Profiling Side-Channel Attack on HQC Polynomial Multiplication Using Machine Learning Methods

Tomáš Rabas[1] [0000−0002−0924−359X], Jiří Buček[1] [0000−0003−1359−4285], Vincent Grosso[3] [0000−0002−3874−7527], Karolína Zenknerová[2] [0009−0006−8706−1894], and Róbert Lórencz[1] [0000−0001−5444−8511]

[1] Faculty of Information Technology, Czech Technical University in Prague, Czech Republic
{tomas.rabas, jiri.bucek, robert.lorencz}@fit.cvut.cz
[2] National Cyber and Information Security Agency, Prague, Czech Republic
[3] Université Jean Monnet Saint-Etienne, CNRS, Institut d'Optique Graduate School, Laboratoire Hubert Curien UMR 5516, F-42023, SAINT-ETIENNE, France
vincent.grosso@cnrs.fr

Abstract. The Hamming Quasi-Cyclic (HQC) cryptosystem was selected for standardization in the 4th round of the NIST post-quantum standardization competition targeting Public-key Encryption and Key-establishment algorithms. In this paper, we propose a profiling power side-channel attack on a HQC cryptosystem exploiting power consumption leakage during polynomial multiplication in the beginning of the decryption. The new attack scheme is based on generic methods such as Welch's ANOVA test or multilayer perceptron with a grid-search algorithm used for the hyperparameter tuning. Consequently, it is easily extendable also to other side-channel attacks. Results of the practical evaluation are presented, using a 32-bit STM32F303 Arm Cortex-M4 processor as a target. We show that a trained model is able to recover the correct key bit from large majority of tests – in the practical evaluation, maximum 6 out of 20 000 tests failed in recovering the correct key bit. For the testing, we follow a single-trace scenario, which assumes that only one decryption power trace is available.

Keywords: Side-Channel Attack, Polynomial Multiplication, HQC, Post-Quantum Cryptography, Machine Learning

1 Introduction

Recent advances in quantum computer design have given rise to one of the greatest security threats in modern cryptography. Indeed, in 1994, Peter Shor published algorithms dedicated to quantum computers, which were able to efficiently solve problems associated with discrete logarithms and factorization of big integers [Sho94] – two mathematical operations on which the classical public-key cryptography is based. This has started a new era of the so-called post-quantum cryptography (PQC) [NSB+23].

In 2016, the National Institute of Standards and Technology (NIST) has started a new public-key algorithm contest aiming to select new quantum-resistant public-key algorithms. In July 2022, NIST closed the 3rd selection round [AAC+22], in which the CRYSTALS-Kyber algorithm has been selected as a new standard for the Public-key

Encryption and Key-establishment algorithms, under the name of ML-KEM in the Federal Information Processing Standard (FIPS) 203. At the same time, NIST has started the 4th selection round aiming to find an alternative algorithm. Four candidates were included in the 4th round: BIKE, Classic McEliece, HQC and SIKE. While SIKE was later found to be critically vulnerable [CD23], the HQC algorithm was later on 11th March 2025 from the remaining candidates chosen as the new standard.

Besides the algorithmic aspects, also the implementation aspects including possible side-channel attacks have to be evaluated in the framework of the selection process. This remains an open task.

Related Works The first published power side-channel attack on HQC [SRSWZ21] presents how to build an oracle that specifies whether the BCH decoder in HQC's decryption algorithm corrects an error for a chosen ciphertext with 10 000 measurements (traces).

Goy et al. [GLG22] show that it is possible to retrieve a static secret key targeting the Hadamard transform in the Reed-Muller (RM) decoding step with an electromagnetic attack using 20 000 traces.

Article	Attacked part	# traces	Method	Targeted secret
[SRSWZ21]	BCH decoder	4[4]/10 000[5]	PC Oracle (templates)	private key
[GLG22]	Hadamard transf. (RM dec.)	20 000	PC Oracle (LDA[6])	private key
[SHR$^+$22]	RSRM decoder	1 000[1]/100 000[2]	PC Oracle (templates)	private key
[GMGL24]	RS decoder	1	PC Oracle (SASCA)	shared secret key
[DG24]	RM decoder	$< 1\,000$	PC Oracle (CNN[7])	private key
This paper	base_mul	1	MLP[8]	private key

Table 1: Known side-channel attacks on the HQC cryptosystem

The attack proposed by Schamberger et al. [SHR$^+$22] targets the Reed-Muller Reed-Solomon (RMRS) version of HQC from the 3rd round of the NIST competition by adapting the idea of the attack from [UXT$^+$22]. The attack exploits side-channel information from the execution of a pseudorandom function (PRF) or pseudorandom number generator (PRG) in the re-encryption of the KEM decapsulation. The attack in [SHR$^+$22] creates a Plaintext-Checking (PC) oracle using SCA. The oracle is then exploited to recover the private key block by block.

Moving to more recent results, the study by Dong and Guo [DG24] presents plaintext-checking oracle based attack using templates constructed from offline access to the publicly available decoding function of Reed-Muller codes. This attack significantly reduces the number of oracle calls.

[4]number of template traces for initialization of the oracle

[5]number of correctly classified traces necessary for oracle calls for full-key recovery

[6]Linear Discriminant Analysis (LDA)

[7]Convolutional Neural Network (CNN)

[8]Multilayer Perceptron (MLP)

The first Soft Analytical Side-Channel Attack (SASCA) on code-based cryptography and HQC was presented by Goy et al. [GMGL24]. They targeted mainly the Reed-Solomon (RS) decoder and achieved to recover a shared secret key using a single power trace. As part of the attack, they also target a polynomial multiplication inside the RS decoder, which computes a multiplication of an 8-bit polynomial representing part of a codeword as the first input and an 8-bit polynomial representing part of the parity check matrix as the second input. In comparison with this paper, we target a polynomial multiplication that occurs before the RS decoder itself and where the private key and ciphertext are the inputs for the multiplication. Also, we focus on recovering the private key (not shared secret key), as the attacker with its possession can simply decode the shared secret key from transmitted data.

It can be observed that all attacks designed to recover the private key necessitate a minimum of one thousand traces. Conversely, the sole single trace attack is capable of recovering the shared secret key.

Regarding HQC implementation protected by a countermeasure, Spyropoulos et al. [SVP$^+$24] presented a masked NIST submission specification-compliant vector generation function. It provides a secure implementation of one part of the whole HQC KEM.

Contributions In this paper, we perform a simple yet efficient profiled side-channel attack against the additional implementation of HQC. It is the first attack targeting the polynomial multiplication in the beginning of the HQC decryption (i.e. before decoding), and also the first single-trace SCA attack against HQC cryptosystem which targets the private key.

As described in Section 4.3, practical evaluation suggests that we can recover the full private key with 51.1% probability from a single trace. The complexity of further brute-forcing the potentially incorrect bits is $2^{14.11}$ following the strategy presented by Zaid et al. [ZBHV21], making the attack highly feasible. We also propose two countermeasures based on masking [GP99,ISW03]. The first countermeasure is a low-entropy approach that leverages the sliding window technique utilized in the base_mul operation. The second countermeasure is a classical Boolean masking implementation of the base_mul sensitive function with two shares. Security evaluation of these countermeasures is provided using our proposed attack, showing that both of them prevent the recovery of the private key in the attack.

Our attack consists of a profiling phase and an attacking phase. In the profiling phase, we:

- measure the training dataset of 40 000 traces and validation dataset of 10 000 traces, both with a unique random key and ciphertext for each trace;
- select points of interest using the Welch's ANOVA statistical tests on the training dataset;
- train the multi-layer perceptron models for each bit using the datasets for training and validation.

