Revisiting Side-Channel Analysis on Polynomial Multiplication in the HQC Implementation

Trevor Yap[0000−0001−8651−574X], Shivam Bhasin[0000−0002−6903−5127], and Léo Weissbart[0000−0003−0288−9686]

Nanyang Technological University, Singapore
{trevor.yap, sbhasin, leojoseph.weissbart}@ntu.edu.sg

Abstract. The Hamming Quasi-Cyclic (HQC) scheme is a post-quantum key encapsulation mechanism recently selected for standardization by NIST. While the security against quantum-capable computation is the focus of the development, the security of its implementations against side-channel analysis is also a critical concern. In this work, we revisit the side-channel analysis of the latest HQC reference implementation, targeting its polynomial multiplication function notably used in decryption algorithm. After prior work demonstrated a simple power analysis (SPA) attack against a lookup-table-based multiplication, the implementation of the function has been updated by replacing it with a bitwise schoolbook multiplication.

We show that the updated implementation is still vulnerable to SPA with clearly distinguishable secret data dependent variations in power consumption when compiled using the `-O0` flag. By analyzing these patterns, an attacker can recover individual bits of the secret polynomial, demonstrating that SPA remains effective against the updated design. However, SPA is not exploitable when using compilation optimization with `-Os` flag. As such, we consider a profiling adversary model and show that with Linear Discriminant Analysis classifiers, one can recover the secret key with 100%.

We discuss various countermeasures. The first countermeasure we consider is swapping the inputs of the multiplication. We show that this countermeasure is ineffective, as training a multi-bit neural network in a profiling setting enables recovery of all bits of the secret key with 100% accuracy. As such, there is a need for a more secure polynomial multiplication implementation in HQC, either through the introduction of hiding countermeasures or through a redesign of the implementation itself.

Keywords: Simple Power Analysis · HQC · Polynomial Multiplication · Post-Quantum Cryptography· Machine Learning · Deep Learning

1 Introduction

Classical cryptographic primitives, such as RSA, will see their security level diminish by the advent of the existence of quantum computer running Shor's algorithm [19]. In response to this threat, the National Institute of Standards and

Technology (NIST) initiated a standardization process for post-quantum cryptography (PQC) in 2016. This effort aims to evaluate and standardize quantum-resistant public-key cryptographic algorithms. On March 11, 2025, Hamming quasi-cyclic (HQC) was selected as a new alternative in the fourth round of the project [12]. In addition to the algorithms themselves, their implementations are also required to be evaluated during the selection process.

In this work, we investigate the security of the polynomial multiplication implementation used in the HQC reference implementation. We target the latest released version (v.5.0.0) of the HQC implementation, [1] which corresponds to the version released when HQC was announced as the fifth algorithm for PQC standardization.

In HQC, the secret key consists of two sparse binary polynomials $(\mathbf{x}, \mathbf{y})$. Decryption multiplies $\mathbf{y}$ by a part of the ciphertext $\mathbf{u}$, which is sent over the network and is therefore known to any observer. The result is then passed to an error-correcting decoder that outputs the message. This multiplication is thus the one step where the secret meets a value the attacker already knows, which makes it an attractive target: if its power consumption reveals $\mathbf{y}$, the scheme is broken. $\mathbf{x}$ can then be computed from $\mathbf{y}$ and the public key, leading to full key recovery. Moreover, such an attack only requires observing the decryption of an ordinary ciphertext in contrast to other works which require maliciously crafted ciphertexts that can be detected [17, 16, 4].

This target has already been shown to leak. Velek et al. [20] analyzed the implementation submitted to the fourth round of the NIST process and showed that a single power trace reveals secret bits of $\mathbf{y}$. A secret dependent lookup table inside the multiplication routine `base_mul` led to exploitable power consumption leakage. The same authors proposed a mitigation that avoids lookup tables altogether, and the HQC team adopted a revised version of it in the latest reference implementation (v5.0.0), as the function `schoolbook_mul`. While the original attack no longer applies, it raises the following question: *does the revised implementation eliminate the leakage, or does it merely relocate it?* This question is of practical relevance, as the reference implementation is likely to serve as the basis for embedded deployments of the forthcoming standard, and vulnerabilities present in it may propagate to those deployments.

Our analysis shows that the revised implementation remains vulnerable to power analysis. Although the revised routine contains no secret-dependent branches, it still processes the secret bit by bit. For each bit, a mask consisting of either all zeros or all ones is derived, and this mask determines whether a shifted copy of the public operand is accumulated into the result. Consequently, the amount of data manipulated, and hence the power consumed by the device, depends on the value of the secret bit. On an ARM Cortex-M4 microcontroller, this dependency is clearly observable when the code is compiled without optimization, allowing the secret bits to be recovered by visual inspection of a single power trace. Enabling compiler optimization reduces the difference to a level that can no longer be distinguished reliably by visual inspection, yet it remains exploitable:

[1] `https://gitlab.com/pqc-hqc/hqc/-/tree/v5.0.0`

an adversary with access to a device under their own control can record profiling traces and train a classifier that recovers the secret bits automatically. Finally, the natural mitigation is to exchange the roles of the two operands, so that the mask depends on the public value rather than on the secret. The secret remains subject to the same computation and can still be recovered by a profiling attack based on a more expressive model.

In summary, this work makes the following contributions.

- We show that `schoolbook_mul` leaks the secret in a single power trace when compiled with `-O0`: all sixteen values of a four-bit window of the secret produce distinguishable trace patterns.
- With `-Os`, visual inspection is no longer reliable, but a profiled Linear Discriminant Analysis (LDA) classifier recovers each secret bit with **100**% accuracy. We also measure the amount of added noise needed to reduce this accuracy to random guessing.
- By analyzing the recursion of the Karatsuba multiplication, we show that observing 32 of the 243 calls to `schoolbook_mul` covers the entire secret in HQC-128 (64 of 729 for HQC-192 and HQC-256).
- We show that swapping the roles of the operands of `schoolbook_mul` defeats the above attacks, but a multi-bit neural network still recovers all 512 secret bits of a leaf call with **100**% accuracy. The masking strategy described by Velek et al. [20] is likewise insufficient.

Consequently, removing the lookup table does not suffice to protect the polynomial multiplication in HQC; hiding countermeasures or a redesign of the routine are required.

Paper Organization. Section 2 reviews related work and Section 3 recalls the necessary background on HQC. Section 4 describes the targeted implementation and our setup. Sections 5 and 6 present the analysis under `-O0` and `-Os`, respectively. Section 7 evaluates countermeasures, and Section 8 concludes the paper.

2 Related Work

Side-channel attacks on HQC differ in the operation they target, in the leakage they exploit, and in the adversary they assume. We organize them accordingly. Schamberger et al. [17] presented the first power side-channel attack on HQC, targeting an older version with a BCH decoder, and later adapted it to the current Reed-Muller–Reed-Solomon (RMRS) variant [16]. Their attack builds an oracle that reveals whether the output of the Reed-Muller decoder is zero. Goy et al. [4] improved the oracle by training a linear discriminant classifier. Later works reduced the number of required traces with offline templates built from the Reed-Muller code [3] or with soft analytic side-channel analysis (SASCA) [5, 1]. All of these attacks operate in a chosen-ciphertext setting, in which the crafted ciphertexts may cause decryption failures and could therefore be detected. Paiva

et al. [13] instead consider valid ciphertexts and construct a distinguisher from the `find_peak` and `expand_and_sum` functions. Recently, [8] target the fixed weight vector sampler and fully recover the secret key using power consumption.

