Due to the chosen parameters, the polynomials h_0 and h_1, or equivalently the circulant matrices they represent, are invertible. This allows the computation of the public key as $h = h_1 \cdot h_0^{-1}$. Additionally, a seed σ is generated for implicit rejection during decapsulation [BP18]. The security of the private key relies on the difficulty of recomputing h_0 and h_1 from h.

Encapsulation The encapsulation process, depicted in Alg. 2, proceeds as follows: It generates a random message m and hashes it using a special hash function $\mathbf{H}$, producing an error vector $e = (e_0, e_1)$ of fixed weight t. The ciphertext is then computed as the concatenation of the syndrome $e_0 + e_1 \cdot h$ and the blinding of the message m through $\mathbf{L}(e_0, e_1)$. The security of the ciphertext lies in the difficulty of computing the error vector (e_0, e_1) from its syndrome $s_{pub} = e_0 + e_1 \cdot h$ given a bad representation of the code $(1, h)$.

Decapsulation The decapsulation process, depicted in Alg. 3, proceeds as follows: It decodes the error (e_0, e_1) using the decoding algorithm in Alg. 6 given the secret key (h_0, h_1). It then recomputes the message m from the ciphertext c_1 and tests its well-formedness by recomputing e and checking its correspondence with the output of the decoder. Depending on the outcome of this check, the real or dummy session key is output.

Table 1 summarizes the parameters corresponding to several levels of security defined by NIST.

Algorithm 1 Key Generation

Output: $(h_0, h_1, \sigma) \in \mathcal{K}$, $h \in \mathcal{R}$

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

Algorithm 2 Encapsulation

Input: h

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

1: $m \leftarrow_{\$} \mathcal{M}$
2: $e_0, e_1 \leftarrow \mathbf{H}(m)$
3: $s_{pub} \leftarrow e_0 \oplus e_1 \cdot h$
4: $(c_0, c_1) \leftarrow (s_{pub}, m \oplus \mathbf{L}(e_0, e_1))$
5: $K \leftarrow \mathbf{K}(m, c_0, c_1)$
6: **return** $K, (c_0, c_1)$

Algorithm 3 Decapsulation

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

$(e'_0, e'_1) \leftarrow \mathsf{decode}(c_0 h_0, h_0, h_1)$
$m' \leftarrow c_1 \oplus \mathbf{L}(e'_0, e'_1)$
if $(e'_0, e'_1) = \mathbf{H}(m')$ **then**
    $K \leftarrow \mathbf{K}(m', c_0, c_1)$
else
    $K \leftarrow \mathbf{K}(\sigma, c_0, c_1)$
end if
return K

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

Table 1. BIKE parameters.

3.1 Decoding Algorithm

This section offers a comprehensive explanation of the Bit Flipping decoding algorithm employed in the decryption process of BIKE. The algorithmic intuition is presented, followed by a step-by-step breakdown of its operation.

Consider a valid codeword, denoted a null error $(e_0, e_1) = (0, 0)$. By design, we have the syndrome $s = h_0 e_0 + h_1 e_1 = 0$, where (h_0, h_1) represents the parity-check matrix. Now, assume we have a non-zero error, denoted as $(e_0, 0)$, obtained by introducing a single error at the i-th position, effectively flipping the i-th bit. This error can be represented as $e_1 = X^i$ in the polynomial representation. Computing the syndrome for $(e_0, 0)$ yields:

$$s = (h_0, h_1) \cdot (e_0, 0) = X^i \cdot h_0 \oplus 0 \cdot h_1 = X^i h_0$$

Here, $X^i h_0$ corresponds to a cyclic shift of i places of the polynomial h_0 since $h_0 \in \mathbb{F}_2/(X^r - 1)$. Equivalently, it corresponds to the i-th row of the parity-check matrix since the matrix is circulant.

By comparing the syndrome s to the rows of h_0, $X^j h_0$, we can determine which bit was erroneously flipped. Specifically, we employ the Hamming weight of $s \wedge X^j h_0$ to assess the similarities between s and the rows of h_0. In the literature, this value is generally denoted as the Unsatisfied Parity Check (UPC).

In the case of multiple errors, the same procedure applies. We flip the i-th bit of c_0 when the syndrome s and $X^i h_0$ are sufficiently close, indicated by $\|s \wedge H_j\|_1 \geq T$, where T represents a threshold value. This approach enables effective correction of multiple errors. The same process is then repeated by comparing the syndrome to rows of h_1 to correct errors occurring in c_1.

The success rate of the decoding algorithm is further influenced by the selection of appropriate threshold values T and the sparsity of the parity-check matrix.

The current decoding algorithm utilized in BIKE, known as the Black-Gray decoder [DGK19], is a variant derived from the original Bit-Flipping algorithm presented in [Gal62]. It has been tailored to minimize the decoding failure rate (DFR) while maximizing efficiency. The Black-Gray algorithm differs in the choice of parameters, the number of Bit-Flip Iterations, and the addition of two so-called masked iterations.

Algorithm 4 Bit-Flipping iteration BFIter

Input: A syndrome s, an error vector (e_0, e_1), a threshold T, a parity-check matrix (h_0, h_1)
Output: An updated error vector e, two masks *black* and *gray*

for $k = 0$ to 1 **do**
  for $j = 1$ to r **do**
    if $\|X^j h_k \wedge s\|_1 \geq T$ **then**
      $e_k^{(j)} \leftarrow e_k^{(j)} \oplus 1$
      $black_k^{(j)} \leftarrow 1$
    elseif $\|X^j h_k \wedge s\|_1 \geq T - \tau$
      $gray_k^{(j)} \leftarrow 1$
    end if
  end for
end for
return $(e_0, e_1), (black_0, black_1), (gray_0, gray_1)$

Algorithm 4 sums up the original Bit-Flip iterations, during the bit flip iterations it also keeps track of the flipped bit in the variable *black* and the bit that was close to the threshold but not flipped in the *gray* variable.

Algorithm 5 Bit-Flipping masked iteration BFMaskedIter

Input: A syndrome s, a error vector (e_0, e_1), a mask $(mask_0, mask_1)$, a parity-check matrix (h_0, h_1)
Output:

for $k = 0$ to 1 **do**
  for $j = 1$ to r **do**
    if $\|X^j h_k \wedge s\|_1 \geq (w/2 + 1)/2 + 1$ **then**
      $e_k^{(j)} \leftarrow e_k^{(j)} \oplus mask_k^{(j)}$
    end if
  end for
end for
return e_0, e_1

Algorithm 5 describe the masked bit-flip iterations that are added in [DGK19]. The mask involved are the *black* and *gray* variable respectively. It proceed similarly to a classic Bit-Flip iteration with a fixed threshold $(w/2 + 1)/2 + 1$. It differs then to the flipping of the error bit which occurs if and only if the bit of the mask, *black* or *gray*, is set. This additional steps allows to correct bits that was wrongfully flipped during the first iterations thanks to the *black* mask and correct bits that should have been flipped during the first iteration thanks to the *gray* mask.

Algorithm 6 Black-Gray Decoder decode

Input: A syndrome $s \in \mathbb{F}_2^r$, a parity-check matrix (h_0, h_1)
Output: An error vector $(e_0, e_1) \in \mathbb{F}_2^{2r}$ such that $s = e_0 h_0 \oplus e_1 h_1$ or a decoding failure $\perp$

$e_0, e_1 \leftarrow 0^{2r}$, $black_0, black_1 \leftarrow 0^{2r}$, $gray_0, gray_1 \leftarrow 0^{2r}$
for $i = 1$ to NbIter **do**
  $T \leftarrow \mathsf{threshold}(\|s\|_1)$
  $(e_0, e_1), (black_0, black_1), (gray_0, gray_1) \leftarrow \mathsf{BFIter}(s, (e_0, e_1), T, (h_0, h_1))$
  $s \leftarrow s \oplus e_0 h_0 \oplus e_1 h_1$
  if $i = 1$ **then**
    $(e_0, e_1) \leftarrow \mathsf{BFMaskedIter}(s, (e_0, e_1), (black_0, black_1), (h_0, h_1))$
    $s \leftarrow s \oplus e_0 h_0 \oplus e_1 h_1$
    $(e_0, e_1) \leftarrow \mathsf{BFMaskedIter}(s, (e_0, e_1), (gray_0, gray_1), (h_0, h_1))$
    $s \leftarrow s \oplus e_0 h_0 \oplus e_1 h_1$
  end if
end for
if $s = 0$ **then**
  return e_0, e_1
else
  return $\perp$
end if

Eventually, the whole algorithm is described in Algorithm 6. It repeats *NbIter* Bit-Flip operations and 2 masked bit-flip iterations for the masks *black* and *gray*, right after the first Bit-Flip. The threshold for the Bit-flip operation is also recomputed for each new iteration depending on the Hamming weight of the syndrome through the following procedure described in Algorithm 7

Algorithm 7 Threshold threshold

Input: The Hamming weight S of a syndrome
Output: The decoding threshold th

if NIST Level 1 **then**
  return $\max(\lfloor 0.0069722 \cdot S + 13.530 \rfloor, 36)$
end if
if NIST Level 3 **then**
  return $\max(\lfloor 0.005265 \cdot S + 15.2588 \rfloor, 52)$
end if
if NIST Level 5 **then**
  return $\max(\lfloor 0.00402312 \cdot S + 17.8785 \rfloor, 69)$
end if

Masking The Scheme Like every cryptographic scheme implemented on embedded devices, the BIKE scheme is susceptible to side-channel attacks. This realization has prompted the development of various masking techniques and frameworks to enhance our countermeasures. Depending on the nature of the operations involved, different types of masking are preferred, with Boolean and Arithmetic masking being common standards for Boolean and arithmetic operations, respectively. Conveniently, in the case of BIKE, the characteristic 2 of arithmetic operations causes the two masking representations to coincide in most cases.

To address potential side-channel vulnerabilities, we discuss a first-order attack on the masking of the decoder proposed by [CEvMS16]. This attack highlights the necessity of masking the Unsatisfied Parity

Check (UPC) value. Subsequently, we delve into the high-order masking of the BIKE scheme, starting with the masking of the decapsulation process, which represents a significant challenge. We then explore the masking strategies for key generation and encapsulation.

For each procedure in the scheme, we provide a comprehensive list of sensitive data and outline the corresponding strategies employed to protect it.

4 Attack on [CEvMS16]

Existing Research. To date, the sole paper addressing the issue of masking in the context of BIKE is [CEvMS16]. The primary contribution of this paper is the proposal of a first-order threshold implementation for the decoder. However, since 2016, significant optimizations have been made to the decoder. While their countermeasures could potentially be adjusted to accommodate these modifications, it is worth noting that the authors of [CEvMS16] applied masking to nearly every step of the algorithm, except for the unsatisfied parity-check value (UPC), denoted as $\|X^j h_k \wedge s\|_1$. They argued that this value is not sensitive to side-channel attacks.

Our Contributions. In the following sections, we demonstrate that the UPC value is actually sensitive to side-channel attacks. Specifically, we describe a first-order attack on the UPC that enables the construction of a side-channel assisted chosen-ciphertext attack (`CCA`) on the private key variables h_0 and h_1. This attack allows for the computation of shifts in the private key, denoted as $h'_0 = X^i h_0$ and $h'_1 = X^i h_1$. These shifted private keys are valid decryption keys corresponding to the same public key h. We outline the underlying principle of this attack. The significance of this attack underscores the necessity to apply masking to the check operation on line 3 of Algorithm 4. The following section will be dedicated to the description of our attack.

4.1 Attack Principle

We assume the ability to probe the value of *upc* at the first iteration of Algorithm 4, i.e., the first iteration of the For loop in line 3. This value corresponds to the value $\|s \wedge h_0\|_1$ for a given input $s = c_0 \cdot h_0 = (e_0 + e_1 h)h_0 = e_0 h_0 + e_1 h_1$, for adversarially chosen (e_0, e_1). Equivalently, we assume we are able to utilize side-channel analysis to construct an oracle, denoted as $\mathcal{O}_{upc}$, which takes an error (e_0, e_1) as input and returns the value $\|(h_0 e_0 + h_1 e_1) \wedge h_0\|_1$.