In the attacking phase, we test the ability of the models to recover the private key bits using an independently measured dataset of 20 000 traces. At the last step, classical post-processing may be needed to correct a few incorrectly predicted bits.

Paper Organization In Section 2, we begin by presenting the targeted Hamming Quasi-Cyclic (HQC) cryptosystem, providing details of the concrete target implementation and explaining algorithms that we use for the attack. In Section 3, we present a profiling side-channel analysis (SCA) process on the polynomial multiplication at the beginning of the HQC decryption. In Section 4, we describe two possible mitigations with different cost and security efficiencies and show the results of our attack with a comparison to the original implementation. In the end, we summarize the results of this paper in Section 5.

2 Background

In this section, we provide overview of the HQC algorithm and describe the hardware and software which was used to validate the attack principle, and briefly describe generic attack methods which have been used.

We denote by $\mathrm{GF}(2)$ the binary finite field consisting of the elements 0 and 1, and we denote by $\mathcal{R}$ the polynomial ring $\mathrm{GF}(2)[X]/(X^n-1)$, where n is a positive integer. For a polynomial $h \in \mathcal{R}$, the Hamming weight w_h refers to the number of non-zero coefficients in the vector representation of h. The notation $\texttt{sample}(\mathcal{R})$ indicates that we uniformly sample an element from $\mathcal{R}$, while $\texttt{sample}(\mathcal{R}, w_h)$ specifies that the Hamming weight of the sampled element must be equal to w_h.

2.1 Overview of the Hamming Quasi-Cyclic (HQC) Cryptosystem

Hamming Quasi-Cyclic (HQC) algorithm [MAB$^+$18] is an IND-CCA2 secure post-quantum cryptographic scheme for Key Encapsulation Mechanisms (KEM) based on HQC Public-Key Encryption (PKE) which is only CPA secure. The HQC PKE relies on the hardness of problems related to codes, specifically leveraging the hardness of the Quasi-Cyclic Syndrom Decoding (QCSD) problem. QCSD is a restriction of the Syndrom Decoding (SD) problem to quasi-cyclic codes. The SD problem can be described as follows: Given a matrix H, a syndrome s, and a weight w, find a vector e of weight w such that $He = s$. The SD problem has been proved to be NP-complete in 1978 by Berlekamp, McEliece and van Tilborg [BMvT78].

Although we only focus our attack on the decryption operation described in Algorithm 3, for the sake of completeness, we also include a concise description of the rest of the cryptosystem.

HQC Public Key Encryption Following the description from [HSC$^+$23], HQC PKE consists of three algorithms – key generation, encryption and decryption. HQC uses two codes. The first one is a public $[n,k]$-code $\mathcal{C}$, generated by G, which can correct at least Δ errors efficiently. The second one is a random double-circulant $[2n,n]$-code, where the parity check matrix can be expressed as $(\mathbf{1},h)$, exploiting the double-circulant property. The code $\mathcal{C}$ has been originally proposed as a concatenated code of BCH and Repetition code. Later on, it was superseded by a concatenation of Reed-Muller and Reed-Solomon code (RMRS) that was found to be strictly better than the BCH-Repetition version [HQCa].

In key generation, polynomials h, x and y are first randomly sampled, where the Hamming weight of x and y is w. Then, the private and public key are set to be $sk = (x, y)$ and $pk = (h, s)$ respectively, where $s = x + h \cdot y$. The process is described in Algorithm 1.

Algorithm 1 The key-generation algorithm of the HQC PKE.

```
Require: HQC security parameters
Ensure: private key $sk = (x,y)$ and public key $pk = (h,s)$
1: $h \leftarrow \texttt{sample}(\mathcal{R})$
2: $x \leftarrow \texttt{sample}(\mathcal{R}, w)$
3: $y \leftarrow \texttt{sample}(\mathcal{R}, w)$
4: $s \leftarrow x + h \cdot y$
5: return $sk = (x,y)$ and $pk = (h,s)$
```

In encryption, first a pseudo-random number generator is initialized by θ. Polynomials r_1 and r_2 with Hamming weight w_r and polynomial e with Hamming weight w_e are (deterministically) sampled. Then, a ciphertext $c = (u, v)$ is formed, where polynomials u and v are computed as follows: $u = r_1 + h \cdot r_2$ and $v = m\mathbf{G} + s \cdot r_2 + e$. The matrix $\mathbf{G}$ is defined by the code $\mathcal{C}$ being used. The process is described in Algorithm 2.

Algorithm 2 The encryption algorithm of the HQC PKE.

```
Require: public key $pk = (h,s)$, plaintext $m$, seed $\theta$
Ensure: ciphertext $c = (u,v)$
1: $r_1 \leftarrow \texttt{sample}(\mathcal{R}, w_r)$
2: $r_2 \leftarrow \texttt{sample}(\mathcal{R}, w_r)$
3: $e \leftarrow \texttt{sample}(\mathcal{R}, w_e)$
4: $u = r_1 + h \cdot r_2$
5: $v = m\mathbf{G} + s \cdot r_2 + e$
6: return $c = (u,v)$
```

Decryption only consists of decoding $v - u \cdot y$, where

$$\begin{aligned} v - u \cdot y &= m\mathbf{G} + s \cdot r_2 + e - (r_1 + h \cdot r_2) \cdot y \\ &= m\mathbf{G} + (x + h \cdot y) \cdot r_2 + e - r_1 \cdot y - h \cdot y \cdot r_2 \\ &= m\mathbf{G} + \underbrace{x \cdot r_2 - r_1 \cdot y + e}_{\text{denoted } e'} \end{aligned}$$

The decoding will be successful if the Hamming weight of the error term e' is sufficiently small so that the code $\mathcal{C}$ can correct it. In this paper, we focus on the side-channel leakage of the polynomial multiplication $u \cdot y$ computed during the decryption process. The process is described in Algorithm 3, where the target operation is highlighted on line 1. More specifically, our attack focuses on recovering the bits in operand y.

Notice that the polynomial multiplication is computed before the decoding even starts. This makes our attack independent of the choice of the concatenated code $\mathcal{C}$.

Also, it allows us to focus on the side-channel leakage during the first operations that manipulate the private key bits in the decryption process.

Algorithm 3 The decryption algorithm of the HQC PKE.

```
Require: private key $sk = (x, y)$, ciphertext $c = (u, v)$
Ensure: plaintext $m$
1: $m = \mathcal{C}.\texttt{Decode}(v - \boxed{u \cdot y})$
2: return $m$
```

2.2 Attacked Target Implementation

As side-channel attacks are in general device dependent and their validation is greatly affected by the hardware and software used, in this subsection we will describe our choice of both for our experiments.

Software The authors of HQC submitted several implementations of this algorithm to the NIST PQC competition. In October 2022 they added a pure constant-time (not optimized) implementation in C referred as "additional implementation". From the implementation of the whole HQC cryptosystem [HQCb], we extracted our target `gf2x.c` which implements the multiplication of two polynomials, together with the necessary dependency files. Note that the additional implementation remained the same also in the latest updated submission package for round 4 [HQCc].

The multiplication is implemented by the Karatsuba algorithm followed by a modular reduction of the resulting polynomial. Once the input into recursive calls of Karatsuba reaches a size of only 64 bits, the implementation computes the algorithm `mul1` from [BGTZ08] in the function called `base_mul`.

The high-level description of `base_mul` multiplication for polynomials over GF(2) is as follows. Each polynomial such as $a = \sum_{i=0}^{63} a_i X^i$ is stored as a sequence of its coefficients $(a_{63}, a_{62}, \ldots, a_0) = a_{63..0}$, and $b = b_{63..0}$, resulting in a 128-bit polynomial $c = a \cdot b = c_{127..0}$.

