BIKE Key-Recovery: Combining Power Consumption Analysis and Information-Set Decoding

Agathe Cheriere, Nicolas Aragon, Tania Richmond, Benoît Gérard

BIKE Key-Recovery: Combining Power Consumption Analysis and Information-Set Decoding

Agathe Cheriere[1], Nicolas Aragon[2], Tania Richmond[3,4], Benoît Gérard[1,3]

[1] CNRS, IRISA, Univ. Rennes, Inria
263 Avenue Général Leclerc, 35042 Rennes Cedex, France
{agathe.cheriere,benoit.gerard}@irisa.fr
[2] NAQUIDIS Center
1 Rue François Mitterrand, 33400 Talence, France
nicolas.aragon@protonmail.com
[3] DGA - Maîtrise de l'Information,
BP7, 35998 Rennes Cedex 9, France
[4] Institute of Exact and Applied Sciences,
University of New Caledonia,
BP R4, 98851 Nouméa Cedex, France
tania.richmond@unc.nc

Keywords: BIKE · QC-MDPC codes · PQC · Side-Channel Attack · Power Consumption Analysis · Key Recovery · Information-Set Decoding

Abstract. In this paper, we present a single-trace attack on a BIKE Cortex-M4 implementation proposed by Chen *et al.* at CHES 2021. BIKE is a key-encapsulation mechanism, candidate to the NIST post-quantum cryptography standardisation process. We attack by exploiting the rotation function that circularly shifts an array depending on the private key. Chen *et al.* implemented two versions of this function, one in C and one in assembly. Our attack uses subtraces clustering combined with a combinatorial attack to recover the full private key. We obtained a high clustering accuracy in our experiments, and we provide ways to deal with the errors. We are able to recover all the private keys for the C implementation, and while the assembly version is harder to attack using our technique, we still manage to reduce BIKE Level-1 security from 128 to 65 bits for a significant proportion of the private keys.

1 Introduction

The currently used public-key cryptography is based on number theory problems, such as integer factorisation for RSA [RSA78]. In 1994, Shor proposed a quantum algorithm to solve the integer factorisation problem in polynomial time [Sho97]. Therefore if a large-scale quantum computer was built, it could break all cryptosystems that rely on this problem. For this reason, the National Institute

of Standards and Technology (NIST) is running a Post-Quantum Cryptography (PQC) standardisation process to select the next encryption and digital signature standards for the quantum era.

BIKE [AAB+21] is one of the Key-Encapsulation Mechanisms (KEM) based on coding theory that was recently selected to the fourth round of the standardisation process. Recently, BIKE received an increased focus from the community regarding its side-channel resilience. The BIKE specification includes a constant-time implementation from [DGK20], protected against timing and cache attacks. However this implementation does not provide resistance against other side channels, such as power consumption analysis, although there are multiple attacks of this type targeting the code family used in the scheme, but not against the scheme itself.

Previous works. Many code-based schemes leverage a low- or moderate-density parity-check codes to obtain better performances. Among them, BIKE is based on binary Quasi-Cyclic Moderate-Density Parity-Check (QC-MDPC) codes which were first proposed for cryptographic purposes in [MTSB13]. Those sparse parity-check codes share a similar iterative decoding algorithm that suffers from timing variations. A first constant-time implementation was proposed by Chou for QcBits in [Cho16]. This one was attacked by a differential power analysis (DPA) on the syndrome computation during the decryption [RHHM17]. In a more recent work, Sim *et al.* improved this DPA attack and also proposed a single-trace attack targeting the same weakness [SKC+19]. Authors discuss the applicability of this later attack to two schemes including BIKE. Two other implementation works have then been published targeting BIKE which was selected by the NIST as a third round alternate candidate in the meantime. Drucker *et al.* proposed a portable C constant-time implementation adapted for 64-bit ARM microcontrollers [DGK20]. This work was used as a basis for the Cortex-M4 optimised implementations by Chen *et al.* [CCK21]. One of the key operations in the decoding of QC-MDPC codes is the computation of unsatisfied parity-check equations. It is usually done by computing circular shifts of the syndrome: this is the operation we target in our side-channel analysis. From this perspective, the implementation from Chen *et al.* differs from the previous ones and this motivated our choice to target it (details are provided in Section 3). Moreover, two versions of this syndrome rotation are proposed: one in C and an optimized assembly one which is an interesting challenge.

Our contribution. In this paper, we go further in the study of resistance of the QC-MDPC decoding implementations against power analysis attacks. We propose a single-trace attack combining unsupervised machine learning with a combinatorial attack against the syndrome rotation in the BIKE Cortex-M4 implementations from [CCK21]. To the best of our knowledge, it is the first full key-recovery attack against the latter, either in C or in assembly. And in a more general way, it is the first attack exploiting a leakage from an assembly instruction on the QC-MDPC decoding algorithm. We provide practical results of our

attack against both the C and assembly versions. We also present a countermeasure in C to improve the side-channel resistance of such implementations.

Organisation of the paper. In Section 2, we recall background of coding theory in the Hamming metric and on QC-MDPC decoders, as well as the BIKE scheme. In Section 3, we present the weakness we target and an outline of our attack. In Section 4, we provide practical results for our attacks with details on adaptations to C and assembly. Finally, we propose a countermeasure in Section 5 and conclude this paper in Section 6.

2 Preliminaries

In this section we recall some background on coding theory and present the BIKE cryptosystem [AAB+21] as well as MDPC codes with their decoding algorithm.

2.1 Coding Theory

Definition 1. *Linear codes. A linear code $\mathcal{C}$ over $\mathbb{F}_q$ of length n and dimension k is a vector subspace of $\mathbb{F}_q^n$ of dimension k.*

Such a code can be represented in two equivalent ways:

- *either by a generator matrix $G \in \mathbb{F}_q^{k\times n}$ where each row of G is an element of a basis of $\mathcal{C}$,*
$$\mathcal{C} = \{mG | m \in \mathbb{F}_q^k\}.$$
- *or by a parity-check matrix $H \in \mathbb{F}_q^{(n-k)\times n}$ such that H is full rank and, for each $c \in \mathcal{C}$:*
$$Hc^\intercal = 0.$$

Code-based cryptography is based on the difficulty of decoding random error-correcting codes, a well known NP-complete problem [BMVT78]:

Definition 2. *Syndrome decoding (SD) problem. Given a parity-check matrix $H \in \mathbb{F}_q^{(n-k)\times n}$, a syndrome $s \in \mathbb{F}_q^{n-k}$, and a positive integer t, the syndrome decoding problem consists in finding an error vector $e \in \mathbb{F}_q^n$ such that:*

- *$He^\intercal = s$,*
- *e is of Hamming weight t.*

In this work we focus on codes with coefficients in $\mathbb{F}_2$, associated with the Hamming metric. The BIKE cryptosystem takes advantage of the structure of circulant matrices and quasi-cyclic codes, as follows.

Definition 3. *Circulant matrices. An $r \times r$ square matrix M is a circulant matrix if it is of the form:*

$$M = \begin{pmatrix} m_0 & m_1 \dots & m_{r-1} \\ m_{r-1} & m_0 & \ddots & m_{r-2} \\ \vdots & \ddots & \ddots & \vdots \\ m_1 & m_2 \dots & & m_0 \end{pmatrix}$$

We say that M is generated by the vector $m = (m_0, \dots, m_{r-1})$.

There exists an isomorphism between the ring of polynomials $\mathcal{R} = \mathbb{F}_2[X]/(X^r - 1)$ and set of circulant $r \times r$ matrices. Operations on matrices (multiplication and inversion in particular) can thus be performed using polynomials in the ring $\mathcal{R}$: to a vector $m = (m_0, \dots, m_{r-1}) \in \mathbb{F}_2^r$ generating a circular matrix, we can associate the polynomial $M(X) = \sum_{i=0}^{r-1} m_i X^i$.