Other works exploit timing or cache behavior rather than power consumption [14, 6, 7, 18]. Like the attacks above, they target the decoder or the sampler rather than the polynomial multiplication.

Only a few works target the multiplication, where the secret is combined directly with public data. Huang et al. [9] proposed a DPA-style attack that exploits the Toeplitz structure of the multiplication. It requires about 20 traces and a chosen-ciphertext setting. Velek et al. [20] showed that a single trace of the previous implementation (round-4) reveals secret bits through the lookup table in `base_mul`, and proposed a table-free replacement. Rabas et al. [15] mounted profiling attacks with machine learning on the multiplication, but on earlier implementations (round-4).

None of the above analyzes the revised implementation that replaced the lookup table. Since the table is gone, the attack of [20] no longer applies, and it is not known whether the replacement is secure. We answer this question with simple power analysis and with profiling attacks. Unlike the chosen-ciphertext attacks above, ours needs only ordinary ciphertexts.

3 Preliminaries of HQC

In this section, we will present the essential parts of HQC for our discussion. HQC is an IND-CCA2 secure PQC scheme for Key Encapsulation Mechanism (KEM) [10]. HQC relies on the Quasi-Cyclic Syndrome Decoding problem as its hardness problem to be secure. To build the KEM, HQC develops its own Public-Key Encryption (PKE). In this paper, we will focus on the decryption algorithm of HQC.

Firstly, we introducet the required notation. We denote bold lowercases (e.g. $\boldsymbol{x}$) as vectors and bold uppercase (e.g. $\boldsymbol{X}$) as matrix. Any binary vector $\boldsymbol{a} = (a_0, \ldots, a_{n-1}) \in \mathbb{F}_2^n$ can also be considered as a polynomial $a_0 + a_1 X^1 + \cdots + a_{n-1} X^{n-1}$ in the ring $\mathcal{R} \in \mathbb{F}_2[X]/(X^n - 1)$.

For any two vectors $u, v \in \mathcal{R}$, the multiplication $\boldsymbol{w} = \boldsymbol{u} \cdot \boldsymbol{v}$ in the ring $\mathcal{R}$ is defined as:

$$w_\ell = \sum_{i+j=\ell} u_i v_j.$$

We can rewrite the multiplication using

$$\boldsymbol{rot}(\boldsymbol{x}) = \begin{pmatrix} x_0 & x_{n-1} & \ldots & x_1 \\ x_1 & x_0 & \ldots & x_2 \\ \vdots & \vdots & \ddots & \vdots \\ x_{n-1} & x_{n-2} & \ldots & x_0 \end{pmatrix}$$

and we have

$$\boldsymbol{u} \cdot \boldsymbol{v} = \boldsymbol{u} \times rot(\boldsymbol{v})^T = \boldsymbol{v} \times rot(\boldsymbol{u})^T = \boldsymbol{v} \cdot \boldsymbol{u}$$

In general, the multiplication algorithm is implemented in an optimized version using the Karatsuba or Toom-Cook algorithm [20].

The HQC scheme makes use of a public code $\mathcal{C} \subseteq \mathbb{F}_2^n$ with dimension k. Current versions of HQC use the duplicated first-order Reed-Muller concatenated with a shortened Reed-Solomon code, we simply call it the Reed-Solomon Reed-Muller Code (RSRM) [11]. We denote $\xleftarrow{\$} \mathcal{X}$ as sampling uniformly from the set $\mathcal{X}$, $\xleftarrow{\$,w} \mathcal{R}$ as sampling uniformly from the polynomial ring $\mathcal{R}$ with the constraint that the sampled element has a Hamming weight w. The sampling can be made deterministic by using a pseudo-random generator with a random seed θ.

We shall briefly describe HQC PKE. HQC PKE has three algorithms: key generation (Algorithm 1), encryption (Algorithm 2) and decryption (Algorithm 3).

For the key generation algorithm, the public vector $\boldsymbol{h} \in \mathcal{R}$ are uniformly sampled and the secret variables $\boldsymbol{x}, \boldsymbol{y} \in \mathcal{R}$ are also uniformly sampled with both their Hamming weight restricted to w. Then, we set the secret key as $sk = (\boldsymbol{x}, \boldsymbol{y})$ and public key as $pk = (\boldsymbol{h}, \boldsymbol{s})$ with $\boldsymbol{s} = \boldsymbol{x} + \boldsymbol{h} \cdot \boldsymbol{y}$. Therefore, this suggests that if either $\boldsymbol{x}$ or $\boldsymbol{y}$ is recovered, the full secret key is recovered.

Algorithm 1. Key Generation

Input: hqc parameters
Output: public key pk, secret key sk

$\boldsymbol{h} \xleftarrow{\$} \mathcal{R}$
$(\boldsymbol{x}, \boldsymbol{y}) \xleftarrow{\$,w} \mathcal{R}^2$
$\boldsymbol{s} \leftarrow \boldsymbol{x} + \boldsymbol{h} \cdot \boldsymbol{y}$
$pk = (\boldsymbol{h}, \boldsymbol{s})$
$sk = (\boldsymbol{x}, \boldsymbol{y})$
return pk, sk.

Next, during encryption $\boldsymbol{e}, \boldsymbol{r_1}, \boldsymbol{r_2}$ are uniformly sampled from $\mathcal{R}$ with some restriction on their Hamming weights. These are initialized by a seed θ. The ciphertext is then created via $\boldsymbol{c} = (\boldsymbol{u}, \boldsymbol{v})$ where

$$\boldsymbol{u} = \boldsymbol{r_1} + \boldsymbol{h} \cdot \boldsymbol{r_2}$$
$$\boldsymbol{v} = \mathit{Encode}(\boldsymbol{m}) + \boldsymbol{s} \cdot \boldsymbol{r_2} + \boldsymbol{e}$$

with $\mathit{Encode}(\boldsymbol{m}) = \boldsymbol{mG}$ encode the message $\boldsymbol{m}$ using the RSRM Code with $\boldsymbol{G}$ being its generator matrix.

Algorithm 2. Encrypt

Input: message $\boldsymbol{m}$, public key pk, randomness seed θ
Output: ciphertext c

$\boldsymbol{e} \xleftarrow{\$,w_e,\theta} \mathcal{R}$
$(\boldsymbol{r_1}, \boldsymbol{r_2}) \xleftarrow{\$,w_r,\theta} \mathcal{R}^2$
$\boldsymbol{u} \leftarrow \boldsymbol{r_1} + \boldsymbol{h} \cdot \boldsymbol{r_2}$
$\boldsymbol{v} \leftarrow \mathit{Encode}(\boldsymbol{m}) + \boldsymbol{s} \cdot \boldsymbol{r_2} + \boldsymbol{e}$
return $\boldsymbol{c} = (\boldsymbol{u}, \boldsymbol{v})$.