Important to note is that the polynomial multiplication function takes the private key as its first operand and the ciphertext as its second operand. This holds true also for the call of the `base_mul` function. Therefore, in the rest of the paper we focus on extracting the bits from the first input operand a, which in fact stores the bits of the private key.

The first step (see Listing 1) is precomputing the table u of sixteen 64-bit words with the partial products of the lower 60 bits of b with all possible 4-bit polynomials (lines 8–13). The first partial product $g = a_{3..0} \cdot b_{59..0}$ is prepared using the table u (lines 14–19), which in effect is $g = u[a_{3..0}]$. This way, the lower 64-bit word l of the result is obtained, and the higher word h is zeroed (lines 20–21).

Step 2: The other bits, $a_{63..4}$ are scanned (in increments of 4), and each partial product $g = u[a_{i+3..i}] = a_{i+3..i} \cdot b_{59..0}$ is computed (lines 24–29). Then it is shifted and added by xoring at the appropriate position in the pair of low and high words l, h (lines 30–31). By the end of Step 2, the intermediate product is $(h, l) = a \cdot b_{59..0}$.

Step 3: The remaining top 4 bits of b are now multiplied bit by bit with a by logical "and" masking and added to the result at the top 4 positions (lines 34–45). The result is $c_{127..0} = (h,l) = a \cdot b$.

In the additional implementation, the algorithm `mul1` from [BGTZ08] is adapted to be time-constant and resistant to cache timing attacks. To protect against cache timing attacks, each time a tabulated pre-calculated value needs to be used, *all* pre-calculated values are accessed so that the attacker cannot determine which value was accessed by cache measurement. This protection measure corresponds to the for loops in lines 16-19 and 26-29.

The processing of the a operand described above is reflected in the power consumption dependency on the values of individual bits of a. This will be visible in the heat map we present in Figure 3 as part of our attack description below.

```
1  void base_mul(uint64_t *c, uint64_t a, uint64_t b) {
2     uint64_t h = 0;
3     uint64_t l = 0;
4     uint64_t g;
5     uint64_t u[16] = {0};
6     uint64_t mask_tab[4] = {0};
7     // Step 1
8     u[0] = 0;
9     u[1] = b & ((1ULL << (64 - 4)) - 1ULL);
10    for(int i=2;i<16;i+=2){
11     u[i] = u[i/2] << 1;
12     u[i+1] = u[i] ^ u[1];
13    }
14    g=0;
15    uint64_t tmp1 = a & 15;
16    for(int i = 0; i < 16; i++) {
17     uint64_t tmp2 = tmp1 - i;
18     g ^= (u[i] & -(1 - ((tmp2 | -tmp2) >> 63)));
19    }
20    l = g;
21    h = 0;
22    // Step 2
23    for (uint8_t i = 4; i < 64; i += 4) {
24       g = 0;
25       uint64_t tmp1 = (a >> i) & 15;
26       for (int j = 0; j < 16; ++j) {
27         uint64_t tmp2 = tmp1 - j;
28         g ^= (u[j] & -(1 - ((tmp2 | -tmp2) >> 63)));
29       }
30       l ^= g << i;
31       h ^= g >> (64 - i);
32    }
33    // Step 3
34    mask_tab [0] = - ((b >> 60) & 1);
35    mask_tab [1] = - ((b >> 61) & 1);
36    mask_tab [2] = - ((b >> 62) & 1);
37    mask_tab [3] = - ((b >> 63) & 1);
38    l ^= ((a << 60) & mask_tab[0]);
39    h ^= ((a >> 4) & mask_tab[0]);
40    l ^= ((a << 61) & mask_tab[1]);
41    h ^= ((a >> 3) & mask_tab[1]);
42    l ^= ((a << 62) & mask_tab[2]);
43    h ^= ((a >> 2) & mask_tab[2]);
44    l ^= ((a << 63) & mask_tab[3]);
45    h ^= ((a >> 1) & mask_tab[3]);
46    c[0] = l;
47    c[1] = h;
}
```

Listing 1: The `base_mul` function from `gf2x.c` from the "additional implementation" in the HQC submission packages (rounds 3 and 4)

The objective of the proposed attack is to recover the bits of the polynomial a while it is being used in the aforementioned `base_mul` function. The proposed countermeasures are based on a modification of the core of the `base_mul` function.

Target hardware and measurement platform Target hardware was a 32-bit STM32F303 RBT6 Arm Cortex-M4 microcontroller with 128 KB Flash and 40 KB SRAM. The target is integrated in a single-board version of NewAE ChipWhisperer-Lite connected through a micro USB cable to a standard PC workstation. A simplified schematic is depicted in Figure 1. For capturing the traces, we used the in-built capturing solution (oscilloscope that is integrated into ChipWhisperer-Lite) synchronized with the target with a limited trace length of around 25k samples per trace.

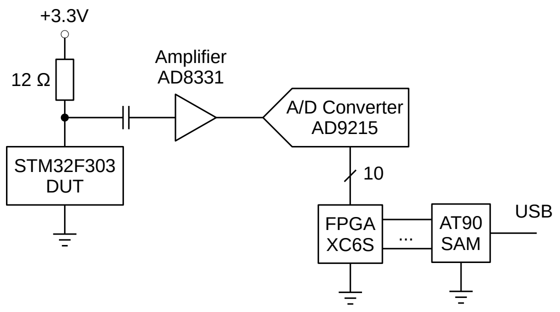


Fig. 1: Simplified block diagram of the ChipWhisperer-Lite power side channel analysis platform. The Device Under Test (DUT) is the microcontroller that runs the attacked cryptosystem. The measured signal is filtered, amplified, sampled and converted by the integrated capture circuitry of ChipWhisperer-Lite. It is then transferred over USB to the PC (not depicted)

We capture the power consumption corresponding to the beginning of the polynomial multiplication involved in decryption of the ciphertext with the private key. In the available trace length, we observe samples influenced by the first approx. 100 bits of the private key. This limitation could be overcome by using an external oscilloscope or capturing device with a streaming mode for long captures like CW1200 ChipWhisperer-Pro or ChipWhisperer-Husky.

Due to the small SRAM memory of the target microcontroller, the parameter n defining the length of the private key had to be reduced for the experiment in this paper and the original size proposed in the NIST submission (for any level of security) unfortunately could not be used.

2.3 Overview of Attack Algorithms

In the side-channel analysis presented in this paper, we used two generic techniques: the Welch's ANOVA test and the multilayer perceptron machine-learning alorithm with grid search hyperparameter tuning.

Welch's ANOVA Test This statistical test was introduced in 1951 by Bernard Lewis Welch in [Wel51]. The test was used to decide whether two or more population means are equal. Compared to the classical analysis of the variance (as it is done in the ANOVA test), the Welch's ANOVA test does not assume equal variances, but it still requires normal distributions of the tested populations and independence of individual samples.

Later, the Welch's ANOVA test started to be used also as a tool for side-channel leakage assessment [YJ21]. The idea is to divide side-channel samples into multiple classes and test the difference among these classes using the test. The leakage points and points which do not leak can be distinguished by determining whether the null hypothesis is accepted or not. The leakage points selected by the tests are commonly called Points of Interest (POIs).

For implementation of the Welch's ANOVA test, we used the `welch_anova()` function from the Pingouin library [Val18] written for the Python programming language.

MLP Multilayer perceptron (MLP) is a simple supervised feed-froward neural network with an input layer, one or more hidden layers and an output layer. We have chosen this model since it is commonly used for classification tasks.

