A PRACTICABLE TIMING ATTACK AGAINST HQC AND ITS COUNTERMEASURE

Guillaume Wafo-Tapa

Worldline, ZI Rue de la pointe
59113 Seclin, France

Slim Bettaieb

Worldline, ZI Rue de la pointe
59113 Seclin, France

Loïc Bidoux

Worldline, ZI Rue de la pointe
59113 Seclin, France

Philippe Gaborit

University of Limoges, XLIM-DMI, 123, Av. Albert Thomas
87060 Limoges, France

Abstract. In this paper, we present a practicable chosen ciphertext timing attack retrieving the secret key of HQC. The attack exploits a correlation between the weight of the error to be decoded and the running time of the decoding algorithm of BCH codes. For the 128-bit security parameters of HQC, the attack runs in less than a minute on a desktop computer using 5441 decoding requests and has a success probability of approximately 93 percent. To prevent this attack, we propose a constant time algorithm for the decoding of BCH codes. Our implementation of the countermeasure achieves a constant time execution of the decoding process without a significant performance penalty.

1. Introduction

HQC [1, 3] is a code-based IND-CCA2-secure public key encryption scheme, whose security is based on the hardness of the quasi-cyclic syndrome decoding problem. It is one of the candidate algorithms that has advanced to the round 2 of the NIST post-quantum standardization project. In particular, HQC relies on tensor product codes (BCH codes tensored with repetition codes) in its decryption algorithm. BCH codes are algebraic codes introduced in two independent works by Bose, Chaudhuri [7] and Hocquenghem [11]. Algorithms to decode BCH codes use Galois field arithmetic operations and basically consists in three steps: syndromes computation; error-locator polynomial computation and roots computation.

So far, BCH codes have been used to mitigate the decryption failure in various public key encryption schemes based on hard problems of either coding theory [1,3] or lattices [15]. However, due to side channel timing leakage, a straightforward use of BCH codes would introduce a security weakness in the underlying cryptographic schemes when implemented in software. In fact, D'Anvers et al. [8] showed that the security of LAC, a lattice-based cryptosystem [15], could be significantly reduced if there is a side channel leakage during the error correction of BCH codes.

Key words and phrases. HQC, BCH decoding, Timing attack, Constant time implementation.

Furthermore, HQC shares the same framework as the RQC [2, 3] cryptosystem. It has been shown in [4] that this framework is vulnerable to a timing attack in the rank metric setting if the decoding of the underlying Gabidulin codes [9] is implemented in a non constant time fashion.

Achieving a constant time implementation of the decoding of BCH codes is challenging. In a recent work, Walters and Sinha Roy [16] proposed such a constant time BCH decoding implementation. However, the algorithms used for syndromes computation and roots computation are not the most efficient known in the litterature.

Contributions. In this paper, we present a practicable timing attack against HQC that completes under the minute. As countermeasure, we give two variants of a constant time algorithm for BCH codes.

Paper organisation. In section 2, we give some preliminaries on code-based cryptography, decoding BCH codes as well as the HQC cryptosystem. Next, in section 3, we present a correlation between the weight of the error to be decoded and the decoding time of BCH codes. This observation is the cornerstone of the timing attack detailed in section 4. In section 5, we introduce a constant time implementation that constitutes a countermeasure to this attack as well as some experimental results. Finally, we conclude this work in section 6.

2. Preliminaries

In this section, we give some preliminaries regarding the Hamming metric, error-correcting codes and the HQC cryptosystem.

2.1. Coding theory.

Let $\mathbb{F}_2$ be the binary finite field and $\mathbb{F}_2^n$ the vector space of dimension n over $\mathbb{F}_2$ for some positive integer n. Elements of $\mathbb{F}_2^n$ are considered as vectors or polynomials in $\mathbb{F}_2[X]/(X^n - 1)$.

Definition 2.1 (Support)**.** Let $\mathbf{x} \in \mathbb{F}_2^n$. The support of $\mathbf{x}$ is the set of indices $i \in [\![0, n-1]\!]$ such that $x_i = 1$.

Definition 2.2 (Hamming weight)**.** Let $\mathbf{x} \in \mathbb{F}_2^n$. The Hamming weight of $\mathbf{x}$, denoted by $w(\mathbf{x})$, is the cardinal of its support, i.e. the number of its non-zero coordinates.

Definition 2.3 (Hamming distance)**.** Let $\mathbf{x}, \mathbf{y} \in \mathbb{F}_2^n$. The Hamming distance from $\mathbf{x}$ to $\mathbf{y}$, denoted by $d(\mathbf{x}, \mathbf{y})$, is defined as $w(\mathbf{x} - \mathbf{y})$, i.e. the number of coordinates $\mathbf{x}$ and $\mathbf{y}$ differ on.

Definition 2.4 (Linear code)**.** A linear $[n, k]$-code $\mathcal{C}$ of length n and dimension k is a linear subspace of $\mathbb{F}_2^n$ of dimension k.

Definition 2.5 (Generator matrix)**.** A matrix $\mathbf{G} \in \mathbb{F}_2^{k \times n}$ is a generator matrix for the $[n, k]$-code $\mathcal{C}$ if $\mathcal{C} = \{\mathbf{mG} \mid \mathbf{m} \in \mathbb{F}_2^k\}$.

Definition 2.6 (Parity-check matrix)**.** A matrix $\mathbf{H} \in \mathbb{F}_2^{(n-k) \times n}$ is a parity-check matrix for the $[n, k]$-code $\mathcal{C}$ if $\mathcal{C} = \{\mathbf{x} \in \mathbb{F}_2^n \mid \mathbf{Hx}^\top = 0\}$.

Definition 2.7 (Correction capacity)**.** Let $\mathcal{C}$ be a linear $[n, k]$-code. The correction capacity of $\mathcal{C}$ is the largest $\delta \in \mathbb{N}$ such that for all $\mathbf{x} \in \mathbb{F}_2^n$, there is at most one $\mathbf{c} \in \mathcal{C}$ such that $d(\mathbf{x}, \mathbf{c}) \leq \delta$. The code $\mathcal{C}$ is called a $[n, k, \delta]$-code.

Definition 2.8 (Cyclic code [14])**.** A code $\mathcal{C}$ is said to be cyclic if every cyclic shift of a codeword in $\mathcal{C}$ is also a codeword. That is, $(c_0, c_1, \ldots, c_{n-1}) \in \mathcal{C}$ implies $(c_{n-1}, c_0, \ldots, c_{n-2}) \in \mathcal{C}$.

Theorem 2.9 (Generator polynomial [14])**.** *Let $\mathcal{C}$ be a cyclic code over $\mathbb{F}_2$. There exists a unique polynomial $g(x)$ in $\mathcal{C}$ of minimal positive degree. Moreover, a polynomial $c(x)$ is a codeword of $\mathcal{C}$ if and only if $g(x)$ divides $c(x)$. The polynomial $g(x)$ is called the generator polynomial of the cyclic code $\mathcal{C}$.*

HQC uses a tensor product code obtained as the combination of a BCH code with a repetition code.

Definition 2.10 (Tensor product code [1])**.** Let $\mathcal{C}_1$ (resp. $\mathcal{C}_2$) be a $[n_1, k_1]$ (resp. $[n_2, k_2]$) linear code over $\mathbb{F}_2$. The tensor product code of $\mathcal{C}_1$ and $\mathcal{C}_2$ denoted $\mathcal{C}_1 \otimes \mathcal{C}_2$ is defined as the set of all $n_2 \times n_1$ matrices whose rows are codewords of $\mathcal{C}_1$ and whose columns are codewords of $\mathcal{C}_2$. More formally, if $\mathcal{C}_1$ (resp. $\mathcal{C}_2$) is generated by $\mathbf{G}_1$ (resp. $\mathbf{G}_2$), then

$$\mathcal{C}_1 \otimes \mathcal{C}_2 = \left\{ \mathbf{G}_2^\top \mathbf{X} \mathbf{G}_1 \;\middle|\; \mathbf{X} \in \mathbb{F}_2^{k_2 \times k_1} \right\}$$

Theorem 2.11 (BCH code [14])**.** *For any positive integers $m \geq 3$ and $t < 2^{m-1}$, there exists a binary cyclic BCH $[n, k, \delta]$-code with the following properties: $n = 2^m - 1$; $n - k \leq mt$; $\delta \geq t$. Let α be a primitive element in $\mathbb{F}_{2^m}$, and let $\phi_i(x)$ be the minimal polynomial of α^i for $1 \leq i \leq 2\delta$. The generator polynomial $g(x)$ of the BCH $[n, k, \delta]$-code is the least common multiple of $\phi_1(x), \phi_2(x), \ldots, \phi_{2\delta}(x)$, that is,*

$$g(x) = LCM\{\phi_1(x), \phi_2(x), \ldots, \phi_{2\delta}(x)\}.$$

BCH codes encoding. Given the generator polynomial $g(x)$ and a message $u(x) = u_0 + u_1 x + \ldots + u_{k-1} x^{k-1}$, the encoding of BCH codes consists of three steps:

(1) Compute $a(x) = x^{n-k} u(x)$.
(2) Compute $b(x) = a(x) \mod g(x)$.
(3) Form the codeword $c(x) = a(x) + b(x)$.

