Masked Vector Sampling for HQC

Maxime Spyropoulos[1,2], David Vigilant[2], Fabrice Perion[2], Renaud Pacalet[1], and Laurent Sauvage[1]

[1] Télécom Paris, Palaiseau, France
{maxime.spyropoulos, renaud.pacalet, laurent.sauvage}@telecom-paris.fr
[2] Thales DIS, Meudon, France
{david.vigilant, fabrice.perion}@thalesgroup.com

Abstract. Anticipating the advent of large quantum computers, NIST started a worldwide competition in 2016 aiming to define the next cryptographic standards. HQC is one of these post-quantum schemes still in contention, with four others already in the process of being standardized. In 2022, Guo *et al.* introduced a timing attack that exploited an inconsistency in HQC rejection sampling function to recover its secret key in 866,000 calls to an oracle. The authors of HQC updated its specification by applying an algorithm to sample vectors in constant time. A masked implementation of this function was then proposed for BIKE but it is not directly applicable to HQC. In this paper we propose a masked specification-compliant version of HQC vector sampling function which relies, to our knowledge, on the first masked implementation of the Barrett reduction.

Keywords: HQC · Masking · Side-channel attack · Post-quantum cryptography

1 Introduction

Post-Quantum Cryptography (PQC) came of interest when NIST launched its competition, anticipating that the current public-key cryptosystems might be broken by quantum computers by the 2030 horizon. In 2023, the competition has reached an important milestone, with the publication of draft standards [9] of 4 selected algorithms, out of 69 initially submitted. An extra 4th round is taking place and it should select at most 2 additional winners for standardization among the remaining 3 algorithms [23]. NIST also posted a call for additional digital signature proposals [23], aiming to standardize extra short signature schemes that would ideally not rely on the problems of structured lattices.

In the meantime regarding the quantum threat, progresses are still ongoing on both technology and algorithms. As an instance, Oded Regev published in 2023 an algorithm able to factor an n-bits integer using $O(n^{3/2})$ quantum gates instead of Shor's original $O(n^2)$ [24]. The same year, IBM presented its 1000+ qbits quantum computer and aims to reach 4000+ qbits by the year 2025 [27].

HQC [21] is one of the schemes participating in the 4th round of the competition. It's a code-based Key-Encapsulation Mechanism (KEM), used to share

securely a symmetric key between 2 parties. Its security relies on the difficulty of decoding a random linear code and is proved to be Indistinguishable under Adaptive Chosen Cipher Attack (IND-CCA2). The quasi-cyclic structure of HQC allows it to benefit from reduced public key and ciphertext sizes. Its fast key generation and decapsulation make it "one of the two best candidates of the 4th round" [1].

The capacity to develop side-channel resistant and efficient implementations of the algorithm is an explicit and non-negligible criterion for selection [2]. Side-Channel Attacks (SCA) have been first introduced by Kocher in 1996 [19]. They take advantage of a physical leakage (power consumption, execution time, ...) emanating from a device running an implementation of a cryptographic algorithm. These leaks are intrinsic to the realisation of the cryptographic algorithm and depend directly on an operation carried out during the computation. If the leak is found to be correlated to the manipulation of parts of a secret, an attacker could exploit it to recover the secret part by part thus breaking the security of the algorithm without needing to break the underlying mathematical hard problem. This security evaluation of the scheme in the "real world" setup also raises questions about performance. Indeed, making the algorithm resistant to SCA may involve adding countermeasures that hinder its performance, thus changing its competitiveness. HQC has already been thoroughly vetted against SCA, either against Timing Attack [22, 28, 17] or Template Attack [15, 16, 3].

In [17], the execution time of the rejection sampling function of HQC is exploited to recover its secret key. The authors proposed to patch this vulnerability with a countermeasure designed by Nicolas Sendrier to sample vectors in constant time [26], a solution that was applied by the HQC team in their following specification of the scheme. Moreover, in early 2024 Demange and Rossi provided the first high-order masked implementation of a code-based algorithm [12]: a fully masked implementation of BIKE in which a masked version of Sendrier's countermeasure is presented. Although the new masked vector sampling function is well suited for BIKE, it is not directly applicable to HQC as it does not respect its specification.

Contributions. This paper presents a masked specification-compliant vector generation function for HQC that relies on the first masked Barrett reduction to our knowledge. We also give a theoretical security proof backed by a practical evaluation of the implementation of our solution on a STM32 board. The practical evaluation part of this study also raises a warning about the secure implementation proposed by Demange and Rossi, showing that compilation optimizations should be handled carefully.