Definition 4. *Quasi-Cyclic codes. An $[sn, k]$ linear code $\mathcal{C}$ is quasi-cyclic (QC) of index s if, for any codeword $c = (c_1, \dots, c_s) \in (\mathbb{F}_2^n)^s$ in $\mathcal{C}$, the vector obtained after applying a circular shift to every block c_i is also a codeword.*

In the following we focus on $[2r, r]$ QC codes: let H be a parity-check matrix of such a code, then it can be represented by a parity-check matrix $H = (H_0 | \ H_1)$, where H_i is a circulant $r \times r$ matrix.

Definition 5. *Quasi-cyclic moderate density parity-check (QC-MDPC) codes. An $[n, r, w]$ QC-MDPC code $\mathcal{C}$ is a quasi-cyclic code that admits a parity-check matrix H such that H has a constant row weight $w = \mathcal{O}(\sqrt{n})$.*

BIKE relies on $[n, r, w]$ QC-MDPC codes, with $n = 2r$. Parity-check matrices are thus represented by two vectors h_0 and $h_1 \in \mathbb{F}_2^r$ of weight $d = \frac{w}{2}$ each.

2.2 BIKE Scheme

BIKE is an alternate candidate to the third round of the NIST post-quantum standardisation process, which moves to the fourth round. We describe the version of the scheme presented in the third round submission package, which was labelled BIKE-2 in previous rounds. **H**, **L** and **K** denote hash functions, and **Decoder** the decoding algorithm described in Section 2.3. In these algorithms, l is the shared secret size.

KeyGen(l)**:**
- Pick $\sigma \xleftarrow{\$} \{0,1\}^l$
- Pick $(h_0, h_1) \xleftarrow{\$} \mathcal{R}^2$ s.t. $|h_0| = |h_1| = \frac{w}{2}$
- Compute $h \leftarrow h_1 h_0^{-1}$
- Return the private key (h_0, h_1, σ) and the public key h

Encap(h, l)**:**
- Pick $m \xleftarrow{\$} \{0,1\}^l$

- Compute $(e_0, e_1) \xleftarrow{\$} \mathbf{H}(m)$
- Compute $(c_0, c_1) \leftarrow (e_0 + e_1 h, m \oplus \mathbf{L}(e_0, e_1))$
- Compute the shared secret $\mathcal{K} \leftarrow \mathbf{K}(m, c_0, c_1)$
- Return the ciphertext (c_0, c_1)

Decap$((h_0, h_1, \sigma), (c_0, c_1))$:

- Compute $e' \leftarrow \mathbf{Decoder}(c_0 h_0, h_0, h_1)$
- Compute $m' \leftarrow c_1 \oplus \mathbf{L}(e')$
- If $e' = \mathbf{H}(m')$ then return $\mathcal{K} \leftarrow \mathbf{K}(m', c)$
- Else return $\mathcal{K} \leftarrow \mathbf{K}(\sigma, c)$

Table 1 contains the parameters for the three security levels of BIKE scheme (length of the code and row weight of the parity-check matrix). The decapsulation procedure relies on the decoding algorithm of the QC-MDPC codes.

Security	r	w
Level-1	12323	142
Level-3	24659	206
Level-5	40973	274

Table 1: BIKE parameters

2.3 Decoding MDPC Codes

While the BIKE scheme can be instantiated using any decoder for the MDPC codes, the choice of the decoding algorithm has an impact on the decoding failure probability, and potentially on the security of the scheme. To meet the security requirements, the authors proposed to use the Black-Gray-Flip (BGF) algorithm as decoder [AAB+21,DGK20]. The BGF algorithm is a variant of the iterative bit-flipping algorithm [Gal62] that was originally described as a decoder for Low-Density Parity-Check codes.

The principle of the bit-flipping algorithm is as follows: in each iteration, the number of unsatisfied parity-check equations is computed for each bit of the error vector. If this number is greater than a threshold value T, then the corresponding bit is flipped and the syndrome is recomputed. This procedure is described in Algorithm 1. The function COUNTER takes as input the parity-check matrix H and the syndrome s, and returns an array containing the number of unsatisfied parity-check equations (in other words, it counts for each column of H the number of ones that appear in the same position in this column and in the syndrome). The function THRESHOLD returns the threshold associated to an iteration. This value can be computed in various ways, the BIKE scheme uses precomputed values that are fixed for each iteration, which can be found in the BIKE specification [AAB+21].

Input: s, H, a number of iterations N
Output: e
$e \leftarrow 0^n$
for $iteration = 1, \ldots, N$ **do**
  $T \leftarrow \text{THRESHOLD}(iteration)$
  $count \leftarrow \text{COUNTER}(H, s)$
  for $i = 0, \ldots, n-1$ **do**
    if $count[i] \geq T$ **then**
      $e[i] \leftarrow e[i] \oplus 1$
      $s \leftarrow s \oplus H[:, i]$
return e

Algorithm 1: Bit-flipping algorithm. $H[:, i]$ denotes the i-th column of H.

In order to reduce the decoding failure rate of this algorithm, BIKE uses the BGF. During the first iteration, the BGF classifies the coordinates of the error as *black* or *gray* using two different thresholds. It then performs two additional iterations to confirm (or not) the choices that were made during the classification. The algorithm is described in Algorithm 2, see [AAB+21] for more details.

The BIKE scheme sets the number of iterations N to 5, which leads to 7 computations of unsatisfied parity-check equations, due to the structure of the BGF algorithm: one in each call to the BFIter procedure, and two additional ones for the iteration where the *black* and *gray* values are processed.

2.4 Optimising the Bit-Flipping Algorithm

As shown in [Cho16], it is possible to take into account the fact that the considered matrices are circulant matrices to optimise the bit-flipping algorithm, by computing the number of unsatisfied parity-check equations as a multiplication in the ring $\mathbb{Z}[X]/(X^r - 1)$. Let u be the vector such that u_j represents the number of unsatisfied parity-check equations for the $j - th$ column. Then this vector u can be computed as $(h_0 \cdot s, h_1 \cdot s)$. Note that both h_0 and h_1 have a low Hamming weight, hence the cost of computing the number of unsatisfied parity-check equations at each iteration of the algorithm is reduced to computing two sparse-dense polynomial multiplications in the ring $\mathbb{Z}[X]/(X^r - 1)$.

Chou proposed a method to perform these operations efficiently in [Cho16], later on reused in [CCK21]. Let f be a dense polynomial and g be a sparse polynomial, represented as an array of coordinates $I = \{i | g_i = 1\}$. Then,

$$fg = \sum_{i \in I} X^i f,$$

where each $X^i f$ can be computed as a cyclic shift of the polynomial f. Each integer $i \in I$ is encoded on j bits, where $j = \lceil \log_2(r) \rceil$: $i = (b_{j-1}, b_{j-2}, \cdots, b_1, b_0)_2$. To compute the sparse-dense polynomial multiplication of the syndrome s with a secret value h_i, the cyclic shift of s is performed $\frac{w}{2}$ times.