In particular, we can query the oracle using the monomials $(X^i, 0)$. This provides us with the values $\|X^i h_0 \wedge h_0\|_1$ for $1 \leq i \leq r$. Notably, for $i = 0$, we observe that $\|h_0 \wedge h_0\|_1 = \|h_0\|_1 = w/2$, which is independent of the key. However, for $i = 1$, we encounter an interesting scenario:

$$\begin{aligned}
\|X h_0 \wedge h_0\|_1 &= \|\mathsf{shift}_1(h_0) \wedge h_0\|_1 \\
&= \sum_{i=1}^{r} h_0^{(i)} h_0^{(i+1)} \\
&= |\{i \mid h_0^{(i)} \text{ and } h_0^{(i+1)} \text{ are } 1\}| \\
&\neq 0 \Leftrightarrow \text{there are two consecutive 1 in } h_0
\end{aligned}$$

This generalizes for $1 \leq i \leq r-1$: $\|X^i h_0 \wedge h_0\|_1 \neq 0$ if and only if there are two 1's separated by $i-1$ places in h_0. Intuitively, knowledge of all the spaces between 1's enables the construction of h_0 up to a shift. However, to ease the attack we can query our oracle $\mathcal{O}_{upc}$ with additional values to recover the key.

4.2 Description of the Attack

First Step We find $i_0 > 1$ such that $\|X^{i_0}h_0 \wedge h_0\|_1 = 1$ - which happens almost certainly due to sparsity of h_0. We therefore deduce there are exactly two 1's separated by $i_0 - 1$ places. We can construct $h'_0 = (1, \star, \ldots, \star, 1, \star, \ldots, \star)$, where the second 1 is in the i_0-th position and $\star$ represents unknown values. Up to a shift of an unknown k_0 places, we know that h'_0 corresponds to h_0, i.e., $h_0^{(k_0)} = h_0^{(k_0+i_0)} = 1$.

Second Step We query our oracle $\mathcal{O}_{upc}$ using combinations $X^{i_0} \oplus X^j$ for $j \neq i_0$. Through some calculations, we remark:

$$\begin{aligned}
\|(X^{i_0} \oplus X^j)h_0 \wedge h_0\|_1 &= \|(X^{i_0}h_0 \oplus X^j h_0) \wedge h_0\|_1 \\
&= \sum_{k=1}^{r} (h_0^{(i_0+k)} \oplus h_0^{(j+k)}) \cdot h_0^{(k)} \\
&= \sum_{k=1}^{r} (h_0^{(i_0+k)} + h_0^{(j+k)} - h_0^{(i_0+k)} h_0^{(j+k)}) \cdot h_0^{(k)} \\
&= \sum_{k=1}^{r} h_0^{(i_0+k)} h_0^{(k)} + h_0^{(j+k)} h_0^{(k)} - h_0^{(i_0+k)} h_0^{(j+k)} h_0^{(k)} \\
&= \|X^{i_0}h_0 \wedge h_0\|_1 + \|X^j h_0 \wedge h_0\|_1 - \sum_{k=1}^{r} h_0^{(i_0+k)} h_0^{(j+k)} h_0^{(k)} \\
\sum_{k=1}^{r} h_0^{(i_0+k)} h_0^{(j+k)} h_0^{(k)} &= \mathcal{O}_{upc}(X^{i_0}) + \mathcal{O}_{upc}(X^j) - \mathcal{O}_{upc}(X^{i_0} \oplus X^j)
\end{aligned}$$

The sum on the left hand side of the equation can be simplified using $\|X^{i_0}h_0 \wedge h_0\| = 1$. Indeed, we have $h_0^{(i_0+k)} h_0^{(k)} \neq 0$ only for $k = k_0$ and $h_0^{(i_0+k_0)} h_0^{(k_0)} = 1$.. Therefore, we deduce:

$$\begin{aligned}
h_0^{(j+k_0)} &= \sum_{k=1}^{r} h_0^{(i+k)} h_0^{(j+k)} h_0^{(k)} \\
&= \mathcal{O}_{upc}(X^{i_0}) + \mathcal{O}_{upc}(X^j) - \mathcal{O}_{upc}(X^{i_0} \oplus X^j)
\end{aligned}$$

Consequently, by setting $h_0'^{(j)} = 1$ if and only if $\mathcal{O}_{upc}(X^{i_0}) + \mathcal{O}_{upc}(X^j) - \mathcal{O}_{upc}(X^{i_0} \oplus X^j) \neq 0$, we obtain a column h'_0 that is equal to h_0 up to an unknown shift of k_0 places, $h'_0 = X^{k_0}h_0$.

Last Step From the public key $h = h_1 h_0^{-1}$, we can then compute $h'_1 = h'_0 h = X^{k_0} h_1 h_0^{-1} h_0 = X^{k_0} h_1$. Eventually, we obtain a couple (h'_0, h'_1) which corresponds to a shift of the private key (h_0, h_1) but equivalently corresponds to a private key leading to the same public key h. Therefore, our constructed (h'_0, h'_1) despite not being the original private key, corresponds to a decryption key for h.

We describe in Alg. 18 in Appendix A a pseudo code for the attack.

Eventually, the attack described above shows the necessity to mask the *upc* value and therefore any computation in which *upc* is involved.

5 Masking the Decapsulation

In this section, we detail the masking strategy employed in the decapsulation process of the BIKE scheme. The critical variables are the session key K and the private key h_0, h_1. For security against side-channel attacks, we therefore need to mask any intermediate computation upon which they directly relies. This

implies masking the syndrome, the error vector, the message and therefore the decoder algorithm and the three hash function $\mathbf{H}$, $\mathbf{K}$ and $\mathbf{L}$.

The following section is organized as follows. It first describes the masking of the decoder by detailing the masking of each individual iteration. Then, we describe the masking of the threshold function and the Hamming weight that are employed for the decoder. Eventually, the masking of the whole decapsultation in summed up in Alg. 12.

In order to assess the effectiveness of our countermeasure we provide for each algorithm a complexity and security analysis.

5.1 Masking the decoder

As explained above, we need to protect the syndrome, the error vector and the private key.

Therefore, when applying our countermeasure to the decoder this implies that we need to mask any intermediate variable it relies on. In particular, we need to mask the Hamming weight and the threshold functions which are described in sections 5.3 and 5.2 respectively. Eventually, we need to mask the Bit-Flip and the Masked Bit-Flip iterations.

BFIter To mask a bit-flipping iteration BFIter in Algorithm 4, we employ the following procedure.

First, we perform a high-order computation of the number of unsatisfied parity checks by applying the high-order Hamming weight, Alg. 11, to the secure And of the Boolean masked syndrome and the j row of the parity-check (PC) matrix. The result is an arithmetic masking modulo 2^α. It is important to note that since each row $X^j h_k$ of the matrix has a fixed weight of $w/2$, we know that the unsatisfied parity check count $upc = \|s_1 \wedge X^j h_k\|_1$ lies within the range of $[0, w/2]$. To ensure correctness of the computation, we choose α such that $w/2 < 2^\alpha$, instead of the constraint $r < 2^\alpha$ that would be required without any further consideration.

Next, we high-order compute the bit $[\![upc \geq th]\!]$ using shifts whose procedure SecIsPositive is described in Alg. 21 in Appendix D.

The result is directly stored in the j-th bit of the variable $black$, as the following equivalence holds:

$$upc \geq th \Leftrightarrow black = 1$$

The bit flips of e are delayed until the end of the for loop since e is not involved in any of the computations within the loop. Additionally, the flips can be performed by updating e as $e \leftarrow e + black$, since the j-th bit of e is flipped if and only if $upc \geq th \Leftrightarrow black^{(j)} = 1$.

Finally, we high-order compute the bits $d = [\![upc \geq th - \tau]\!]$ and obtain the gray bits through $gray = black \oplus d$. This is based on the following equivalences:

$$\begin{aligned} gray = 1 &\Leftrightarrow upc \geq th - \tau \wedge upc < th \\ &\Leftrightarrow d = 1 \wedge black = 0 \\ &\Leftrightarrow d \oplus black = 1 \end{aligned}$$

The last equivalence holds since $black = 1$ implies $d = 1$.

The entire procedure is described in detail in Algorithm 8.

Algorithm 8 Masked Bit-Flipping Iteration SecBFIter

Input: A Boolean sharing $(s_1, \ldots, s_n)$ of a syndrome, a Boolean sharing $(e_1, \ldots, e_n)$ of the error vector, an arithmetic sharing modulo 2^α $(th_1, \ldots, th_n)$ of the threshold, a Boolean Sharing of the parity-check $(h_{0,1}, \ldots, h_{0,n}), (h_{1,1}, \ldots, h_{1,n})$

Output: A Boolean sharing $(e_1, \ldots, e_n)$ of updated error vector, a Boolean sharing $(black_1, \ldots, black_n)$ of the black mask, a Boolean sharing $(gray_1, \ldots, gray_n)$ of the gray mask

1: **for** $k = 0$ to 1 **do**
2: **for** $j = 1$ to r **do**
3: $r_1, \ldots, r_n \leftarrow \mathsf{SecAnd}((s_1, \ldots, s_n), (X^j h_{k,1}, \ldots, X^j h_{k,n}))$
4: $upc_1, \ldots, upc_n \leftarrow \mathsf{SecHW}(2^\alpha, (r_1, \ldots, r_n))$
5: $black_1^{(j)}, \ldots, black_n^{(j)} \leftarrow \mathsf{SecIsPositive}(upc_1 - th_1, \ldots, upc_n - th_n)$
6: $d_1^{(j)}, \ldots, d_n^{(j)} \leftarrow \mathsf{SecIsPositive}(upc_1 - th_1 + \tau, upc_2 - th_2, \ldots, upc_n - th_n)$
7: **end for**
8: **end for**
9: **for** $i = 1$ **to** n **do** $e_i \leftarrow e_i + black_i$
10: **for** $i = 1$ **to** n **do** $gray_i \leftarrow black_i + d_i$
11: **return** $(e_1, \ldots, e_n), (black_1, \ldots, black_n), (gray_1, \ldots, gray_n)$

Complexity. Counting operations as above, we deduce the complexity of Algorithm 8

$$\begin{aligned} T_{\mathsf{SecBFIter}}(n, r, \alpha) &= 2r \cdot (T_{\mathsf{SecAnd}}(n) + T_{\mathsf{SecHW}}(n, r) + 2T_{\mathsf{SecIsPositive}}(n, \alpha)) + n + n \\ &= \mathcal{O}((r^2 + \alpha r)n^2) \end{aligned}$$

Security. The following theorem shows our algorithm achieves NI

Theorem 1 ($(n-1)$ − NI **of** SecBFIter)**.** *For any set of t_1 probes, there exists a subset $I \subset [1, n]$, with $|I| \leq t_1$ such that the t_1 probes can be perfectly simulated from inputs $e_{|I}$, $th_{|I}$, $H_{|I}$.*

Proof. The SNI follows from the composition of SNI SecAnd, SecHW and NI SecIsPositive with sharewise computations.

BFMaskedIter The high-order bit-flip masked iteration closely resembles Algorithm 8. The main difference lies in the conditional bit flip based on the value of $mask$, which can be computed using the following equivalences:

$$\begin{aligned} e \leftarrow e + mask &\Leftrightarrow d = 1 \\ &\Leftrightarrow e \leftarrow e + d \wedge mask \end{aligned}$$

where $d = [\![upc \geq (w/2 + 1)/2 + 1]\!]$ is the result of the test. This means that if $d = 1$, the bit flip is performed by updating e with the value of $mask$. Otherwise, if $d = 0$, then $d \wedge mask = 0$ and there are no flip. The entire procedure is outlined in detail in Algorithm 9.