In this paragraph we briefly describe MLP and provide some details about our training. For more information about MLP and backpropagation we refer to the Pattern Recognition and Machine Learning by Christopher Bishop [Bis06].

The i-th output of neuron y_i from a hidden layer in MLP can be described as follows: let $x_1, \dots x_n$ be outputs from the previous layer, let $w_1, \dots w_n$ be weights, β_i is a bias and f is an activation function. Then

$$y_i = f\left(\left(\sum_{j=1}^{n} x_j \cdot w_j\right) + \beta_i\right),$$

where f is applied to every neuron in the hidden layer. We use two types of activation functions, ReLU and tanh, which are defined as follows:

$$\begin{aligned} \mathrm{ReLU}(x) &= \begin{cases} x & \text{if x} > 0 \\ 0 & \text{otherwise,} \end{cases} \\ \tanh(x) &= \frac{e^x - e^{-x}}{e^x + e^{-x}}. \end{aligned}$$

Our input layer consists of POIs of one trace. The output layer is a single neuron with the sigmoid activation function that performs classification of the chosen bit of the key. The sigmoid function is defined as follows:

$$\mathrm{sigmoid}(x) = \frac{1}{1 + e^{-x}}.$$

Sometimes we also add a batch normalization layer that normalizes a mini-batch of samples. The batch normalization is described in the paper [IS15] by S. Ioffe and Ch. Szegedy.

The goal of model training is to obtain parameters, weights and biases which minimize the loss function. We use the binary cross-entropy loss function and we evaluate the result using the binary accuracy metric. The corresponding functions are described in The Basics of Machine Learning [Cer23].

The training algorithm automatically minimizes the loss function and finds parameters using backpropagation. We use the AdamW optimizer algorithm for the training. AdamW was introduced in 2017 as a modification of the Adam optimizer [LH17].

For the implementation of MLP model and for training we used the Keras [C$^+$15] and PyTorch [AYH$^+$24] libraries written in Python.

Hyperparameter Tuning and Grid Search Hyperparameters are parameters that cannot be trained and they have to be set manually, e.g., the number of neurons in one layer. Searching for hyperparameters which optimize the training procedure and give us the best model can be automated. Grid search exhaustively searches through all chosen values of hyperparameters, trains a model for every setting and returns the model with the best accuracy on validation data.

We combined the grid search with an early stopping. The early stopping is used to stop the training when the loss on the validation data stops improving. We used EarlyStopping from Keras library. We implemented grid search and early stopping using the Keras [C$^+$15], KerasTuner [OBL$^+$19] and PyTorch [AYH$^+$24] libraries.

3 Profiling SCA on Polynomial Multiplication in HQC Multiplication

In this section, we describe our attack and experimental results against polynomial multiplication in HQC decryption implemented by `gf2x.c` function in the additional implementation submitted to the NIST PQC competition. It is a side-channel attack with the assumption of the attacker's ability to get access to the device and measure side-channel leakage physically – in our case, power consumption of the device while running the target operation. We also use a profiling phase in our attack scheme, which assumes the ability of the attacker to have the same kind of device in hand together with the ability to measure the side-channel leakage of the target operation with known inputs (including private keys) on this device.

Due to the limitations of the utilised hardware, we present in this paper experimental validation of our attack scheme only on the HQC polynomial multiplication with reduced parameter $n = 1234$, which is shorter than the size of the private key $n = 17\,669$ for level 1 according to NIST security categories, and would make the cryptosystem practically insecure against classical cryptanalysis. We argue that the shortening does not conflict with the main idea of exploiting power leakage from the implemented algorithm to gain knowledge about the private key, since we did not take any advantage of the shortening of the private key in our attack scheme and it does not alter the implementation in any significant way other than in number of recursive calls of the Karatsuba algorithm.

3.1 Attack Scheme

For implementing the attack we measured three datasets: 40 000 traces for training, 10 000 traces for validation in hyperparameter tuning, and 20 000 traces for testing.

For each bit of the private key, we selected POIs using Welch's ANOVA test on the training dataset, trained machine-learning multilayer perceptron model with grid search hyperparameter tuning using POIs from the training and validation datasets, and then tested the best model on 20 000 traces from the testing dataset.

Note that following this scheme we get one multilayer perceptron model for each bit of the private key.

The POI selection is done by dividing all training traces into two groups according to the possible values of the key-bit 0 or 1. For each sample of traces (where each trace consists of around 24 000 samples), we run the Welch's ANOVA test and store the resulting F-statistics. We select 480 points with the highest F-statistics and those are the POIs (features) used for machine-learning training, validation and testing.

The grid-search for MLP hyperparameter tuning was run through the following parameters: First layer: 512 or 1024 nodes, second layer: 128 or 256 nodes, activation function: ReLU or tanh, batch size: 200, 500 or 800. The last parameter run through 3 selected ways how to construct MLP: two layers with the same chosen activation function and batch normalization between them, two layers with the same chosen activation function but without batch normalization, and MLP just from one layer.

3.2 Attack Complexity

For training, validation and testing of MLP models, we used hardware with the following parameters: Memory – 96 GiB; Processor – Intel® Xeon(R) Gold 6136 CPU @ 3.00GHz × 48; Disk Capacity – 3TB; OS – Ubuntu 22.04.5 LTS. Note that we did not use hardware with a GPU.

The most time-consuming part of the attack scheme is the grid search for the MLP hyperparameter tuning, which uses 40 000 samples for training and independent 10 000 samples for validation, both with 480 features (i.e. chosen POIs). If 24 cores (as a parameter for the PyTorch library) are used with the hardware specified above, we get the following times:

For one bit of the private key, it takes 4 234 seconds (1h 10min 34s) to do the grid search hyperparameter tuning. For all the bits of the private key with parameter $N = 17\,669$, that would be 74 810 546 seconds (865d 20h 42min 26s).

Such an effort would probably not be justifiable by an actual attacker, so further optimizations of the attack would need to occur despite achieving slightly worse results in total. Such optimizations could be to 1) avoid tuning MLP models for each bit and instead re-use the hyperparameters trained on a smaller subset of selected bits for the others or 2) train the MLP models on several bits at once.

Our results from the grid search show that from the first 25 bits, almost all winning models had the following structure: two hidden layers with ReLU activation function with 512 and 128 nodes, respectively, with batch normalization layer between them and batch size 200. The only exceptions have been winning models for the 4th and 21st bit, where for the 4th bit, better results had a model with 256 nodes in the second layer and batch size 800; for the 21st bit, the only difference was in batch size to be 500. As we described earlier, in all cases, the output layer consists of 1 node with a sigmoid activation function.

To see how difficult the classification problem we are facing in terms of SCA, we provide Signal-to-Noise Ratio (SNR) [Man04] for our data in Table 2. More specifically, a maximum, mean and min of SNR from 480 POIs for each bit.

private key bit	0	1	2	3	4	5	6	7	8	9	10	11
max SNR	12.38	10.93	11.04	21.91	9.70	8.60	10.83	14.80	9.06	6.89	9.97	14.32
mean SNR	0.49	0.47	0.54	0.97	0.88	0.85	0.87	1.18	0.83	0.80	0.83	1.16
min SNR	0.02	0.03	0.02	0.02	0.14	0.08	0.08	0.11	0.13	0.07	0.10	0.11

private key bit	12	13	14	15	16	17	18	19	20	21	22	23
max SNR	9.17	7.02	10.32	15.18	9.00	7.00	10.21	14.61	9.04	6.68	10.32	14.46
mean SNR	0.86	0.80	0.83	1.14	0.87	0.81	0.84	1.15	0.84	0.79	0.83	1.13
min SNR	0.14	0.07	0.10	0.11	0.08	0.07	0.10	0.08	0.13	0.07	0.07	0.09

Table 2: Max, mean and min of SNR from POIs for individual bits of the private key (restricted just to the first 24 bits as an example).

