Efficient Microcontroller Implementation of BIKE

Mario Bischof[1], Tobias Oder[2], and Tim Güneysu[2,3]

[1]Swiss Distance University of Applied Sciences (FFHS) Brig, Switzerland
`mario.bischof@ffhs.ch`
[2]Horst Görtz Institute for IT Security, Ruhr-Universität Bochum, Germany
`{tobias.oder,tim.gueneysu}@rub.de`
[3]DFKI, Germany

Abstract. In the digital world, public-key cryptography is ubiquitous. Current public-key crypto schemes like RSA or Diffie-Hellmann are in widespread use and they represent an indispensable asset of our technological toolbox. However, the discovery of Shor's algorithm and the rapid progression in the field of quantum computers became a painful reminder of our alerting dependency on such technologies. At the same time, this realization started a demand for new cryptographic algorithms withstanding the power of quantum computers. The National Institute of Standards and Technology (NIST) aimed to satisfy this urge by initiating a standardization process in 2017 with a call for proposals of post-quantum key exchange mechanisms and signature algorithms. One of the submissions that made it to the second round is the key encapsulation mechanism BIKE.
This work investigates various techniques to achieve an efficient and secure implementation of BIKE on embedded devices. We show that it is possible for BIKE to run on a Cortex-M4 microcontroller using reduced data representation and adequate decoding algorithms. Our implementation achieves a performance of 6 million cycles for key generation, 7 million cycles for encapsulation, and 89 million cycles for decapsulation for BIKE-1.

Keywords: Post-quantum cryptography, code-based cryptography, BIKE, KEM, microcontroller, timing attacks, Cortex-M4.

1 Introduction

The advancements in the development of quantum computers impose an increasing threat [14, 16] to most of the currently existing public-key crypto schemes. Already decades ago, Peter Shor [23] developed a quantum algorithm that is able to break them. Since current public-key cryptosystems like RSA are in such widespread use, it is crucial that extensive research for new cryptographic schemes is done before the arrival of large enough quantum computers. For this reason, the United States National Institute of Standards and Technology (NIST) announced a call for proposals in 2017 [12, 17] to submit new cryptographic schemes able to withstand such attacks. One of these proposals is the

BIt Flipping **K**ey **E**ncapsulation- or **BIKE**-suite [2]. The NIST standardization is a multi-round process aiming to select new algorithms and publish first drafted standards by the years 2022/2024 [18]. NIST announced a list of candidates that have been selected for the second round in early 2019 [19], with BIKE being among the selected schemes. The main NIST selection-criteria are security, cost and performance, as well as algorithm and implementation characteristics on a large variety of platforms [20]. We contribute to the NIST standardization process by presenting the first microcontroller implementation of BIKE.

1.1 Related Work

Most code-based cryptographic schemes nowadays are improved and/or optimized adaptations of the initial works of Robert McEliece [8] and Hermann Niederreiter [15], who published cryptographic schemes of the same names in the late seventies and eighties. Both of them are widely considered to be very well studied and remained essentially unbroken up until this day and provide sufficient security properties for the upcoming quantum computing age. To eliminate the major downside of these older schemes, being its very large key sizes, newer proposals often use codes that have some cyclic structure like quasi-cyclic (QC) low-density-parity-check (LDPC) or modest-density-parity-check (MDPC) codes [6,21]. This becomes especially essential when targeting embedded devices like we do in this work, due to their limited memory. The list of NIST submissions [19] shows that the majority of standardization candidates are either code- or lattice-based schemes. Examples of other code-based candidates closely related to BIKE are classic McEliece [3] or Hamming Quasi-Cyclic (HQC) [1].

The pqm4 post-quantum crypto library for the ARM Cortex-M4 [9] consolidates most microcontroller implementation efforts of post-quantum key exchange mechanisms and signature schemes. Great efforts towards efficient and side-channel attack resistant implementations of the McEliece crypto scheme using quasi-cyclic MDPC codes for embedded platforms has been made by von Maurich, Oder, Güneysu and Heyse [25,27–29] and is also a central topic of von Maurich's dissertation [26].