Input: s, H, a number of iterations N
Output: e

$e \leftarrow 0^n$
for $iteration = 1, \ldots, N$ **do**
  $T \leftarrow$ THRESHOLD$(iteration)$
  $e, black, gray \leftarrow$ BFITER$(s + eH^t, e, T, H)$
  if $iteration = 1$ **then**
    $e \leftarrow$ BFMASKEDITER$(s + eH^t, e, black, (d+1)/2 + 1, H)$
    $e \leftarrow$ BFMASKEDITER$(s + eH^t, e, gray, (d+1)/2 + 1, H)$
return e

`procedure` BFITER(s, e, T, H)
$count \leftarrow$ COUNTER(H, s)
for $i = 0, \ldots, n-1$ **do**
  if $count[i] \geq T$ **then**
    $e[i] \leftarrow e[i] \oplus 1$
    $black[j] \leftarrow 1$
  else if $count[i] \geq T - \theta$ **then**
    $gray[j] \leftarrow 1$
return e, $black$, $gray$

`procedure` BFMASKEDITER(s, e, mask, T, H)
$count \leftarrow$ COUNTER(H, s)
for $i = 0, \ldots, n-1$ **do**
  if $count[i] \geq T$ **then**
    $e[i] \leftarrow e[i] \oplus mask[i]$
  end
return e

Algorithm 2: BGF algorithm

The fact that the secret nonzero coordinates of h_i are manipulated during the computation of the unsatisfied parity-check equations is the key for our attack to succeed. In the next section we describe how this rotation operation is implemented in [CCK21] and how the countermeasures against cache and timing attacks allow us to recover information using a side-channel attack by power analysis.

3 Overview of our Attack

The purpose of constant-time implementation is to remove the dependence between the execution time of a program and the secret values it manipulates. Chen *et al.* [CCK21] proposed an optimised and constant-time implementation of the syndrome rotation for Cortex-M4, which is itself based on the portable implementation proposed by Drucker *et al.* [DGK20].

3.1 Constant-Time Syndrome Rotation for Cortex-M4

We consider the Cortex-M4 as a target, which is a 32-bit ARM embedded processor. Most computations such as arithmetic or logical operations are thus made on 32-bit registers. Consequently, in the considered BIKE implementation, the syndrome is stored as an array of integers of size $\lceil \frac{r}{32} \rceil$, and the secret vectors h_0 and h_1 are stored as two arrays of $\frac{w}{2}$ each, where each element of the array is the position of a nonzero coordinate in the vector. We now describe the techniques used in [CCK21] to compute the syndrome rotations in constant time. We provide in Algorithm 3 a description of the cyclic shift of the syndrome s by ind positions. ind is an integer encoded as $(b_{j-1}, b_{j-2}, \cdots, b_1, b_0)_2$. The algorithm is split into two parts: first, the high order bits of ind (from b_{j-1} to b_5) are processed, to perform a shift of $ind - (ind \mod 32)$ bits. Then the remaining bits (b_4 to b_0) are processed to perform a shift of $(ind \mod 32)$ bits.

Input: s, ind
Output: Cyclic shift of s by ind bits
1 $dupS \leftarrow (s|s)$
2 $(b_{j-1}, b_{j-2}, \ldots, b_1, b_0) \leftarrow (ind)_2$
`// Process most significant bits`
3 **for** $elt = j-1, \cdots, 5$ **do**
4 **if** $b_{elt} == 1$ **then**
5 **for** $i = 0, \cdots, 2 \times \lceil \frac{r}{32} \rceil - 2^{elt-5}$ **do**
6 $dupS[i] \leftarrow dupS[i + 2^{elt-5}]$
`// Process less significant bits`
7 $shift \leftarrow (b_4, \ldots, b_0)$
8 **for** $i = 0, \ldots, i < \lceil r/32 \rceil$ **do**
9 $dupS[i] = (dupS[i] >> shift)|(dupS[i+1] << shift)$
10 $s \leftarrow dupS[0:r]$

Algorithm 3: Non constant-time rotation for Cortex-M4

Algorithm 3 is not in constant time due to the conditional branching, line 4, which depends on the bits values. One way to remove this conditional branching is to replace it by an operation such as:

$$s[k] = (s[k] \wedge \neg mask) \oplus (s[k + 2^{elt-5}] \wedge mask) \tag{1}$$

where $mask$ is an unsigned 32-bit word whose value is determined by the bit b_{elt} as follows:

$$mask = -b_{elt} = \begin{cases} \texttt{0xFFFFFFFF} & \text{if } b_{elt} == 1 \\ \texttt{0x00000000} & \text{if } b_{elt} == 0 \end{cases}$$

Thus, if the bit b_{elt} is at 1 then $s[k]$ takes the value $s[k + 2^{elt-5}]$, otherwise there is no modification. Operation (1) is targeted by Sim *et al.* in the QcBits

implementation [SKC+19]. Drucker *et al.* replaced the XOR in Operation (1) by a logical OR, c.f. [DGK20], giving Operation (2):

$$s[k] = (s[k] \wedge \neg mask) \vee (s[k + 2^{elt-5}] \wedge mask) \tag{2}$$

However, by studying the C code of [CCK21] we noticed that Operation (1) is implemented as:

$$s[k] \oplus = (s[k] \oplus s[k + 2^{elt-5}]) \wedge mask. \tag{3}$$

In this work we target both implementations of Operation (3) from [CCK21], one in plain C and the other one leveraging the SEL assembly instruction.

3.2 Determine Bit Values

We want to recover the coordinates of the private key (h_0, h_1). Those coordinates are decomposed in binary representation to shift the syndrome thanks to Operation (3) or the SEL instruction. To obtain the values of the bits for each index $ind = (b_{j-1}, b_{j-2}, \ldots, b_0)$, we exploit leakages of information embedded in power traces. We first need to identify time features corresponding to the syndrome rotation. Second, we also need to separate the execution for the bits from $(b_{j-1}, \ldots, b_5)$ from the last 5 bits. Once it is done, we can determine the masks values with a clustering algorithm and by consequence the bits values. Clustering is an unsupervised machine-learning method that interprets the input data and finds natural groups called clusters. So the subtraces containing the syndrome rotation can be sorted into clusters under the condition that, at some points, there are differences between traces. To obtain their values, we need to execute a clustering for each bit with only the time features corresponding. Thus, the clustering algorithm will return two clusters, one with the bits at 1 and one with those at 0. There are many clustering algorithms: we used the most known namely the k-means algorithm [Mac67] and obtained good enough results. More advanced clustering algorithms may slightly improve the attack but this is left for further research. The k-means algorithm partitions a set of points (resp. vectors) into k groups with the objective of minimizing the distance between the points (resp. vectors) in each group and the different means of the groups. The algorithm is repeated until it converges or if a maximum number of iterations, fixed in advance, is reached.

Algorithm 4 presents the different steps of recovering a bit b_{elt}. We do not use k-means directly with the subtraces. We first rescale the traces using their standard deviation to help the clustering step (Algorithm 4, line 1). The k-means algorithm, line 2, uses the Euclidean distance on the set of rescaled traces to cluster and returns a label for each one, either 0 or 1, and a centroid for each cluster (a.k.a the means). The last step is to determine the value of the bits for each cluster. We noticed that the power consumption is higher for the bits at 1 than for the ones at 0. Hence, we take the maximum values in both centroids and compare them to decide if we need to permute the labels, line 3. The permutation, line 4, is just the action to change the label 1 into label 0 and

Input: $traces$ for b_{elt}, $iter$
Output: Values of b_{elt}
$traces_r \leftarrow rescale(traces)$
$labels, centroid \leftarrow kmeans(traces_r, 2, iter)$
if $max(centroid[0]) > max(centroid[1])$ **then**
    $labels \leftarrow Permute(labels)$
return labels

Algorithm 4: Bits b_{elt} Recovery

vice versa. Then Algorithm 4 returns the labels which correspond to the values of the bits.

From this technique, we theoretically obtain up to $j - 5$ bits values for each coordinate. Thus we have partially recovered the private key (h_0, h_1). The last step is to finish the attack by doing a full key-recovery attack thanks to a mathematical approach.