Now, during the decryption algorithm, it decode the $\boldsymbol{v} - \boldsymbol{u} \cdot \boldsymbol{y}$. Since

$$\boldsymbol{v} - \boldsymbol{u} \cdot \boldsymbol{v} = \mathit{Encode}(\boldsymbol{m}) + \boldsymbol{x} \cdot \boldsymbol{r_2} - \boldsymbol{r_1} \cdot \boldsymbol{y} + \boldsymbol{e}$$

Now, suppose the error term $\boldsymbol{e'} = \boldsymbol{x} \cdot \boldsymbol{r_2} - \boldsymbol{r_1} \cdot \boldsymbol{y} + \boldsymbol{e}$ is sufficiently small, then the RSRM code $\mathcal{C}$ can correct it and recover the message $\boldsymbol{m}$. In this paper, we

Algorithm 3. Decrypt

Input: secret key $sk = (\boldsymbol{x}, \boldsymbol{y})$, ciphertext $\boldsymbol{c} = (\boldsymbol{u}, \boldsymbol{v})$
Output: message $\boldsymbol{m}$
1: $\boldsymbol{v'} \leftarrow \boldsymbol{v} - \boldsymbol{u} \cdot \boldsymbol{y}$
2: $\boldsymbol{m} \leftarrow \mathit{Decode}(\boldsymbol{v'})$
3: **return** $\boldsymbol{m}$.

shall mainly focus on the multiplication of $\boldsymbol{u} \cdot \boldsymbol{y}$ in the decryption (green in Algorithm 3). We note that our analysis can also be extended to the multiplication $\boldsymbol{h} \cdot \boldsymbol{y}$ (line 3 of Algorithm 1).

4 HQC Implementation and Experimental Set-up

Target Function. This work analyzes the reference implementation of the HQC as in the version v5.0.0.[2] We observe that the multiplication $\boldsymbol{u} \cdot \boldsymbol{y}$ is implemented using a Karatsuba algorithm followed by a modular reduction. The Karatsuba algorithm recursively calls itself until the input size reaches 16 words (each 64 bits), at which point it invokes the function `schoolbook_mul`. By following the inputs of the recursive calls within the Karatsuba algorithm, we observe that `schoolbook_mul` takes a portion of the secret $\boldsymbol{y}$ as input `a`, and a portion of the ciphertext $\boldsymbol{u}$ as input `b`. In this work, we focus on extracting the bits of $\boldsymbol{y}$.

The `schoolbook_mul` (Fig.1) implements a schoolbook-style carry-less multiplication of two large binary polynomials over $\mathbb{F}_2$. Each polynomial is represented as arrays of 64-bit words, with $n \leq 16$ corresponding to maximum of 1024-bit inputs. It iterates over each word of the first operand `a` and, within each word, over all 64 bits. For every bit of `a[i]`, a mask is generated to determine whether that bit is set, allowing the computation to proceed without conditional branches. If the bit is 1, a shifted version of the second operand `b` is XORed into the result array at the appropriate position; if the bit is 0, no change occurs. The shifting is handled across word boundaries to correctly align contributions in the 2048-bit result. Overall, this corresponds to a bitwise schoolbook multiplication where partial products are accumulated using XOR.

Since the mask is derived from the bits of `a`, which correspond directly to the secret $\boldsymbol{y}$, one would expect observable differences in the power consumption trace depending on whether the mask bit is 0 or 1. Consequently, differences within the trace can be exploited to recover the secret $\boldsymbol{y}$. The results are presented in Section 5, where the observable differences in the patterns are shown.

[2] `https://gitlab.com/pqc-hqc/hqc/-/tree/v5.0.0`

```
1  static void schoolbook_mul(uint64_t *r, const uint64_t *a, const uint64_t
       *b, size_t n) {
2      memset(r, 0, 2 * n * sizeof(uint64_t));
3      for (size_t i = 0; i < n; i++) {
4          uint64_t ai = a[i];
5          for (int bit = 0; bit < 64; bit++) {
6              uint64_t mask = -((ai >> bit) & 1ULL);
7              size_t base = i;
8              int sh = bit;
9              int inv = 64 - sh;
10             if (sh == 0) {
11                 for (size_t j = 0; j < n; j++) {
12                     r[base+j]^=b[j] & mask;
13                 }
14             } else {
15                 for (size_t j = 0; j < n; j++) {
16                     r[base + j] ^= (b[j] << sh) & mask;
17                     r[base + j + 1] ^= (b[j] >> inv) & mask;
18                 }
19             }
20         }
21     }
22 }
```

Fig. 1: C code snippet of the `schoolbook_mul` operation used by the multiplication in HQC. The target operations and variables for SCA are highlighted in red and in blue.

Experimental Set-up. In this work, the target device used to collect the power consumption of the `schoolbook_mul` function is the ChipWhisperer-Lite board, embedding a 32-bit STM32F303 ARM Cortex-M4 microcontroller. The reference implementation is written in C and has been compiled with arm-none-eabi-gcc (version 15.2.0). Compilation optimization used is either -O0 or -Os, and is specified in each section. Other compilation flags are the following: -mcpu=cortex-m4 -mthumb -mfloat-abi=soft -fmessage-length=0 -ffunction-sections -gdwarf-2 -funsigned-char -funsigned-bitfields -fshort-enums -Wall -std=gnu99.

The power traces are captured using the board's built-in oscilloscope, which is limited to approximately 25,000 sample points per trace. The sampling frequency is set to four times the target clock frequency.

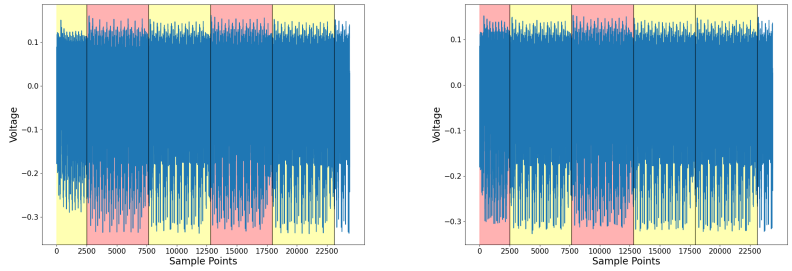
5 Analysis under `-O0` Optimization

In this section, we analyze the power consumption traces when the underlying code is compiled with the `-O0` optimization flag. We then describe the SPA attack performed on these collected traces, highlighting how information can be extracted from the observed leakage.

Simple Power Analysis. Now we describe an SPA attack. The attack is purely visual in nature. Figure 2 presents the power consumption traces corresponding to the computation of the first five bits of `a[0]`. These traces are extracted from the execution of lines 5-20 in Figure 1, restricted to `bit` $= 0, \ldots, 4$.

Suppose `a[i]` $= a_{63}a_{62}\ldots a_0$., then Figure 2a depicts the trace obtained when $a_4a_3a_2a_1a_0 = 01010$ while Figure 2b shows the trace for $a_4a_3a_2a_1a_0 = 00101$.

From Figure 2, we identify clearly distinguishable temporal segments within the traces. Each segment corresponds to the processing of a single bit of `a[0]`, as determined by the iteration over `bit` in the implementation. Specifically, the interval from sample points 0 to 2500 corresponds to the processing of a_0, 2500 to 7650 corresponds to a_1, 7650 to 12800 corresponds to a_2, 12800 to 17950 corresponds to a_3, and 17950 to 23100 corresponds to a_4. These segment boundaries are approximate and serve as empirical estimates derived from visual inspection of the traces.