Algorithm 9 Masked Bit-Flipping Masked Iteration SecBFMaskedIter

Input: A Boolean sharing $(s_1, \ldots, s_n)$ of a syndrome, a Boolean sharing $(e_1, \ldots, e_n)$ of the error vector, a Boolean sharing $(mask_1, \ldots, mask_n)$ of the iteration mask, a Boolean Sharing of the parity-check matrix $(H_1, \ldots, H_n)$

Output: A Boolean sharing $(e_1, \ldots, e_n)$ of updated error vector

1: **for** $j = 1$ to r **do**
2: $\quad r_1, \ldots, r_n \leftarrow \mathsf{SecAnd}((s_1, \ldots, s_n), (H_{j,1}, \ldots, H_{j,n}))$
3: $\quad upc_1, \ldots, upc_n \leftarrow \mathsf{SecHw}(2^\alpha, (r_1, \ldots r_n))$
4: $\quad d_1^{(j)}, \ldots, d_n^{(j)} \leftarrow \mathsf{SecIsPositive}(upc_1 - (w/2+1)/2 + 1, upc_2, \ldots, upc_n)$
5: **end for**
6: $mask_1, \ldots, mask_n \leftarrow \mathsf{SecAnd}((mask_1, \ldots, mask_n), (d_1, \ldots, d_n))$
7: **for** $i = 1$ **to** n **do** $e_i \leftarrow e_i + mask_i$
8: **return** $e_1, \ldots, e_n$

Complexity. The number of operations is

$$\begin{aligned} T_{\mathsf{SecBFMaskedIter}}(n, r, \alpha) &= 2r \cdot (T_{\mathsf{SecAnd}} + T_{\mathsf{SecHW}}(n, r) + T_{\mathsf{SecIsPositive}}(n, \alpha)) + T_{\mathsf{SecAnd}}(n) + n \\ &= \mathcal{O}((r^2 + \alpha r)n^2)) \end{aligned}$$

Security. The theorem below shows that Algorithm 9 achieves NI security.

Theorem 2 (NI security of SecBFMaskedIter). *For any set of t_1 probes, there exists a subset $I \subset [1, n]$, with $|I| \leq t_1$, such that the t_1 can be perfectly simulated from inputs $s_{|I}$, $e_{|I}$, $mask_{|I}$, $H_{|I}$.*

Proof. The security follows from the composition of SNI SecAnd and the NI security of SecHW and SecIsPositive

5.2 Masking the Threshold

In the BIKE specification, the authors opt to compute the threshold based on the Hamming weight of the syndrome, as depicted above in Algorithm 7. This choice was influenced by an analysis conducted in [DGK19], aiming to minimize the decoding failure rate (DFR), which is crucial for achieving IND-CCA security in the scheme.

While masking the maximum function is relatively straightforward, dealing with the floor value computation using floating-point parameters poses a significant challenge when applying our masking countermeasures. In order to address this challenge, we opt to replace the floating-point computations with integer operations and boolean shifts. This modified algorithm retains the same functionality for our range of inputs while benefiting from much simpler masking techniques. Ultimately, we address the secure computation of the maximum value in Appendix D

Masking the floor value As described previously, we have replaced the floor value computation with a more suitable approach that achieves the same functionality within our desired value range of $0 \leq S \leq r$. Specifically, we performed an algorithmic search to find integer values m, p, and k such that:

$$\mathsf{threshold}(S) = \max(\lfloor \frac{m \cdot S + p}{2^k} \rfloor, lower)$$

This floor value can then be effectively masked using arithmetic masking and arithmetic shifts. We have discovered the following parameters, where each equality holds when $0 \leq S \leq r$:

$$\lfloor 0.0069722 \cdot S + 13.530 \rfloor = \left\lfloor \frac{58487 \cdot S + 113498112}{2^{23}} \right\rfloor$$
$$\lfloor 0.005265 \cdot S + 15.2588 \rfloor = \left\lfloor \frac{22083 \cdot S + 64000256}{2^{22}} \right\rfloor$$
$$\lfloor 0.00402312 \cdot S + 17.8785 \rfloor = \left\lfloor \frac{269987 \cdot S + 1199806464}{2^{26}} \right\rfloor$$

This revised computation approach allows us to more easily and efficiently apply our masking countermeasures. In particular, the floor operation can now be regarded as a shift of k positions for the integer value $m \cdot S + p$. The former can be computed using arithmetic masking modulo $2^{k+\alpha}$, while the latter can be implemented using the efficient ShiftMod operation from [CGTZ23], as illustrated in Appendix C, to achieve arithmetic masking modulo 2^{α}.

The entire procedure is summarized in Algorithm 10.

Algorithm 10 Masked Threshold SecThreshhold

Input: An arithmetic masking modulo $2^{k+\alpha}$, $S_1, \ldots, S_n$ of an integer $0 \leq S \leq r$.
Output: An arithmetic masking modulo 2^{α}, $th_1, \ldots, th_n$ of the value threshold(S)

1: **for** $i = 1$ **to** n **do** $S_i \leftarrow m \cdot S_i \bmod 2^{k+\alpha}$
2: $S_1 \leftarrow S_1 + p \bmod 2^{k+\alpha}$
3: $th_1, \ldots, th_n \leftarrow S_1, \ldots, S_n$
4: **for** $j = 0$ to $k - 1$ **do**
5: $\quad th_1, \ldots, th_n \leftarrow \mathsf{ShiftMod}(2^{k-j+\alpha}, (th_1, \ldots, th_n))$
6: **end for**
7: $th_1, \ldots, th_n \leftarrow \mathsf{SecMax}((th_1, \ldots, th_n), (lower, 0, \ldots, 0))$
8: **return** $th_1, \ldots, th_n$

Complexity. The number of operations of ShiftMod in [CGTZ23] is $T_{\mathsf{ShiftMod}}(n) = 2n^2 + 10n - 9$. Therefore, we deduce the complexity of Algorithm 10 :

$$\begin{aligned} T_{\mathsf{SecThreshold}}(n, k, \alpha) &= n + 1 + k \cdot T_{\mathsf{ShiftMod}}(n) + T_{\mathsf{SecMax}}(\alpha, n) \\ &= n + 1 + 2kn^2 + 10kn - 9k + (2\alpha + 6)n^2 + (10\alpha - 5)n - 9\alpha + 2 \\ &= \mathcal{O}((k + \alpha) \cdot n^2) \end{aligned}$$

Security. The following theorem shows our implementation of the threshold achieves SNI.

Theorem 3 **($(n-1)$ – SNI of SecThreshold).** *For any t_1 intermediate variables and any subset $O \subset [1, n]$ such that $t_1 + |O| < n$. There exists a subset $I \subset [1, n]$, with $|I| \leq t_1$, such that the t_1 intermediate variables and the outputs $th_{|O}$ can be perfectly simulated from inputs $S_{|I}$.*

Proof. The theorem results from the composition of NI gadget ShiftMod from [CGTZ23] and SNI gadget SecMax, see Theorem 12, altogether with sharewise operations.

5.3 Masking the Hamming Weight

We now show how to mask the computation of the Hamming weight. This procedure is inspired from [CEvMS16] where the authors convert each bit from boolean masking to arithmetic masking.

The correctness of the algorithm relies on the following equation:

$$\|s\|_1 = \sum_{i=1}^{r} s^{(i)}$$

Each bit $s^{(i)}$ being masked in Boolean, the sum described shows the necessity of converting from Boolean to arithmetic. In our countermeasure, we instantiated this conversion from 1 bit Boolean to arithmetic with the efficient conversion from [SPOG19] recalled in Alg. 19 in App. B.

Algorithm 11 Masked Hamming Weight SecHW

Input: A modulus 2^k, a Boolean sharing $s_1, \ldots, s_n$ of $s \in \mathbb{F}_2^r$
Output: An arithmetic sharing $S_1, \ldots, S_n$ modulo 2^k of $\|s\|_1$

1: $S_1, \ldots, S_n \leftarrow (0, \ldots, 0)$
2: **for** $j = 1$ to r **do**
3: $\quad s'_1, \ldots, s'_n \leftarrow \mathsf{1bitB2A}_{2^k}(s_1^{(j)}, \ldots, s_n^{(j)})$
4: $\quad$ **for** $i = 1$ **to** n **do** $S_i \leftarrow S_i + s'_1 \bmod 2^k$
5: **end for**
6: **return** $(S_1, \ldots, S_n)$

Complexity. Given the complexity of 1bitB2A, $T_{\mathsf{1bitB2A}}(n) = 2n^2 + 4n - 6$ from [SPOG19], we deduce the complexity of Alg. 11:

$$\begin{aligned} T_{\mathsf{SecHW}}(n, r) &= n + r \cdot (T_{\mathsf{1bitB2A}})(n) + n) \\ &= n + r \cdot (2n^2 + 4n - 6 + n) \\ &= 2rn^2 + 5rn - 6r + n \\ &= \mathcal{O}(2rn^2) \end{aligned}$$

Security. The following theorem shows Algorithm 11 achieves free-SNI.

Theorem 4 ($(n-1)-$SNI of SecHW). *For any set of t_1 intermediates variables and any subset $O \subset [1, n]$, such that $t_1 + |O| < n$, there exists $I \subset [1, n]$ such that the t_1 intermediate variables and outputs $S_{|O}$ can be perfectly simulated from inputs $s_{|I}$, with $|I| \leq t_1$.*

Proof. The proof follows from the composition of SNI 1bitB2A gadget with sharewise computations.

5.4 Decapsulation

Given the masking of the decoder we deduce the masking of the whole decapsulation. The whole procedure is sum up in Algorithm 12.

Our countermeasures assume we are given a high-order implementation of the SHA3 − 384 in order to compute the hash functions **L** and **K**. We refer to [GSM17] where the secure implementation of SHA3 − 384 is described. The high-order computation of the last hash function **H** is described in the following section.

Algorithm 12 Sec Decapsulation

for $i = 1$ **to** n, $c_{0,i} \leftarrow c_0 \cdot h_{0,i}$
$e'_{0,1}, \ldots, e'_{0,n}, e'_{1,1}, \ldots, e'_{1,n} \leftarrow \mathsf{SecDecoder}((c_{0,i}, h_{0,i}, h_{1,i})_{1 \le i \le n})$
$m'_1, \ldots, m'_n \leftarrow \mathbf{SecL}((e'_{0,i}, e'_{1,i})_{1 \le i \le n})$
$m'_1 \leftarrow c_1 \oplus m'_1$
$e_{s,1}, \ldots, e_{s,n} \leftarrow \mathsf{SecSampleH}((m'_1, \ldots, m'_n), 2 \cdot r, t)$
$((e_{0,1}, e_{1,1}), \ldots, (e_{0,n}, e_{1,n}) \leftarrow \mathsf{DSSecMult}((1, 0, \ldots, 0), (e_{s,1}, \ldots, e_{s,n}))$
for $i = 1$ **to** n, $e_{0,i} \leftarrow e_{0,i} \oplus e'_{0,i}$
for $i = 1$ **to** n, $e_{1,i} \leftarrow e_{1,i} \oplus e'_{1,i}$
$b_1, \ldots, b_n \leftarrow \mathsf{ZeroTestBool}((e_{0,i}, e_{1,i})_{1 \le i \le n})$
$K_1, \ldots, K_n \leftarrow \mathsf{HOTSwap}((b_1, \ldots, b_n), \mathbf{SecK}((m'_i, c)_{1 \le i \le n}), \mathbf{SecK}((\sigma_i, c)_{1 \le i \le n}))$
return $K_1, \ldots, K_n$

Complexity. Given the complexity of each part of the Decapsulation, we deduce the complexity of Alg. 12 is:

$$T_{\mathsf{SecDecaps}}(r, w, n) = \mathcal{O}(r^2 n^2)$$

Security. The following theorem shows the Decapsulation achieves NI security.