Organization. The remaining of this paper is organized as follows. Section 2 gives a preliminary description of HQC, masking and what motivated our approach. In Section 3, we introduce our proposition to adapt BIKE secure implementation to HQC, by proposing a masked version of the vector sampling

suited to HQC, along with its security proof. Section 4 summarizes the experimental results we obtained when evaluating practically our implementation on an STM32 device. Finally, Section 5 concludes the paper.

2 Background

2.1 HQC

HQC is a post-quantum scheme based on the theory of quasi-cyclic codes, where the secret key is generated independently from the code so the security reduction is independent from the decoding algorithm used for decryption. Our work refers to the specification of February 2024.

The authors use the Hofheinz-Hövelmanns-Kiltz transformation (a variant of the Fujisaki-Okamoto transformation that can handle decryption failures) to turn the IND-CPA secure PKE into an IND-CCA2 KEM.

HQC.PKE The Public Key Encryption (PKE) version of HQC consists of three algorithms. k represents the length of the shared key (128, 256 or 512 bits), it is a parameter set at initialization. Algorithm 1 generates the public key pk and the secret key sk given a seed $\mathcal{R}$. The weights ω are parameters that depend on the value of k. Algorithm 2 encrypts a message $\mathbf{m}$ using pk and a seed θ. The term $\mathbf{mG}$ corresponds to the encoding of the message $\mathbf{m}$ by the concatenated code $\mathcal{C}$ using its generator matrix $\mathbf{G}$. Finally, Algorithm 3 details how to decrypt the ciphertext $\mathbf{c}$ knowing sk. The secret key consists of $\mathbf{x}, \mathbf{y}$ and σ although only $\mathbf{y}$ is needed for decryption.

Algorithm 1 HQC.KeyGen

Input: $\mathcal{R}$
Output: sk, pk
$\mathbf{h} = \text{Sample}(\mathcal{R})$
$\mathbf{x} = \text{Sample}(\mathcal{R}, \omega)$
$\mathbf{y} = \text{Sample}(\mathcal{R}, \omega)$
$\sigma = \text{rand}(\mathbb{F}_2^k)$
$sk = (\mathbf{x}, \mathbf{y}, \sigma)$
$pk = (\mathbf{h}, \mathbf{s} = \mathbf{x} + \mathbf{h} \cdot \mathbf{y})$

Algorithm 2 HQC.Encrypt

Input: $pk, \mathbf{m}, \theta$
Output: $\mathbf{c} = (\mathbf{u}, \mathbf{v})$
$\mathbf{r}_1 = \text{Sample}(\theta, \omega_r)$
$\mathbf{r}_2 = \text{Sample}(\theta, \omega_r)$
$\mathbf{e} = \text{Sample}(\theta, \omega_e)$
$\mathbf{u} = \mathbf{r}_1 + \mathbf{h} \cdot \mathbf{r}_2$
$\mathbf{v} = \mathbf{mG} + \mathbf{s} \cdot \mathbf{r}_2 + \mathbf{e}$

Algorithm 3 HQC.Decrypt

Input: $sk, \mathbf{c}$
Output: $\mathbf{m}$
$\mathbf{m} = \mathcal{C}.\text{Decode}(\mathbf{v} - \mathbf{u} \cdot \mathbf{y})$

HQC.KEM The KEM version of HQC relies on 3 hash functions $(\mathcal{G}, \mathcal{H}, \mathcal{K})$ and consists of 2 algorithms. The sender executes Algorithm 4 to generate a random symmetric key K and exchange it in the form of a ciphertext. The receiver applies Algorithm 5 to retrieve the symmetric key.

Algorithm 4 HQC.Encaps

Input: pk
Output: $K, \mathbf{c}, salt$
$\mathbf{m} = \text{rand}(\mathbb{F}_2^k)$
$salt = \text{rand}(\mathbb{F}_2^{128})$
$\theta = \mathcal{G}(\mathbf{m} \| pk \| salt)$
$\mathbf{c} = \text{HQC.Encrypt}(pk, \mathbf{m}, \theta)$
$K = \mathcal{K}(\mathbf{m}, \mathbf{c})$

Algorithm 5 HQC.Decaps

Input: $sk, \mathbf{c}, salt$
Output: K
$\mathbf{m}' = \text{HQC.Decrypt}(sk, \mathbf{c})$
$\theta' = \mathcal{G}(\mathbf{m}' \| pk \| salt)$
$\mathbf{c}' = \text{HQC.Encrypt}(pk, \mathbf{m}', \theta')$
if $\mathbf{m}' = \perp \vee \mathbf{c} \neq \mathbf{c}'$ **then**
    $K = \mathcal{K}(\sigma, \mathbf{c})$
else
    $K = \mathcal{K}(\mathbf{m}, \mathbf{c})$
end if

2.2 Guo *et al.* Timing Attack

HQC June 2021 specification suffered from a timing flaw in its encryption function. Indeed, HQC encryption relies on the sampling of 3 random vectors with fixed weights. For the sampling of each vector, a call was made to the function `seedexpander` in order to generate bytes of randomness that would determine the support of the vector. In most cases there was no collision between the indexes drawn and the algorithm called `seedexpander` only 3 times, once for each vector. However, if at least one collision occurred, a second call to `seedexpander` was made.