BCH codes decoding. The decoding of BCH codes also consists of three steps:

(1) Compute the 2δ syndromes from the received polynomial $r(x)$.
Let $c(x)$ denote the sent codeword and $e(x)$ the error word, one has:

$$r(x) = c(x) + e(x)$$

For $1 \leq i \leq 2\delta$, the syndromes S_i are defined as:

$$S_i = r(\alpha^i) = e(\alpha^i)$$

(2) Compute the Error Locator Polynomial (ELP) $\sigma(x)$ using the syndromes $(S_i)_{1 \leq i \leq 2\delta}$.
Let v be the number of errors and let $j_1, j_2, \ldots, j_v$ be the error positions. Then:

$$e(x) = x^{j_1} + x^{j_2} + \ldots + x^{j_v}$$

So:

$$S_i = (\alpha^i)^{j_1} + (\alpha^i)^{j_2} + \ldots + (\alpha^i)^{j_v} \qquad (1 \leq i \leq 2\delta)$$

Introducing the error locators $\beta_s = \alpha^{j_s}$, with $s = 1, 2, \ldots, v$, one can write the syndromes more explicitely:

$$\begin{aligned} S_1 &= \beta_1 + \beta_2 + \ldots + \beta_v \\ S_2 &= \beta_1^2 + \beta_2^2 + \ldots + \beta_v^2 \\ &\vdots \\ S_{2\delta} &= \beta_1^{2\delta} + \beta_2^{2\delta} + \ldots + \beta_v^{2\delta} \end{aligned}$$

These are known as power sum symmetric functions. They lead to the definition of the error locator polynomial:

$$\sigma(x) = \prod_{r=1}^{v}(1 + \beta_r x) = \sum_{r=0}^{v} \sigma_r x^r$$

$(\sigma_i)_{1 \leq i \leq v}$ and $(S_i)_{1 \leq i \leq 2\delta}$ are then related by Newton's identities:

$$\begin{aligned} & S_1 + \sigma_1 = 0 \\ & \vdots \\ & S_\delta + \sigma_1 S_{\delta-1} + \ldots + \sigma_{\delta-1} S_1 + \delta\sigma_\delta = 0 \\ & S_{\delta+1} + \sigma_1 S_\delta + \ldots + \sigma_{\delta-1} S_2 + \delta\sigma_\delta S_1 = 0 \\ & \vdots \\ & S_{2\delta} + \sigma_1 S_{2\delta-1} + \ldots + \delta\sigma_\delta S_1 = 0 \end{aligned} \tag{1}$$

(3) Compute the roots of the error locator polynomial $\sigma(x)$.
These roots $\beta_1^{-1}, \beta_2^{-1}, \ldots, \beta_v^{-1}$ are the inverses of the error locators. Once found, one can retrieve error positions $j_1, j_2, \ldots, j_v$ and correct $r(x)$.

Definition 2.12 (Repetition code)**.** The binary repetition code $\mathbb{1}_n$ of length n is the set of two codewords $\mathbf{1}_n$ (the all ones) and $\mathbf{0}_n$ (the all zeros). It has dimension 1 and correction capacity $\lfloor \frac{n-1}{2} \rfloor$.

The $\mathbb{1}_n$ code is an error-correcting code where encoding is done by repeating the message bit n times. Decoding is done by majority decision; it outputs 1 if there is a majority of 1 and 0 otherwise.

2.2. **The HQC public key encryption scheme.** Hamming Quasi-Cyclic [1,3] is a code-based IND-CCA2 secure encryption scheme whose security relies on the syndrome decoding problem. It is obtained by applying the HHK transformation [12] on the IND-CPA construction denoted HQC.PKE (depicted in Figure 1). HQC uses two types of codes: a tensor code $\mathcal{C}$ of generator matrix $\mathbf{G}$ and a random double-circulant $[2n, n]$-code with a parity check matrix $(1, \mathbf{h})$.

The correctness of HQC relies on the decoding capability of the code $\mathcal{C}$. Indeed, $\mathsf{Decrypt}(\mathbf{sk}, \mathsf{Encrypt}(\mathbf{pk}, \mathbf{m})) = \mathbf{m}$ when $\mathcal{C}.\mathsf{Decode}$ correctly decodes $\mathbf{v} - \mathbf{u} \cdot \mathbf{y}$, namely whenever $w(\mathbf{x} \cdot \mathbf{r}_2 - \mathbf{r}_1 \cdot \mathbf{y} + \mathbf{e}) \leq \delta$.

The tensor product code $\mathcal{C}$ is defined by $\mathcal{C} = \mathcal{B} \otimes \mathcal{R}$, where $\mathcal{B}$ is a $[n_1, k, \delta]$ BCH code and $\mathcal{R}$ is the $[n_2, 1, \lfloor \frac{n_2-1}{2} \rfloor]$ repetition code $\mathbb{1}_{n_2}$. Encoding a given message $\mathbf{m} \in \mathbb{F}_2^{k_1}$ is done in two steps. Firstly, it is encoded into $\mathbf{b} \in \mathbb{F}_2^{n_1}$ using the aforementioned BCH code $\mathcal{B}$. Secondly, each coordinate b_i of $\mathbf{b}$ is re-encoded into $\mathbf{c}^i \in \mathbb{F}_2^{n_2}$, for $0 \leq i \leq n_1 - 1$, with the repetition code $\mathcal{R} = \mathbb{1}_{n_2}$. This yields the codeword

$(\mathbf{c}^0\ \mathbf{c}^1\ \dots\ \mathbf{c}^{n_1-1})$. Similarly, decoding $\mathbf{a} = (\mathbf{a}^0\ \mathbf{a}^1\ \dots\ \mathbf{a}^{n_1-1})$ with $\mathbf{a}^i \in \mathbb{F}_2^{n_2}$ for $0 \leq i \leq n_1 - 1$ is also done in two steps. Firstly, the repetition code $\mathcal{R}$ decodes each $\mathbf{a}^i$ into a bit b_i. Secondly the BCH code $\mathcal{B}$ decodes the word $\mathbf{b} = (b_i)_{0 \leq i \leq n_1-1}$ into the message.

- $\mathsf{Setup}(\mathbf{1}^\lambda)$: Generate and return parameters $\mathsf{param} = (n, k, \delta, \mathbf{G}, \omega, \omega_\mathbf{r}, \omega_\mathbf{e})$
- $\mathsf{KeyGen}(\mathsf{param})$:
  - $\mathbf{sk} = (\mathbf{x}, \mathbf{y}) \xleftarrow{\$} (\mathbb{F}_2^n)^2$ such that $\omega(\mathbf{x}) = \omega(\mathbf{y}) = \omega$
  - $\mathbf{h} \xleftarrow{\$} \mathbb{F}_2^n$
  - $\mathbf{pk} = (\mathbf{h}, \mathbf{s} = \mathbf{x} + \mathbf{h} \cdot \mathbf{y})$
  - Return $(\mathbf{pk}, \mathbf{sk})$
- $\mathsf{Encrypt}(\mathbf{pk}, \mathbf{m})$:
  - $\mathbf{r} = (\mathbf{r}_1, \mathbf{r}_2) \xleftarrow{\$} (\mathbb{F}_2^n)^2$ such that $\omega(\mathbf{r}_1) = \omega(\mathbf{r}_2) = \omega_\mathbf{r}$
  - $\mathbf{u} = \mathbf{r}_1 + \mathbf{h} \cdot \mathbf{r}_2$
  - $\mathbf{e} \xleftarrow{\$} \mathbb{F}_2^n$ such that $\omega(\mathbf{e}) = \omega_\mathbf{e}$
  - $\mathbf{v} = \mathbf{mG} + \mathbf{s} \cdot \mathbf{r}_2 + \mathbf{e}$
  - Return $\mathbf{c} = (\mathbf{u}, \mathbf{v})$
- $\mathsf{Decrypt}(\mathbf{sk} = (\mathbf{x}, \mathbf{y}), \mathbf{c} = (\mathbf{u}, \mathbf{v}))$:
  - $\mathbf{a} = \mathbf{v} - \mathbf{u} \cdot \mathbf{y}$
  - $\mathbf{b} = (\mathcal{R}.\mathsf{Decode}(\mathbf{a}^0), \mathcal{R}.\mathsf{Decode}(\mathbf{a}^1), \dots, \mathcal{R}.\mathsf{Decode}(\mathbf{a}^{n_1-1}))$
  - $\mathbf{m} = \mathcal{B}.\mathsf{Decode}(\mathbf{b})$
  - Return $\mathbf{m}$

FIGURE 1. Description of HQC.PKE [1].