3.3 Attack Results

Trained models after hyper-parameter tuning are tested on 20 000 samples, each with 480 features (selected POIs), corresponding to 20 000 independently measured traces, each with unique random ciphertext and private key. In Figure 2, we see each model's number of misclassifications on the corresponding key-bit it was trained for. From 20 000 traces, trained models made very few wrong guesses for the key-bit. Note that each testing trace is independent (with a unique random private key), and we do not exploit information from other traces from the testing dataset to find the key bits that correspond to the trace we test at the moment. Therefore, we are in the single-trace attack scenario. That is important for us because HQC, as other KEMs from NIST PQC competition, is expected to be used in an ephemeral way so that for each shared key which results from the KEM, a new pair of private and public keys is generated and used. When HQC is used in an ephemeral way, the attacker can effectively run only single-trace attacks.

Taking in mind that the length of the private key according to the NIST submission parameters would in practice be 17 669, 35 851, or 57 637, depending on the security level, for full recovery of the private key it would be necessary to do some further processing using classical cryptanalytical techniques or brute-force through residual entropy in the private key.

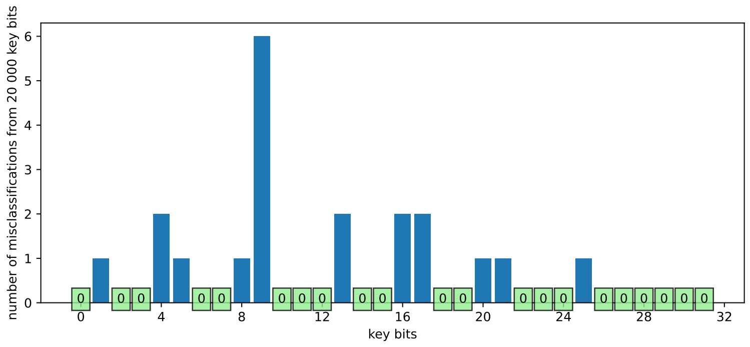


Fig. 2: Results of the attack phase for recovering the individual bits from the private key. The height of the columns corresponds to the misclassifications of individual bits. Most of the bits have been recovered with perfect accuracy. The worst result was for the 9th bit with 6 errors of 20 000, i.e. with the accuracy of 0.9997.

4 Proposed Countermeasures

In this section, we present and evaluate two countermeasures implemented at the `base_mul` level. The first one, presented in Section 4.1, exploits the windows technique used in Brent et al. [BGTZ08] to introduce some randomness but can protect only one out of three bits. The second countermeasure corresponds to a classical masking technique [GP99,ISW03]. Due to the high linearity of the polynomial multiplication over GF(2), Boolean masking presented in Section 4.2 is highly competitive.

Both countermeasures are based on Boolean secret sharing. The idea of such countermeasure is to split the sensitive value s into several shares s_i such that $\bigoplus_i s_i = s$.

4.1 Windowing the Self-Unmasking

The idea of this countermeasure is to exploit the linearity of polynomial multiplication and the windows technique. Instead of scanning the bits of the first input a in groups of 4 bits independently through the loop described in Line 23 of Listing 1, we use Line 28 of Listing 2, i.e. we overlap the scanning of one bit. If we look at two consecutive iterations of the loop, we can see that the value of the MSB of the ith iteration $MSB(a,i) = a[i \times 3 + 3]$ and the LSB of the $(i+1)$th iteration $LSB(a,i+1) = a[(i+1) \times 3]$ are the same value. So we can mask only this bit, and thus unmask on the fly.

Example 1. With the overlapping window, we scan the bits of a by groups of 4 bits but shift the windows of three. For $a = a_0 + a_1X + a_2X^2 + a_3X^3 + a_4X^4 + a_5X^5 + a_6X^6 + a_7X^7$, we first consider $a_0 + a_1X + a_2X^2 + a_3X^3$ and next $a_3X^3 + a_4X^4 + a_5X^5 + a_6X^6$, which means by using bit operation we can choose to perform the multiplication by a_3X^3 either in the first iteration or the second iteration.

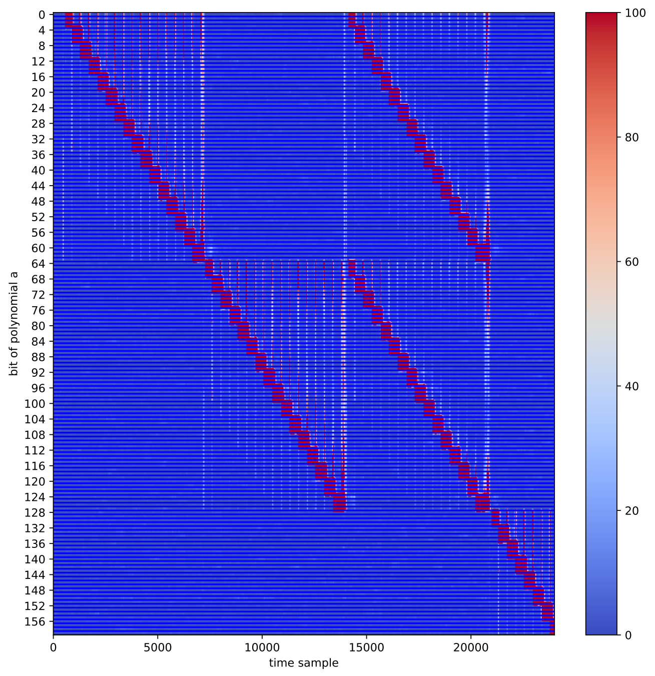


Fig. 3: Heat map of F-statistics from the Welch's ANOVA test for the first 160 bits of the "additional implementation" polynomial multiplication. Three whole calls to `base_mul` are visible; Dependence on $a_{63..0}$, then $a_{127..64}$, and then $a_{63..0} + a_{127..64}$ corresponds to the base case of a recursive Karatsuba multiplication.