3.3 Information Set Decoder

Using our side-channel attack, we extract some information about the secret vectors h_0 and h_1. Each h_i is represented as an array of $\frac{w}{2}$ integers which are the positions of the nonzero coordinates in h_i. Each coordinate h_{ij} is encoded by $l = \lceil log_2(r) \rceil$ bits, and the information we got from the side-channel analysis is a subset of these l bits. In other words, this means that for each h_{ij} we know a subset L of $\{1, \ldots, r\}$ such that $h_{ij} \in L$. The size of L depends on the number of recovered bits: more information we recover, smaller L is. Our goal is therefore to get the smallest search space L from the side-channel analysis to ease our attack.

We explored two ways of performing the ISD: one that we call classical Prange and the approach from [HPR$^+$21]. More details about the complexity analysis for both of these methods can be found in Appendix A. Results are given in Figure 1.

As we can see, using the approach from [HPR$^+$21] allows us to theoretically reduce the complexity of our attack. Nevertheless, we will need in practice to recover half of the bits representing each coordinate for the attack to be feasible in a reasonable time, which corresponds to a probability of success close to 1 for both approaches. The theoretical gap we could gain by using the second approach is not verified in practice. For this reason, in the rest of the paper we compute the attack complexities given by the classical Prange algorithm (*i. e.* the first approach). Nonetheless, in the cases where the success probability gets low, the more advance approach from [HPR$^+$21] can be used to reduce the overall complexity.

Remark 1 *We also tried to adapt the approach from [RHHM17] by taking into account the fact that we have information on the nonzero coordinates, not only in h_0 but also in h_1. However, it led to higher complexities than the classical Prange algorithm.*

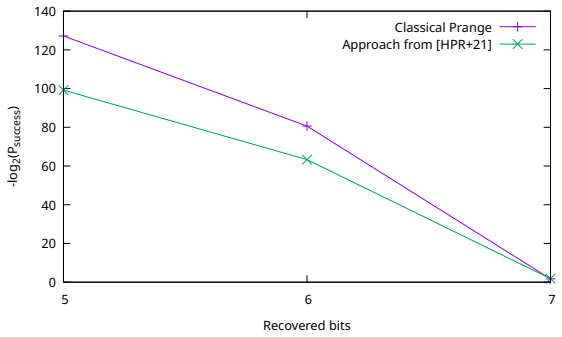


Fig. 1: Probability of success of the ISD algorithms depending on the number of recovered bits from our side-channel analysis.

To compute the actual complexity of the Prange algorithm, we need to take into account the cost of linear algebra, which is not negligible in our case. This cost is equal to the cost of inverting an $r \times r$ matrix over $\mathbb{F}_2$, i.e r^ω bit operations. In our computations we use the value $\omega = 2.8$.

4 Experimentation on the Cortex-M4

The necessary material for reproducing our experiments have been made available on GitHub [CARG].

We performed experiments on both the plain C implementation (referred to as C implementation) and the implementation leveraging the SEL instruction (referred to as assembly implementation in the following). Both were made by running the implementation provided on PQM4 GitHub [CCK21]. Implementations for Level-1 and Level-3 are available, and we chose to test the attack strategy previously described on the Level-1 parameters. Nonetheless, as the rotation function works using the same method for both sets of parameters, we can assume that the attack will work for Level-3 parameters despite an extra bit. Since the BIKE scheme can be used with ephemeral keys, we provide experimental results using a single trace of power consumption.

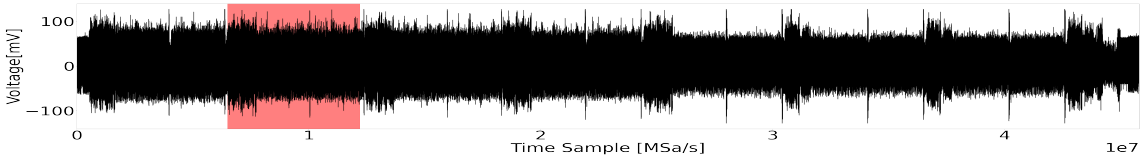
4.1 Measurement Setup

The experimentation setup was made up of a Chipwhisperer STM32F4 based on a CW308 UFO Board. The board was connected to a computer using an ST-LINK/V2 for debugging purposes. Power consumption was measured through an oscilloscope with a 3GHz bandwidth. We took traces of the full decapsulation process for both implementations using the same oscilloscope configuration except for the acquisition sampling rate. In fact, the execution of the assembly implementation takes less time than the C implementation, so it allows us to use

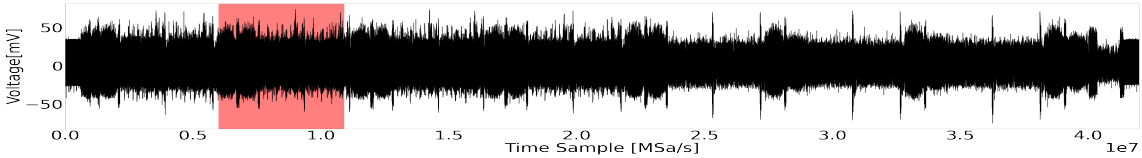
a higher sampling rate. The Cortex-M4 was flashed with files generated using the GCC compiler with the optimisation flag set to -Og. For our experimentation, we generated valid keys (resp. ciphertexts) using the key generation (resp. encapsulation) function provided in the implementations, and passed these values as inputs to the decapsulation function.

Figure 2 shows power consumption traces for the whole decapsulation process. The first trace, Figure 2a, is for the C implementation and the one below, Figure 2b, is for the assembly implementation. One thing to notice is that the scale for the voltage is not the same for both traces: the power consumption is significantly lower for the assembly one (approximately half of the one for the C trace). Nevertheless, both traces follow the same general pattern which can be cut into seven smaller patterns, highlighted by a red rectangle in Figure 2.

Remark 2 *For each iteration we can observe an overall drop of the power consumption as the iterations progress. It is directly linked to the decreasing of the syndrome weight processed at each iteration.*



(a) C implementation



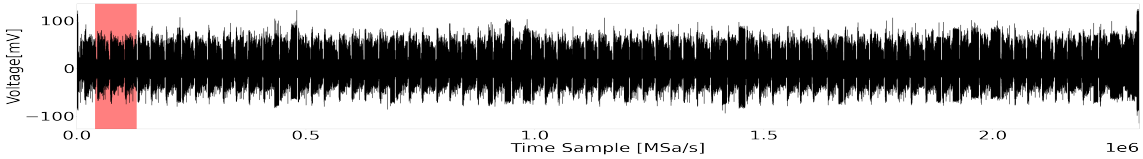
(b) Assembly implementation

Fig. 2: Full traces of the decapsulation execution, respectively for the C and the assembly implementation. One iteration of syndrome rotation is highlight in red.

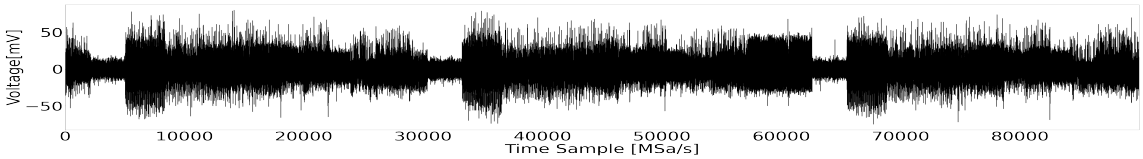
The BIKE specification [AAB+21] gives us information to determine relations between different parts of the traces and the decoding algorithm. In fact, the number of iterations in the Bit-Flipping algorithm is fixed to 5 by the authors. By adding the two iterations generated by the Black-Gray part of the algorithm (Subsection 2.3), we obtain a correspondence between the number of patterns and the number of iterations in the algorithm: one of these patterns is highlighted in red in Figure 2. In each pattern, the first part is the syndrome computation, and the two other parts (more visible at the end of the traces, where the consumption is overall lower) correspond to the syndrome rotation (the operation we are targeting) and to the computation of the number of unsatisfied parity-check equations. In the rest of this section, full syndrome rotation

will refer to the execution of the rotations for all the coordinates, and syndrome rotation will refer to the rotation of the syndrome for one specific coordinate.