Theorem 5. *For any set of t_1 intermediate variables such that $t_1 < n$. There exists a subset $I \subset [1, n]$ such that the t_1 intermediate variables can be perfectly simulated from $(h_{0,i}, h_{1,i}, \sigma_i)_{i \in I}$ and c, with $|I| \le t_1$.*

Proof. The proof follows from the composition of each individual par composed with sharewise operations.

6 Masking the Key Generation

In this section, we give an overview of the masking the key generations. In particular, we need to mask the secret key (h_0, h_1) and the seed for implicit rejection σ. To that extent, we need to mask the fixed weight sampling algorithm of BIKE represented by $\mathbf{H}$. The remaining operations are mainly arithmetic operations.

6.1 Sampling fixed weight polynomials

In BIKE, we need to sample uniform vectors of fixed weight. The generic procedure to sample fixed weight vectors is depicted in Alg. 13. The same procedure is hence used both for sampling of the private key h_0 and h_1 and the error vector e by adjusting the parameters.

Algorithm 13 H implementation

Input: a seed *seed*, a vector size *len*, and a weight *wt*
Output: A list *wlist* of *wt* distinct elements in $\{0, \ldots, len - 1\}$.

$wlist \leftarrow \{\}$
$s_0, \ldots, s_{wt-1} \leftarrow \mathrm{SHAKE256} - \mathrm{Stream}(seed, 32 \cdot wt)$
for $i = wt - 1$ to 0 **do**
    $pos \leftarrow i + \lfloor (len - i) s_i / 2^{32} \rfloor$
    if $pos \in wlist$ **then**
        $wlist \leftarrow wlist \cup \{i\}$
    else
        $wlist \leftarrow wlist \cup \{pos\}$
    end if
end for
return *wlist*

To sample a vector of length len with weight wt, the algorithm generates a sparse representation of the vector by sampling a list of wt distinct entries in $[\![0, len-1]\!]$. More precisely, the algorithm first hashes the *seed* and extends it to $32 \times wt$ bits using SHAKE256 in XOF mode. Each 32-bit word is then used to sample a position in $[\![wt-1, len-1]\!]$, then $[\![wt-2, len-1]\!]$, ..., $[\![0, len-1]\!]$. To ensure only distinct positions are generated without rejection sampling, the algorithm uses the fact that at step i, all previous indexes pos_j generated satisfy $pos_j > i$. Hence, the algorithm samples a new candidate $pos_i \in [\![i, len-1]\!]$ and adds it to the list. If the new candidate pos_i collides with a previous index, we add i instead of pos_i to the list. Since $pos_j > i$, we know i is distinct from every other element. Eventually, this leads to a sparse representation of the private key as a list of wt positions in $\mathbb{F}_2^r$ that correspond to the positions of the coefficient 1 in the usual dense polynomial representation. While this procedure induces a bias in the distribution, [Sen21] showed that the distribution is sufficiently close to a uniform one to remain secure.

Algorithm 14 Masked Sample H SecSampleH

Input: A Boolean mask $seed_1, \ldots, seed_n$ of the seed, a parameter wt, a length len
Output: An arithmetic masking mod len $wlist_1, \ldots, wlist_n$ of a vector in $\mathbb{F}_2^{len}$ of distinct elements.

$s_1, \ldots, s_n \leftarrow \mathsf{SECSHAKE256-Stream}((seed_1, \ldots, seed_n), 32 \cdot wt)$
$wlist_1, \ldots, wlist_n \leftarrow ((0, \ldots, 0), \ldots, (0, \ldots, 0))$
for $j = wt - 1$ to 0 **do**
  $(t_1, \ldots, t_n) \leftarrow \mathsf{BtoA}_{2^{32} len}(s_1^{(j)}, \ldots, s_n^{(j)})$
  for $i = 1$ **to** n **do** $pos_i \leftarrow (r-j)t_i \bmod 2^{32} len$
  for $i = 0$ to 31 **do**
    $pos_1, \ldots, pos_n \leftarrow \mathsf{ShiftMod}(2^{32-i}, (pos_1, \ldots, pos_n))$
  end for
  $pos_1 \leftarrow pos_1 + j \bmod len$
  $b_1, \ldots, b_n \leftarrow \mathsf{SecMembershipTest}((pos_1, \ldots, pos_2), (wlist_1^{\leq wt-1-j}, \ldots, wlist_n^{\leq wt-1-j}))$
  $wlist_1^{(i)}, \ldots, wlist_n^{(i)} \leftarrow \mathsf{HOTSwap}((b_1, \ldots, b_n), (pos_1, \ldots, pos_n), (j, 0, \ldots, 0))$
end for
return $wlist_1, \ldots, wlist_n$

We describe in Alg. 14 a masked implementation of the sampler.

Given the high-order masked implementation of SHAKE256 as described in [GSM17], we are provided with a Boolean masking of the output $s_1, \ldots, s_n$. We split this s into 32-bit words, denoted $s^{(j)}$, and convert each Boolean mask $s_1^{(j)}, \ldots, s_n^{(j)}$ into an arithmetic mask modulo $r2^{32}$. We can therefore high-order compute arithmetic masking modulo $r2^{32}$ by linearity and apply the ShiftMod algorithm from [BDK+21] to compute the 32 shifts. Consequently, we obtain arithmetic sharing modulo r of the candidate positions. It remains to high-order compute a Boolean masking of a bit b corresponding to the membership test of $[\![pos \in wlist]\!]$. The procedure is based on the high-order comparison described in [CGMZ23], and detailed in Appendix F. Eventually, we can append the new position pos or i depending on b through the HOTSwap, Alg. 22.

As in the original function, we opt to use a masked sparse representation of the output. More precisely, the output is represented as a list of wt such that each index is arithmetically masked modulo r, i.e., $i = i_1 + \cdots + i_n \bmod r$.

Complexity Given the complexity of each part of Algorithm 14, we deduce the complexity to be:

$$\begin{aligned} T_{\mathsf{SecSampleH}}(n, r, wt) &= T_{\mathsf{SecSHAKE}}(n, 32, wt) \\ &+ \sum_{i=0}^{wt-1} (T_{\mathsf{BtoA}_{32}}(n) + n + 32 \cdot T_{\mathsf{ShiftMod}}(n) + 1 \\ &+ T_{\mathsf{SecMembershipTest}}(n, i) + T_{\mathsf{HOTSwap}}(n)) \\ &= \mathcal{O}(wt^2 n^2). \end{aligned}$$

Security The following theorem shows our masked implementation of the sampler, Alg. 14, achieves the NI security.

Theorem 6 (NI security of SecSampleH). *For any set of t_1 probes, there exists a subset $I \subset [1, n]$, with $|I| \leq t_1$, such that the t_1 probes can be perfectly simulated from inputs $s_{|I}$ and $H_{|I}$.*

Proof. The proof follows from the composition of NI and SNI parts with sharewise operations.

6.2 Conversion Sparse to Dense Representation of Polynomials

The output of our sampler is represented as a sparse masked representation. This more compact representation nonetheless can not be directly used for further arithmetic operations. In this section, we describe a conversion algorithm that takes as input a masked sparse representation and outputs a dense representations.

This conversion is actually represented as a multiplication algorithm between a sparse and a dense representation whose output is dense whose input are the polynomial we need to convert and the constant 1 polynomial.

Assuming we have a masked sparse representation $h_{s,0,1}, \ldots, h_{s,0,n}$ of sparse $h_{s,0}$ where each 1 position is arithmetically shared modulo r, i.e for all j $h_{s,0}^{(j)} = \sum_{i=1}^{n} h_{s,0,i}^{(j)} \mod r$. We remark that the dense polynomial can be retrieved from:

$$h_0 = \sum_{j=1}^{w/2} X^{h_{s,0}^{(j)}} = \sum_{j=1}^{w/2} \prod_{i=1}^{n} X^{h_{s,0,i}^{(j)}}. \tag{1}$$

In particular, this equality is based on the equivalence:

$$X^i = X^j \bmod (X^r - 1) \Leftrightarrow i = j \bmod r. \tag{2}$$

This leads to a masked multiplication algorithm between a masked dense polynomial and a masked sparse polynomial that returns a masked dense polynomial, see Algorithm 15.

$$\begin{aligned} (f_1 + \cdots + f_n) \cdot \sum_{k=1}^{w/2} X^{g_{s,1}^{(k)}} \cdots X^{g_{s,n}^{(k)}} &= \sum_{k=1}^{w/2} (f_1 + \cdots + f_n) \cdot X^{g_{s,1}^{(k)}} \cdots X^{g_{s,n}^{(k)}} \\ &= \sum_{i=1}^{n} \sum_{k=1}^{w/2} f_i \cdot X^{g_{s,1}^{(k)}} \cdots X^{g_{s,n}^{(k)}} \end{aligned}$$

Algorithm 15 Secure Multiplication of dense and sparse polynomial DSSecMult

Input: A masked dense polynomial $f_1, \ldots, f_n$ and a masked sparse polynomial $g_{s,1}, \ldots, g_{s,n}$ of weight wt.
Output: A masked dense polynomial $p_1, \ldots, p_n$ of the product $f \cdot g$

$p_1, \ldots, p_n \leftarrow (0, \ldots, 0)$
for $k = 1$ to wt **do**
  $sum_1, \ldots, sum_n \leftarrow f_1, \ldots, f_n$
  for $i = 1$ to n **do**
    for $j = 1$ **to** n **do** $sum_j \leftarrow sum_j \cdot X^{g_{s,i}^{(k)}} \bmod (X^r - 1)$
    $sum_1, \ldots, sum_n \leftarrow \mathsf{LinearRefresh}(sum_1, \ldots, sum_n)$
  end for
  For $i = 1$ **to** n, $p_i \leftarrow p_i + sum_i$
end for
return $p_1, \ldots, p_n$

Eventually, the multiplication algorithm can be used with input $h_1, \ldots, h_n = (1, 0, \ldots, 0)$ in order to convert from sparse representation to dense representation.

Complexity. Counting each modular operation as unit we deduce the complexity of Algorithm 15 to be :

$$\begin{aligned} T_{\mathsf{DSSecMult}}(n, wt) &= wt \cdot (n \cdot (n + T_{\mathsf{LinearRefresh}}(n)) + n) \\ &= wt \cdot n \cdot (n + 3n - 3 + 1) = 4wt \cdot n^2 - 2wt \cdot n \\ &= \mathcal{O}(wt \cdot n^2) \end{aligned}$$

Security. The following theorem shows our algorithm achieves the SNI security.

Theorem 7 (**$(n-1)-$SNI security of DSSecMult**). *For any t_1 intermediate variables and any subset $O \subset [1, n]$ such that $t_1 + |O| < n$. There exists a subset $I \subset [1, n]$ such that the t_1 intermediate variables and the outputs $f_{|O}$ can be perfectly simulated from inputs $h_{|I}$ and $g_{|I}$, with $|I| \leq t_1$.*

The proof of Theorem 7 can be found in Appendix G.

6.3 Masking of the Keygen

The masking of the key generation is straightforward by replacing each gadget with its masked alternative. We note that we reuse the strategy from [KLRBG22] to perform the inversion, denoted as SecInv. Their procedure is recalled in Appendix H. Namely, it consists of converting the additive shares into multiplicative shares. Therefore, the inversion can then be handled by inverting each multiplicative share. Eventually, the multiplicative shares are converted back to additive shares.

Algorithm 16 describes the whole process.

Algorithm 16 SecKeyGeneration

Output: A Boolean masking of the private key $((h_{0,i}, h_{1,i}, \sigma_i))_{1 \leq i \leq n}$ and the corresponding public key h.