3. Correlation between decoding time and error weight

In this section, we show that there exists a correlation between the weight of the error to be decoded and the running time of the BCH codes decoding algorithm, assuming Berlekamp's simplified algorithm [14] (see appendix A) is used for the second step of decoding. We next describe an oracle distinguishing BCH codewords without errors from those with one error exactly using the running time of the HQC.Decrypt algorithm (see Figure 1).

Berlekamp's simplified algorithm (see appendix A) is an iterative algorithm solving the set of equations (1). It completes in δ iterations. It starts with $\sigma(x) = 1$. At iteration μ, it computes a quantity d_μ, called discrepancy, whose value is 0 if the μth equation from system (1) holds. If not, it corrects $\sigma(x)$ such that equation μ holds. The loop invariant is that after μ iterations, the first μ equations of system (1) are verified. Looking at the pseudocode from appendix A, one can see that:

- For a codeword without error, all discrepancies are zero and the algorithm completes without corrections.
- For a codeword with one error, the first syndrome is α^{j_1} where j_1 is the error position and one correction is needed.

Assuming constant running time for the other steps of $\mathcal{B}.\mathsf{Decode}$ (syndromes computation and roots search) as well as the other parts of the HQC.Decrypt subroutine (multiplication and repetition code decoding), one can build the aforementioned oracle. Let $\mathcal{O}_{\text{Time}}^{HQC}$ denote a timing oracle returning the running time of the

HQC.Decrypt algorithm. We now explain how to construct an oracle, denoted by $\mathcal{O}_{01}^{HQC}$, returning the weight (0 or 1) of the error corrected by the BCH code, using $\mathcal{O}_{\text{Time}}^{HQC}$. The oracle $\mathcal{O}_{01}^{HQC}$ takes as input an HQC public key $\mathbf{pk}$ (which implicitely defines a BCH code $\mathcal{B}$) and a ciphertext $\mathbf{c} = (\mathbf{u}, \mathbf{v})$.

The oracle features an initialization step Init (see Algorithm 1) and an evaluation step Eval (see Algorithm 2). The Init step computes the expected running times T_0 and T_1 when the BCH code corrects 0 and 1 error respectively. To obtain these times T_0 and T_1, the proper requests have to be submitted to $\mathcal{O}_{\text{Time}}^{HQC}$. In order to construct them, one has to account for the additionnal layers of multiplication and $\mathcal{R}$ decodings on top of BCH decoding. The repetition code layer sees its input $\mathbf{a}$, of length $n = n_1 n_2$, as n_1 blocks of n_2 bits:

$$\mathbf{a} = (\mathbf{a}^0, \mathbf{a}^1, \ldots, \mathbf{a}^{n_1-1}) \qquad \mathbf{a}^i \in \mathbb{F}_2^{n_2}$$

Each block $\mathbf{a}^i$ gives a bit b_i of the output vector $\mathbf{b}$ (fed to the BCH decoder) where $b_i = 1$ if the block contains a majority of 1 and $b_i = 0$ otherwise. To compute T_0 and T_1 we simply query the timing oracle $\mathcal{O}_{\text{Time}}^{HQC}$ and measure its response time with $\mathbf{u} = \mathbf{0}_n$ and $\mathbf{v} = \mathbf{0}_n$ to get an estimation of T_0 and $\mathbf{u} = \mathbf{0}_n$ and $\mathbf{v} = (\mathbf{1}_{n_2}\ \mathbf{0}_{n-n_2})$ to get an estimation of T_1 as $\mathbf{b} = (1\ \mathbf{0}_{n_1-1})$.

As described in Algorithm 1, for T_1 we make a sample of p requests and retain their mean as the estimate. The complexity of this initialization step is that of $1+p$ decodings which will be negligible with respect to the rest of the attack.

The Eval step takes a word $\mathbf{c}$ as input and guesses whether or not the BCH code corrects an error during the HQC decryption of $\mathbf{c}$. To this end, it calls $\mathcal{O}_{\text{Time}}^{HQC}(\mathbf{pk}, \mathbf{c})$, yielding the running time t, and outputs the error weight i such that $\mid t - T_i \mid$ is minimal.

The complexity of a $\mathcal{O}_{01}^{HQC}$ request (i.e. an Eval step) is equal to the complexity of an HQC decryption, namely $\mathcal{O}(n\sqrt{n})$ operations in $\mathbb{F}_{q^m}$ (under the assumption $\delta = \mathcal{O}(\sqrt{n})$ as is the case in HQC, see section 4.2).

Algorithm 1: Init step of $\mathcal{O}_{01}^{HQC}$

Input: A public key $\mathbf{pk}$
A precision parameter p
Output: A couple (T_0, T_1) of expected running times

$T_0 \longleftarrow \mathcal{O}_{\text{Time}}^{HQC}(\mathbf{pk}, \mathbf{0}_{2n})$
$T_1 \longleftarrow 0$
for $i \in (0, 1, \ldots, p-1)$ **do**
  $b \xleftarrow{\$} \{1, 2, \ldots, n_1\}$
  $\mathbf{c} \longleftarrow (\mathbf{0}_{(b-1)*n_2}\ \mathbf{1}_{n_2}\ \mathbf{0}_{n-b*n_2})$
  $T_1 \longleftarrow T_1 + \mathcal{O}_{\text{Time}}^{HQC}(\mathbf{pk}, \mathbf{c})$
$T_1 \longleftarrow T_1/p$

return (T_0, T_1)

Algorithm 2: Eval step of $\mathcal{O}_{01}^{HQC}$

Input: A public key $\mathbf{pk}$ and a ciphertext $\mathbf{c}$
Expected running times T_0 and T_1
Output: The error weight 0 or 1 that the BCH code $\mathcal{B}$ corrected during HQC.Decrypt($\mathbf{sk}, \mathbf{c}$)

$t \longleftarrow \mathcal{O}_{\text{Time}}^{HQC}(\mathbf{pk}, \mathbf{c})$

return i such that $\mid t - T_i \mid$ is minimal

4. Practicable timing attack against HQC

In this section, we present a side-channel chosen ciphertext attack against HQC. This attack is a real threat as it has a polynomial complexity and requires a reasonable amount of requests. It proceeds by iterations until the key $\mathbf{y}$ is recovered. We first give a brief overview of the attack in section 4.1. We follow by describing its first two iterations in sections 4.2 and 4.3. Finally, we estimate its success probability in section 4.4 and discuss the attack complexity and bandwidth cost in section 4.5.

4.1. Attack overview.

The key $\mathbf{y}$ has a Hamming weight of ω, meaning it contains ω bits 1 and $n - \omega$ bits 0. The objective of the attack is to recover the support of $\mathbf{y}$, i.e. (the positions of) all 1's. Consider secret key $\mathbf{y}$ as n_1 blocks of n_2 bits. After initializing the oracle $\mathcal{O}_{01}^{HQC}$, the attack proceeds by iterations. At iteration i, the attack searches block by block, finding out all 1's from each block containing exactly i. This is done by querying the oracle with appropriate requests. For all requests, the vector $\mathbf{u}$ is chosen as

$$\mathbf{u} = \bar{\mathbf{u}} := (1\ \mathbf{0}_{n-1})$$

such that $\mathbf{u} \cdot \mathbf{y} = \mathbf{y}$ and $\mathbf{a} = \mathbf{v} \oplus \mathbf{y}$. The input $\mathbf{a}$ isn't fed directly to the BCH code decoder but needs to go through the repetition code decoder first. So one wants to pick $\mathbf{v}$ such that $\mathbf{v} \oplus \mathbf{y}$ establishes a majority of 1's in the block that $\mathbf{v}$ alone wouldn't have. This naturally leads us to consider vectors $\mathbf{v}$ having a 1 in $\lfloor \frac{n_2}{2} \rfloor$ positions of a block $\mathbf{v}^i$. Doing so,

- either block $\mathbf{y}^i$ has a 1 in one of the remaining positions which leads $(\mathbf{v} \oplus \mathbf{y})^i$ to have a majority of 1's, and the oracle returns 1;
- or block $\mathbf{y}^i$ has no 1's in the remaining positions, $(\mathbf{v} \oplus \mathbf{y})^i$ has no majority of 1's, and the oracle returns 0.

Either way the oracle response leaks information on block $\mathbf{y}^i$'s content. Nevertheless, this strategy does not always work as $\mathbf{y}$ can have multiple 1's per block. When it does, these 1's could cancel those we set in $\mathbf{v}$ and break our majority, preventing us to gain information. This complexifies our task and is the reason why we split the attack in different iterations, each designed to search within $\mathbf{y}$'s blocks for a certain number of 1's. For the sake of clarity and simplicity, we only describe the first two iterations.