The traces display in Figure 3 show, respectively, a full syndrome rotation in Figure 3a and a zoom on three syndrome rotations in Figure 3b for the C implementation. On Figure 3a, 71 blocks can be counted, which exactly corresponds to the number of coordinates in h_i. The syndrome rotation function always takes the same amount of time, mandatory to remove time dependence. Figure 3b shows that each syndrome rotation is isolated from the others, thus the beginning of the execution can be easily determined.



(a) Syndrome rotation for all h_i coordinates



(b) Zoom on three coordinates of h_i

Fig. 3: Syndrome rotations for h_i with a zoom on three coordinates.

The implementation structure of the syndrome rotation makes the distinction between the bits easier. Indeed, the treatment of the syndrome for each bit is composed of two **for** loops, the outer and the inner. The inner **for** loop is executed t times, the value of t changes for each bit b_i so there is a specific pattern for each b_i. Those specific patterns help to determine which bit is treated at one instant of the trace. We used them to cut the traces into subtraces corresponding to only one bit b_i, the one we want to recover.

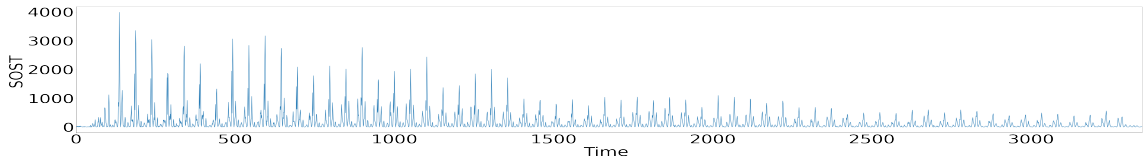
In assembly traces, we find the same structural characteristics with the blocks and a specific pattern for each b_i.

Figure 4 displays the points of interest (PoI) for the section of the traces corresponding to b_{13}. They were computed by using the sum of squared pairwise t-differences T-test (SOST) [GLRP06]:

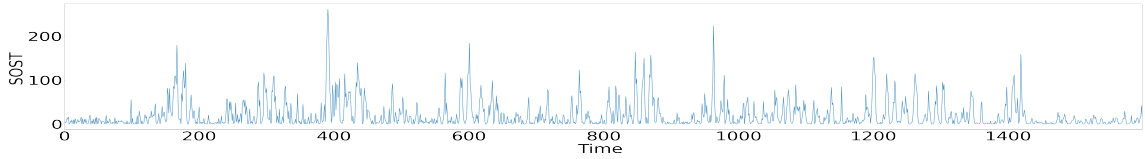
$$\left(\frac{E(T_0) - E(T_1)}{\sqrt{\frac{\sigma(T_0)^2}{\#T_0} + \frac{\sigma(T_1)^2}{\#T_1}}} \right)^2 \tag{4}$$

T_l is the mean of the traces with the value of bit at l.

In Figure 4, the PoIs for C traces are numerous while assembly traces have noticeably less PoIs, and the SOST values are much lower, which explains why



(a) SOST for the C implementation



(b) SOST for the assembly implementation

Fig. 4: SOST for b_{13} subtraces, for a C trace and an assembly trace.

we can not recover the mask values with the exact same method for both implementations. Both approaches are detailed in the following subsections.

Remark 3 *The optimisation flags -O0 and -O3 were also tested for both implementations to have an idea of their impact on the power consumption of the function. On a general aspect, the traces are following the pattern displayed in Figure 2. The execution time is obviously different, either longer or shorter, but the extraction of the information is also possible.*

4.2 Attack on the Plain C Implementation

We first recall how the plain C implementation performs its conditional assignment in order to shift the syndrome:

$$Rx \oplus = (Rx \oplus Ry) \wedge mask \tag{5}$$

Looking at this operation, there are two possible PoI to detect leakages. More specifically, either the final XOR (denoted $\oplus$) between Rx and the result of $(Rx \oplus Ry) \wedge mask$, or the logical AND (denoted $\wedge$) could leak information about the mask value, as the other XOR (*i. e.* $Rx \oplus Ry$) is not impacted by the mask. Furthermore, both logical operations correlated with $mask$ can leak at the same time. Figure 4 shows that, at least for the bit b_{13}, the XOR and AND leak even though we cannot determine which one (or both) leaks information. Figure 4a shows the leakages instant. However, we detected that an iteration of the syndrome rotation for a zero syndrome (i.e a zero vector) has higher leakage than for a nonzero syndrome. The SOST of the former in Figure 5 is five times higher than the one for the latter. The iteration with zero syndrome often occurs twice at the end of the decapsulation process, so we will use these last two executions to perform our attack. And as we use the k-means algorithm, the greater the difference is, the better its efficiency is. Thus we focus our attack on the exploitation of the last two iterations.

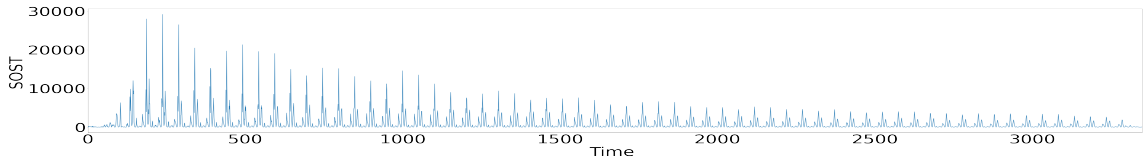
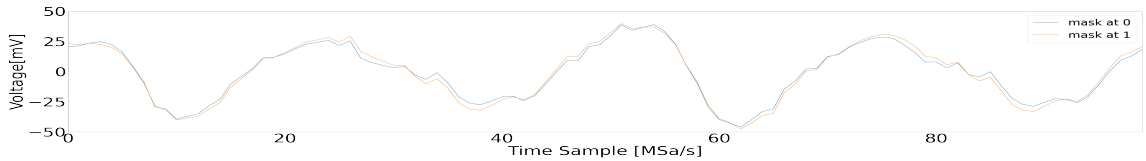


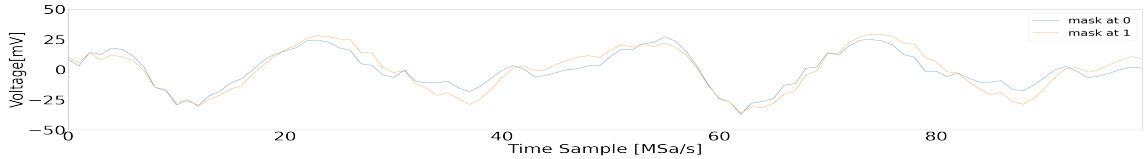
Fig. 5: SOST for zero syndrome subtraces for C implementation

Remark 4 *In Figure 5 and 4a, the values decrease with the time due to the desynchronisation of the subtraces. Those are very small discrepancies that occur in each trace.*

Leakage Exploitation We show in Figure 6 two comparisons between means of two groups of traces, either for masks at 1 or at 0. Figure 6a is the comparison for nonzero syndrome traces and Figure 6b for zero syndrome traces (with 2840 traces in both groups). Both mean traces have some points of divergence at the same instant, however the difference between means is greater in the second one (for zero syndrome). We guess that it is due to the reduction of noise generated by the logical operations. Indeed, as Rx and Ry in Operation 5 are at 0, no bit is flipping from 0 to 1 nor from 1 to 0 on them. For instance, the two XORs return 0 independently of $mask$ value, as Rx, Ry, and $(Rx \oplus Ry) \wedge mask$ are equal to 0. Thus, there is less power consumption by the operations unrelated to the $mask$. We conclude, in this specific case, that the logical AND is leaking as it is the only operation manipulating $mask$.