$sh_{0,1}, \ldots, sh_{0,n} \leftarrow \mathcal{M}^n$
$sh_{1,1}, \ldots, sh_{1,n} \leftarrow \mathcal{M}^n$
$h_{0,s,1}, \ldots, h_{0,s,n} \leftarrow \mathsf{SecSampleH}((sh_{0,1}, \ldots, sh_{0,n}), w/2, r)$
$h_{1,s,1}, \ldots, h_{1,s,n} \leftarrow \mathsf{SecSampleH}(((sh_{1,1}, \ldots, sh_{1,n}), w/2, r)$
$h_{0,1}, \ldots, h_{0,n} \leftarrow \mathsf{DSSecMult}((1, 0, \ldots, 0), (h_{0,s,1}, \ldots, h_{0,s,n})$
$h_{1,1}, \ldots, h_{1,n} \leftarrow \mathsf{DSSecMult}((1, 0, \ldots, 0), (h_{1,s,1}, \ldots, h_{1,s,n}))$
$h_{0,1,inv}, \ldots, h_{0,n,inv} \leftarrow \mathsf{SecInv}(h_{0,1}, \ldots, h_{0,n})$
$h_1, \ldots, h_n \leftarrow \mathsf{SecMult}((h_{1,1}, \ldots, h_{1,n}), (h_{0,1,inv}, \ldots, h_{0,n,inv}))$
$h_1, \ldots, h_n \leftarrow \mathsf{Refresh}(h_1, \ldots, h_n)$
$h \leftarrow \sum_{i=1}^{n} h_i$
$\sigma_1, \ldots, \sigma_n \leftarrow \mathcal{M}^n$
return $(h_{0,i}, h_{1,i}, \sigma_i)_{1 \leq i \leq n}, h$

Complexity. Given the complexity of each part of the Key Generation, we deduce the complexity of Alg. 16 is:

$$T_{\mathsf{SecKeygen}}(r, w, n) = \mathcal{O}(w^2 \cdot n^2).$$

Security. The following theorem shows the Key generation achieves NIo security when the public key h is given to the simulator.

Theorem 8 ($(n-1)$-NIo security of SecKeyGen). *For any set of t_1 intermediate variables such that $t_1 < n$. The t_1 intermediate variables can be perfectly simulated from h*

Proof. The proof of Theorem 8 follows from the composition of each individual part.

7 Masking of the Encapsulation

The masking of the encapsulation is straightforward given the masking of its individual parts described above.

In particular, the main difficulty in masking the encapsulation is the sampling of the fixed weight error vector, which is performed by the hash function $\mathbf{H}$. The masking of $\mathbf{H}$ has already been studied in Sec. 6.1.

This procedure outputs a list of t indexes arithmetically masked modulo r. This sparse representation of the error vector can then be converted into a dense representation using Alg. 15 with polynomial arithmetic in $\mathbb{F}_2[X]/(X^{2r}-1)$ instead of $\mathbb{F}_2[X]/(X^r-1)$. The result is therefore an arithmetic sharing of a polynomial $e \in \mathbb{F}_2[X]/(X^{2r}-1)$ from which we deduce the error vector by splitting the polynomial into low and high monomials: $e = e_0 + X^r e_1$.

The whole procedure is depicted in Alg. 17.

Algorithm 17 SecEncaps

Input: A public key h.

Output: A Boolean masking of a session key $(K_1, \ldots, K_n)$ and its encapsulation c under h.

$m_1, \ldots, m_n \leftarrow_{\$} \mathcal{M}^n$
$e_{s,1}, \ldots, e_{s,n} \leftarrow \mathsf{SecSampleH}((m_1, \ldots, m_n), 2 \cdot r, t)$
$((e_{0,1}, e_{1,1}), \ldots, (e_{0,n}, e_{1,n}) \leftarrow \mathsf{DSSecMult}((1, 0, \ldots, 0), (e_{s,1}, \ldots, e_{s,n}))$
for $i = 1$ **to** n, $c_{0,i} \leftarrow e_{0,i} \oplus e_{1,i} \cdot h$
$c_{0,1}, \ldots, c_{0,n} \leftarrow \mathsf{Refresh}(c_{0,1}, \ldots, c_{0,n})$
$c_0 \leftarrow \sum_{i=1}^{n} c_{0,i}$
$mask_1, \ldots, mask_n \leftarrow \mathbf{SecL}((e_{0,i}, e_{1,i})_{1 \le i \le n})$
for $i = 1$ **to** n, $c_{1,i} \leftarrow m_i \oplus mask_i$
$c_{1,1}, \ldots, c_{1,n} \leftarrow \mathsf{Refresh}(c_{1,1}, \ldots, c_{1,n})$
$c_1 \leftarrow \sum_{i=1}^{n} c_{1,i}$
$c \leftarrow (c_0, c_1)$
$K_1, \ldots, K_n \leftarrow \mathbf{SecK}((m_i, c)_{1 \le i \le n})$
return $(K_1, \ldots, K_n), c$

Complexity. Given the complexity of each part of the Encapsulation, we deduce the complexity of Alg. 17 is:

$$T_{\mathsf{SecEncaps}}(r, w, n) = \mathcal{O}(w^2 \cdot n^2).$$

Security. The following theorem shows the Encapsulation achieves NIo security when the ciphertext is given to the simulator c.

Theorem 9 (**$(n-1)$-NIo security of SecEncaps**). *For any set of t_1 intermediate variables such that $t_1 < n$. The t_1 intermediate variables can be perfectly simulated from c.*

Proof. The proof follows from the composition of each individual part composed with sharewise operations.

8 Implementation Results

In order to assess the correctness and efficiency of our countermeasures, we performed a proof-of-concept implementation in the language `C`.

In the following section, we review the performance of our implementation. It's worth noting that all benchmarks were conducted on an `AMD RYZEN 5 PRO 3400G with Radeon Vega Graphics 3.70GHz` CPU.

Hereafter, we present the benchmark results for the entire execution of the BIKE scheme. The results are shown in Table 2 for several masking orders up to order 6 in thousands of CPU cycles. We also compare our results to the reference implementation described in [ABB+22], which was submitted to the fourth round of the NIST competition.

Order t		Ref	0	1	2	4	6
Level 1	Keygen	589	1 512 316	1 791 149	2 952 493	7 874 062	15 242 222
	Encaps	97	76 650	146 884	280 769	778 527	1 227 860
	Decaps	1 135	189 968 064	4 187 647 683	5 641 387 112	19 211 783 573	33 996 803 704
Level 3	Keygen	1 823	8 106 113	14 685 250	24 131 779	106 584 362	180 601 352
	Encaps	223	355 894	1 890 992	3 560 430	8 010 906	13 575 219
	Decaps	3 887	2 616 924 564	11 233 101 479	23 388 097 962	52 435 693 469	81 828 924 691
Level 5	Keygen	–	72 568 264	88 779 611	116 292 757	347 387 738	412 328 052
	Encaps	–	4 772 084	6 828 964	13 115 061	24 088 345	33 576 154
	Decaps	–	6 482 717 125	26 717 510 118	57 632 897 876	103 659 520 225	155 281 028 553

Table 2. Cycle Counts for full implementation of BIKE for each level of security. Results in thousand of Cycles

We can observe a significant overhead between the reference implementation and our unmasked implementation. This overhead, particularly evident in the cases of key generation and encapsulation, is mainly due to the implementation of polynomial arithmetic in our implementation. Specifically, the polynomial multiplication we implemented is the naive schoolbook multiplication, which lacks many optimizations. As shown in the complexity breakdown by operations in Tables 3 and 4, polynomial arithmetic represents the bottleneck of these algorithms.

Order t	Ref	0	1	2	4	6	8
$h_0, h_1 \leftarrow \mathcal{H}_w$	–	1 245	8 730	21 086	61 205	109 608	193 278
Conversion	–	1 797	13 539	28 113	93 806	203 952	338 355
$h_{inv} \leftarrow h_0^{-1}$	–	1 434 019	1 532 175	2 200 004	5 486 895	10 403 040	17 422 752
Total	589	1 512 316	1 791 149	2 952 493	7 874 062	15 242 222	27 271 347

Table 3. Cycle Counts of KeyGen for NIST Level 1 of security. Cycle Counts are split into each main step of the process and expressed in thousands of cycles.

Order t	Ref	0	1	2	4	6	8
$e'_0, e'_1 \leftarrow \mathbf{H}(m)$	–	1 235	6 458	19 068	78 832	148 568	250 517
$s_{pub} \leftarrow e_0 \oplus e_1 \cdot h$	–	73 727	128 952	208 726	437 339	639 743	1 132 106
$c_1 \leftarrow m \oplus \mathbf{L}(e_0, e_1)$	–	196	7 127	31 027	192 434	287 010	600 875
$k \leftarrow \mathbf{K}(m, c)$	–	100	3 733	21 080	65 940	146 502	290 965
Total	97	76 650	146 884	280 769	778 527	1 227 860	2 566 731

Table 4. Cycle Counts of Encapsulation for NIST Level 1 of security. Cycle Counts are split into each main step of the process and expressed in thousands of cycles.

Nonetheless, Tables 3 and 4 demonstrate the efficiency and scalability of our countermeasure for fixed-weight sampling of polynomials, along with the conversion of the sparse representation, which was at the core of our countermeasures. We also note that our results are consistent with the theoretical complexity we provided for each gadget. However, a practical implementation should focus on implementing better arithmetic, particularly benefiting from `AVX2` and `AVX512` instruction sets as in the reference implementation.

Eventually, as shown in Table 5, the significant overhead suffered by our implementation of the decapsulation is primarily due to the decoder, for which achieving first-order security requires millions of cycles. This substantial overhead is explained by the theoretical complexity of our countermeasure, which is $\mathcal{O}(r^2n^2)$. In particular, our implementation differs from the reference implementation in the computation of a bit-flip iteration. In the reference implementation, the authors leverage the sparsity of the parity check matrix and bit-slice techniques to compute all unique parity checks (UPCs) in parallel. More precisely, the UPC $|s \wedge h_0|_1$ can be efficiently computed using only the positions of the nonzero coefficients of h_0 through:

$$|s \wedge h_0|_1 = \sum_{i, h_0^{(i)}=1} s^{(i)}$$

We describe in Appendix I an earlier attempt at an implementation of the computation of the UPC, following the same strategy as the Bike Specification [ABB+22]. However, this alternative method has an $\mathcal{O}(\log(w) \cdot r)$ space complexity, which makes it unrealistic for embedded implementations with limited RAM. We leave as an open problem to further improve this strategy to fit the limitations of embedded cryptography.

Order t		Ref	0	1	2	4	6
Decoder	$S \leftarrow \|s\|_1$	–	732	4 719	11 739	46 259	91 796
	threshold	–	2	18	57	186	301
	BFIter	–	13 377 779	234 574 115	408 338 435	1 384 092 908	3 077 727 375
	BFMaskedIter	–	14 218 186	584 971 905	1 128 762 724	3 020 288 121	6 218 525 136
	Total	–	187 465 013	4 185 318 401	5 639 079 456	19 209 286 698	33 993 736 284
$m \leftarrow c_1 \oplus \mathbf{L}(e_0, e_1)$		–	287	14 880	46 293	352 475	555 930
$e'_0, e'_1 \leftarrow \mathbf{H}(m)$		–	1 697	12 386	29 701	83 890	300 499
$K \leftarrow \mathbf{K}(m, c)$		–	156	9 582	23 984	70 068	336 181
Total		1 135	189 968 064	4 187 647 683	5 641 387 112	19 211 783 573	33 996 803 704

Table 5. Cycle Counts of Decapsulation for NIST Level 1 of security. Cycle Counts are split into each main step of the process and expressed in thousands of cycles.

9 Conclusion

In this work, we introduced the first high-order implementation of the entire BIKE scheme, marking a significant advancement in masking code-based cryptography. Additionally, we scrutinized a prior article addressing the masking technique applied to a previous decoder version, uncovering vulnerabilities and suggesting enhancements. Through a `C` implementation, we evaluated the efficiency and scalability of our countermeasures, presenting compelling experimental performance data. It is imperative to note that further optimization of polynomial arithmetic could greatly enhance the efficiency of key generation and encapsulation processes. Eventually, our implemented countermeasure introduces a significant overhead during decoder, impacting overall performance of the decapsulation. Despite efforts to optimize polynomial arithmetic, this overhead persists, posing a challenge in secure embedded implementations.