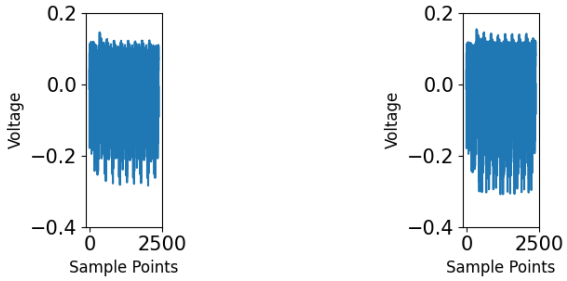


(a) 01010 (b) 00101

Fig. 2: Power consumption traces corresponding to the computation of the first 5 bits of `a[0]` under the `-O0` compiler. Traces segments corresponding to bit value 0 are shown in yellow, while those corresponding to bit value 1 are shown in red.

In particular, lines 11-13 (highlighted in red in Figure 1) correspond to the computation of a_0. For this case, we observe two distinct patterns in the power consumption trace, depending on the value of the processed bit. This is the leftmost segment of the trace from sample point 0 to 2500. When $a_0 = 0$, the mask evaluates to `0x0000000000000000`, which results in a trace segment with lower power consumption compared to the case where $a_0 = 1$, for which the mask evaluates to `0xFFFFFFFFFFFFFFFF`. Figure 3 shows a side-by-side comparison of the trace segments corresponding to the two possible values of a_0. This is line 16-19. A similar pattern can also be observed for `bit` > 0. lines 15-18 (highlighted in blue in Figure 1) corresponds to the computation of a_i for $i \neq 0$.

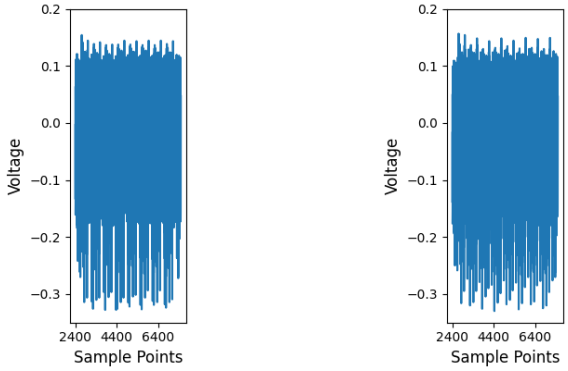
For simplicity, we restrict our discussion to the case `bit` $= 1$, as the remaining cases exhibit analogous behaviour. In this setting, we again observe two distinct patterns in the power consumption trace, depending on the value of the processed bit. Similarly to a_0, when $a_1 = 0$, the mask evaluates to `0x0000000000000000`, resulting in a less dense power consumption segment compared to when $a_1 = 1$, which the mask evaluates to `0xFFFFFFFFFFFFFFFF`, producing a visibly different trace pattern. Figure 4 presents a side-by-side comparison of the trace segments



(a) $a_0 = 0$ (b) $a_0 = 1$

Fig. 3: Trace Segment for different values of a_0 (sample point 0 to 2500)

for both possible values of a_1. We evaluate all $2^4 = 16$ possible key-dependent



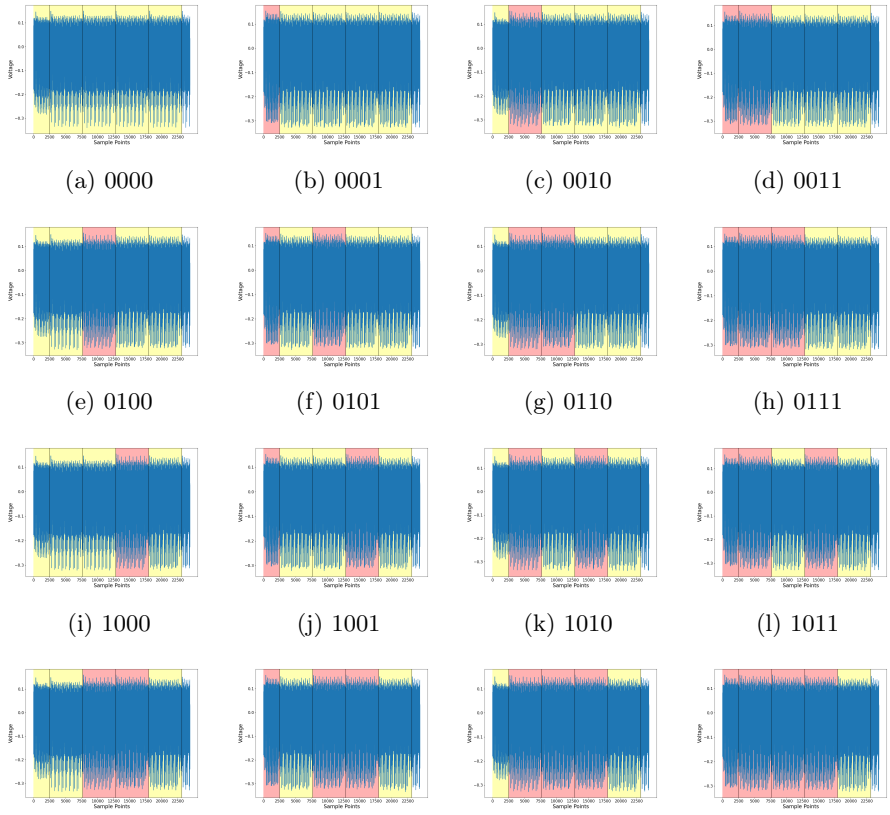
(a) $a_1 = 0$ (b) $a_1 = 1$

Fig. 4: Trace Segment for different values of a_1 (sample point 2400 to 7500)

patterns, corresponding to the complete set of 4-bit binary values from 0000 to 1111. Each pattern induces a distinct power consumption trace, reflecting how different combinations of key bits influence the underlying computation. As shown in Figure 5, the traces are organized according to their binary key values, enabling a systematic comparison across all cases.

Each trace is partitioned into regions based on the processed bit values. Regions corresponding to a bit value of 0 are colored yellow, while those corresponding to a bit value of 1 are colored red. We observe clear and distinguishable side-channel patterns between the two bit values, consistent with the discussion above. These results indicate that the four key bits can be visually distinguished with high reliability.

Using Figure 5, we further argue that the secret information can, in principle, be recovered through simple visual pattern matching. Even without the figure,



(a) 0000 (b) 0001 (c) 0010 (d) 0011

(e) 0100 (f) 0101 (g) 0110 (h) 0111

(i) 1000 (j) 1001 (k) 1010 (l) 1011

(m) 1100 (n) 1101 (o) 1110 (p) 1111

Fig. 5: Power consumption traces for all 2^4 key-dependent executions, covering all key values from 0000 to 1111 under the `-O0` compiler. Each subplot corresponds to one key value.

an adversary observing the two distinct trace patterns could perform a brute-force comparison and map each pattern to either 0 or 1, thereby recovering the key.

6 Analysis under `-Os` Optimization

Next, we investigate whether the SPA leakage persists when the firmware is compiled with the `-Os` optimization flag. Even under this optimization level, we are still able to identify trace segments corresponding to the processing of individual bits. As in the previous setting, the segment associated with a_0 is shorter, consisting of 628 sample points, while the segments for a_i with $i \neq 0$ are longer, each containing 1708 sample points. With our measurement setup, which captures approximately 25,000 sample points per trace, each trace contains the

computation of up to 15 bits when the code is compiled with the `-Os` flag. Figure 6 shows an example of a full trace.

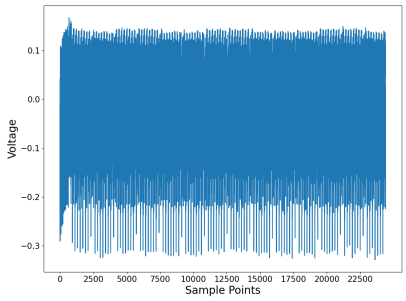


Fig. 6: Example of a full power trace collected under `-Os` optimization.