This difference in the number of calls is noticeable in timing and the authors used this distinguisher to perform a Timing Attack and recover the secret key in $\approx \mathbf{8.7} \times \mathbf{10^5}$ decapsulations [17].

2.3 Sendrier's countermeasure

In 2021, Nicolas Sendrier proposed an algorithm to sample vectors in constant time in order to patch vulnerabilities in both BIKE's and HQC rejection sampling functions. The idea is to always ask for the same amount of pseudo randomness. In the event of a collision between two indices, the index that was drawn is replaced by the number of the current index of the loop iteration (see Algorithm 6). This is made possible by the introduction of a small bias that has no significant impact on the distribution [26].

In Algorithm 6, `compare` is a constant-time function that returns $0\text{xF}\cdots\text{F}$ if the two input values are equal and $0\text{x}0\cdots0$ otherwise. Following recommendations of Guo *et al.*, the HQC team explicitly added this countermeasure in their 2023 update of the scheme.

In 2024, Demange and Rossi proposed a fully masked and provably secure implementation of BIKE [12] featuring a masked version of Sendrier's algorithm.

Algorithm 6 Sendrier's constant time vector sampling

Input: $s, seed, n$
Output: $support[0], \ldots, support[s-1]$

1: $prng \leftarrow \text{prng\_init}(seed)$
2: **for** $i = 0$ **to** $s - 1$ **do**
3: $\quad support[i] \leftarrow i + \text{rand}(prng, n - i)$ ▷ $support[i] \in [i, n[$
4: **end for**
5: **for** $i = s - 1$ **downto** 0 **do**
6: $\quad found \leftarrow 0$ ▷ collision flag
7: $\quad$ **for** $j = i + 1$ **to** $s - 1$ **do**
8: $\quad\quad found \leftarrow found \vee \text{compare}(support[i], support[j])$
9: $\quad$ **end for**
10: $\quad support[i] \leftarrow (i \wedge found) \oplus (support[i] \wedge \neg found)$
11: **end for**
12: **return** $support[0], \ldots, support[s-1]$

2.4 Masking the vector sampling

Masking The principle of masking is to add random values to any sensitive variable in the algorithm. A common masking scheme is the *Boolean masking* [8, 14] where any sensitive variable x is split into $d + 1$ boolean shares such that:

$$x = x_0 \oplus \ldots \oplus x_d$$

d is called the masking order. During the computation, the shares are processed in such a way that any combination of d intermediate variables is independent of any sensitive variable, ensuring that the leakage observed by an attacker with access to up to d intermediate variables is independent of any secret. The algorithm is said d-probing secure.

Security proof To prove the security of an algorithm, the easiest way is to prove the security property of individual functions that make it up, referred to as "gadgets". To achieve overall security one can follow a property outlined in [6, Proposition 4]:

"An algorithm P is t-NI provided all its gadgets are t-NI, and all masked variables are used at most once as argument of a gadget call other than `refresh`*".*
"t-NI" is a concept also developed in [6]:

Definition 1. *A gadget is t-non-interfering (t-NI) iff any set of at most t observations can be perfectly simulated from at most t shares of each input.*

Remark 1. t-NI implies t-probing secure.

Remark 2. Any linear operation in $\mathbb{F}_2$ (the binary finite field) is a t-NI gadget, as long as it is applied share-wise.

Remark 3. Masked values manipulated by a linear gadget don't have to be refreshed afterwards.

The "`refresh`" function was introduced in [10] to increase the overall randomness of an algorithm by re-randomizing the values of the shares encoding a secret variable. Informally, Proposition 4 states that a masked value has to be refreshed after a non-linear manipulation before being used in another gadget.

Masked implementation We give a description of Demange *et al.* version of Sendrier's algorithm, referred to as "`SecFisherYates`" in Algorithm 7. The gadgets are detailed in the appendix of [12] . The authors have proved the t-NI resistance of their gadgets, and they provided a C implementation on GitHub [20]. In the following, $[\![\mathbf{x}]\!]$ denotes the array containing the boolean shares of the masked value $\mathbf{x}$.
All binary secure operations for which it makes sense come in two flavours: $\texttt{sec}_{op}([\![A]\!], [\![B]\!])$ (both operands masked), $\texttt{sec}_{op}([\![A]\!], B)$ (left operand masked, right operand unmasked).