The primary factor behind this is the theoretical complexity of the Unsatisfied Parity Check (UPC) computation, which scales as $\mathcal{O}(r \cdot n^2)$. Addressing this complexity remains an open problem for future research, warranting further exploration and innovation.

We also note that we leave as an open problem the further evaluation of the practical security of our countermeasures by performing concrete leakage evaluation.

References

[ABB+22] Nicolas Aragon, Paulo SLM Barreto, Slim Bettaieb, Loic Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Shay Gueron, Tim Guneysu, Carlos Aguilar Melchor, et al. Bike: bit flipping key encapsulation, 2022.

[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 *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 116–129, 2016. Publicly available at `https://eprint.iacr.org/2015/506.pdf`.

[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 *Advances in Cryptology - EUROCRYPT 2018 - Proceedings, Part II*, pages 354–384, 2018.

[BDK+21] Shi Bai, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. Crystals-dilithium algorithm specifications and supporting documentation (version 3.1), 2021. `https://pq-crystals.org/dilithium/data/dilithium-specification-round3-20210208.pdf`.

[BP18] Daniel J Bernstein and Edoardo Persichetti. Towards kem unification. *Cryptology ePrint Archive*, 2018.

[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, *Selected Areas in Cryptography – SAC 2015*, pages 293–309, Cham, 2016. Springer International Publishing. https://eprint.iacr.org/2015/924.pdf.

[CGMZ23] Jean-Sébastien Coron, François Gérard, Simon Montoya, and Rina Zeitoun. High-order polynomial comparison and masking lattice-based encryption. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2023(1):153–192, 2023. `https://ia.cr/2021/1615`.

[CGTZ22] Jean-Sebastien Coron, François Gérard, Matthias Trannoy, and Rina Zeitoun. High-order masking of ntru. Cryptology ePrint Archive, Paper 2022/1188, 2022. `https://eprint.iacr.org/2022/1188`.

[CGTZ23] Jean-Sébastien Coron, François Gérard, Matthias Trannoy, and Rina Zeitoun. Improved gadgets for the high-order masking of dilithium, 2023.

[Cor14] Jean-Sébastien Coron. Higher order masking of look-up tables. In *Proceedings of EUROCRYPT 2014*, pages 441–458, 2014.

[DGK19] Nir Drucker, Shay Gueron, and Dusan Kostic. Qc-mdpc decoders with several shades of gray. Cryptology ePrint Archive, Paper 2019/1423, 2019. `https://eprint.iacr.org/2019/1423`.

[DR24] Loïc Demange and Mélissa Rossi. A provably masked implementation of bike key encapsulation mechanism. Cryptology ePrint Archive, Paper 2024/076, 2024. `https://eprint.iacr.org/2024/076`.

[Gal62] Robert Gallager. Low-density parity-check codes. *IRE Transactions on information theory*, 8(1):21–28, 1962.

[GSM17] Hannes Gross, David Schaffenrath, and Stefan Mangard. Higher-order side-channel protected implementations of keccak. Cryptology ePrint Archive, Paper 2017/395, 2017. `https://eprint.iacr.org/2017/395`.

[ISW03] Yuval Ishai, Amit Sahai, and David A. Wagner. Private circuits: Securing hardware against probing attacks. In *CRYPTO 2003, Proceedings*, pages 463–481, 2003.

[KLRBG22] Markus Krausz, Georg Land, Jan Richter-Brockmann, and Tim Güneysu. Efficiently masking polynomial inversion at arbitrary order. Cryptology ePrint Archive, Paper 2022/707, 2022. `https://eprint.iacr.org/2022/707`.

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

[SPOG19] Tobias Schneider, Clara Paglialonga, Tobias Oder, and Tim Güneysu. Efficiently masking binomial sampling at arbitrary orders for lattice-based crypto. In *PKC 2019, Proceedings, Part II*, pages 534–564, 2019.

A Attack on [CEvMS16]

Algorithm 18 Attack Pseudo-Code

Input: A public key h, An oracle returning the first UPC $\mathcal{O}_{upc}$
Output: A decryption key (h_0, h_1) for the public key h.

$h_0 \leftarrow 1$
$i_0 \leftarrow 0$
for $i = 0$ to $r - 1$ **do**
  if $\mathcal{O}_{upc}(X^i) = 1$ **then**
    $i_0 \leftarrow i$
    $h_0 \leftarrow h_0 \oplus X^{i_0}$
    break for
  end if
end for
for $j = 0$ to $r - 1$ **do**
  if $j \neq i_0$ **then**
    if $\mathcal{O}(X^{i_0}) + \mathcal{O}(X^j) - \mathcal{O}(X^{i_0} \oplus X^j) \neq 0$ **then**
      $h_0 \leftarrow h_0 \oplus X^j$
    end if
  end if
end for
$h_1 \leftarrow h \cdot h_0$
return h_0, h_1

B 1bitB2A from [SPOG19]

We recall hereafter algorithm introduced in [SPOG19] that performs free-SNI - proved in [CGTZ23] - conversion from 1 bit Boolean sharing to arithmetic sharing over arbitrary modulus.

Algorithm 19 1bitB2A

Input: $x_1, \ldots, x_n \in \{0, 1\}$
Output: $v_1, \ldots, v_n \in \mathbb{Z}_q$ such that $v_1 + \cdots + v_n \bmod q = x_1 \oplus \cdots \oplus x_n$

$x_{n+1} \leftarrow x_1$, $v_1 \leftarrow x_2$ ▷ $v_1 = x_2 \pmod q$
for $i = 2$ to n **do**
  $(v_1, \ldots, v_i) \leftarrow \mathsf{LinearRefresh}_{\mathbb{Z}_q}(v_1, \ldots, v_{i-1}, 0)$
  $(v_1, \ldots, v_i) \leftarrow (1 - 2x_{i+1}) \cdot (v_1, \ldots, v_i) \bmod q$
  $v_1 \leftarrow v_1 + x_{i+1} \bmod q$ ▷ $\sum_{j=1}^{i} v_i = x_2 \oplus \cdots \oplus x_{i+1} \pmod q$
end for
$(v_n, \ldots, v_1) \leftarrow \mathsf{LinearRefresh}_{\mathbb{Z}_q}(v_n, \ldots, v_1)$
return $(v_1, \ldots, v_n)$

C ShiftMod from [CGTZ23]

We recall hereafter algorithm introduced in [CGTZ23] that performs t-NI Boolean shift over arithmetic sharing on even moduli with complexity $\mathcal{O}(n^2)$

Algorithm 20 ShiftMod

Input: A modulus $q' = 2q$ and $x_1, \ldots, x_n \in \mathbb{Z}_{2q}$
Output: $a_1, \ldots, a_n \in \mathbb{Z}_q$ such that $a_1 + \cdots + a_n = \lfloor (x_1 + \cdots + x_n)/2 \rfloor \pmod q$

for $i = 1$ to n **do** $b_i \leftarrow x_i \,\&\, 1$
$(y_1, \ldots, y_n) \leftarrow \mathsf{1bitB2A}(2q, (b_1, \ldots, b_n))$
for $i = 1$ to n **do** $z_i \leftarrow x_i - y_i \bmod 2q$
for $i = 1$ to $n - 1$ **do**
  $z_n \leftarrow z_n + (z_i \,\&\, 1) \bmod 2q$
  $z_i \leftarrow z_i - (z_i \,\&\, 1) \bmod 2q$
end for
for $i = 1$ to n **do** $a_i \leftarrow z_i \gg 1$
return $a_1, \ldots, a_n$

D Masking the maximum

In this section, we demonstrate how to mask the maximum operation. We observe that the maximum of two values, a and b, can be expressed as follows: First, we compute the bit $d = [\![a < b]\!]$. Then, based on the value of d, we return b if $d = 1$, and a otherwise. The bit d can be computed in a high-order manner by shifting an arithmetic sharing of the value $b - a$ using the ShiftMod operation from [CGTZ23], as recalled in Algorithm 20 in Appendix C. This approach is similar to the one in [SPOG19], but we avoid converting the arithmetic sharing into Boolean sharing and performing individual shifts on each sharing. This optimization is made possible by utilizing a power of 2 modulus instead of a prime modulus as in [SPOG19]. The algorithm for testing the positivity of a number is described in Algorithm 21.

Algorithm 21 Masked Test of Positivity SecIsPositive

Input: A power of 2 modulus 2^α, an arithmetic sharing modulo 2^α $(x_1, \ldots, x_n)$ of a value $-2^{\alpha-1} \leq x < 2^\alpha$.
Output: A Boolean masking $(b_1, \ldots, b_n)$ of a bit b such that $b = 1$ iff $x \geq 0$.

$b_1, \ldots, b_n \leftarrow x_1, \ldots, x_n$
for $j = 1$ to $\alpha - 1$ **do**
    $b_1, \ldots, b_n \leftarrow \mathsf{ShiftMod}(2^{\alpha-j+1}, (b_1, \ldots, b_n))$
end for
$b_1 \leftarrow b_1 \oplus 1$
return $b_1, \ldots, b_n$

Once the bit d is computed in Boolean sharing, we can apply the high-order table-based computation from [Cor14] to the function $f : \{0, 1\} \to \{a, b\}$, where $f(0) = a$ and $f(1) = b$. The description of this high-order computation named HOTSwap is described in Alg. 22.

Algorithm 22 Secure Swap HOTSwap

Input: A Boolean sharing $(d_1, \ldots, d_n)$ of a bit d, two sharings $(a_1, \ldots, a_n)$ and $(b_1, \ldots, b_n)$ of two values a and b
Output: A sharing of the value a if $d = 0$ or b if $d = 1$

$T[0] \leftarrow (a_1, \ldots, a_n)$
$T[1] \leftarrow (b_1, \ldots, b_n)$
for $i = 1$ to $n - 1$ **do**
    $T' \leftarrow T[1]$
    $T[d_i] \leftarrow T[0]$
    $T[d_i \oplus 1] \leftarrow T'$
    $T[0] \leftarrow \mathsf{LinearRefresh}(T[0])$
    $T[1] \leftarrow \mathsf{LinearRefresh}(T[1])$
end for
return $T[d_n]$

Eventually, the masking of the entire procedure is described in Algorithm 23.

Algorithm 23 Masked maximum SecMax

Input: An arithmetic sharing mod 2^α $a_1, \ldots, a_n$ of a value a and an arithmetic sharing mod 2^α $b_1, \ldots, b_n$ of a value b.
Output: An arithmetic sharing mod 2^α $c_1, \ldots, c_n$ of $\max(a, b)$

for $i = 1$ **to** n **do** $d_i \leftarrow b_i - a_i$
$d_1, \ldots, d_n \leftarrow \mathsf{SecIsPositive}(2^\alpha, (d_1, \ldots, d_n))$
$d_1, \ldots, d_n \leftarrow \mathsf{Refresh}(d_1, \ldots, d_n)$
$(c_1, \ldots, c_n) \leftarrow \mathsf{HOTSwap}((d_1, \ldots, d_n), (a_1, \ldots, a_n), (b_1, \ldots, b_n))$
return $c_1, \ldots, c_n$

Complexity. Given the complexity of the ShiftMod: $T_{\mathsf{ShiftMod}} = 2n^2+10n-9$ and LinearRefresh: $T_{\mathsf{LinearRefresh}}(n) = 3n-3$, we deduce the complexity of Alg. 23 to be:

$$\begin{aligned} T_{\mathsf{SecMax}}(n,\alpha) &= n + \alpha \cdot T_{\mathsf{ShiftMod}}(n) + T_{\mathsf{Refresh}}(n) + 2 \\ &\quad + (n-1)\cdot(3 + 2\cdot T_{\mathsf{LinearRefresh}}(n)) + T_{\mathsf{LinearRefresh}}(n)) \\ &= n + 2\alpha n^2 + 10\alpha n - 9\alpha + 3n^2 - 3n + 2 + (n-1)(6n-3) + 3n - 3 \\ &= (2\alpha+9)n^2 + (10\alpha - 8)n - 9\alpha + 2 \\ &= \mathcal{O}(\alpha n^2) \end{aligned}$$

Security. The following theorem shows our algorithm achieves the stronger SNI security.