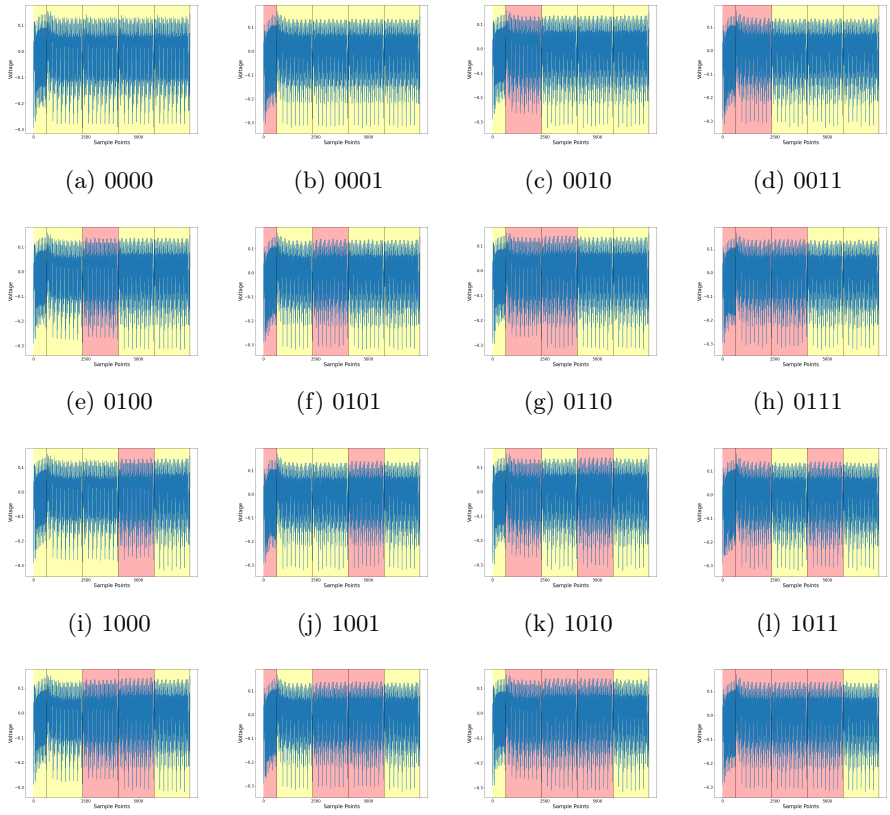
Figure 7 illustrates the $2^4 = 16$ possible key-dependent patterns, corresponding to all 4-bit binary values from 0000 to 1111, when the code is compiled with `-Os`. However, the patterns are no longer as visually distinguishable as before. For example, Figures 7m (i.e. key value is 1100) makes it difficult to distinguish whether the fourth bit (i.e. correct value is 1) is 0 or 1.

6.1 Profiling Attack

Now, we investigate whether, under the profiling setting, secret recovery can be automated using a classifier instead of relying on visual inspection. We performed 100 calls to `schoolbook_mul`. In each call, the inputs consist of a random secret value `a` and a random public variable `b`, where each `a` and `b` is composed of 8 words of 64 bits each. In total, this results in 51200 trace segments, all extracted from a single trace.

As described above, there are two types of trace segments: one corresponding to a_0, and another corresponding to a_i for $i \neq 0$. From processing a_0, we obtained a dataset of 800 trace segments, each containing 628 sample points. In contrast, the dataset derived from computations involving a_i with $i \neq 0$ consists of 50400 trace segments, each with 1708 sample points. Note that there are fewer sample points than those in Section 5 as we compile the code in `-Os` instead. But the underlying computation remains the same. Furthermore, the trace segments are acquired with labels that are uniformly sampled; hence, there is no bias in the dataset.

For each type of trace segment, we divide the dataset into profiling and attack dataset. For the dataset corresponding to a_0, we use 500 profiling trace segments and 300 attack trace segments. For a_i with $i \neq 0$, we use 40000 trace segments for the profiling phase and 10400 trace segments for the attack phase. Using



(a) 0000 (b) 0001 (c) 0010 (d) 0011

(e) 0100 (f) 0101 (g) 0110 (h) 0111

(i) 1000 (j) 1001 (k) 1010 (l) 1011

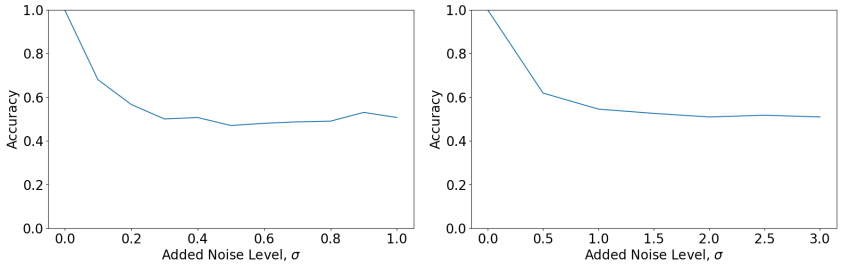
(m) 1100 (n) 1101 (o) 1110 (p) 1111

Fig. 7: Power consumption traces for all 2^4 key-dependent executions, covering all key values from 0000 to 1111 under `-Os` compiler. Each subplot corresponds to one key value. For illustration purposes, we only plot the first 7400 sample points of the trace.

the profiling dataset, we then train an LDA classifier [2] to predict whether the underlying computation in a trace corresponds to a value of a_i equal to zero or one. Then, we compute the accuracy by using the trained LDA classifier to predict the attack traces.

We believe that adding noise can serve as a hiding countermeasure, hence we investigate how much noise must be added to render the profiling attack ineffective. Figure 8 shows the accuracy of the trained classifier under varying levels of noise added to the traces. The injected noise follows a Gaussian distribution with mean 0 and standard deviation σ, which determines the noise level. Based on our experiments, the added noise level required for a_0 to reach random-guessing performance is lower than that for a_i with $i \neq 0$. Accordingly, we vary the added noise level in increments of 0.1 for a_0, and in increments of 0.5 for a_i with $i \neq 0$.

For both types of trace segments, we obtained a 100% accuracy when no noise is injected into the traces. However, as noise level increases the accuracy of the classifier decreases. For a_0, the classifier accuracy decreases to random guessing i.e. $\approx 50\%$ at around $\sigma = 0.5$. As for a_i with $i \neq 0$ the classifier accuracy decreases to random guessing i.e. $\approx 50\%$ at around $\sigma = 2.0$.



(a) Accuracy for a_0. (b) Accuracy for a_i where $i \neq 0$.

Fig. 8: Accuracy for classifier trained to predict under various noise level.

This suggest that without any countermeasure, the attacker can recover the secret key with 100% accuracy using a single long traces, as described in Section 6.2. As such one should use countermeasures like adding jitter or noise to prevent a trained classifier to recover the secret key.

6.2 Attack Evaluation

For HQC, the private key $\boldsymbol{y}$ has 17669, 35851, or 57637 bits, depending on the security level, and is stored as 277, 561, or 901 words of 64 bits. The function `karatsuba_mul` implements the Karatsuba multiplication algorithm with a threshold of 16 on the block size n; below this threshold, it invokes `schoolbook_mul`. Fig. 9 presents the function `karatsuba_mul` used in the reference implementation.