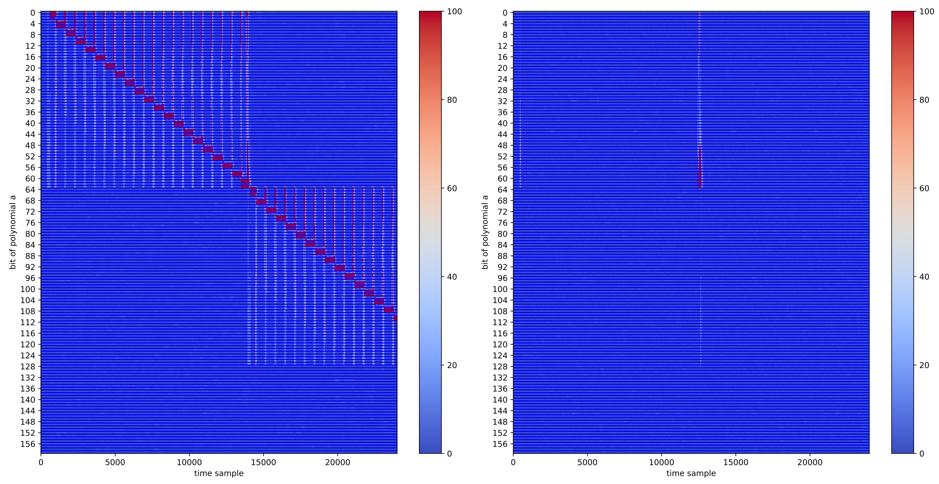
```
void base_mul_prot(uint64_t *c, uint64_t a, uint64_t b) {
   uint64_t h = 0;
   uint64_t l = 0;
   uint64_t g;
   uint64_t u[16] = {0};
   uint64_t mask_tab[4] = {0};
   // Step 1
   u[0] = 0;
   u[1] = b & ((1ULL << (64 - 4)) - 1ULL);
   for(int i=2;i<16;i+=2){
    u[i] = u[i/2] << 1;
    u[i+1] = u[i] ^ u[1];
   }
   g=0;
   uint64_t tmp1 = a & 15;
   uint64_t at=a;
   uint64_t bit=rand()%2;
   tmp1^= (bit<<3);
   for(int i = 0; i < 16; i++) {
    uint64_t tmp2 = tmp1 - i;
    g ^= (u[i] & -(1 - ((tmp2 | -tmp2) >> 63)));
   }
   at&=((-1ull)^(1ull<<(3)));
   at|=(bit<<(3));
   l = g;
   h = 0;
   // Step 2
   for (uint8_t i = 3; i < 60; i += 3) {
      g = 0;
      uint64_t bit=rand()%2;
      uint64_t tmp1 = (at >> i) & 15 ;
      tmp1^= (bit<<3);
      for (int j = 0; j < 16; ++j) {
        uint64_t tmp2 = tmp1 - j;
        g ^= (u[j] & -(1 - ((tmp2 | -tmp2) >> 63)));
      }
      at&=((-1ull)^(1ull<<(i+3)));
      at|=(bit<<(i+3));
      l ^= g << i;
      h ^= g >> (64 - i);
   }
   uint8_t i=60;
   g = 0;
   tmp1 = (at >> i) & 15 ;
   for (int j = 0; j < 16; ++j) {
    uint64_t tmp2 = tmp1 - j;
    g ^= (u[j] & -(1 - ((tmp2 | -tmp2) >> 63)));
  }
  l ^= g << i;
  h ^= g >> (64 - i);
  // Step 3
  mask_tab [0] = - ((b >> 60) & 1);
  mask_tab [1] = - ((b >> 61) & 1);
  mask_tab [2] = - ((b >> 62) & 1);
  mask_tab [3] = - ((b >> 63) & 1);
  l ^= ((a << 60) & mask_tab[0]);
  h ^= ((a >> 4) & mask_tab[0]);
  l ^= ((a << 61) & mask_tab[1]);
  h ^= ((a >> 3) & mask_tab[1]);
  l ^= ((a << 62) & mask_tab[2]);
  h ^= ((a >> 2) & mask_tab[2]);
  l ^= ((a << 63) & mask_tab[3]);
  h ^= ((a >> 1) & mask_tab[3]);
  c[0] = l;
  c[1] = h;
}
```

Listing 2: Windowing the self-unmasking implementation of `base_mul`.

The introduction of this low entropy masking helps us to reduce the effectiveness of the attack, as shown by the heat map in Figure 4a. Compared with the heat map of the additional implementation, we can notice that instead of having a vertical block of 4 bits, we have a block of 2 bits plus a third one with less information. The two effects can be explained by the fact that 2 bits remain unmasked so it is natural to have information about them, the third one is masked so we have no information, and this third bit appears in two consecutive multiplications, once as the *MSB* and then as the *LSB*.

We can still see the thin vertical line for each bit before each computation, confirming that this leakage is due to the selection of the window of a.

It is obvious that the execution time of a single base multiplication is considerably longer than that of the additional implementation. In the additional implementation, this corresponds to 6 500 points in the trace, while with the window self-unmasking, the multiplication corresponds to slightly less than 14 000 points. The observed increase in execution time is unexpected given the number of iterations of the loop, which is



(a) Self-unmasking `base_mul_prot` function

(b) Masked implementation with 2 shares, `base_mul_mask`

Fig. 4: Heat map of F-statistics from the Welch's ANOVA test for the first 160 bits of the two variants of the protected implementation of `base_mul`.

15 in the additional implementation and 20 in the proposed approach. A 33% increase was anticipated. However, the proposed approach involves a significant amount of bit manipulation of 64-bit words in a 32-bit microcontroller, which results in a considerable increase in the number of instructions.

4.2 Full Mask Implementation

In this section, we evaluate the security of the classical Boolean masking at the `base_mul` level. The inputs, designated as a and b, are passed to the function unmasked. In our case study, only the value a is sensitive; consequently, we only mask this value. This is illustrated by Lines 10-11 in Listing 3. Subsequently, a random mask, designated as *mask*, is generated and xored with the sensitive value, $a = a \oplus mask$. The subsequent step involves the exploitation of the linearity of polynomial multiplication to perform operations on the two shares independently. This is then followed by the unmasking of the result, as illustrated in lines 77 and 78.

Of course, since the inputs are unmasked, we expect some leakage, but we wanted to evaluate if some security could be achieved even in a lazy engineering way by a first order masking with only two shares for long computation in between the shares, this may indicate that the results of [BGG+14] are pessimistic.

```
1  void base_mul_mask(uint64_t *c, uint64_t a, uint64_t b) {
2     uint64_t h = 0;
3     uint64_t l = 0;
4     uint64_t hm = 0;
```

```
    uint64_t lm = 0;
    uint64_t g;
    uint64_t u[16] = {0};
    uint64_t mask_tab[4] = {0};
    uint64_t mask = rand();
    mask=(mask*(1ull<<31)+rand())*(1ull<<33)+(rand()%4);
    a=a^mask;
    // Step 1
    u[0] = 0;
    u[1] = b & ((1ULL << (64 - 4)) - 1ULL);
    for(int i=2;i<16;i+=2){
     u[i] = u[i/2] << 1;
     u[i+1] = u[i] ^ u[1];
    }
     g=0;
    uint64_t tmp1 = (a) & 15;
    for(int i = 0; i < 16; i++) {
     uint64_t tmp2 = tmp1 - i;
     g ^= (u[i] & -(1 - ((tmp2 | -tmp2) >> 63)));
    }
    l = g;
    h = 0;
    g=0;
    tmp1 = (mask) & 15;
    for(int i = 0; i < 16; i++) {
     uint64_t tmp2 = tmp1 - i;
     g ^= (u[i] & -(1 - ((tmp2 | -tmp2) >> 63)));
    }
    lm = g;
    hm = 0;
    // Step 2
    for (uint8_t i = 4; i < 64; i += 4) {
       g = 0;
       uint64_t tmp1 = ((a) >> i) & 15;
       for (int j = 0; j < 16; ++j) {
         uint64_t tmp2 = tmp1 - j;
         g ^= (u[j] & -(1 - ((tmp2 | -tmp2) >> 63)));
       }
       l ^= g << i;
       h ^= g >> (64 - i);
    }
    for (uint8_t i = 4; i < 64; i += 4) {
       g = 0;
       uint64_t tmp1 = ((mask) >> i) & 15;
       for (int j = 0; j < 16; ++j) {
         uint64_t tmp2 = tmp1 - j;
         g ^= (u[j] & -(1 - ((tmp2 | -tmp2) >> 63)));
       }
       l ^= g << i;
       h ^= g >> (64 - i);
    }
    // Step 3
    mask_tab [0] = - ((b >> 60) & 1);
    mask_tab [1] = - ((b >> 61) & 1);
    mask_tab [2] = - ((b >> 62) & 1);
    mask_tab [3] = - ((b >> 63) & 1);
    l ^= (((a) << 60) & mask_tab[0]);
    h ^= (((a) >> 4) & mask_tab[0]);
    l ^= (((a) << 61) & mask_tab[1]);
    h ^= (((a) >> 3) & mask_tab[1]);
    l ^= (((a) << 62) & mask_tab[2]);
    h ^= (((a) >> 2) & mask_tab[2]);
    l ^= (((a) << 63) & mask_tab[3]);
    h ^= (((a) >> 1) & mask_tab[3]);
    lm ^= (((mask) << 60) & mask_tab[0]);
    hm ^= (((mask) >> 4) & mask_tab[0]);
    lm ^= (((mask) << 61) & mask_tab[1]);
    hm ^= (((mask) >> 3) & mask_tab[1]);
    lm ^= (((mask) << 62) & mask_tab[2]);
    hm ^= (((mask) >> 2) & mask_tab[2]);
    lm ^= (((mask) << 63) & mask_tab[3]);
    hm ^= (((mask) >> 1) & mask_tab[3]);
    c[0] = l^lm;
    c[1] = h^hm;
}
```

Listing 3: Masked implementation of `base_mul`.