4.2. First iteration.

During the first iteration, we aim to recover all 1's of $\mathbf{y}$ alone in their block. Let's consider the $(i{+}1)$-th block $\mathbf{y}^i$ of $\mathbf{y}$ $(0 \leq i \leq n_1 - 1)$ and $\mathbf{v}^i$ the

corresponding block of $\mathbf{v}$. In order to determine the position of an eventual lone 1 in $\mathbf{y}^i$, we start querying the oracle with $(\bar{\mathbf{u}}, \mathbf{v})$ such that:

- $\mathbf{v}^j = \mathbf{0}_{n_2}$ if $j \neq i$
- $\mathbf{v}^i = (\mathbf{1}_{\lfloor \frac{n_2}{2} \rfloor}\ \mathbf{0}_{\lceil \frac{n_2}{2} \rceil})$

If the oracle response is 1, it means $\mathcal{B}$ corrected an error, thus $\mathbf{y}^i$ has a 1 in one of its last $\lceil \frac{n_2}{2} \rceil$ positions. Proceeding by dichotomy, we can then submit to the oracle the query $(\bar{\mathbf{u}}, \mathbf{v})$ with:

- $\mathbf{v}^j = \mathbf{0}_{n_2}$ if $j \neq i$
- $\mathbf{v}^i = (\mathbf{0}_{\left\lfloor \frac{\lceil \frac{n_2}{2} \rceil}{2} \right\rfloor}\ \mathbf{1}_{\lfloor \frac{n_2}{2} \rfloor}\ \mathbf{0}_{\left\lceil \frac{\lceil \frac{n_2}{2} \rceil}{2} \right\rceil})$

For example, if $n_2 = 31$, our first request would be with $\mathbf{v}^i = (\mathbf{1}_{15}\ \mathbf{0}_{16})$. Assuming a response 1 we would identify a 1 in one of the last 16 positions and follow with a second request where $\mathbf{v}^i = (\mathbf{0}_8\ \mathbf{1}_{15}\ \mathbf{0}_8)$, reducing by half the set of remaining candidates for the position of the 1. This allows us to pinpoint the position in $\lfloor \log_2 n_2 \rfloor + 1$ requests.

If we get a response 0 to our first request, the same amount of requests is enough to either find the position of the lone 1 or know there aren't any. However, since there are many more blocks without 1 than blocks with any, one can reduce the number of requests. Instead the second request is $(\bar{\mathbf{u}}, \mathbf{v})$ with:

- $\mathbf{v}^j = \mathbf{0}_{n_2}$ if $j \neq i$
- $\mathbf{v}^i = (\mathbf{0}_{\lceil \frac{n_2}{2} \rceil}\ \mathbf{1}_{\lfloor \frac{n_2}{2} \rfloor})$

This way, if the oracle returns 0, one can immediately dismiss the block with this second request as it does not have exactly one 1. This implies to perform an extra request if it turns out there's a 1 to find but saves us $\lfloor \log_2 n_2 \rfloor - 1$ requests most of the time. Since there are a total of n_1 blocks, and that $\mathbf{y}$ has at most ω blocks containing a single 1, the first iteration requires at most $2(n_1 - \omega) + \omega(\lfloor \log_2 n_2 \rfloor + 1)$ requests. Let's examine the complexity of this iteration. A request amounts to:

- the computation of $\mathbf{v} - \mathbf{u} \cdot \mathbf{y}$. The product complexity is $2\omega n + (\omega - 1)n$ (rotating ω arrays of size n and summing the resulting vectors). With the final addition, this iteration's complexity is $3\omega n$.
- n_1 $\mathcal{R}$-decodings of complexity $n_1((n_2 - 1) + 1) = n$ (for each of the n_1 blocks, its n_2 bits are summed and a comparison is done).
- a $\mathcal{B}$-decoding of complexity $\mathcal{O}(n_1^2)$ under the HQC hypothesis $\delta = \mathcal{O}(\sqrt{n})$.

Under the assumption $\omega = \mathcal{O}(\sqrt{n})$, we get a request complexity of $\mathcal{O}(n\sqrt{n})$ and an overall complexity in $\mathcal{O}(n^{\frac{5}{2}})$ for the first iteration.

The probability that the attack is successful after this first iteration is low enough (see section 4.4) that it calls for a second iteration.

4.3. **Second iteration.** The first iteration of the attack identified all 1's alone in their blocks. We now look for blocks of $\mathbf{y}$ containing exactly two 1's. In order to do so, we need to analyze what happens when one encounters such a block during the first iteration. There are two kinds of situations:

- case *a*: both 1's are in the same half of the block (including the middle position if n_2 is odd). If they're in the upper half, our first request gets a response 1 and we end up identifying the position of the 1 closer to the middle of the block. If they're in the lower half, our first request gets a

response 0 but our second request gets a response 1 and we again end up identifying the position of the 1 closer to the middle of the block.
- case b: both halves have a 1 (note that the case where n_2 is odd and there is a 1 in the middle would have been detected already). In that case the first two requests return 0 and the block is discarded.

The second iteration will be divided in two phases treating blocks falling in each case. One can remark that there should be roughly the same amount of blocks falling in each case, simply because if one fixes a position in a block and randomly picks another position of the block, there's almost as many positions left in the same half as in the other half.

4.3.1. *Phase 1.* Here the search is focused on blocks in which a 1 has already been identified. Clearly this situation is very similar to the first iteration. We can just ignore the 1 we know of, consider the block is of length $n_2 - 1$ and assume we need one less to achieve majority. This can be done using dichotomy as in the first iteration except each time we pick $\lfloor \frac{n_2}{2} \rfloor - 1$ positions out of these $n_2 - 1$. This phase can be performed efficiently as at most $\lfloor \frac{\omega}{2} \rfloor$ blocks have to be looked into. This makes a maximum of $\lfloor \frac{\omega}{2} \rfloor (\lfloor \log_2 n_2 \rfloor + 2)$ requests. Under the hypothesis $\omega = \mathcal{O}(\sqrt{n})$, this phase complexity is:

$$\lfloor \frac{\omega}{2} \rfloor (\lfloor \log_2 n_2 \rfloor + 2)\mathcal{O}(n\sqrt{n}) = \mathcal{O}(n^2 \log_2 n)$$

4.3.2. *Phase 2.* Now we turn to the remaining blocks. We want to catch those containing precisely two 1's. Let's recall that in the event of such a block, it has a 1 in each block half (and none in the middle if n_2 is odd). We can generalize the same strategy applied in the first iteration; we can distinguish if the block contains or not a pair of 1's in four requests $(\bar{\mathbf{u}}, \mathbf{v})$ with $\mathbf{v}^j = 0$ if $j \neq i$ and:

- $\mathbf{v}^i = (\mathbf{1}_{\left\lfloor \frac{\lfloor \frac{n_2}{2} \rfloor - 1}{2} \right\rfloor}\ \mathbf{0}_{\left\lceil \frac{\lceil \frac{n_2}{2} \rceil + 1}{2} \right\rceil}\ \mathbf{1}_{\left\lceil \frac{\lfloor \frac{n_2}{2} \rfloor - 1}{2} \right\rceil}\ \mathbf{0}_{\left\lfloor \frac{\lceil \frac{n_2}{2} \rceil + 1}{2} \right\rfloor})$
- $\mathbf{v}^i = (\mathbf{0}_{\left\lceil \frac{\lceil \frac{n_2}{2} \rceil + 1}{2} \right\rceil}\ \mathbf{1}_{\left\lfloor \frac{\lfloor \frac{n_2}{2} \rfloor - 1}{2} \right\rfloor}\ \mathbf{0}_{\left\lfloor \frac{\lceil \frac{n_2}{2} \rceil + 1}{2} \right\rfloor}\ \mathbf{1}_{\left\lceil \frac{\lfloor \frac{n_2}{2} \rfloor - 1}{2} \right\rceil})$
- $\mathbf{v}^i = (\mathbf{1}_{\left\lfloor \frac{\lfloor \frac{n_2}{2} \rfloor - 1}{2} \right\rfloor}\ \mathbf{0}_{\left\lceil \frac{\lceil \frac{n_2}{2} \rceil + 1}{2} \right\rceil}\ \mathbf{0}_{\left\lfloor \frac{\lceil \frac{n_2}{2} \rceil + 1}{2} \right\rfloor}\ \mathbf{1}_{\left\lceil \frac{\lfloor \frac{n_2}{2} \rfloor - 1}{2} \right\rceil})$
- $\mathbf{v}^i = (\mathbf{0}_{\left\lceil \frac{\lceil \frac{n_2}{2} \rceil + 1}{2} \right\rceil}\ \mathbf{1}_{\left\lfloor \frac{\lfloor \frac{n_2}{2} \rfloor - 1}{2} \right\rfloor}\ \mathbf{1}_{\left\lceil \frac{\lfloor \frac{n_2}{2} \rfloor - 1}{2} \right\rceil}\ \mathbf{0}_{\left\lfloor \frac{\lceil \frac{n_2}{2} \rceil + 1}{2} \right\rfloor})$