Taking HQC-128 as an example, `karatsuba_mul` is initially invoked with $n = 277$ words. Let `a` and `b` be the input to the `karatsuba_mul`. We further split the inputs into two halves $\mathtt{a} = [\mathtt{a}_l, \mathtt{a}_h]$ and $\mathtt{b} = [\mathtt{b}_l, \mathtt{b}_h]$. For $n > 16$, the function applies the Karatsuba recursion by setting $m = \lfloor n/2 \rfloor$ and performing three recursive calls: one on the lower halves ($z_0 = \mathtt{a}_l * \mathtt{b}_l$, with $n_0 = m$ words), one on the upper halves ($z_2 = \mathtt{a}_h * \mathtt{b}_h$, with $n_1 = n - m$ words), and one on the XOR of the halves ($z_{\mathtt{mid}} = (\mathtt{a}_h + \mathtt{a}_l) * (\mathtt{b}_h + \mathtt{b}_l)$, also with n_1 words). Figure 10 illustrates the resulting recursion tree. The recursion terminates after five levels, producing $3^5 = 243$ invocations of `schoolbook_mul`. Note that the leaves are reached with 8 or 9 words as the threshold value of in the implementation is fixed to be 16. As such, sizes of 17 and 18 still exceed the threshold and are therefore split once more, into 8/9 and 9/9 respectively. For HQC-128 the leaf

```c
static void karatsuba_mul(uint64_t *r, const uint64_t *a, const
 uint64_t *b, size_t n, uint64_t *tmp_buffer) {
if (n <= KARATSUBA_THRESHOLD) {
    schoolbook_mul(r, a, b, n);
    return;
}

size_t m = n >> 1;
size_t n0 = m;
size_t n1 = n - m;

/* take successive chunks of tmp_buffer for each intermediate result
 */
uint64_t *z0 = tmp_buffer;   /* a_l*b_l, size 2*n words */
uint64_t *z2 = z0 + 2 * n;   /* a_h*b_h, size 2*n words */
uint64_t *zmid = z2 + 2 * n; /* (a_h+a_l)*(b_h+b_l), size 2*n words */

/* ta and tb hold the sums of low and high halves: */
/* ta[i] = a0[i] XOR a1[i], tb[i] = b0[i] XOR b1[i] for i < n1 */
uint64_t *ta = zmid + 2 * n;
uint64_t *tb = ta + n;

/* buffer for child recursions */
uint64_t *child_buffer = tmp_buffer + 8 * n;

/* 1) a_l*b_l */
karatsuba_mul(z0, a, b, n0, child_buffer);

/* 2) a_h*b_h */
karatsuba_mul(z2, a + m, b + m, n1, child_buffer);

/* 3) (a_h+a_l)*(b_h+b_l) */
for (size_t i = 0; i < n1; i++) {
    uint64_t loa = (i < n0 ? a[i] : 0);
    uint64_t lob = (i < n0 ? b[i] : 0);
    ta[i] = loa ^ a[m + i];
    tb[i] = lob ^ b[m + i];
}
karatsuba_mul(zmid, ta, tb, n1, child_buffer);

/* 4) assemble into r */
memset(r, 0, 2 * n * sizeof(uint64_t));
for (size_t i = 0; i < 2 * n0; i++) r[i] ^= z0[i];
for (size_t i = 0; i < 2 * n1; i++) r[2 * m + i] ^= z2[i];
for (size_t i = 0; i < 2 * n1; i++) {
    uint64_t z0i = (i < 2 * n0 ? z0[i] : 0);
    uint64_t z2i = (i < 2 * n1 ? z2[i] : 0);
    uint64_t mid = zmid[i] ^ z0i ^ z2i;
    r[m + i] ^= mid;
}
}
```

Fig. 9: C code snippet of the `karatsuba_mul` operation used by the multiplication in HQC.

sizes are 37×8 and 206×9 words; for HQC-192 the tree has depth 6 with leaves of 8 or 9 words, and for HQC-256 it has depth 6 with leaves of 14 or 15 words.

To recover the key, not all of these calls need to be observed. The z_{mid} branch multiplies the XOR of two halves of $\boldsymbol{y}$, whereas the z_0 and z_2 branches pass a contiguous slice of $\boldsymbol{y}$ through unchanged. Following only z_0 and z_2 edges from the root therefore yields $2^5 = 32$ leaves whose `a` operands are part of $\boldsymbol{y}$ and which

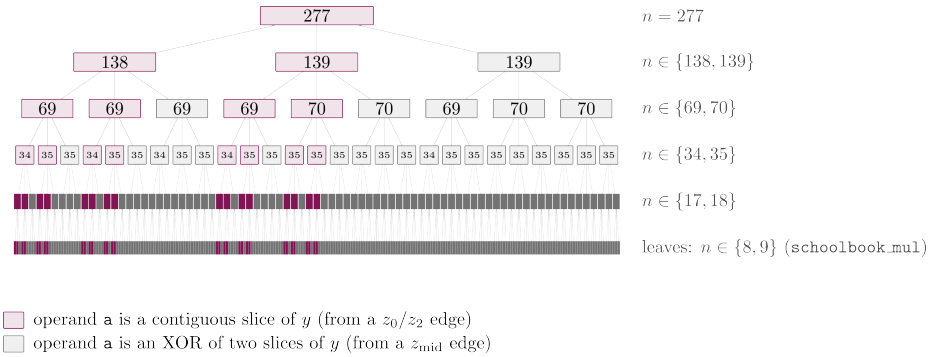


Fig. 10: Full recursion tree of `karatsuba_mul` for HQC-128, where the secret $\boldsymbol{y}$ occupies 277 words of 64 bits. Every one of the 364 nodes across the 5 recursion levels is shown; from level 4 onward nodes are drawn as small unlabelled marks since their sizes only alternate between the two values given at the right of each row. Pink nodes carry a contiguous slice of $\boldsymbol{y}$ (reached only via z_0/z_2 edges); gray nodes carry an XOR of two slices (reached via at least one $z_{\texttt{mid}}$ edge). The bottom row is the 243 actual calls to `schoolbook_mul`, each of which is a point where the attacker can capture a power trace as in Section 5.

together cover all 277 words. Recovering the full secret thus requires observing only 32 of the 243 invocations of `schoolbook_mul` (resp. 64 of the 729 invocations for HQC-192 and HQC-256), each leaking $8 \times 64 = 512$ or $9 \times 64 = 576$ bits of $\boldsymbol{y}$ by the analysis of Section 5. As a result, for HQC-128, the attacker will have to apply the aforementioned SPA methodology of Section 5 to the purple leaf node while ignoring the computations corresponding to the grey leaf node. A similar tree structure can be constructed for HQC-192 and HQC-256 as well.

We measure the power consumption at the start of the polynomial multiplication phase when decrypting the ciphertext using the private key. With the available trace length, we can observe sample points related to a few bits of the private key (5 bits when compiling under `-O0` and 15 when compiling under `-Os`). This can be easily overcome by using either an external oscilloscope or capturing devices with streaming mode like CW1200 ChipWhisperer-Pro or ChipWhisperer-Husky that could capture long traces. Overall, the proposed attacks for recovering the full secret key can be performed using a single trace; in other words, our attack is a horizontal attack.

7 Evaluation of Possible Countermeasures

In this section, we discuss possible countermeasures against the SPA and profiling attacks described above.