(a) Nonzero syndrome traces



(b) Zeros syndrome traces

Fig. 6: Extract from the comparison between the trace means for the 1 and 0 masks, for nonzero syndrome first, and zero syndrome secondly.

Computing the SOST gives the PoIs which are the exact instants where the difference between traces should be the highest. The bigger the difference is, the

easier it will be for the classification algorithm to correctly separate traces in two clusters. Indeed, as the algorithm bases its classification on the Euclidean distance between the means and the traces in the cluster, if the distance between the two means is important (when choosing a good PoI) the number of errors is smaller. In Table 2 we give the accuracy of the k-means algorithm for each bit. Those accuracies were computed by executing the k-means algorithm under the condition of a single-trace attack, which signifies that the groups of traces to sort into two clusters are composed of $2 * w = 2 * 142 = 284$ subtraces corresponding to the last two iterations. The accuracy was computed by running the algorithm 100 times on the same groups and also for various full traces. The results shown in Table 2 is the average of all the computed accuracies.

Bit	13	12	11	10	9	8	7	6	5
Clustering accuracy	0.974	0.987	0.972	0.985	0.962	0.983	0.985	0.504	0.536

Table 2: Average k-means accuracy

Clustering Accuracy In Table 2, we can notice that for the bits from b_{13} to b_7 accuracies are high. But, curiously for the last two bits, b_6 and b_5, their accuracies are hardly higher than 0.5. The difference with the other bits can be explained by the fact that the number of shifts executed in the inner **for** loop is different. In fact, for b_6 (resp. b_5) the number is divided by two (resp. four). In other words, when there are 8 operations for bits from b_{13} to b_7 in the inner **for** loop, there is only 4 for b_6 and 2 for b_5. This explains the observed difference on multiple PoIs. For instance, b_6 leaks only in one time sample and with less amplitude than previous bits. Focusing on this single leaking point, the accuracy moves from 0.5 to 0.92.

Our goal is to recover the bit values with as few errors as possible. So, to increase the success rate in our bits detection, we proceeded in three steps. Firstly, we ran l times the k-means algorithm ($l = 50$ for our different tests) with the same subgroup of traces. For each bit, we reduce the subtraces to an interval of 20 samples (except for b_6 for which we focused on a single sample) centered to the highest PoIs of the SOST computed: ideally we would like to consider less samples to precisely target the PoIs, but using a few samples allows to reduce the impact of the noise. We obtain the best results by taking ten samples from each side of the highest PoIs. The second step is to look at the l results returned by k-means, gather identical results into groups, and count the number of occurrences of each group. We select the result with the highest occurrence as labels. We chose the value of l that led to the best experimental results.

The last step is to identify the potential errors. We applied a maximum likelihood strategy. Since we ran the k-means with the subtraces for the two last iterations, we had at the end two labels for each bit. If both labels were identical, then we supposed that it was the right value, otherwise the k-means algorithm

failed and we considered this as a clustering error. During our experiments, we never encountered the case where both labels were wrong, so we do not treat this case. Our technique returned between 0 and 3 errors on the $7 * 284 = 1988$ labeled subtraces for bits from b_{13} to b_7. For bit b_6, there are between 15 and 20 errors in the labelisation of the 284 associated subtraces. It is due to the fact that the subtraces consist of a single leaking point, as previously discussed.

We will use an ISD algorithm to finish the attack, and as we will see, recovering the values of the bits from b_{13} to b_6 is enough to recover the secret using a single iteration of the algorithm. Hence, recovering b_5 is not necessary since it would not make the ISD algorithm run faster.

At this stage of the attack, the coordinates are partially recovered. We have the values of half of the bits, from b_{13} to b_7 with potentially up to 3 errors, as well as the values for the bit b_6 with up to 20 errors.

Recovering the Remaining Bits First, we assume that there is no error during the classification process, so we correctly recovered the binary representation of each coordinate of h_i, up to the bit b_6 included. To compute the complexity of our attack, we use the discussion from Section 3.3. Basically, for each nonzero coordinate, we have a subset of 64 positions where it might be. So, to create the square submatrix of size r used in the ISD, we can select every columns in the $64 * 142 = 9088$ possibilities where we know nonzero coordinates might be, and complete the information set with $12323 - 9088 = 3235$ other random columns. Since we are guaranteed to have selected every nonzero coordinate in the information set, the algorithm will succeed with probability 1.

To handle the errors that occur during the classification process, we use the fact that we know where these errors occur. Indeed, we use traces from 2 different syndrome rotations for each bit, and if we obtain different labels for this bit then obviously one is wrong and we count it as an error. For example, if we get different labels for the bit b_6 for a coordinate, we select the 128 possible columns as if we did not recover information about b_6. In theory, it could happen that all labels are the same and the information we get is still erroneous, but it never occurred during our experiments.

In our experiments, the total of possible positions did not go higher than r, whether there are errors or not, so the probability of success of the ISD algorithm was always 1.

4.3 Attack on an Assembly Instruction

The assembly implementation replaces the series of AND and XOR of the full C implementation by a unique assembly instruction SEL. This instruction sets the destination register (Rx) to either Ry if the flag is set (a.k.a bit at 1) or Rx if the flag is not set (a.k.a. bit at 0). The leakage exploited for the full C implementation does not exist anymore, but we can still recover the bit values by adapting our strategy.

Selection of the Exploited Subtraces To extract the information about coordinates, we used the same process than previously explained for the C implementation. The main difference is on the subgroup of traces given as input to the k-means algorithm. The traces for the iterations with zero syndromes do not show specific leakages. In fact, using the SEL instruction the main source of leakage comes from bit values flipping in the destination register when the mask is at 1. However in the case of a zero syndrome, there are no bit-flips in the destination register no matter the mask, since both source registers only contain zeros. Therefore we chose to exploit the 4 first bit-flipping iterations that correspond to nonzero syndromes.

Impact of the Syndrome We noticed that the syndrome has a big impact on where the leakages will appear. Indeed, even if in Figure 4 the SOST seems to show PoIs for one trace, by selecting other traces the PoIs in their SOST will appear at other instants on the traces, and the syndrome is the only element which differentiates the different traces used in the SOST computation. The most obvious consequence is that we cannot select one specific instant to classify the traces as it changes depending on the syndrome. We need to take the whole power consumption trace for one bit to make this attack feasible for any syndrome. We noticed that the first three iterations of the BGF decoder often use the same syndrome as input, hence the leakage points are mostly identical for these three iterations. Taking as input for the k-means algorithm the $3 * w$ traces was inconclusive, and the solution we found was to resynchronize and compute the mean of the three subtraces for each bit. Precise resynchronization is critical to our attack because the slightest discrepancy modifies the mean and can lead to a different k-means output. The modification of the syndrome during the decoding process makes the bit-recovery process dependent of the previous bits values. In fact, if the treated bit b_i is 1 then the syndrome will be rotated and thus totally different from the non-rotated syndrome (bit at 0) when treating the next bit b_{i-1}, so the following bits will leak at different temporal instants and will depend on the value of b_i. So, the traces with a bit at 1 need to be treated separately from those with a bit at 0 for the following bits, which divides the traces into more and more subgroups as we progress in the recovery process.