Since one knows the 1's are in different halves of the block, there are only four different pairs of quarters they can be in. Each of the aforementioned requests tests one such pair. Therefore, if the oracle returns 0 to these four requests, the block contains either no 1's or more than two. If the oracle answers 1 to one of these requests, one retrieves two ranges of indices, both containing a 1. Then proceeding by dichotomy for each range, one can narrow it down to a singleton in $\left\lfloor \log_2 \left\lceil \frac{\lceil \frac{n_2}{2} \rceil + 1}{2} \right\rceil \right\rfloor + 1$ requests. In the worst case scenario, we have $\lfloor \frac{\omega}{2} \rfloor$ blocks containing two 1's, none of which have been detected yet. This takes

$$\left\lfloor \frac{\omega}{2} \right\rfloor \left(4 + 2 \left\lfloor \log_2 \left\lceil \frac{\lceil \frac{n_2}{2} \rceil + 1}{2} \right\rceil \right\rfloor + 2 \right) + 4 \left(n_1 - \left\lfloor \frac{\omega}{2} \right\rfloor \right)$$

requests to find them all, from which we derive the second iteration complexity of:

$$\left[2 \left\lfloor \frac{\omega}{2} \right\rfloor \left(\left\lfloor \log_2 \left\lceil \frac{\lceil \frac{n_2}{2} \rceil + 1}{2} \right\rceil \right\rfloor + 1 \right) + 4n_1 \right] \mathcal{O}(n\sqrt{n}) = \mathcal{O}(n^{\frac{5}{2}})$$

4.4. **Success probability estimation.** Let's calculate the probabilities that $\mathbf{y}$ has been retrieved after each iteration. Let the following events be for $0 \leq i \leq \lfloor \frac{n_1}{2} \rfloor$:

$$\mathbf{A}_i : \text{“}\mathbf{y} \text{ has exactly } i \text{ blocks with two 1's and no block with more.”}$$
$$\mathbf{A} : \text{“}\mathbf{y} \text{ has at most two 1's per block.”}$$

The event $\mathbf{A}_0$ can also be described as the attack being successful after the first iteration. This means $\mathbf{y}$ has ω blocks containing a single 1 for which we have n_2 positions to choose from and $n_1 - \omega$ blocks containing none. Therefore:

$$\mathbb{P}(\mathbf{A}_0) = \binom{n_1}{\omega}\binom{n_2}{1}^{\omega}$$

With HQC-128-1 [1] parameters $n_1 = 796$, $n_2 = 31$ and $\omega = 67$, one has $\mathbb{P}(\mathbf{A}_0) \simeq 0.0625$. One recover 6.25 percent of potential keys $\mathbf{y}$ after the first iteration.

Let's now compute the probability $\mathbb{P}(\mathbf{A})$ that the attack is successful after at most two iterations. $\mathbf{A}$ is the disjoint union of the $\mathbf{A}_i$:

$$\mathbb{P}(\mathbf{A}) = \sum_{i=0}^{\lfloor n_1/2 \rfloor} \mathbb{P}(\mathbf{A}_i)$$

$$\mathbb{P}(\mathbf{A}) = \binom{n}{\omega}^{-1} \sum_{i=0}^{\lfloor n_1/2 \rfloor} \binom{n_1}{i}\binom{n_2}{2}^{i}\binom{n_1 - i}{\omega - 2i}\binom{n_2}{1}^{\omega-2i}$$

With $n_1 = 796$, $n_2 = 31$ and $\omega = 67$, one finds $\mathbb{P}(\mathbf{A}) \simeq 0.9344$. 93 percent of potential keys have been retrieved after the second iteration. One could show that the attack success probability after three iterations is above 99 percent.

4.5. **Attack complexity and bandwidth cost.** Table 1 presents the attack complexity and the number of required requests with respect to HQC parameters. Since the multiplication takes most of the decryption workload, we took twice its complexity (i.e. $6\omega n$) as an upper bound of a request complexity.

We implemented the attack locally for HQC-128-1. Table 1 assumes each oracle request is done once. However, in a real life scenario, different runs of the same request usually yield slightly different execution times. This derails the attack if the real execution time is closer to T_i than T_{1-i} but the measured execution time is closer to T_{1-i} than T_i for $i = 0, 1$. To mitigate this effect, we take the standard approach of repeating each request several times, each time measuring the execution time, and taking the median of the batch as execution time estimate. The tests were performed on a machine with 16GB of memory, equipped with an Intel core i7-7820X CPU@3.60GHz and with Hyper-Threading, Turbo Boost and SpeedStep features disabled. On this machine, repeating each request nine times, the attack against HQC-128-1 takes less than a minute to complete. We ran a thousand attacks. As expected, 7% of them fail because the key $\mathbf{y}$ has a block with at least three 1's. 5% of them also fail because of the aforementioned random nature of measurements. This can be lowered by raising the repeat count to the expense of a higher running time. Overall 88% of attacks succeed.

5. Constant time decoding of BCH codes

A constant time BCH code decoding algorithm naturally thwarts the attack. In this section we discuss how to construct such an algorithm. We start by precising

	Complexity upper bound			Requests		
	$128-1$	$192-2$	$256-3$	$128-1$	$192-2$	$256-3$
Oracle Init ($p=1$)	2^{25}	2^{26}	2^{27}	2	2	2
First iteration	2^{35}	2^{36}	2^{37}	1793	1936	2257
Second iteration - phase 1	2^{31}	2^{34}	2^{35}	198	350	528
Second iteration - phase 2	2^{35}	2^{37}	2^{38}	3448	3564	3844
Total	2^{36}	2^{38}	2^{39}	5441	5852	6631

128-1: 128-bit security and a Decoding Failure Rate (DFR) less than 2^{-64}
192-2: 192-bit security and a DFR less than 2^{-96}
256-3: 256-bit security and a DFR less than 2^{-128}

TABLE 1. Attack complexity and bandwidth cost against HQC

the constant time model we are considering and discuss how one can tranform a non constant time algorithm into a constant time one (section 5.1). We then apply these techniques to finite field arithmetic (section 5.2), syndromes and roots computation (section 5.3) and ELP computation (section 5.4). This allows us to provide two variants of a constant time algorithm to BCH code decoding. To finish, we provide the results of our tests and discuss which variant should be considered depending on the chosen BCH code and the targeted material (section 5.5).

5.1. **Constant time implementation.** For constant time implementation, two security models are usually considered: *full constant time*, where the algorithm running time is indeed constant; and *timing attack resistant*, where the algorithm running time is independant of its secrets (although its running time may vary). Since an attacker can force the BCH code decoder to use the secret $\mathbf{y}$ as its input (with ciphertext $(\mathbf{0}_n, \mathbf{1}_{n_2}\ \mathbf{0}_{n-n_2})$ for example), we hereafter consider the full constant time model.

There are three kinds of obstacles to constant time implementation: loops whose bound is input-dependant, branches whose condition is input-dependant and input-dependant memory accesses. Natural fixes for each of these obstacles would respectively be [16]:

- To patch loops whose condition depends upon inputs by supplying a constant bound (the maximum number of iterations) and performing dummy operations once the original bound has been reached.
- To patch branches whose condition depends upon inputs by executing both branches and using a flag to control which branch is effectively executed.
- To patch array accesses whose index depends upon inputs either by eliminating them or by ensuring the corresponding address is already cached.

Dealing with leaking array accesses can be done in several ways. Walters and Sinha Roy [16] suggest patching each such access by scanning the whole array to load it into the cache. For nested array accesses, this operation may induce a huge

performance penalty. One may scan the array less often, but it requires being careful about addresses not being evicted from the L1 cache. One also has to be wary of the compiler with this approach, as compilers tend to identify these kinds of "do nothing" loops and optimize them out. We will denote the approach of scanning an array having potential leaking accesses once (and only once) as a *cache-dependant patch* as it works only if the cache is big enough or if code parameters are small enough. Note that even if the access doesn't leak anymore, it still, stricly speaking, depends on the inputs. The second approach is a *cache-independant patch*, which consists of removing the array access entirely. The idea is to first determine the range of indices that can potentially be accessed, then loop on all these indices, each time performing either a dummy operation or the real one as needed.

Now recall from section 2.1 that BCH code decoding has three steps: syndromes computation, ELP computation and roots computation. To provide a constant time implementation of BCH code decoding, we need to achieve constant time for Galois field arithmetic as well as for each of these three steps. We propose two variants: one with some cache-dependant array accesses and one without any cache-dependant array access.