7.1 Switching Inputs for `schoolbook_mul`

Since multiplication is commutative in nature, a simple mitigation against the previously explored attack would be to change the inputs into the operands of `schoolbook_mul`. In particular, operand `a` should not contain any secret-dependent data. By swapping the inputs, the ciphertext $\boldsymbol{u}$ is assigned to operand `a`, while the secret $\boldsymbol{y}$ is assigned to operand `b`. This can be achieved without modifying the `schoolbook_mul` function; instead, it suffices to change line 3 of `karatsuba_mul` from `schoolbook_mul(r,a,b,n)` to `schoolbook_mul(r,b,a,n)`.

As a result, the `mask` computation in `schoolbook_mul` no longer depends on the secret $\boldsymbol{y}$, but instead on the public variable $\boldsymbol{u}$. As a result, even if the same leakage patterns described in Section 5 are observed, they correspond only to bits of $\boldsymbol{u}$, which is publicly known. Consequently, no sensitive information is revealed, and the attack yields no exploitable knowledge about the secret key. Note that the profiling attack described above is also ineffective, as the labels are identical to a public variable, and therefore no additional information is revealed.

However, this implementation may still be susceptible to other profiling attacks, such as those described in [15]. In the following, we attempt to recover the secret information when the operands are switched.

Profiling Attack When Inputs Are Switched. Here, we assume that the operands are switched. We again consider the `-Os` compiler. In other words, the operand `b` in `schoolbook_mul` contains the secret value, while `a` contains the public value.

We train a multi-bit neural network that takes as input the trace segments corresponding to a_i for $i \neq 0$, and predicts all bits of the secret contained in `b`. Note that all bits of `b` can be recovered from these trace segments; therefore, we do not need to consider the trace segments corresponding to a_0. In our experiments, we assume that `b` consists of 8 words of 64 bits each, resulting in a total of 512 bits to be recovered. Accordingly, the multi-bit neural network comprises 512 branches, each responsible for predicting a single bit of `b`. We use a Multi-Layer Perceptron (MLP) architecture, where each branch consists of two fully connected layers. A sigmoid activation function is applied to the output layer of each branch. The architecture of the network is illustrated in Figure 11. We perform a hyperparameter search over 50 neural network configurations, exploring the search space defined in Table 1.

Each model is trained for 100 epochs using binary cross-entropy loss applied independently to each branch. The overall training objective is defined as the sum of the losses across all branches. Note that this profiling attack differs from Section 5, where the LDA classifier predicts a single bit of the operand `a`.

We use 40000 trace segments for the profiling phase and 10400 trace segments for the attack phase. Since `b` is sampled uniformly at random, each bit is also randomly sampled. Therefore, there is no bias in the attack dataset that would affect the results. Our experiments show that all bits are successfully recovered with 100% accuracy using the best trained model. This suggests that the imple-

mentation of the schoolbook multiplication is not secure even with the proposed countermeasure to switch the input.

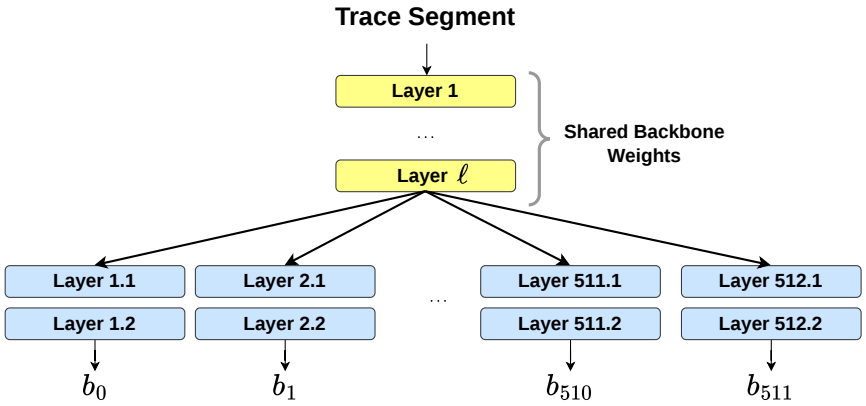


Fig. 11: Multi-bit Neural Network Architecture

Table 1: Hyperparameter search space.

Hyperparameter	**Options**
	MLP
Number of Dense Layers	1 to 8 in a step of 1
Neurons per layer	$10, 20, 50, 100, 200, 300, 400, 500$
Batch size	100 to 1000 in a step of 100
Activation function	$ReLU, SeLU, ELU$ or $tanh$
Optimizer	Adam or RMSprop
Learning Rate	$1e^{-3}, 1e^{-4}, 5e^{-4}, 1e^{-5}, 5e^{-5}$
Weight Initializer	Random Uniform or Glorot Uniform or He Uniform

7.2 Boolean Masking

Next, we consider the masking scheme proposed by [20]. The corresponding pseudocode is given below:

$$
\begin{aligned}
k_0 &\xleftarrow{\$} (\mathbb{F}_2^{64})^{16} \\
k_1 &= k \oplus k_0 \\
r_0 &= \texttt{schoolbook_mul}(k_1, u) \\
r_1 &= \texttt{schoolbook_mul}(k_0, u) \\
r &= r_0 \oplus r_1
\end{aligned}
$$

However, as noted by the authors of [20], an attacker can still successfully recover the secret key when both multiplications are targeted, recovering the a_0 and a_1. As such recovering the secret $a = a_0 \oplus a_1$. The same observation applies in our SPA and both profiling attacks settings. Therefore, the masking does not provide effective protection against this type of analysis. As a result, to mitigate the above attacks, one needs to consider hiding countermeasures that introduce noise or jitter or to redesign multiplication function to mitigate all these side-channel attacks.

8 Conclusion

Table 2: Overview of the attack strategies evaluated under different mitigation and compiler optimization settings.

Mitigation	Optimization Flag	Technique
None	`-O0`	Simple Power Analysis
None	`-Os`	Profiling Attack with LDA
Switch operands role	`-Os`	Profiling Attack with Multi-bit MLP
Masking	`-O0`/`-Os`	Same techniques as above, provided individual shares can be recovered.

In this work, we presented side-channel attacks on the latest HQC implementation, focusing on the `schoolbook_mul` function used within the Karatsuba multiplication. Although the update removed earlier lookup-table-based implementation and replaced them with a schoolbook bitwise multiplication strategy, we show that the implementation still exhibits visually distinguishing power consumption patterns based on the secret when the code is compile under the `-O0` flag, enabling the recovery of key information from direct inspection of power traces. As such, we compile the code in `-Os` flag and noticed that SPA is no

longer effective in recovering the key. Next, we explore a profiling attack on the traces obtained when computing code compiled using the `-Os` flag with a LDA classifier. Our results demonstrate that the attack achieves 100% accuracy in predicting individual secret bits using an LDA classifier.

To mitigate this vulnerability, we explore various countermeasures. The first countermeasure considered is to swap the inputs of the `schoolbook_mul` function. While effective against the SPA and the first profiling technique presented in this work, it is still not secure. We show that profiling attacks using multi-bit neural network could still to predict the each bits of the secret values with 100% accuracy. We also establish that masking countermeasure are not effective against all the our proposed attacks. As a result, our findings highlight that either hiding countermeasures need to be considered or further care is required when designing the multiplication for HQC. Table 2 provides an overview of the attack scenarios investigated in this work, along with the corresponding mitigation strategies and compiler optimization settings.