Cluster Management With assembly traces and the method using k-means, we are able to recover the bits from b_{13} to b_9 without any error. Yet, the clusters containing the traces get smaller and smaller as the classification progresses, so for the bit b_9 some clusters were just composed of one or zero trace. Some others contained two traces. We detail how we treat clusters with only a few traces in them.

Cluster with a unique trace. Suppose we have one trace in the cluster, we are unable to determine if the bit b_i is either a 0 or a 1 as we cannot use k-means. And this applies also for the following bits, so we stop the detection process at this bit for these traces.

Two traces in a given cluster. Suppose now that we have two traces in one cluster. Then it is possible that both traces correspond to the same bit value (0 or 1) but the k-means algorithm will distribute them into two clusters regardless, hence we chose to stop our detection process and assume that we can not recover the value of b_i when the cluster contains only two traces.

Bigger cluster with the same mask value. There are some clusters with three to five traces for which the bit b_i has the same value. And it is not possible to determine in advance if we will be in this specific case. However, the impact is not that important because the errors during the classification does not block the bits detection process: it will just slightly increase the ISD complexity.

Starting from bit b_6, the clusters are on average too small to be treated, so we chose to stop our classification problem at the bit b_7.

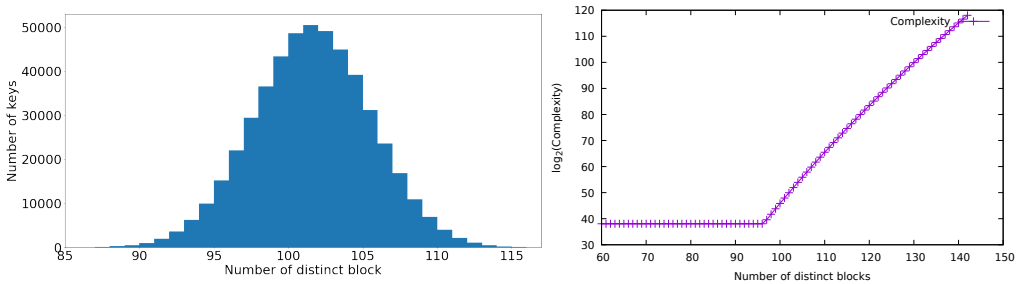
At the end of the classification step, we were able to recover for each coordinate the bits from b_{13} to b_9 without errors. For some coordinates, we could not go further in their recovery but for the majority of them we had the value for b_8 and b_7.

Full Private Key Recovery with the ISD Recovering the private key (h_0, h_1) with the information obtained from the classification for the C implementation is done in polynomial time since we have enough information to reach a probability of success of 1. However, for the assembly one, it is not that straightforward because we can recover less bits.

ISD with half of the bits recovered Let us first suppose that we are in the best case and we are able to recover correctly all the bits from b_{13} to b_7 without errors. Then we have a subset of 128 possible positions for each coordinate. In the worst case this represents $128 * 142 = 18176$ positions, which leads to a complexity of 2^{120}. However, in practice, multiple coordinates can belong to the same subset of 128 positions, which reduces the number of combinations to test and by consequence reduces the complexity of the ISD algorithm.

The diagram on Figure 7a shows the distribution of the number of distinct subsets of 128 positions that contain a nonzero coordinate for 500000 randomly generated BIKE private keys. Figure 7b shows the complexity to recover the private key using the ISD algorithm depending on the number of distinct subsets, using the techniques presented Section 3.3. As we can see, there is a non negligible proportion of private keys straightforward to recover, up to 96 distinct subsets. Then the complexity grows exponentially, and while some keys are really hard to attack using this method, a high proportion can be recovered in practice.

Recovery of the private key with lack of information In practice we do not recover all the information about the bits b_8 and b_7. We thought of two methods to compensate this lost. The first method is to select columns for the information set from bigger subsets, of size 256 if b_7 is missing or size 512 if b_8 is missing.



(a) Distribution of distinct subsets (b) ISD attack complexity

Fig. 7: Distribution of the number of distinct subsets of size 128 containing nonzero coordinates, and the respective attack complexity for 500000 BIKE private keys.

In our experimentation with the generated private keys, the number of missing bits can be as high as 25 which can make the cost of the ISD quite high. Some keys that would have been vulnerable with more information, are now out of our capabilities. The second method is to exhaust all of the 2^k possibilities for the k missing bits, then running the ISD algorithm as if every bit up to b_7 was correctly recovered. If $k = 25$ for example, this leads to an overhead of 25 bits in the complexity of our attack. In particular, for keys where the number of distinct blocks of size 128 containing nonzero coordinates do not exceed 96, our attack still succeeds in less than 2^{65} operations considering 25 unrecovered bits.

5 Countermeasure

As a reminder, the exploited leakage comes from the fact that the logical AND is either done with two 32-bit words at 0 or with one filled with 1. We choose to make a countermeasure that used both words in the operation, with a bit flipping for each part to avoid difference of power consumption.

To do so, we first randomly generate two 32-bit words $Rx2$ and $Ry2$. Then we can compute $Rx1$ (resp. $Ry1$) such that $Rx1 = Rx \oplus Rx2$ (resp. $Ry1 = Ry \oplus Ry2$). The last step before the operation is to redefine Rx such that $Rx = Rx1 \oplus Ry1$. Then the operation can be executed as follows:

$$Rx \oplus = ((Rx1 \oplus Ry2) \wedge mask) \vee ((Rx2 \oplus Ry1) \wedge \neg mask) \tag{6}$$

Operation (3) is rewritten in Operation (6).

The SOST computed for the countermeasure, Figure 8, is not showing any specific points of interest. Additionally, its highest value is 22 times smaller than for the SOST of the assembly implementation, Figure 4b, and 220 times smaller than the one for the C implementation, Figure 4a.

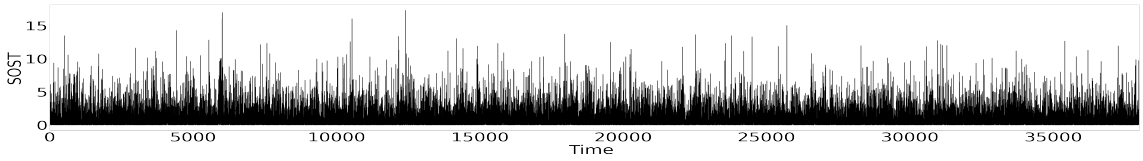


Fig. 8: SOST for the countermeasure

6 Conclusion

We show in this paper that the Cortex-M4 implementation provided in [CCK21] is sensitive to power analysis attack. The full private key can be recovered using a single trace of power consumption of the decapsulation process. The weakness is directly linked to the manipulation of the coordinates in the private key. Given partial information about these coordinates obtained through clustering, we can then use an Information-Set Decoding algorithm and are able to recover the whole private key. Both the C and the assembly implementations are vulnerable to this attack. We recover all the private keys for the C version, and by using some adaptations we recover a high proportion of the private keys for the assembly one. Finally we propose a countermeasure to our attack in C using a binary masking.

In this paper we targeted software implementations of BIKE. However, efficient hardware implementations were recently proposed [RMGS20,RBCGG21]. It would be of interest to assess the potential vulnerability of the proposed attack to those implementations where we expect significantly less leakages from the syndrome rotation operation.

References

[AAB+21] Carlos Aguilar Melchor, Nicolas Aragon, Paulo Barreto, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Shay Gueron, Tim Güneysu, Rafael Misoczki, Edoardo Persichetti, Nicolas Sendrier, Jean-Pierre Tillich, and Gilles Zémor. BIKE. Round 3 Submission to the NIST Post-Quantum Cryptography Call, v. 4.2, September 2021.

[BMVT78] Elwyn Berlekamp, Robert McEliece, and Henk Van Tilborg. On the inherent intractability of certain coding problems (corresp.). *IEEE Transactions on Information Theory*, 24(3):384–386, 1978.