5.2. **Constant time field arithmetic.** All three steps of decoding make abundant use of field operations (mostly additions and multiplications) that need be constant time.

Addition. For addition we use coefficient-wise xor.

Multiplication. We propose two implementations for multiplication:

- lookup tables. Given log and antilog tables (relative to a primitive element $\alpha \in \mathbb{F}_{2^m}$), multiplying two elements of $\mathbb{F}_{2^m}$ is done by taking their logarithms, adding them modulo $2^m - 1$, and taking the antilog.
- the CLMUL instruction set. This is an extension to the x86 instruction set for microprocessors from Intel and AMD. The pclmulqdq instruction computes the 128-bit carry-less product of two 64-bit values. We then reduce modulo the primitive polynomial using bitwise operations.

Implementation 2 is constant time but requires support for the CLMUL instruction set. Note that if one knows of a more efficient multiplication implementation or if the CLMUL instruction set it not available, one can use any other multiplication implementation as long as it is constant time. Implementation 1 is faster but not constant time by itself because it uses three input-dependant array accesses. However, using the aforementioned cache-dependant patch, that is scanning both log and antilog tables at the beginning of decoding, we may have implementation 1 run in constant time, depending on cache size and code parameters. These two underlying implementations for field multiplication distinguish our two constant time implementation variants.

Squaring. For squaring we use bitwise operations with constant shift amounts.

Inversion. For inversion we use fast exponentiation.

5.3. **Constant time syndromes computation and roots computation.** We start with steps 1 and 3 of BCH decoding, i.e. computation of syndromes and roots.

For both we benefit from fast algorithms developped by Bernstein et al. [6], who built on previous work from Gao and Mateer [10]. They use an additive Fast Fourier Transform (FFT) algorithm to compute the syndromes and its transpose algorithm to compute the ELP roots. Both these algorithms are constant time. We refer the reader to the aforementioned papers for more details on the additive FFT.

We describe a small adjustment to these algorithms. Additive FFT is a recursive algorithm which calls two copies of itself. At each recursion level, some constants (called gammas and deltas) are computed using field operations. Bernstein et al. propose a bitsliced version of the algorithm. Since we use a non bitsliced version here, field operations are more costly. As a result, recomputing these constants is more expensive than accessing them from an array (even factoring some L1 cache misses). Therefore, we compute these constants only once and store them in lookup tables for our subsequent needs. Note that the array accesses to these tables are not subject to timing leaks.

5.4. **Constant time error locator polynomial computation.** Here we start with Berlekamp's simplified algorithm [5, 13] (see appendix A). We then use the standard techniques described in section 5.1 to make it constant time, opting for the cache-independant approach when we encounter input-dependant array accesses. Because pseudocode hides implementation details by nature whereas constant time is an implementation-sensitive property, we give a constant time C implementation of Berlekamp's simplified algorithm in appendix B.

5.5. **Test results.** The benchmarks are performed on a machine which has 16GB of memory and is equipped with an Intel core i7-7820X CPU @ 3.60GHz. Hyper-Threading, Turbo Boost and SpeedStep features are disabled. L1 data cache is 32 kilobytes. We pick six BCH codes of various parameters. For each chosen BCH code $[n, k, \delta]$, we conduct two tests (one for each implementation of field multiplication) as follows. We generate 10 000 erroneous codewords with a distribution of error weights between 0 and 1.1δ where errors positions are picked randomly. Each codeword is decoded 100 times. Out of each batch, the minimum execution time is taken as estimate execution time for decoding that codeword. For each error weight of the distribution, we also monitor minimum and maximum of these minimum running times. At the end of the test, for each of the two codewords giving global minimum or maximum, we run another 100 decodings and take the minimum to confirm whether or not these extremums are circumstantial (these recomputed values don't appear on the graphs). For the two BCH codes $[796, 256, 60]$ and $[766, 256, 57]$ used in HQC, we use some optimizations. Firstly we use hardcoded lookup tables for both $\mathbb{F}_{1024}$ and the FFT constants. Secondly we use all optimizations suggested by Bernstein et al. [6] regarding the additive FFT, namely picking an ideal basis to avoid twisting; dealing with 2-coefficent and 3-coefficient polynomials more efficiently and unrolling both the FFT and its transpose. Note that these codes are shortened BCH codes. Because it doesn't fundamentally impact our case, we won't discuss it here but we refer the reader to [1] for more details. An implementation will be made available at pqc-hqc.org.

We give the results in the form of graphs (see figures 2 and 3). Figure 2 features the decoding of all six codes using lookup tables for field multiplication whereas figure 3 features these same codes using the pclmulqdq instruction for field multiplication. Each graph is vertically centered around the mean execution time

t_{mean}. Vertical axes spread from $0,95t_{mean}$ to $1,05t_{mean}$, except for the last code $[32767, 16412, 1315]$ where it stretches from $0,85t_{mean}$ to $1,15t_{mean}$.

As expected, on one hand, the second implemention of multiplication looks perfectly constant time (see figure 3). For all six codes, regardless of number of errors, the relative difference between any extremum and the mean decoding time always stays under 1%. On the other hand, the first implementation appears to be constant time only for the first three codes, that is if $m \leq 12$, i.e. up to $\mathbb{F}_{4096}$ (see the first three graphs of figure 2). Above that, the first implementation runs into cache issues. Indeed, our implementation uses `uint16_t` to represent field elements, which means two bytes per element. For $\mathbb{F}_{8192}$, log and antilog tables require $2 * 2 * 8192 = 32767$ bytes, which completely fill the L1 data cache of 32 kilobytes for the considered machine. From there, any computation will lead to addresses being evicted from the cache, which in turn will cause timing leaks (see the last three graphs of figure 2). For $\mathbb{F}_{4096}$, the lookup tables take only half the memory, which seems to leave enough for our decoding needs. However, for the small fields where it is constant time, the first implementation has better performance than the second (see tables 3 and 4). For the BCH codes used in HQC, observed decoding times are 30% faster. So our recommendation would be to use the first multiplication implementation (lookup tables) for BCH codes on field $\mathbb{F}_{4096}$ or smaller, which is the case of HQC, and to use the second multiplication implementation (via `pclmulqdq`) for larger fields.

We integrated the constant time BCH decoding algorithm in the optimized implementation of HQC IND-CCA2 to measure the performance overhead. We restrict our measurements to the lookup tables variant of the BCH decoding. In table 2 we report CPU cycles counts for the decapsulation step of HQC across the different security levels with either the original BCH implementation or the constant time variant. One can see that our constant time implementation only adds a little overhead between 3.21% and 11.06%.

TABLE 2. Running time (CPU cycles) and overhead when original or constant time BCH decoding is used in the decapsulation step of HQC

	HQC.Decaps		Overhead
	Original BCH	Constant time BCH	
HQC-128-1	507285	563414	11.06%
HQC-192-1	947552	995272	5.05%
HQC-192-2	992057	1047054	5.54%
HQC-256-1	1490993	1538824	3.21%
HQC-256-2	1562207	1616673	3.49%
HQC-256-3	1617269	1675195	3.58%

6. Conclusion

In this work, we have highlighted a correlation between the weight of the error to be decoded and the running time of decoding BCH codes when Berlekamp's simplified algorithm is straightforwardly implemented. Next, we have devised an efficient chosen ciphertext timing attack against HQC based on that correlation. We then implemented it in software and carried it out against different security levels of HQC. The attack is very efficient as it recovers the secret key $\mathbf{y}$ often enough in a couple iterations and its overall complexity is $\mathcal{O}(n^{\frac{5}{2}})$. In order to thwart this attack, we proposed two variants of a constant-time decoding algorithm for BCH codes. Furthermore, we integrated our new constant time algorithm in the latest version of HQC and showed that this countermeasure results in minimal overhead performance.

References

[1] C. Aguilar-Melchor, N. Aragon, S. Bettaieb, L. Bidoux, O. Blazy, J.-C. Deneuville, P. Gaborit, E. Persichetti and G. Zémor, Hamming Quasi-Cyclic (HQC), 2017.

[2] C. Aguilar-Melchor, N. Aragon, S. Bettaieb, L. Bidoux, O. Blazy, J.-C. Deneuville, P. Gaborit and G. Zémor, Rank Quasi-Cyclic (RQC), 2017.

[3] C. Aguilar-Melchor, O. Blazy, J.-C. Deneuville, P. Gaborit and G. Zémor, Efficient Encryption from Random Quasi-Cyclic Codes, *IEEE Transactions on Information Theory*, **64** (2018), 3927–3943.

[4] S. Bettaieb, L. Bidoux, P. Gaborit and E. Marcatel, Preventing timing attacks against RQC using constant time decoding of Gabidulin codes, in *International Conference on Post-Quantum Cryptography*, Springer, (2019), 371–386.