References

1. Baïsse, C., Moran, A., Goy, G., Maillard, J., Aragon, N., Gaborit, P., Lecomte, M., Loiseau, A.: Secret and shared keys recovery on hamming quasi-cyclic with SASCA. Des. Codes Cryptogr. **93**(6), 2137–2157 (2025), https://doi.org/10.1007/s10623-025-01575-2
2. Bishop, C.M.: Pattern recognition and machine learning, 5th Edition. Information science and statistics, Springer (2007), https://www.worldcat.org/oclc/71008143
3. Dong, H., Guo, Q.: OT-PCA: New Key-Recovery Plaintext-Checking Oracle Based Side-Channel Attacks on HQC with Offline Templates. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2025**(1), 251–274 (2025), https://doi.org/10.46586/tches.v2025.i1.251-274
4. Goy, G., Loiseau, A., Gaborit, P.: A new key recovery side-channel attack on HQC with chosen ciphertext. In: Cheon, J.H., Johansson, T. (eds.) Post-Quantum Cryptography - 13th International Workshop, PQCrypto 2022, Virtual Event, September 28-30, 2022, Proceedings. Lecture Notes in Computer Science, vol. 13512, pp. 353–371. Springer (2022), https://doi.org/10.1007/978-3-031-17234-2_17
5. Goy, G., Maillard, J., Gaborit, P., Loiseau, A.: Single trace HQC shared key recovery with SASCA. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2024**(2), 64–87 (2024), https://doi.org/10.46586/tches.v2024.i2.64-87
6. Guo, Q., Hlauschek, C., Johansson, T., Lahr, N., Nilsson, A., Schröder, R.L.: Don't Reject This: Key-Recovery Timing Attacks Due to Rejection-Sampling in HQC and BIKE. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2022**(3), 223–263 (2022), https://doi.org/10.46586/tches.v2022.i3.223-263
7. Guo, Q., Nabokov, D., Nilsson, A., Johansson, T.: SCA-LDPC: A Code-Based Framework for Key-Recovery Side-Channel Attacks on Post-quantum Encryption Schemes. In: Guo, J., Steinfeld, R. (eds.) Advances in Cryptology - ASIACRYPT 2023 - 29th International Conference on the Theory and Application of Cryptology and Information Security, Guangzhou, China, December 4-8, 2023, Proceedings, Part IV. Lecture Notes in Computer Science, vol. 14441, pp. 203–236. Springer (2023), https://doi.org/10.1007/978-981-99-8730-6_7

8. Hesse, D., Krausz, M., Murugananthan, R., Wollinger, T., Güneysu, T.: Power reveals timing conceals - side-channel attacks and hiding countermeasures for HQC's fixed-weight vector sampling. Cryptology ePrint Archive, Paper 2026/1462 (2026), https://eprint.iacr.org/2026/1462
9. Huang, Z., Wang, W., Zhou, X., Yu, Y.: DPA-Style Attacks on HQC. IACR Cryptol. ePrint Arch. **2025**, 1966 (2025), https://eprint.iacr.org/2025/1966
10. Melchor, C.A., Aragon, N., Bettaieb, S., Bidoux, L., Blazy, O., Deneuville, J.C., Gaborit, P., Persichetti, E., Zémor, G., Bos, J.: Hamming Quasi-Cyclic (HQC). Algorithm Specification (Version 2026-10-12) (2026), https://pqc-hqc.org/doc/hqc_specifications_2026_10_12.pdf
11. Melchor, C.A., Aragon, N., Deneuville, J., Gaborit, P., Lacan, J., Zémor, G.: Efficient error-correcting codes for the HQC post-quantum cryptosystem. Des. Codes Cryptogr. **92**(12), 4511–4530 (2024), https://doi.org/10.1007/s10623-024-01507-6
12. National Institute of Standards and Technology: NIST Selects HQC as Fifth Algorithm for Post-Quantum Encryption. NIST News (March 2025), https://www.nist.gov/news-events/news/2025/03/nist-selects-hqc-fifth-algorithm-post-quantum-encryption
13. Paiva, T.B., Ravi, P., Jap, D., Bhasin, S., Das, S., Chattopadhyay, A.: Et tu, brute? side-channel assisted chosen ciphertext attacks using valid ciphertexts on HQC KEM. In: Niederhagen, R., Saarinen, M.O. (eds.) Post-Quantum Cryptography - 16th International Workshop, PQCrypto 2025, Taipei, Taiwan, April 8-10, 2025, Proceedings, Part II. Lecture Notes in Computer Science, vol. 15578, pp. 294–321. Springer (2025), https://doi.org/10.1007/978-3-031-86602-9_11
14. Paiva, T.B., Terada, R.: A timing attack on the HQC encryption scheme. In: Selected Areas in Cryptography–SAC 2019: 26th International Conference, Waterloo, ON, Canada, August 12–16, 2019, Revised Selected Papers 26. pp. 551–573. Springer (2020)
15. Rabas, T., Bucek, J., Grosso, V., Zenknerová, K., Lórencz, R.: Profiling Side-Channel Attack on HQC Polynomial Multiplication Using Machine Learning Methods. In: Rivain, M., Sasdrich, P. (eds.) Constructive Approaches for Security Analysis and Design of Embedded Systems - First International Conference, CASCADE 2025, Saint-Etienne, France, April 2-4, 2025, Proceedings. pp. 580–602. Lecture Notes in Computer Science, Springer (2025), https://doi.org/10.1007/978-3-032-01405-4_24
16. Schamberger, T., Holzbaur, L., Renner, J., Wachter-Zeh, A., Sigl, G.: A Power Side-Channel Attack on the Reed-Muller Reed-Solomon Version of the HQC Cryptosystem. In: Cheon, J.H., Johansson, T. (eds.) Post-Quantum Cryptography - 13th International Workshop, PQCrypto 2022, Virtual Event, September 28-30, 2022, Proceedings. Lecture Notes in Computer Science, vol. 13512, pp. 327–352. Springer (2022), https://doi.org/10.1007/978-3-031-17234-2_16
17. Schamberger, T., Renner, J., Sigl, G., Wachter-Zeh, A.: A Power Side-Channel Attack on the CCA2-Secure HQC KEM. In: Liardet, P., Mentens, N. (eds.) Smart Card Research and Advanced Applications - 19th International Conference, CARDIS 2020, Virtual Event, November 18-19, 2020, Revised Selected Papers. Lecture Notes in Computer Science, vol. 12609, pp. 119–134. Springer (2020), https://doi.org/10.1007/978-3-030-68487-7_8
18. Schröder, R.L., Gast, S., Guo, Q.: Divide and Surrender: Exploiting Variable Division Instruction Timing in HQC Key Recovery Attacks. In: Balzarotti, D., Xu, W. (eds.) 33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia,

PA, USA, August 14-16, 2024. USENIX Association (2024), https://www.usenix.org/conference/usenixsecurity24/presentation/schr\%C3\%B6der
19. Shor, P.W.: Algorithms for Quantum Computation: Discrete Logarithms and Factoring. In: 35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico, USA, November 20-22, 1994. pp. 124–134. IEEE Computer Society (1994), https://doi.org/10.1109/SFCS.1994.365700
20. Velek, P., Rabas, T., Bucek, J.: Simple Power Analysis of Polynomial Multiplication in HQC. CoRR **abs/2601.07634** (2026), https://doi.org/10.48550/arXiv.2601.07634