Algorithm 7 Demange *et al.* masked SecFisherYates

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

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

3 Proposed countermeasure

BIKE and HQC differ on the way they draw a random number in an interval $\mathbb{Z}_n$. Given a p bits random number a, BIKE multiplies it by n then shifts the result p bits to the right (Algorithm 8, Line 4), whereas HQC returns the remainder of a modulo n (Algorithm 9, Line 3). This difference means that the `SecFisherYates` algorithm cannot be directly transposed to an HQC implementation, as it would not follow its specification. We have to design a side-channel resistant function that computes the remainder of a boolean masked value a modulo a public value n. One particular constraint is that we don't want to use the division or modulo instructions to fulfil this task. Firstly, because on some architectures they

have a variable execution time that depends on the numerator. In our case, a variation in execution time may translate to a leak because the numerator is a secret variable in the vector sampling. Secondly because they are not directly applicable to Boolean masked data. We have chosen to solve this problem by designing a masked Barrett reduction. Non-masked Barrett reduction was actually implemented in September 2023 in the PQClean version of HQC before being added to the latest HQC specification in February 2024.

Algorithm 8 BIKE's vector sampling

Input: $seed, len, wt$
Output: $wlist$, a list of wt distinct elements of $\{0, \ldots, len-1\}$.
1: $wlist \leftarrow ()$ ▷ empty list
2: $s_0, \ldots, s_{wt-1} \leftarrow$ `SHAKE256-Stream`$(seed, 32 \cdot wt)$
▷ parse as a sequence of wt non negative 32-bits integers
3: **for** $i = wt - 1$ **downto** 0 **do**
4: $pos \leftarrow i + \lfloor (wt - i) s_i / 2^{32} \rfloor$
5: $wlist \leftarrow wlist, (pos \in wlist)\ ?\ i : pos$
6: **end for**
7: **return** wlist

Algorithm 9 HQC vector sampling

Input: $n, w, seed$
Output: w distinct elements of $\{0, \ldots, n\}$
1: $prng \leftarrow$ prng_init$(seed)$
2: **for** $i = w - 1$ **downto** 0 **do**
3: $l \leftarrow i + (\text{rand}(prng) \bmod (n - i))$
4: $pos[i] \leftarrow (l \in \{pos[j], i < j < t)\ ?\ i : l$
5: **end for**
6: **return** $pos[0], \ldots, pos[w-1]$

3.1 Barrett reduction

The Barrett reduction [5] is a constant-time procedure to efficiently compute the remainder of an integer division. To compute the remainder of x mod n one can use:

$$r = x - \lfloor x/n \rfloor \times n$$

In order not to use the division operation, Barrett's idea was to approximate the quotient by finding an integer m such that:

$$\frac{m}{2^p} \approx \frac{1}{n}$$

We usually take: $m = \lfloor \frac{2^p}{n} \rfloor$.

Remark 4. In our case, $p = 32$, because the function rand of HQC (Algorithm 9, Line 3) outputs pseudo-random 32-bits unsigned integers.

Since the quotient $\frac{m}{2^p}$ is only guaranteed to be less or equal to $\frac{1}{n}$, a final subtraction is sometimes required. The main advantage of this technique is that it relies on a variable m that can be pre-computed, and efficient operations (dividing by 2^p comes down to a shift p bits to the right, which is virtually free).

Algorithm 10 Barrett reduction

Input: a, n, p and m s.t. $m = \lfloor \frac{2^p}{n} \rfloor$
Output: $r = a \bmod n$
1: $q \leftarrow (a \times m) \gg p$
2: $r \leftarrow a - q \times n$
3: **if** $r \geq n$ **then**
4: $\quad r \leftarrow r - n$
5: **end if**
6: **return** r

3.2 Masked implementation

In this section we describe the algorithm that we designed to achieve the masked implementation of the Barrett reduction. Thanks to the gadgets introduced in [12] we have an implementation with no mask conversion (which are known to be demanding in terms of computation time), that works with any degree of masking. It is constant-time by design and provably secure.

New gadgets We developed a new gadget, $\texttt{sec\_}([\![a]\!], [\![b]\!])$, that computes the subtraction $[\![a]\!] - [\![b]\!]$ of 2 masked values, and a partly masked variant of the secure equality: $\texttt{sec}_=([\![a]\!], b)$, which computes the equality of a masked value $[\![a]\!]$ and a public value b (presented in Annex)

Algorithm To keep the constant-time property, we are forced to always perform the conditional subtraction at the end. We start by subtracting n from the computed value "a_qn" $(= a - q \times n)$, and store the result in A (Line 8). There are 2 possibilities: either A is negative and a_qn is the correct result or A is positive and the correct result. To evaluate the sign of A we shift it by 31 bits to the right and store the result in z; if A is negative then z is equal to 1, else A is positive and z is equal to 0.