[5] E. R. Berlekamp, Non-binary BCH decoding, Technical report, North Carolina State University. Dept. of Statistics, 1966.

[6] D. J. Bernstein, T. Chou and P Schwabe, Mcbits: fast constant-time code-based cryptography, In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 250–272. Springer, 2013.

[7] R. Chandra. Bose and D. K. Ray-Chaudhuri, On a class of error correcting binary group codes, *Information and control*, **3** (1960), 68–79.

[8] J.-P. D'Anvers, F. Vercauteren and Ingrid Verbauwhede, On the impact of decryption failures on the security of LWE/LWR based schemes, IACR Cryptology ePrint Archive, 2018:1089, 2018.

[9] E. M. Gabidulin, Theory of codes with maximum rank distance, *Problemy Peredachi Informatsii*, **21** (1985), 3–16.

[10] S. Gao and T. Mateer, Additive fast fourier transforms over finite fields, *IEEE Transactions on Information Theory*, **56** (2010), 6265–6272.

[11] A. Hocquenghem, Codes correcteurs derreurs, *Chiffres*, **2** (1959), 147–56.

[12] D. Hofheinz, K. Hövelmanns and E. Kiltz, A modular analysis of the Fujisaki-Okamoto transformation, in *Theory of Cryptography Conference*, pages 341–371. Springer, 2017.

[13] L. L. Joiner and J. J. Komo, Decoding binary BCH codes, In *Southeastcon '95*, 1995.

[14] S. Lin and D. J. Costello, in *Error control coding*, Prentice Hall Englewood Cliffs (2004).

[15] X. Lu, Y. Liu, Z. Zhang, D. Jia, H. Xue, J. He, B. Li, K. Wang, Z. Liu and H. Yang, LAC: Practical Ring-LWE based Public-Key Encryption with Byte-Level Modulus, IACR Cryptology ePrint Archive, 2018:1009, 2018.

[16] M. Walters and S. Sinha Roy, Constant-time BCH Error-Correcting Code, IACR Cryptology ePrint Archive, 2019:155, 2019.

TABLE 3. Decoding of some BCH codes with multiplication by tables

BCH code $[n, k, \delta]$	Running time (in CPU cycles)				
	LuT	Syndromes	ELP	Roots	Total
$[766, 256, 57]$	0	34240	30089	26778	91873
$[796, 256, 60]$	0	34646	33359	27086	95861
$[4095, 418, 501]$	82491	291827	2145899	187004	2711521
$[8191, 7580, 47]$	124587	278191	23216	186407	616569
$[16383, 14598, 130]$	245850	789651	166062	552630	1760773
$[32767, 16412, 1315]$	503337	2531258	17361393	1786677	22217535

TABLE 4. Decoding of some BCH codes with multiplication by pclmulqdq

BCH code $[n, k, \delta]$	Running time (in CPU cycles)				
	LuT	Syndromes	ELP	Roots	Total
$[766, 256, 57]$	0	42799	50735	34017	128226
$[796, 256, 60]$	0	43560	55562	34404	134157
$[4095, 418, 501]$	96997	474817	4585893	321102	5482880
$[8191, 7580, 47]$	134176	443016	61288	311542	953739
$[16383, 14598, 130]$	260450	1501411	474177	1106680	3352090
$[32767, 16412, 1315]$	484200	2143567	14832791	1514189	18996691

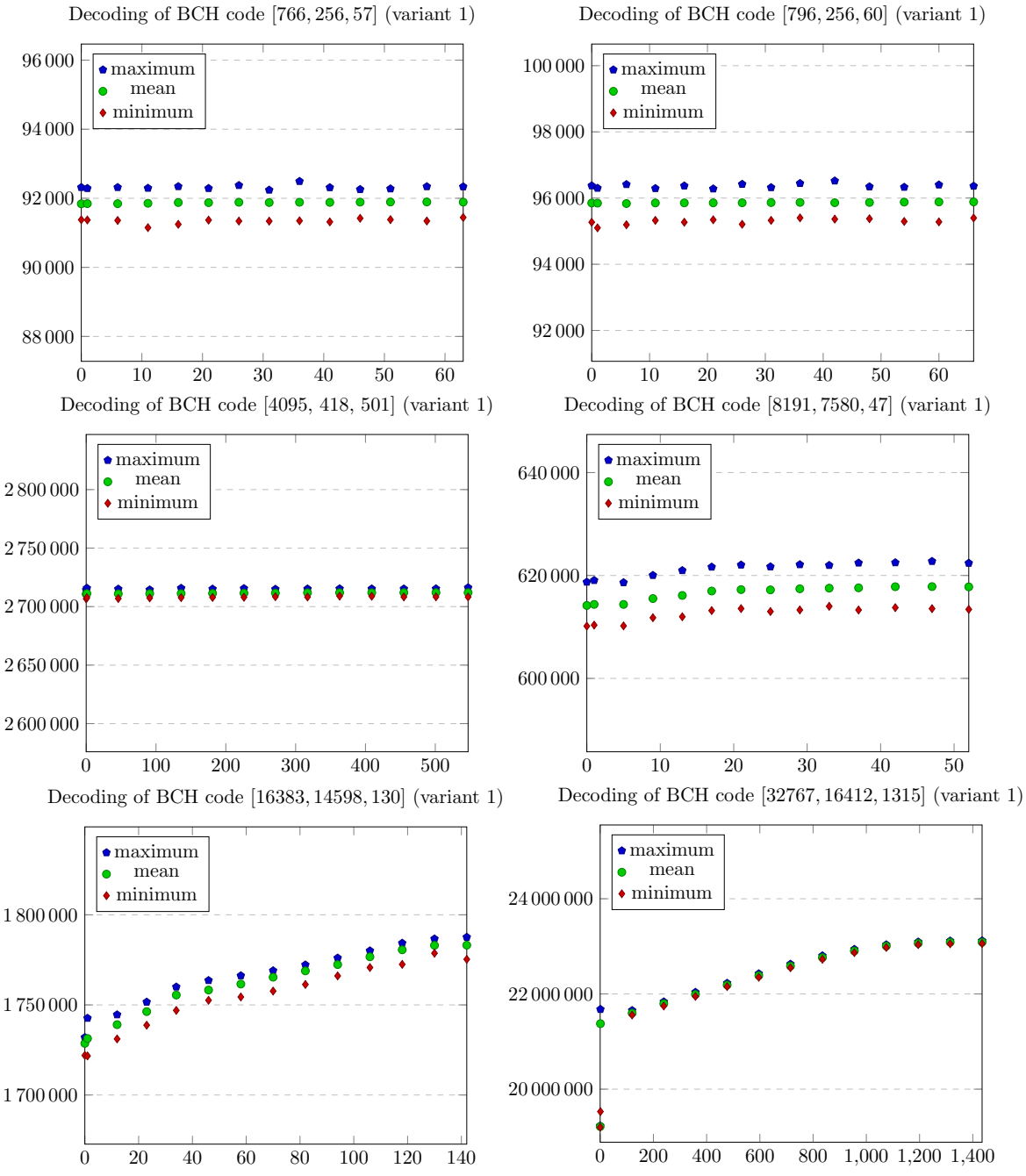

FIGURE 2. Decoding execution times (in CPU cycles) of various BCH codes for different error weights with field mutliplication implemented by lookup tables (variant 1).