As demonstrated by the heat map in Figure 4b, the computation demonstrates minimal leakage of the value of a. However, the emergence of such leakage is discernible when computing the final two lines, specifically in the unmasking segment. Additionally, a minor leakage was initially observed during the execution of the computation, which is anticipated given that the data loaded into the function is unprotected. It is also worth mentioning that the highest and lowest bits of a are more susceptible to leakage. This phenomenon can be attributed to the fact that, in the context of multiplication, the output monomials of the highest and lowest degrees are more dependent on a single bit, as a result of the inherent nature of the multiplication operation. Furthermore, as we are utilizing sparse polynomials, this may be valid for several monomials.

It is somewhat unexpected that the computation does not exhibit significant leakage when manipulating the share. This suggests that in the case of long linear operations, where shares are manipulated that are longer than the word considered in the result of reference [BGG$^+$14], the aforementioned security estimate may be overly pessimistic.

It can be observed that the computation time of the full masked implementation is comparable to that of the windowing self-unmasked implementation and is approximately twice that of the unprotected `base_mul`. This is to be expected, given that the operations are GF(2)-linear and Boolean secret sharing is employed.

4.3 Comparison of Implementations

We compare the three different implementations in terms of efficiency and security improvement for the attack. Because of the hardware memory limitations (see Section 2.2), we did not perform the attack on all bits of the private key, but we restricted our practical evaluation to the first 25 bits. In Table 3, we provide the mean of bit error ε_{bit} for this subset and compute extrapolated values of naive complexity [ZBHV21] and success rate for $n = 1234$ and $n = 17\,669$. The success rate is calculated simply as $(1-\varepsilon_{bit})^n$, which gives us a probability of perfect recovery of the private key from a single trace. For estimating the remaining entropy for the attacker to brute-force potentially wrongly predicted bits, we use naive complexity $\mathcal{C}_{NC}$ computed as

$$\mathcal{C}_{NC}(n, \varepsilon_{bit}) = \log_2 \left(\sum_{i=0}^{\lceil n\cdot\varepsilon_{bit} \rceil} \binom{n}{i} \right).$$

We can see that for NIST security level 1 with parameter $n = 17\,669$ we get 51.1% probability for the additional implementation with `base_mul` from the NIST submission. The naive complexity is only 14.11, which shows that the key recovery attack is highly practical. For the proposed mitigation described in Section 4.1, we get the naive complexity of 80.86, putting the attack on the border of feasibility. For the full mask implementation described in Section 4.2, we get the naive complexity of 337.69, making this particular SCA attack worse than the state-of-the-art cryptanalytic attacks on code-based cryptosystems.

	Additional implementation	Self-unmasking	Full-mask
# clock cycles	6.5k	14k	14k
random bits	0	20	64
Max F-statistic	198 607	41 226	4 381
Mean bit-error ε_{bit}	3.8e-05	0.008	0.048
Naive complexity ($n = 1234$)	10.27	80.86	337.69
Success rate ($n = 1234$)	0.954	4.5e-05	8.22e-27
Naive complexity ($n = 17\,669$)	14.11	1194.30	4868.05
Success rate ($n = 17\,669$)	0.511	7.4e-63	3.2e-374

Table 3: Comparison of the different approaches

5 Conclusion

In this paper, we presented a new attack on HQC post-quantum cryptosystem targeting only polynomial multiplication used in the decryption operation. We have shown and verified that a significant part of the private key can be extracted using a power side-channel attack on the Karatsuba algorithm with `base_mul` function. This function is used by the additional implementation of HQC in the NIST submission, a pure C constant-time implementation of the cryptosystem provided by the submitters into the NIST PQC competition. This implementation is of high relevance because it is imported also by the PQClean library [KSSW22,PQC] from the Open Quantum Safe (OQS) project, and the algorithm itself is used by BouncyCastle libraries for Java [Boub] and C# [Boua].

The proposed attack is the first attack targeting the polynomial multiplication in the beginning of the HQC decryption and the first single-trace SCA attack against HQC cryptosystem which targets the private key. In our attack, we assume that the attacker can acquire power traces during the profiling/training phase on the same or similar device. During the attacking/testing phase we use only one trace for the private key extraction. We used the Welch's ANOVA test for POI selection and multi-layer perceptron models for classification of the private-key bits.

We presented experimental results of the attack, which has been realized on the ChipWhisperer-Lite platform featuring the 32-bit STM32F303RBT6 Arm Cortex-M4 target processor. For the practical evaluation, we used 40 000 traces for training, 10 000 traces for validation and independent 20 000 traces for the attack itself, with reduced size of the private key because of the hardware limitations. We used a unique random ciphertext and private key for each trace with the intention to get as close to the real scenario as possible. The results show that we can recover the private key with 51.1%

probability for NIST security level 1, threatening the security of the cryptosystem if such implementation is used in practice to protect sensitive data.

We also implemented two possible mitigations with different performance and randomness costs and evaluated our attack on them. Our results show that both mitigations protect the private key to a sufficient level, making the attack unfeasible in practice.

5.1 Future Work

There are several directions for future work that we consider beneficial. The first is an evaluation of the attack scheme on a more powerful processor that would be able to run HQC with full parameters. The second is to implement more machine learning methods and classical template attacks with a comparison of their effectiveness. The third would be a more granular study of specific assembler operations inside Karatsuba and `base_mul` functions with respect to the compiling process and target processor together with evaluation of their differences in leakage and possible more precise targeting of these operations during the attack. The fourth is relaxing the attacker's abilities by exploring distant electromagnetic leakage in contrast to power analysis.

Acknowledgements This work was supported by the Grant Agency of the Czech Technical University in Prague, grant No. SGS23/211/OHK3/3T/18 funded by the MEYS of the Czech Republic, the OP VVV MEYS funded project CZ.02.1.01/0.0/0.0/16 019/ 0000765 "Research Center for Informatics", and also by the Project Barrande No. 8J23FR012, funded by the MEYS of the Czech Republic. This work received funding from the France 2030 program, managed by the French National Research Agency under grant agreement No. ANR-22-PETQ-0008 PQ-TLS

References

AAC+22. Gorjan Alagic, Daniel Apon, David Cooper, Quynh Dang, Thinh Dang, John Kelsey, Jacob Lichtinger, Yi-Kai Liu, Carl Miller, Dustin Moody, Rene Peralta, Ray Perlner, Angela Robinson, and Daniel Smith-Tone. Status report on the third round of the NIST post-quantum cryptography standardization process. Technical report, US Department of Commerce, National Institute of Standards and Technology, 2022.

AYH+24. Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, CK Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Michael Suo, Phil Tillet, Eikan Wang, Xiaodong Wang, William Wen, Shunting Zhang, Xu Zhao, Keren Zhou, Richard Zou, Ajit Mathews, Gregory Chanan, Peng Wu, and Soumith Chintala. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24)*. ACM, April 2024.