[CARG] Agathe Cheriere, Nicolas Aragon, Tania Richmond, and Benoît Gérard. Github repository sca-bike. https://github.com/benoitgerard/sca-bike.

[CCK21] Ming-Shing Chen, Tung Chou, and Markus Krausz. Optimizing bike for the intel haswell and arm cortex-m4. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 97–124, 2021.

[Cho16] Tung Chou. QcBits: constant-time small-key code-based cryptography. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 280–300. Springer, 2016.

[DGK20] Nir Drucker, Shay Gueron, and Dusan Kostic. QC-MDPC decoders with several shades of gray. In *International Conference on Post-Quantum Cryptography*, pages 35–50. Springer, 2020.

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

[GLRP06] Benedikt Gierlichs, Kerstin Lemke-Rust, and Christof Paar. Templates vs. stochastic methods. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 15–29. Springer, 2006.

[HPR+21] Anna-Lena Horlemann, Sven Puchinger, Julian Renner, Thomas Schamberger, and Antonia Wachter-Zeh. Information-set decoding with hints. In *Code-Based Cryptography Workshop*, pages 60–83. Springer, 2021.

[Mac67] J MacQueen. Classification and analysis of multivariate observations. In *5th Berkeley Symp. Math. Statist. Probability*, pages 281–297, 1967.

[MTSB13] Rafael Misoczki, Jean-Pierre Tillich, Nicolas Sendrier, and Paulo SLM Barreto. Mdpc-mceliece: New mceliece variants from moderate density parity-check codes. In *2013 IEEE international symposium on information theory*, pages 2069–2073. IEEE, 2013.

[RBCGG21] Jan Richter-Brockmann, Ming-Shing Chen, Santosh Ghosh, and Tim Güneysu. Racing BIKE: Improved Polynomial Multiplication and Inversion in Hardware. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(1):557–588, Nov. 2021.

[RHHM17] Mélissa Rossi, Mike Hamburg, Michael Hutter, and Mark E. Marson. A Side-Channel Assisted Cryptanalytic Attack Against QcBits. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems – CHES 2017*, pages 3–23, Cham, 2017. Springer International Publishing.

[RMGS20] Andrew H Reinders, Rafael Misoczki, Santosh Ghosh, and Manoj R Sastry. Efficient bike hardware design with constant-time decoder. In *2020 IEEE International Conference on Quantum Computing and Engineering (QCE)*, pages 197–204. IEEE, 2020.

[RSA78] Ronald L. Rivest, Adi Shamir, and Len Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[Sho97] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509, 1997.

[SKC+19] Bo-Yeon Sim, Jihoon Kwon, Kyu Young Choi, Jihoon Cho, Aesun Park, and Dong-Guk Han. Novel side-channel attacks on quasi-cyclic code-based cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 180–212, 2019.

A Analysis of the ISD Complexity

We first formalize the instance of the syndrome decoding problem we want to solve. Let $\mathcal{C}$ be the code that admits as a generator matrix the public key $(\mathbf{1}, h)$ and let H be a parity-check matrix of this code. By definition we know that the vector (h_0, h_1) is a codeword of $\mathcal{C}$, hence $H(h_0, h_1)^T = 0$. This gives us an instance of the SD problem [2] with H of size $r \times n$ over $\mathbb{F}_2$, where $n = 2r$, that admits the private key (h_0, h_1) of weight w as a solution.

To solve this instance, we use Prange's information set decoding (ISD) algorithm. The principle is as follows: in each iteration, we choose a subset of r columns among n (called an information set), and we solve the linear system $H'e^T = 0$, where H' is the square submatrix of H obtained by extracting the r chosen columns. If all the nonzero coordinates of the desired solution (h_0, h_1) are in the set of the chosen columns, then we obtain a solution to our ISD instance, otherwise we obtain a random solution and try again with another subset. The probability of success is $\frac{\binom{r}{w}}{\binom{n}{w}}$, and this is what we aim to improve using the additional information from our side-channel analysis.

The information we obtain is the same as the hints described in [HPR$^+$21, Section 4]. The only difference being that we use a parity-check matrix instead of a generator matrix: we study how well this algorithm behaves compare to the classical Prange algorithm in our particular case. We use the same definitions as in [HPR$^+$21]: let $\mathcal{W}$ be the set $\{1, \ldots, n\}$ that is partitioned in subsets $\mathcal{W}_i$. We assume that we know the weight of the secret (h_0, h_1) restricted to each subset $\mathcal{W}_i$. Recall that in our case, each nonzero position of (h_0, h_1) is encoded by $l = \lceil log_2(r) \rceil$ bits. If we suppose that we recover the l_1 most significant bits of each position, then the set $\mathcal{W}$ is partitioned into $N = 2\lceil \frac{r}{2^{l-l_1}} \rceil$ subsets $\mathcal{W}_i$ of cardinality 2^{l-l_1}, except for the two subsets covering the highest coordinates in both h_i, which are of size $r \mod 2^{l-l_1}$. From our side-channel analysis we know the weight t_i of the private key restricted to each subset $\mathcal{W}_i$.

To improve the probability of success of the Prange algorithm, we changed our strategy of choosing the information set. We tested two different approaches:

1. Randomly choose the information set among the $\mathcal{W}_i$ corresponding to values of $t_i \neq 0$ (we refer to this technique as the classical Prange algorithm in the following),
2. Fix the number of columns chosen in each $\mathcal{W}_i$ depending on the value of t_i.

Theorem 1. *Let $\mathcal{I}$ be the set $\{i | t_i \neq 0\}$, and let $c = \sum_{i \in \mathcal{I}} |\mathcal{W}_i|$. The probability of success of the classical Prange algorithm using hints from our side-channel analysis is:*

$$\frac{\binom{r}{w}}{\binom{c}{w}}.$$

The second approach is the one described in [HPR$^+$21], adapted to work with parity-check matrices. We fix a vector $x \in \mathbb{Z}^N$ such that $x_i \geqslant t_i$ and $\sum_i x_i = r$. Then, to sample an information set, we choose for each i a random subset $\mathcal{X}_i \subseteq \mathcal{W}_i$ of cardinality x_i, and then proceed to solve the linear system as in the Prange algorithm.

Theorem 2. *[HPR$^+$21] The probability of success of the Prange algorithm using hints from our side-channel analysis is:*

$$\prod_{i=1}^{N} \frac{\binom{x_i}{t_i}}{\binom{|\mathcal{W}_i|}{t_i}}.$$

The last step to evaluate the complexity of this algorithm is to choose the best vector x (i.e the one that maximizes the success probability of the algorithm). We use a greedy approach as in [HPR+21] to perform this step:

- Initially, choose $x_i = |\mathcal{W}_i|$ if $t_i \neq 0$ and $x_i = 0$ otherwise.
- While $\sum_i x_i > r$, decrease by one the x_i that reduces the probability of success the least.

[HPR+21, Appendix E] gives a proof of why this approach yields the optimal choice for x.

Remark 5 *If after the initial step of the algorithm, we have $\sum_i x_i \leqslant r$, then the probability of success of the algorithm is 1.*

Among the BIKE private keys, some are easier to recover than others using this technique. Indeed, the more t_i equals to 0 we have, the better the attack will perform, since it will allow to choose more positions in the subsets containing at least one nonzero position. For this reason, we compute the complexity of our attack by averaging the complexities for 10 random private keys. We run the following experiment: for BIKE-level-1 parameter set ($r = 12323, w = 142$), we compute the probability of success of the Prange algorithm for different values of l_1, *i. e.* the number of recovered bits among the 14 bits used to encode each coordinate for this parameter.