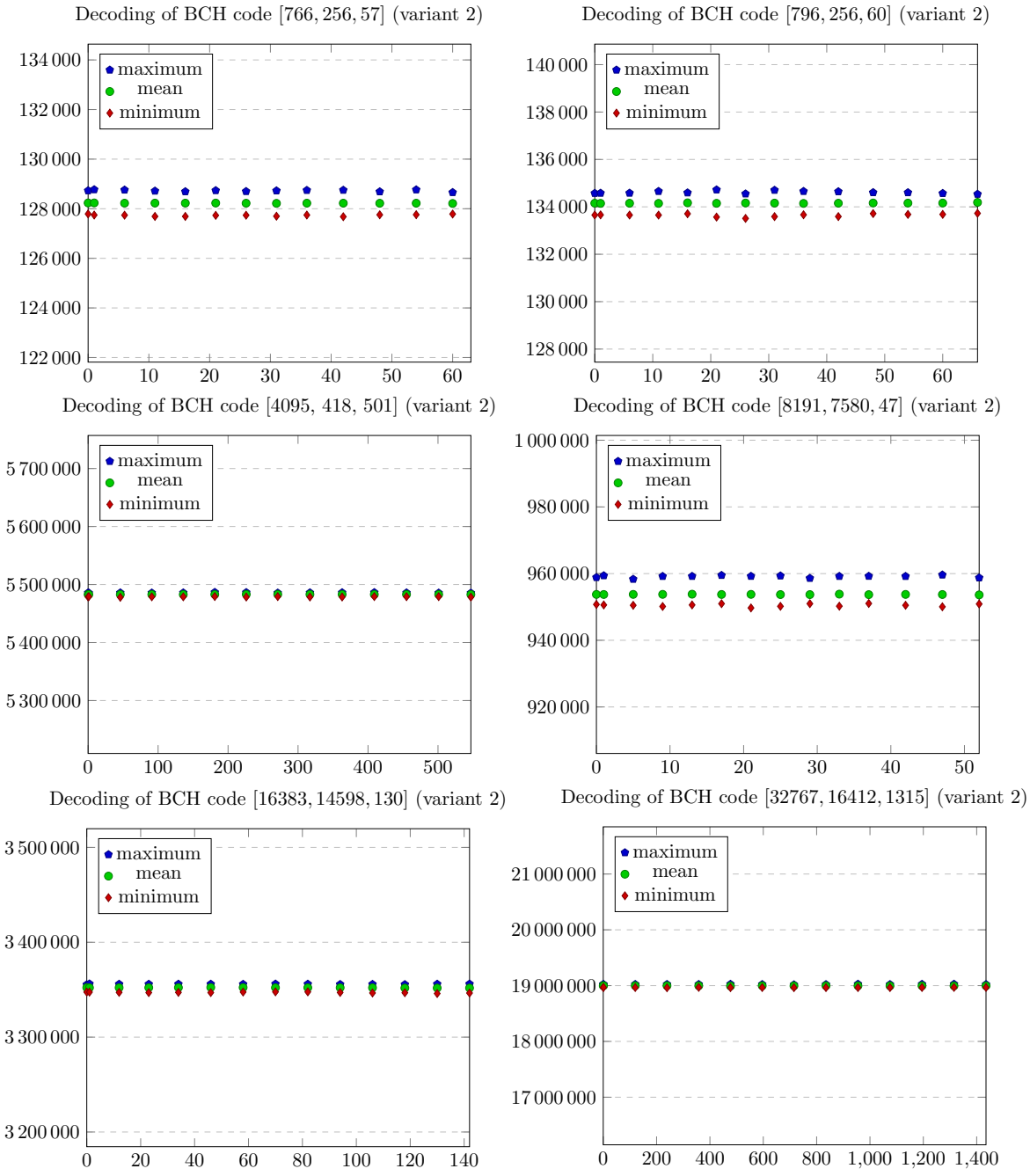

FIGURE 3. Decoding execution times (in CPU cycles) of various BCH codes for different error weights with field multiplication implemented via pclmulqdq instruction (variant 2).

Appendix A. ELP computation

Algorithm 3: Simplified Berlekamp algorithm [5, 13]

Input: A list of syndromes S_1, S_2, ..., $S_{2\delta}$
Output: The corresponding error locator polynomial $\sigma(X)$

`/* Initialize the following array */`

μ	$\sigma^{(\mu)}(X)$	d_μ	l_μ	$2\mu - l_\mu$
$-\frac{1}{2}$	1	1	0	-1
0	1	S_1	0	0

`/* Fill the array's next lines as follows */`

repeat
- **if** $d_\mu = 0$ **then**
  - $\sigma^{(\mu+1)}(X) = \sigma^{(\mu)}(X)$
  - $l_{\mu+1} = l_\mu$
- **if** $d_\mu \neq 0$ **then**
  - Find another line ρ, where $d_\rho \neq 0$ and $2\rho - l_\rho$ is maximal
  - $\sigma^{(\mu+1)}(X) = \sigma^{(\mu)}(X) + d_\mu d_\rho^{-1} X^{2(\mu-\rho)} \sigma^{(\rho)}(X)$
  - $l_{\mu+1} = \max(l_\mu, l_\rho + 2(\mu - \rho))$
- Compute $d_{\mu+1} = S_{2\mu+3} + \sigma_1^{(\mu+1)} S_{2\mu+2} + \ldots + \sigma_{l_{\mu+1}}^{(\mu+1)} S_{2\mu+3-l_{\mu+1}}$
- Increment μ and compute $2\mu - l_\mu$

until $\mu = \delta$

return $\sigma^{(\delta)}(X)$

Appendix B. Constant time ELP computation

The C function below computes the error locator polynomial using a constant time version of Berlekamp's simplified algorithm. It has the following features:

- The constant `PARAM_DELTA` is the correction capacity $\delta > 1$ of the BCH code.
- Elements of $\mathbb{F}_{2^m}$ are represented by `uint16_t` as polynomials ($m \leq 15$).
- `gf_mul` is the Galois field multiplication. It takes two elements and returns their product.
- `gf_inverse` computes an element inverse. It returns 0 for input 0.
- `syndromes` is an array of size `2*PARAM_DELTA` storing the 2δ syndromes.
- `sigma` is an array of size `PARAM_DELTA + 1` that will receive the $\delta + 1$ coefficients of the ELP.
- The function returns the degree of $\sigma(X)$ (and its coefficients in the array `sigma`).
- The array `X_sigma_p` represents the polynomial $X^{2(\mu-\rho)}\sigma_\rho(X)$.
- Instead of maintaining a list of $\sigma^{(i)}(X)$, we update in place both $\sigma(X)$ (array `sigma`) and the corrective term $X^{2(\mu-\rho)}\sigma_\rho(X)$ (array `X_sigma_p`).
- We don't care about $\sigma(X)$ if its degree exceeds `PARAM_DELTA` [13]. So we don't care about `X_sigma_p` if its degree exceeds `PARAM_DELTA` either.
- `sigma_copy` serves as a temporary save of `sigma` in case we need it to update `X_sigma_p`. We only need to save the first `PARAM_DELTA`-1 coefficients of `sigma`.

```
size_t
compute_elp(uint16_t             *sigma,
            const uint16_t       *syndromes)
{
    memset(sigma, 0, 2*(PARAM_DELTA+1));
    sigma[0] = 1;
    size_t deg_sigma = 0;
    size_t deg_sigma_p = 0;
    uint16_t sigma_copy[PARAM_DELTA - 1] = {0};
    size_t deg_sigma_copy = 0;
    uint16_t X_sigma_p[PARAM_DELTA + 1] = {0, 1};
    int pp = -1; // 2*rho
    uint16_t d_p = 1;
    uint16_t d = syndromes[0];

    for (size_t mu = 0; mu < PARAM_DELTA; ++mu) {
        // Save sigma in case we need it to update X_sigma_p
        memcpy(sigma_copy, sigma, 2*(PARAM_DELTA-1));
        deg_sigma_copy = deg_sigma;

        uint16_t dd = gf_mul(d, gf_inverse(d_p)); // 0 if (d == 0)
        for (size_t i = 1; (i <= 2*mu+1) && (i <= PARAM_DELTA); ++i)
            sigma[i] ^= gf_mul(dd, X_sigma_p[i]);

        size_t deg_X = 2*mu-pp; // 2*(mu-rho)
        size_t deg_X_sigma_p = deg_X + deg_sigma_p;

        // mask1 = 0xffff if (d != 0) and 0 otherwise
        int16_t mask1 = -((uint16_t)-d >> 15);
```

```
        // mask2 = 0xffff if (deg_X_sigma_p > deg_sigma) and 0 otherwise
        int16_t mask2 = -((uint16_t)(deg_sigma - deg_X_sigma_p) >> 15);

        // mask12 = 0xffff if the deg_sigma increased and 0 otherwise
        int16_t mask12 = mask1 & mask2;

        deg_sigma = (mask12 & deg_X_sigma_p) ^ (~mask12 & deg_sigma);

        if (mu == PARAM_DELTA-1)
            break;

        // Update pp, d_p and X_sigma_p if needed
        pp = (mask12 & (2*mu)) ^ (~mask12 & pp);
        d_p = (mask12 & d) ^ (~mask12 & d_p);
        for (size_t i = PARAM_DELTA-1; i; --i)
            X_sigma_p[i+1] = (mask12 & sigma_copy[i-1])
                ^ (~mask12 & X_sigma_p[i-1]);
        X_sigma_p[1] = 0;
        X_sigma_p[0] = 0;
        deg_sigma_p = (mask12 & deg_sigma_copy)
            ^ (~mask12 & deg_sigma_p);

        // Compute the next discrepancy
        d = syndromes[2*mu+2];
        for (size_t i = 1; (i <= 2*mu+1) && (i <= PARAM_DELTA); ++i)
            d ^= gf_mul(sigma[i], syndromes[2*mu+2-i]);
    }

    return deg_sigma;
}
```

Email address: kyzdra@yahoo.fr
Email address: slim.bettaieb@worldline.com
Email address: loic.bidoux@worldline.com
Email address: gaborit@unilim.fr