BGG$^+$14. Josep Balasch, Benedikt Gierlichs, Vincent Grosso, Oscar Reparaz, and François-Xavier Standaert. On the cost of lazy engineering for masked software implementations. In Marc Joye and Amir Moradi, editors, *Smart Card Research and Advanced Applications - 13th International Conference, CARDIS 2014, Paris, France, November 5-7, 2014. Revised Selected Papers*, volume 8968 of *Lecture Notes in Computer Science*, pages 64–81. Springer, 2014.

BGTZ08. Richard P Brent, Pierrick Gaudry, Emmanuel Thomé, and Paul Zimmermann. Faster multiplication in GF(2)[x]. In *Algorithmic Number Theory: 8th International Symposium, ANTS-VIII Banff, Canada, May 17-22, 2008 Proceedings 8*, pages 153–166. Springer, 2008.

Bis06. Christopher Bishop. *Pattern Recognition and Machine Learning*. Springer, January 2006.

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

Boua. Bouncy Castle. Bouncy Castle – C#, HQC implementation, GF2PolynomialCalculator. `https://github.com/bcgit/bc-csharp/blob/master/crypto/src/pqc/crypto/hqc/GF2PolynomialCalculator.cs`. Accessed: 2024-12-17.

Boub. Bouncy Castle. Bouncy Castle – Java, HQC implementation, GF2PolynomialCalculator. `https://github.com/bcgit/bc-java/blob/main/core/src/main/java/org/bouncycastle/pqc/crypto/hqc/GF2PolynomialCalculator.java`. Accessed: 2024-12-17.

C$^+$15. François Chollet et al. Keras. `https://keras.io`, 2015.

CD23. Wouter Castryck and Thomas Decru. An efficient key recovery attack on SIDH. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 423–447. Springer, 2023.

Cer23. Giovanni Cerulli. *The Basics of Machine Learning*, pages 30–33. Springer International Publishing, Cham, 2023.

DG24. Haiyue Dong and Qian Guo. OT-PCA: New key-recovery plaintext-checking oracle based side-channel attacks on HQC with offline templates. *Cryptology ePrint Archive*, 2024.

GLG22. 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*, pages 353–371. Springer, 2022.

GMGL24. Guillaume Goy, Julien Maillard, Philippe Gaborit, and Antoine Loiseau. Single trace HQC shared key recovery with SASCA. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024(2):64–87, 2024.

GP99. Louis Goubin and Jacques Patarin. DES and differential power analysis (the ”duplication” method). In Çetin Kaya Koç and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems, First International Workshop, CHES’99, Worcester, MA, USA, August 12-13, 1999, Proceedings*, volume 1717 of *Lecture Notes in Computer Science*, pages 158–172. Springer, 1999.

HQCa. HQC authors. HQC specification. `https://pqc-hqc.org/doc/hqc-specification_2025-02-19.pdf`. Accessed: 2025-3-8.

HQCb. HQC Authors. Submission of HQC – round 3 – with additional implementation. `https://pqc-hqc.org/doc/hqc-submission_2023-04-30.zip`. Accessed: 2024-2-27.

HQCc. HQC Authors. Submission of HQC – round 4 – latest – February 2025. `https://pqc-hqc.org/doc/hqc-submission_2025-02-19.zip`. Accessed: 2024-2-27.

HSC$^+$23. Senyang Huang, Rui Qi Sim, Chitchanok Chuengsatiansup, Qian Guo, and Thomas Johansson. Cache-timing attack against HQC. *Cryptology ePrint Archive*, 2023.

IS15. Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

ISW03. Yuval Ishai, Amit Sahai, and David A. Wagner. Private circuits: Securing hardware against probing attacks. In Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, volume 2729 of *Lecture Notes in Computer Science*, pages 463–481. Springer, 2003.

KSSW22. Matthias J. Kannwischer, Peter Schwabe, Douglas Stebila, and Thom Wiggers. Improving software quality in cryptography standardization projects. In *IEEE European Symposium on Security and Privacy, EuroS&P 2022 - Workshops, Genoa, Italy, June 6-10, 2022*, pages 19–30, Los Alamitos, CA, USA, 2022. IEEE Computer Society.

LH17. Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

MAB$^+$18. Carlos Aguilar Melchor, Nicolas Aragon, Slim Bettaieb, Loıc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Edoardo Persichetti, Gilles Zémor, and IC Bourges. Hamming quasi-cyclic (HQC). *NIST PQC Round*, 2(4):13, 2018.

Man04. Stefan Mangard. Hardware countermeasures against dpa–a statistical analysis of their effectiveness. In *Topics in Cryptology–CT-RSA 2004: The Cryptographers' Track at the RSA Conference 2004, San Francisco, CA, USA, February 23-27, 2004, Proceedings*, pages 222–235. Springer, 2004.

NSB$^+$23. William Newhouse, Murugiah Souppaya, William Barker, Chris Brown, Panos Kampanakis, Jim Goodman, Julien Prat, John Gray, Mike Ounsworth, Cleandro Viana, et al. Migration to post-quantum cryptography quantum readiness: Testing draft standards. Technical Report NIST Special Publication 1800-38C, US Department of Commerce, National Institute of Standards and Technology, 2023.

OBL$^+$19. Tom O'Malley, Elie Bursztein, James Long, François Chollet, Haifeng Jin, Luca Invernizzi, et al. Kerastuner. `https://github.com/keras-team/keras-tuner`, 2019.

PQC. PQClean. PQClean, HQC implementation, gf2x.c. `https://github.com/PQClean/PQClean/blob/master/crypto_kem/hqc-128/clean/gf2x.c`. Accessed: 2024-12-16.

Sho94. Peter W Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th annual symposium on foundations of computer science*, pages 124–134. Ieee, 1994.

SHR$^+$22. Thomas Schamberger, Lukas Holzbaur, Julian Renner, Antonia Wachter-Zeh, and Georg Sigl. A power side-channel attack on the Reed-Muller Reed-Solomon version of the HQC cryptosystem. In *International Conference on Post-Quantum Cryptography*, pages 327–352. Springer, 2022.

SRSWZ21. Thomas Schamberger, Julian Renner, Georg Sigl, and Antonia Wachter-Zeh. A power side-channel attack on the CCA2-secure HQC KEM. In *Smart Card Research and Advanced Applications: 19th International Conference, CARDIS 2020, Virtual Event, November 18–19, 2020, Revised Selected Papers 19*, pages 119–134. Springer, 2021.

SVP$^+$24. Maxime Spyropoulos, David Vigilant, Fabrice Perion, Renaud Pacalet, and Laurent Sauvage. Masked vector sampling for HQC. *Cryptology ePrint Archive*, 2024.

UXT^{+}22. Rei Ueno, Keita Xagawa, Yutaro Tanaka, Akira Ito, Junko Takahashi, and Naofumi Homma. Curse of re-encryption: a generic power/EM analysis on post-quantum KEMs. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 296–322, 2022.

Val18. Raphael Vallat. Pingouin: statistics in Python. *J. Open Source Softw.*, 3(31):1026, 2018.

Wel51. Bernard Lewis Welch. On the comparison of several mean values: an alternative approach. *Biometrika*, 38(3/4):330–336, 1951.

YJ21. Wei Yang and Anni Jia. Side-channel leakage detection with one-way analysis of variance. *Security and Communication Networks*, 2021(1):6614702, 2021.

ZBHV21. Gabriel Zaid, Lilian Bossuet, Amaury Habrard, and Alexandre Venelli. Efficiency through diversity in ensemble models applied to side-channel attacks:–a case study on public-key algorithms–. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 60–96, 2021.