Theorem 10 ($(n-1)-$SNI of SecMax). *For any t_1 intermediate variables and any subset $O \subset [1,n]$ such that $t_1 + |O| < n$. There exists a subset $I \subset [1,n]$, with $|I| \leq t_1$, such that the t_1 intermediate variables and the outputs $c_{|O}$ can be perfectly simulated from inputs $a_{|I}$ and $b_{|I}$*

Proof. The proof follows from the compositions of 2 SNI parts.

E ZeroTestBool from [CGMZ23]

We recall hereafter the algorithm introduced in [CGMZ23] that performs t-SNI zero-test over Boolean sharing with complexity $\mathcal{O}(\log(k)\cdot n^2)$.

Algorithm 24 ZeroTestBool

Input: $k \in \mathbb{Z}$ and $x_1, \ldots, x_n \in \{0,1\}^k$
Output: $b_1, \ldots, b_n \in \{0,1\}$ with $\oplus_{i=1}^n b_i = 1$ if $\oplus_{i=1}^n x_i = 0$ and $\oplus_{i=1}^n b = 0$ otherwise

1: $m \leftarrow \lceil \log_2 k \rceil$
2: $y_1 \leftarrow \overline{x_1}$ or $(2^{2^m} - 2^k)$
3: **for** $i = 2$ to n **do** $y_i \leftarrow x_i$
4: **for** $i = 0$ to $m-1$ **do**
5: $\quad (z_1, \ldots, z_n) \leftarrow \mathsf{RefreshMasks}(y_1 \gg 2^i, \ldots, y_n \gg 2^i)$
6: $\quad (y_1, \ldots, y_n) \leftarrow \mathsf{SecAnd}(m, (y_1, \ldots, y_n), (z_1, \ldots, z_n))$
7: **end for**
8: **return** $(y_1 \,\&\, 1, \ldots, y_n \,\&\, 1)$

F High-Order Membership Test

In order to perform the membership test needed for the sampling of fixed weight polynomials in 13 we describe 2 methods based on the parameters.

Given an arithmetic sharing $x_1, \ldots, x_n$ of a value x and a list of t arithmetic sharings $(w_1^{(i)}, \ldots, w_n^{(i)})_{1\leq i\leq t}$ we actually test:

$$\begin{aligned} x \in (w^{(i)})_{1\leq i\leq t} &\Leftrightarrow \exists i,\ x = w^{(i)} \\ &\Leftrightarrow \bigvee_{1\leq i\leq t} [\![x - w^{(i)} = 0]\!] = 1 \\ &\Leftrightarrow \bigoplus_{1\leq i\leq t} [\![x - w^{(i)} = 0]\!] = 1 \end{aligned}$$

The last equivalence holds since we assume the elements of $(w^{(i)})$ to be distinct and therefore there is at most one i for which $x = w^{(i)}$. In order to high-order compute the zero-test $[\![x - w^{(i)} = 0]\!]$ we employ

Alg. 24 from [CGMZ23] preceded by an arithmetic to Boolean conversion since the value x and $w^{(i)}$ are arithmetically masked. The whole procedure is described in Alg. 25.

Algorithm 25 Masked Membership Test SecMembership

Input: An arithmetic sharing modulo q $(x_1, \ldots, x_n)$ of a value x, a list of t arithmetic sharings modulo q $(w_1^{(i)}, \ldots, w_n^{(i)})_{1 \leq i \leq t}$ of distinct values $(w^{(i)})_{1 \leq i \leq t}$

Output: A Boolean masking $(b_1, \ldots, b_n)$ of a bit b such that $b = 1$ iff $x = w^{(i)}$ for some i

1: **for** $i = 1$ to t **do**
2: **for** $k = 1$ **to** n **do** $p_k^{(i)} \leftarrow w_k^{(i)} - x_k \bmod q$
3: **end for**
4: $b_1, \ldots, b_n \leftarrow (0, 0, \ldots, 0)$
5: **for** $i = 1$ to t **do**
6: $a_1, \ldots, a_n \leftarrow \mathsf{AtoB}_q(p_1^{(i)}, \ldots, p_n^{(i)})$
7: $b'_1, \ldots, b'_n \leftarrow \mathsf{ZeroTestBool}(a_1, \ldots, a_n)$
8: **for** $k = 1$ to n **do**
9: $b_k \leftarrow b'_k \oplus b_k$
10: **end for**
11: **end for**
12: **return** $b_1, \ldots, b_n$

Complexity. Counting the operations as above, we deduce the complexity of Alg. 25 to be:

$$\begin{aligned} T_{\mathsf{SecMembership}}(n, q, t) &= n + t \cdot (T_{\mathsf{AtoB}}(n, q) + T_{\mathsf{ZeroTestBool}}(n, \log(q)) + n) \\ &= \mathcal{O}(t \log(q) \cdot n^2) \end{aligned}$$

Security. The following theorem shows our algorithm achieves the NI security.

Theorem 11 **($(n-1)-$NI of SecMembership).** *For any t_1 intermediate variables and any subset $O \subset [1, n]$ such that $t_1 + |O| < n$. There exists a subset $I \subset [1, n]$, with $|I| \leq t_1 + |O|$, such that the t_1 intermediate variables and the outputs $b_{|O}$ can be perfectly simulated from inputs $x_{|I}$ and $(w_{|I}^{(i)})_{1 \leq i \leq t}$*

Proof. The proof follows from the compositions of the NI AtoB conversion, the NI ZeroTestBool zero-test and sharewise operations.

In the particular case of BIKE, the memebership test is used during the sampling of the private key in Alg. 1 and the sampling of the error vector in Alg. 2 and 3. For the sampling of the private key, the masking modulus, r, is a prime number and therefore we can improve the membership test using the following equivalence:

$$x \in (w^{(i)})_{1 \leq i \leq t} \Leftrightarrow \prod_{1 \leq i \leq t} (x - w^{(i)}) = 0$$

The equivalence is based on the integrity of the field $\mathbb{Z}_r$, i.e $ab = 0 \Rightarrow a = 0 \vee b = 0$. This structure allows us to perform only one arithmetic to boolean conversion and one high-order zero test as described in Alg. 26. We note that this optimization can not be applied to the sampling of the error vector in Encapsulation and Decapsulation since the modulus involved, $2 \cdot r$, is composite.

Algorithm 26 Masked Membership Test SecMembershipPrime

Input: A prime number r, An arithmetic sharing modulo r $(x_1, \ldots, x_n)$ of a value x, a list of t arithmetic sharings modulo r $(w_1^{(i)}, \ldots, w_n^{(i)})_{1 \leq i \leq t}$ of values $(w^{(i)})_{1 \leq i \leq t}$

Output: A Boolean masking $(b_1, \ldots, b_n)$ of a bit b such that $b = 1$ iff $x = w^{(i)}$ for some i

1: **for** $i = 1$ to t **do**
2: **for** $k = 1$ **to** n **do** $p_k^{(i)} \leftarrow w_k^{(i)} - x_k \bmod r$
3: **end for**
4: $prod_1, \ldots, prod_n \leftarrow (1, 0, \ldots, 0)$
5: **for** $i = 1$ to t **do**
6: $prod_1, \ldots, prod_n \leftarrow \mathsf{SecMult}((prod_1, \ldots, prod_n), (p_1^{(i)}, \ldots, p_n^{(i)}))$
7: **end for**
8: $prod_1, \ldots, prod_n \leftarrow \mathsf{AtoB}_r(prod_1, \ldots, prod_n)$
9: $b_1, \ldots, b_n \leftarrow \mathsf{ZeroTestBool}(prod_1, \ldots, prod_n)$
10: **return** $b_1, \ldots, b_n$

Complexity. Counting the operations as above, we deduce the complexity of Alg. 26 to be:

$$\begin{aligned} T_{\mathsf{SecMembership}}(n, r, t) &= n + t \cdot T_{\mathsf{SecMult}}(n) + T_{\mathsf{AtoB}}(n, r) + T_{\mathsf{ZeroTestBool}}(n, \log(r)) \\ &= \mathcal{O}(t \cdot n^2 + \log(r) n^2 \end{aligned}$$

Security. The following theorem shows our algorithm achieves the stronger SNI security.

Theorem 12 ($(n-1)$ − SNI **of** SecMembershipPrime)**.** *For any t_1 intermediate variables and any subset $O \subset [1, n]$ such that $t_1 + |O| < n$. There exists a subset $I \subset [1, n]$, with $|I| \leq t_1 + |O|$, such that the t_1 intermediate variables and the outputs $b_{|O}$ can be perfectly simulated from inputs $x_{|I}$ and $(w_{|I}^{(i)})_{1 \leq i \leq t}$*

Proof. The proof follows from the compositions of the NI AtoB conversion, the NI ZeroTestBool with the SNI gadget SecMult. We note that the multiplication at the first iteration of the **for** loop with $prod = 1$ corresponds to a SNI full refresh on the masks of $p^{(1)}$ therefore it decorrelates the masks of $p^{(i)}$ and $prod$ ensuring the SNI composition.

G Proof of Theorem 7

In order to prove the SNI security of Alg. 15, we introduce Alg. 28 which corresponds to the processing of the ℓ first shares of an index $g_s^{(k)}$ of the sparse representations. We remark the algorithm can be rewritten as in Alg. 27 where the core of the for loop is presented in Alg. 28 for $\ell = n$. The proof strategy is as follow, we first show by induction on ℓ that Alg. 28 achieves SNI security and then we conclude by composition that the overall algorithm achieve SNI security.

Algorithm 27 Secure Multiplication of dense and sparse polynomial DSSecMult

Input: A masked dense polynomial $h_1, \ldots, h_n$ and a masked sparse polynomial $g_{s,1}, \ldots, g_{s,n}$ of weight wt.

Output: A masked dense polynomial $f_1, \ldots, f_n$ of the product $h \cdot g$

1: $f_1, \ldots, f_n \leftarrow (0, \ldots, 0)$
2: **for** $k = 1$ to wt **do**
3: $sum_1, \ldots, sum_1 \leftarrow \mathsf{P_n}((h_1, \ldots, h_n), (g_{s,1}^{(k)}, \ldots, g_{s,n}^{(k)}))$
4: **For** $i = 1$ **to** n, $f_i \leftarrow f_i + sum_i$
5: **end for**
6: **return** $f_1, \ldots, f_n$

Algorithm 28 Core of for loop of P_ℓ

Input: A masked dense polynomial $h_1, \ldots, h_n$ and an arithmetic sharing modulo r $x_1, \ldots, x_n$ of an index of a term X^x in g_s.

Output: A masked dense polynomial $f_1, \ldots, f_n$ of the product $h \cdot X^{x_{|\ell}}$ where $x_{|\ell} = \sum_{i=1}^{\ell} x_i \bmod r$.

$f_1, \ldots, f_n \leftarrow h_1, \ldots, h_n$
for $i = 1$ to ℓ **do**
  for $j = 1$ **to** n **do** $f_j \leftarrow f_j \cdot X^{x_i} \bmod (X^r - 1)$
  for $j = 1$ to $n - 1$ **do** ▷ Linear Refresh
    $p_{i,j} \leftarrow_\$ \mathbb{F}_2[X]/(X^r - 1)$
    $f_j \leftarrow f_j + p_{i,j}$
    $f_n \leftarrow f_n - p_{i,j}$
  end for
end for
return $f_1, \ldots, f_n$

SNI security of P_ℓ We show by induction on $1 \leq \ell \leq n$ the following property: If $\mathsf{P}_{\ell-1}$ achieves NI security then P_ℓ achieves SNI security.