Theorem 1. *The masked Barrett reduction Algorithm 11 is t-NI secure*

Algorithm 11 Masked Barrett reduction

Input: $[\![a]\!], m, n$
Output: $[\![r]\!] = [\![a]\!] \bmod n$

1: $[\![q]\!] \leftarrow \texttt{sec}_{\times}([\![a]\!], m)$
2: $[\![q]\!] \leftarrow [\![q]\!] \gg 32$ ▷ Shift on each share
3: $[\![qn]\!] \leftarrow \texttt{sec}_{\times}([\![q]\!], n)$
4: $[\![a]\!] \leftarrow \texttt{refresh}([\![a]\!])$
5: $[\![a\_qn]\!] \leftarrow \texttt{sec}_{-}([\![a]\!], [\![qn]\!])$
6: $[\![a\_qn]\!] \leftarrow [\![a\_qn]\!] \& (2^{32} - 1)$ ▷ $(a - q \times n) \bmod 2^{32}$
7: $minus\_n \leftarrow 2^{32} - n$
8: $[\![A]\!] \leftarrow \texttt{sec}_{+}([\![a\_qn]\!], minus\_n)$ ▷ conditional final subtraction
9: $[\![z]\!] \leftarrow [\![A]\!] \gg 31$
10: $[\![A]\!] \leftarrow [\![A]\!] \& (2^{32} - 1)$
11: $[\![t]\!] \leftarrow \texttt{sec}_{=}([\![z]\!], 1)$
12: $[\![a\_qn]\!] \leftarrow \texttt{refresh}([\![a\_qn]\!])$
13: $[\![r]\!] \leftarrow \texttt{sec}_{if}([\![a\_qn]\!], [\![A]\!], [\![t]\!])$ ▷ $[\![t]\!]$? $[\![a\_qn]\!]$: $[\![A]\!]$
14: **return** $[\![r]\!]$

Proof. m and n are public values, we do not need to mask them. Since it operates on each share individually, the shift operation (Lines 2, 9) is t-NI. The same reasoning applies to the AND operation (Lines 6, 10). Algorithm 11 uses t-NI gadgets, and the only masked variables that are used twice without modification are a (Lines 1, 5) and a_qn (Lines 8, 13). They are both refreshed (Lines 4, 12) before their re-use. We don't have to refresh A before using it on Line 13 because it is only manipulated by linear operations (Lines 9, 10).

4 Experimental setup and results

To conduct our experiments we used the STM32 Nucleo-F439ZI development board which is equipped with an ARM Cortex-M4 core running at 180 MHz. We selected the *pqm4* [18] implementation of HQC. We retrieved the code of the gadgets from Github [20] and implemented our solution. After setting the masking order to one, we ran 10,000 executions of our masked Barrett reduction, once with fixed inputs and once with random ones; and recorded the resulting electro-magnetic traces. For comparison purposes, we ran this first experiment while fixing the masks to zero (virtually unmasking the secret value). Using a Test Vector Leakage Assessment (TVLA) [7] we confront these two sets of traces and obtain Figure 1. The numerous peaks well above and below the ± 4.5 threshold [25] (dotted red lines) inform us of the presence of leaks. We then ran a second experiment, this time relying on the integrated True Random Number Generator (TRNG) of the board to produce the random masks required for the computation. This time, there are no visible peaks (Figure 2) which is expected from a first order leakage analysis of a first order masking. Hence, the absence of leaks in the second experiment gives us better confidence concerning the soundness of our solution.

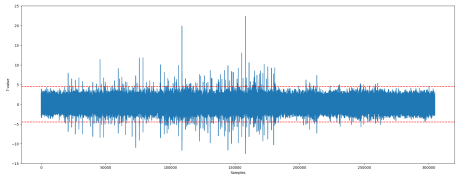

Fig. 1. TVLA of the masked Barrett reduction with masks set to zero.

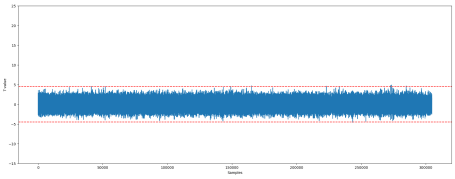

Fig. 2. TVLA of the masked Barrett reduction.

Remark 5. A formal security proof on the pseudo or source code is not a complete guarantee. Due to all optimizations performed by the compilation tool chain and even by the hardware, executing the corresponding compiled software on a real hardware CPU can leak secret data, even when the abstract algorithm was proved t-NI secure. Further analyses on the assembly code, on the linked and loaded binary or even on the actual execution by the processor are needed before one can conclude that the masking provides the expected security level, as demonstrated for instance in [11].

Our results were obtained with the compilation flag `-Og` and all caches ON. While testing another configuration, we found out that compiling with the `-O3` flag introduced leaks (see Figure 3 in Annex). As we explain in Annex these leaks are due to a different mapping of the various shares on the physical CPU registers.

4.1 Performances

In this section we discuss the performance of our solution. We implemented some 32-bit variants of the gadgets to best fit our STM32 Nucleo-144 target board before running a benchmark. The following results were all obtained using the compilation flag `-Og`

Table 1. Comparison of the number of cycles needed to sample a random vector of weight $w = 75$ (ARM Cortex-M4 @180 MHz, caches ON).

Algorithm	Execution time (cycles)	Overhead
HQC	747,500	–
Our solution (order 1)	17,650,000	2361%
Our solution (order 2)	36,625,000	4899%
BIKE	746,500	–
Masked BIKE (order 1)	11,606,000	1554%

Our solution is 52% more computationally intensive compared to the one proposed for BIKE. This is due to the fact that BIKE specification only requires a masked multiplication and a shift. In comparison, following HQC specification

requires to execute the entire masked Barrett reduction (Algorithm 11). Since, in an unmasked implementation, the vector sampling phase represents only 0.14% of the total cycles required for a decapsulation, we think that the impact of our solution on a masked version should be limited while providing the key advantage of preserving the HQC specification.

5 Conclusion

This paper proposes a first masked version of HQC vector sampling, which has never been proposed to our knowledge. To bridge this gap, we based our work on Demange *et al.* SecFisherYates, and we have introduced some new gadgets, mainly the masked Barrett reduction, which was not publicly available so far. It is now trivial to adapt `SecFisherYates` to HQC, one has to replace Line 2 in Algorithm 7 with our masked Barrett reduction to get a masked vector sampling function for HQC.

Moreover, one main advantage is that it does not require any mask conversion and works for any order of masking. The security assessment of our modular Barrett reduction is two-fold: we provide a security proof for our new gadgets, and we have performed a practical side-channel evaluation of its implementation on an STM32 device, that shows the absence of remaining leakages in this specific setting. This practical evaluation has raised some warnings about compiler options when using our and Demange *et al.* gadgets, since some leakage appears with aggressive compiler optimization settings. It reminds the importance of validating practical implementations against attacks.

This work covers the vector sampling, by providing a publicly available C implementation. Vector sampling is only a part of HQC algorithm. Even if side-channel resistant implementations of some other parts of HQC have been studied in [15], building a full HQC masked implementation is still let as future work.

Acknowledgements This work was realized thanks to the grant 2022154 from the Appel à projets 2022 thèses AID Cifre-Défense by the Agence de l'Innovation de Défense (AID), Ministère des Armées (French Ministry of Defense).
We wish to thank Benoît Cogliati for helpful discussions.

References

[1] Gorjan Alagic et al. "Status report on the third round of the NIST post-quantum cryptography standardization process". In: *US Department of Commerce, NIST* (2022).

[2] *Announcing Request for Nominations for Public-Key Post-Quantum Cryptographic Algorithms.* `https://csrc.nist.gov/News/2016/Public-Key-Post-Quantum-Cryptographic-Algorithms`. NIST, 2016.

[3] Chloé Baïsse et al. *Secret and Shared Keys Recovery on Hamming Quasi-Cyclic with SASCA*. Cryptology ePrint Archive, Paper 2024/440. https://eprint.iacr.org/2024/440. 2024. URL: https://eprint.iacr.org/2024/440.

[4] Josep Balasch et al. "On the Cost of Lazy Engineering for Masked Software Implementations". In: *Smart Card Research and Advanced Applications*. Ed. by Marc Joye and Amir Moradi. Cham: Springer International Publishing, 2015, pp. 64–81. ISBN: 978-3-319-16763-3.

[5] Paul Barrett. "Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor". In: *Advances in Cryptology — CRYPTO' 86*. Ed. by Andrew M. Odlyzko. Berlin, Heidelberg: Springer Berlin Heidelberg, 1987, pp. 311–323. ISBN: 978-3-540-47721-1.

[6] Gilles Barthe et al. "Strong Non-Interference and Type-Directed Higher-Order Masking". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS '16. Vienna, Austria: Association for Computing Machinery, 2016, pp. 116–129. ISBN: 9781450341394. DOI: 10.1145/2976749.2978427. URL: https://doi.org/10.1145/2976749.2978427.