The base step is trivial, since P_0 is the algorithm that returns the input $h_1, \ldots, h_n$ and is therefore NI.

We will now show the induction step.

We assume $\mathsf{P}_{\ell-1}$ achieves NI security for some $1 \leq \ell \leq n$. Let $\mathcal{W}$ be any set of t_0 intermediate variables in P_ℓ and $O \subset [1, n]$ such that $t_0 + |O| \leq n$. We show how to construct I of cardinality $|I| \leq t_0$ such that the t_0 intermediate variables of $\mathcal{W}$ and the outputs $f_{|O}$ can be perfectly simulated from $h_{|I}$ and $x_{|I}$.

First, we split $\mathcal{W} = \mathcal{W}_1 \cup \mathcal{W}_2$ where $|\mathcal{W}_i| = t_i$ and $\mathcal{W}_1$ corresponds to the variables probed during the execution of the first $\ell - 1$ iteration of the main loop, i.e in $\mathsf{P}_{\ell-1}$ and $\mathcal{W}_2$ the variables probed during the execution of the ℓ-th iteration of the main loop.

By induction hypothesis, there exists I_1, such that the t_1 variables form $\mathcal{W}_1$ can be perfectly simulated from $h_{|I_1}$ and $x_{|I_1}$.

We now show how to simulate the remaining $\mathcal{W}_2$ variables by extending our current input set. We start with $I = I_1$.

- For each probe $p_{\ell,j} \in \mathcal{W}_2$, we add j to I
- For each probe $f_j + p_{\ell,j} \in \mathcal{W}_2$, we add j to I
- For each probe $f_n - p_{\ell,j} \in \mathcal{W}_2$, we add n to I
- For each probe $f_j \cdot X^{x_\ell} \in \mathcal{W}_2$, we add j to I if $j \notin I$ or l to I if $j \in I$.

By construction, we have $|I| \leq |I_1| + t_2 \leq t_1 + t_2 = t_0$. Since the new set I contains I_1, by extension of the input set we can perfectly simulate all variables in $\mathcal{W}_1$ with input $h_{|I}$ and $x_{|I}$.

By construction, we can trivially simulate each variables of $\mathcal{W}_2$ of the form $p_{\ell,j}$, $f_j + p_{\ell,j}$, $f_n - p_{\ell,j}$ by simulating $p_{\ell,j}$ with uniform random variables and propagating the simulation since f_j can be simulated from $h_{|I}$.

To simulate the remaining variables of $\mathcal{W}_2$ (of the form $f_j \cdot X^{x_\ell}$), we distinguish two cases depending on $l \in I$ or not.

Case $l \in I$: If $\ell \in I$ by construction $j \in I$, therefore any probe $f_j \cdot X^{x_\ell}$ can be perfectly simulated from input $h_{|I}$ and $x_{|I}$.

Case $l \notin I$: If $l \notin I$, then by construction $f_j \cdot X^{x_\ell}$ is the only probe involving f_j and $j \in I$. Therefore we can perfectly simulate f_j before the shift by propagating simulation from $h_{|I}$ and $x_{|I}$. The value $f_j \cdot X^{x_i}$ can then be simulated by $f_j \cdot X^u$ with a uniform u.

It remains to show how to simulate the outputs $f_{|O}$ from inputs $h_{|I}$ and $x_{|I}$.

For $o \in O \cap I$, the output f_o can be perfectly simulated from inputs $h_{|O}$ and $x_{|O}$ by propagating the simulations.

For $o \in O \setminus I \setminus \{n\}$, by construction none of $p_{\ell,o}$, $f_o + p_{\ell,o}$ and $f_n - p_{\ell,o}$ has been probed. Therefore, $p_{\ell,o}$ acts as a one time pad on f_o and we can perfectly simulate the output with a uniform random.

For $n \in O \setminus I$, the output f_n corresponds to $f'_n - \sum_j p_{\ell,j}$ where f'_n represents the value of f_n at the beginning of the last iteration $i = l$. From the hypothesis $t_1 + |O| < n$, we know there exists $j^\star$ such that $j \in [1, n] \setminus (I \cup O)$. By expressing f_n as $f_n = (f'_n - \sum_{j \neq j^\star} p_{\ell,j}) - p_{\ell,j^\star}$ we remark that $p_{\ell,\star}$ is independent of all simulated variables and acts as a one time pad on f_n. Henceforth, the output f_n can be perfectly simulated with a uniform random.

Finally, we have shown the induction step, i.e $\mathsf{P}_{\ell-1}$ is NI implies P_ℓ achieves SNI. Since P_0 is NI, we conclude that P_n achieves the SNI security.

SNI security of DSSecMult Given the SNI security of the core of the for loop P_n we deduce the SNI of Alg. 27 since it consists in the composition of wt SNI part with sharewise operations.

H SecInv from [KLRBG22]

In their work, the authors of [KLRBG22] demonstrated an efficient method for inversing an element that is arithmetically masked over an arbitrary ring. Their method is based on converting from arithmetic sharing into multiplicative sharing, as inversion is linear under the multiplicative law. To that extent, they also described the converse conversion from multiplicative to additive masking. We recall their algorithm to convert from additive masking to multiplicative masking in Algorithm 29, and the reverse conversion from multiplicative to additive sharing in Algorithm 30.

Algorithm 29 Additive to multiplicative conversion ($\mathsf{A2M_{INV}}$)

Input: An arithmetic sharing $(a_1, \ldots, a_n)$ of an invertible $a \in \mathbb{F}_2[X]/(X^r - 1)$
Output: A multiplicative sharing $(m_1, \ldots, m_n)$ of the inverse of a^{-1}

1: **for** $i = 1$ to n **do**
2: $\quad r_i \leftarrow \mathbb{F}_2[X]/(X^r - 1)^\star$
3: $\quad$ **for** $j = 1$ **to** n **do** $a_j \leftarrow r_i \cdot a_j$
4: $\quad a_1, \ldots, a_n \leftarrow \mathsf{LinearRefresh}(a_1, \ldots, a_n)$
5: $\quad m_i \leftarrow r_i$ $\quad \triangleright\ a = \left(\sum_{j=1}^n a_j\right) \prod_{j=1}^i m_j^{-1}$
6: **end for**
7: $m_1 \leftarrow m_1 \cdot (\sum_{j=1}^n a_j)^{-1}$ $\quad \triangleright\ a^{-1} = m_1 \cdot m_2 \cdot \cdots \cdot m_n$
8: **return** $m_1, \ldots, m_n$

Algorithm 30 Multiplicative to additive conversion (M2A)

Input: A multiplicative sharing $(m_1, \ldots, m_n)$ of an invertible element $m \in \mathbb{F}_2[X]/(X^r - 1)$
Output: An arithmetic sharing $(a_1, \ldots, a_n)$ of m

1: $a_1 \leftarrow m_1$
2: **for** $i = 1$ to $n - 1$ **do**
3: $\quad a_1, \ldots, a_{i+1} \leftarrow \mathsf{LinearRefresh}(a_1, \ldots, a_i, 0)$
4: $\quad$ **for** $j = 1$ to $i + 1$ **do** $a_j \leftarrow a_j \cdot m_{i+1}$
5: **end for**
6: **return** $a_1, \ldots, a_n$

We note that Algorithm 29 actually converts into the inverse of the input. This optimization allows for only 1 inversion instead of $2n-1$ in the original algorithm, where $n-1$ shares were inverted twice. Moreover, Algorithm 29 is the corrected version by [CGTZ22], as the original version suffered from a third-order attack.

Therefore we can deduce the following inversion algorithm depicted in Alg. 31

Algorithm 31 Inversion based on multiplicative masking $\mathsf{INV}_{\mathsf{Mul}}$

Input: An arithmetic sharing $(a_1, \ldots, a_n)$ of an invertible $a \in \mathbb{F}_2[X]/(X^r - 1)$
Output: An arithmetic sharing $(b_1, \ldots, b_n)$ of the inverse a^{-1}

1: $m_1, \ldots, m_n \leftarrow \mathsf{A2M}_{\mathsf{INV}}(a_1, \ldots, a_n)$
2: $b_1, \ldots, b_n \leftarrow \mathsf{M2A}(m_1, \ldots, m_n)$
3: **return** $b_1, \ldots, b_n$

I Alternative Computation of the UPC

In this section, we introduce an alternative approach to computing the Unsatisfied Parity Check (UPC). Rather than calculating the UPC dynamically during each Bit-Flip iteration, we choose to precompute all UPC values and store them in an array. This strategy results in a substantial complexity improvement of Algorithm 9 by a factor of $\mathcal{O}(w/r)$. However, this optimization requires storing all UPC values, leading to a space complexity of $\mathcal{O}(\log(w)r)$.

The idea of Alg. 32 is to perform only one conversion from 1 bit Boolean sharing to additive sharing. Then, it computes the UPC following the same idea as in the reference implementation [ABB+22]:

$$|s \wedge X^i h_0|_1 = \sum_{j \mid h_0^{(j)}=1} s^{(i+j \bmod r)}$$

Where s is arithmetically mask modulo $2^{\log(w)}$ and h_0 is masked using sparse representation. Using this representation of h_0, the equation becomes:

$$|s \wedge X^i h_0|_1 = \sum_{j=1}^{w/2} s^{(i+\sum_{k=0}^{n} h_{s,0,k}^{(j)}) \bmod r}$$

Therefore, we deduce the following algorithm to perform the precomputation of all UPCs:

Algorithm 32 Masked computation of UPCs SecUPCalter

Input: A Boolean sharing $(s_1, \ldots, s_n)$ of a syndrome s, a masked sparse representation $(h_{s,1}, \ldots, h_{s,n})$ of the polynomial h

Output: An array of arithmetic sharing modulo 2^α of the values $(|s \wedge X^i h|_1)_{1 \leq i \leq n}$

1: **for** $i = 1$ to r **do**
2: $\quad t_1^{(i)}, \ldots, t_n^{(i)} \leftarrow \mathsf{1bitB2A}_{2^\alpha}()s_1^{(i)}, \ldots, s_n^{(i)})$
3: $\quad upc_1^{(i)}, \ldots, upc_n^{(i)} \leftarrow (0, \ldots, 0)$
4: **end for**
5: **for** $j = 1$ to $w/2$ **do**
6: $\quad$ **for** $i = 1$ to r **do**
7: $\quad\quad u_1^{(i)}, \ldots, u_n^{(i)} \leftarrow t_1^{(i)}, \ldots, t_n^{(i)}$
8: $\quad$ **end for**
9: $\quad$ **for** $k = 1$ to n **do**
10: $\quad\quad$ **for** $i = 1$ to r **do**
11: $\quad\quad\quad u_1^{(i)}, \ldots, u_n^{(i)} \leftarrow u_1^{(i+h_{s,k}^{(j)})}, \ldots, u_n^{(i+h_{s,k}^{(j)})}$
12: $\quad\quad$ **end for**
13: $\quad\quad$ **for** $i = 1$ to r **do**
14: $\quad\quad\quad u_1^{(i)}, \ldots, u_n^{(i)} \leftarrow \mathsf{LinearRefresh}(u_1^{(i)}, \ldots, u_n^{(i)})$
15: $\quad\quad$ **end for**
16: $\quad$ **end for**
17: $\quad$ **for** $i = 1$ to r **do**
18: $\quad\quad$ **for** $k = 1$ to n **do**
19: $\quad\quad\quad upc_k^{(i)} \leftarrow upc_k^{(i)} + u_k^{(i)}$
20: $\quad\quad$ **end for**
21: $\quad$ **end for**
22: **end for**
23: **return** $upc_1, \ldots, upc_n$