[7] Georg T. Becker et al. "Test Vector Leakage Assessment (TVLA) methodology in practice". In: 2013. URL: https://api.semanticscholar.org/CorpusID:28168779.

[8] Suresh Chari et al. "Towards sound approaches to counteract power-analysis attacks". In: *Advances in Cryptology—CRYPTO'99: 19th Annual International Cryptology Conference Santa Barbara, California, USA, August 15–19, 1999 Proceedings 19*. Springer. 1999, pp. 398–412.

[9] *Comments Requested on Three Draft FIPS for Post-Quantum Cryptography*. https://csrc.nist.gov/News/2023/three-draft-fips-for-post-quantum-cryptography. NIST, 2023.

[10] Jean-Sebastien Coron. *Higher Order Masking of Look-up Tables*. Cryptology ePrint Archive, Paper 2013/700. https://eprint.iacr.org/2013/700. 2013. URL: https://eprint.iacr.org/2013/700.

[11] Jean-Sébastien Coron et al. "Conversion of Security Proofs from One Leakage Model to Another: A New Issue". In: *Constructive Side-Channel Analysis and Secure Design*. Ed. by Werner Schindler and Sorin A. Huss. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 69–81. ISBN: 978-3-642-29912-4.

[12] Loic Demange and Mélissa Rossi. "A provably masked implementation of BIKE Key Encapsulation Mechanism". In: *Cryptology ePrint Archive* (2024).

[13] Barbara Gigerl et al. "Coco: Co-Design and Co-Verification of Masked Software Implementations on CPUs". In: *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 1469–1468. ISBN: 978-1-939133-24-3. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/gigerl.

[14] Louis Goubin and Jacques Patarin. "DES and differential power analysis the "Duplication" method". In: *Cryptographic Hardware and Embedded Systems: First InternationalWorkshop, CHES'99 Worcester, MA, USA, August 12–13, 1999 Proceedings 1*. Springer. 1999, pp. 158–172.

[15] Guillaume Goy, Antoine Loiseau, and Philippe Gaborit. "A new key recovery side-channel attack on HQC with chosen ciphertext". In: *International Conference on Post-Quantum Cryptography*. Springer. 2022, pp. 353–371.

[16] Guillaume Goy et al. "Single trace HQC shared key recovery with SASCA". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2024.2 (Mar. 2024), pp. 64–87. DOI: 10.46586/tches.v2024.i2.64-87. URL: https://tches.iacr.org/index.php/TCHES/article/view/11421.

[17] Qian Guo et al. "Don't reject this: Key-recovery timing attacks due to rejection-sampling in HQC and BIKE". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2022), pp. 223–263.

[18] Matthias J. Kannwischer et al. *PQM4: Post-quantum crypto library for the ARM Cortex-M4*. https://github.com/mupq/pqm4.

[19] Paul C Kocher. "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems". In: *Annual International Cryptology Conference*. Springer. 1996, pp. 104–113.

[20] *masked BIKE code*. https://github.com/loicdemange/masked_BIKE_code. GitHub, 2024.

[21] Carlos Aguilar Melchor et al. "Hamming quasi-cyclic (HQC)". In: *NIST PQC Round* 2.4 (2018), p. 13.

[22] Thales Bandiera Paiva and Routo Terada. "A timing attack on the HQC encryption scheme". In: *Selected Areas in Cryptography–SAC 2019: 26th International Conference, Waterloo, ON, Canada, August 12–16, 2019, Revised Selected Papers 26*. Springer. 2020, pp. 551–573.

[23] *PQC Standardization Process: Announcing Four Candidates to be Standardized, Plus Fourth Round Candidates*. https://csrc.nist.gov/News/2022/pqc-candidates-to-be-standardized-and-round-4. NIST, 2022.

[24] Oded Regev. "An Efficient Quantum Factoring Algorithm". In: (2023). DOI: 10.48550/ARXIV.2308.06572. URL: https://arxiv.org/abs/2308.06572.

[25] Tobias Schneider and Amir Moradi. "Leakage assessment methodology: Extended version". In: *Journal of Cryptographic Engineering* 6 (2016), pp. 85–99.

[26] Nicolas Sendrier. "Secure sampling of constant-weight words–application to bike". In: *Cryptology ePrint Archive* (2021).

[27] *The IBM Quantum Development Roadmap*. https://www.ibm.com/quantum/roadmap. IBM, 2023.

[28] Guillaume Wafo-Tapa et al. "A practicable timing attack against HQC and its countermeasure". In: *Cryptology ePrint Archive* (2019).

Annex

Algorithm 12 $\texttt{sec}_-([\![a]\!], [\![b]\!])$ (secure minus)

Input: $[\![a]\!], [\![b]\!]$
Output: $[\![r]\!] = [\![a]\!] - [\![b]\!]$
1: $[\![nb]\!] \leftarrow [\![b]\!]$
2: $[\![nb]\!]_0 \leftarrow \neg[\![nb]\!]_0$ ▷ NOT of the first share
3: $[\![mb]\!] \leftarrow \texttt{sec}_+([\![nb]\!], 1)$ ▷ $-b = (\neg b) + 1$
4: $[\![r]\!] \leftarrow \texttt{sec}_+([\![a]\!], [\![mb]\!])$
5: **return** $[\![r]\!]$

Theorem 2. *The masked subtraction Algorithm 12 is t-NI.*

Proof. The only gadget manipulating the data is $\texttt{sec}_+$, which is t-NI, and the negation only manipulates the first share.

Algorithm 13 $\texttt{sec}_=([\![x]\!], y)$ (masked vs. unmasked secure equality)

Input: $[\![x]\!], y$
Output: $[\![r]\!] = 0$ iff $[\![x]\!] = y, 1$ otherwise
1: $[\![r]\!] \leftarrow [\![x]\!] \oplus y$
2: **for** $i = \frac{n}{2}$ **to** 1 **step** $-\frac{i}{2}$ **do**
3: $[\![a]\!] \leftarrow$ left_half($[\![r]\!]$)
4: $[\![b]\!] \leftarrow$ right_half($[\![r]\!]$)
5: $[\![a]\!]_0 \leftarrow \neg[\![a]\!]_0$ ▷ NOT of the first share
6: $[\![b]\!]_0 \leftarrow \neg[\![b]\!]_0$
7: $[\![r]\!] \leftarrow \texttt{sec}_\&([\![a]\!], [\![b]\!])$
8: $[\![r]\!]_0 \leftarrow \neg[\![r]\!]_0$
9: **end for**
10: **return** $[\![r]\!]$

Theorem 3. *The partly masked equality Algorithm 13 is t-NI*

Proof. The only gadget manipulating the data is $\texttt{sec}_\&$ which is t-NI. Moreover, the negation (Lines 5, 6) only manipulates the first share, it is not able to leak anything (since the values are updated at each loop turn).

Example of leak introduced by the compiler optimizations When testing our solution with the more aggressive `-O3` compilation option we discovered that the compiler decided to reuse the same register for two shares as can be seen on Listing 1.1. Since we use a first order masking, the transition between the two states of the register induces a XOR between the two values, which unmasks the secret as illustrated by Figure 3.

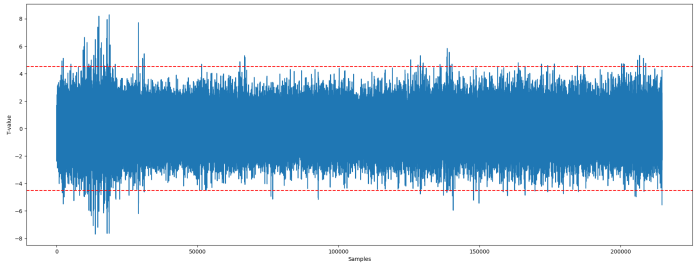

Fig. 3. TVLA of the masked Barrett reduction compiled with `-O3`

```
1 <boolean_sec_and>:
2   push   {r3, r4, r5, r6, r7, lr}
3   mov    r5, r1         ;store address of y in r5
4   ldrd   r1, r3, [r0]   ;r1 = x[0], r3 = x[1]
5   mov    r4, r2
6   ldr    r2, [r5, #0]   ;r2 = y[0]
7   ands   r1, r2         ;r1 = x[0] & y[0]
8   ldr    r2, [r5, #4]   ;r2 = y[1]
9   ...
```

Listing 1.1. Abstract of the assembly code of $\texttt{sec}_{\&}$ for `-O3`

On Line 6, the value of `y[0]` is loaded in `r2` then on Line 8 the value of `y[1]` is loaded in `r2`. Physically, for the register to switch from the value `y[0]` to `y[1]` there is an implicit XOR between the two values. As $\texttt{y} = \texttt{y}[0] \oplus \texttt{y}[1]$, for all 0-bits of `y` there is no transition (that is, no consumed energy) between the corresponding bits of `y[0]` and `y[1]`, while for all 1-bits of `y` there is a transition from 0 to 1 or from 1 to 0, that consumes energy.

This involuntarily exposes the secret `y` and advocates for further analyses on the assembly code, on the linked and loaded binary or even on the actual execution by the processor as proposed for instance by [13]. Increasing the masking order, as suggested by [4], is another option but it is costly.