1.2 Contribution

In this work, we present the first microcontroller implementation of the NIST round 2 candidate BIKE. We replace any external dependencies that have been a major issue preventing microcontroller adoption of BIKE [9] with stand-alone components. Furthermore, we replace the decoding algorithm of the reference implementation with a more memory-efficient one to make the implementation fit the memory of our target device. Finally, we apply countermeasures to protect the implementation against timing side-channels. We implemented all three variants of BIKE, each at three different security levels, leading to a total number of 9 implementations. Up to our knowledge, no microcontroller implementations of code-based NIST round 2 candidates have been published so far. This work is therefore an important contribution to the evaluation of the practicability of code-based cryptography in the ongoing NIST standardization process. To allow independent verification of our results, we will make our source code publicly available with the publication of this work [1].

2 Preliminaries

In this section, we discuss the mathematical background that is necessary for the understanding of this paper. We use the following notation. n describes the size of the whole parity check matrix and r of one circulant block. w is the parity check matrix' row weight and t the weight of error, the code is able to correct. By H we denote the parity check matrix of a linear code and H^T denotes the transpose of the parity check matrix. The syndrome of an input vector e is defined as $s = eH^T$.

2.1 BIKE - Bit Flipping Key Encapsulation

BIKE is a key encapsulation mechanism (KEM) based on QC-MDPC codes. It is composed out of three different variants: BIKE-1, BIKE-2 and BIKE-3 that can be instantiated at three different security levels: Level 1, 3, and 5. These correspond to the security recommendations given by NIST. The suggested BIKE parameters are shown in Table 1. The three variants of BIKE are described in Figures 1-3. For a more detailed description of the scheme, we refer to the specification of the NIST submission [2].

Table 1: Parameters for every BIKE variant

variant	**security**	**r**	**n**	**w**	**t**
BIKE 1/2	Level 1	10,163	20,326	142	134
BIKE 1/2	Level 3	19,853	39,706	206	199
BIKE 1/2	Level 5	32,749	65,498	274	264
BIKE 3	Level 1	11,027	22,054	134	154
BIKE 3	Level 3	21,683	43,366	198	226
BIKE 3	Level 5	36,131	72,262	266	300

2.2 Decoding Using Bit Flipping Algorithms

To decode QC-MDPC codes we can use bit flipping algorithms. Many different bit flipping decoders do exist. Algorithm 1.1 [2] shows a classical variant. After termination, the algorithm should output an error pattern e which corresponds to the inputted syndrome s, e.g. $s = eH^T$.

[1] https://www.seceng.ruhr-uni-bochum.de/research/publications/efficient-microcontroller-implementation-bike/

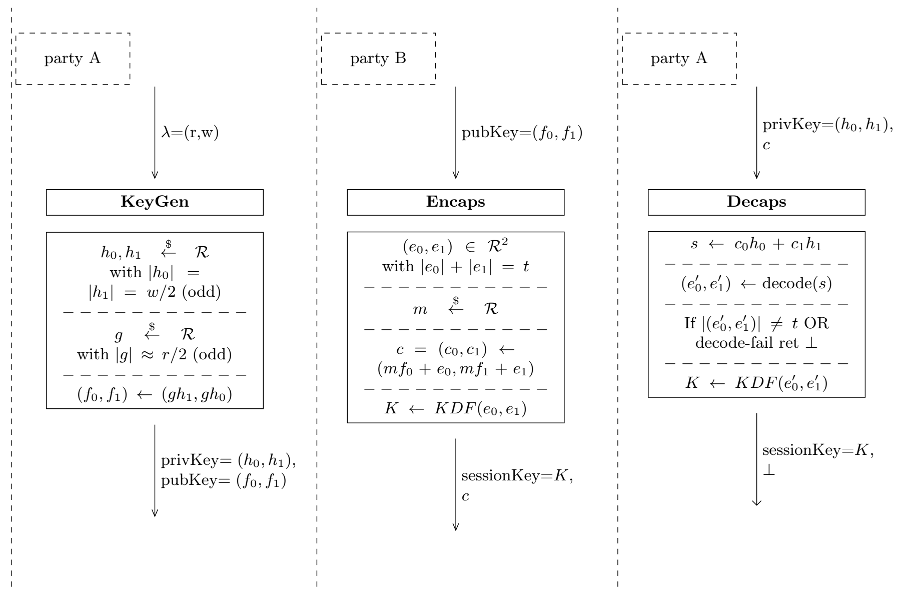
party A
λ=(r,w)
KeyGen
$h_0, h_1 \xleftarrow{\$} \mathcal{R}$ with $|h_0| = |h_1| = w/2$ (odd)
$g \xleftarrow{\$} \mathcal{R}$ with $|g| \approx r/2$ (odd)
$(f_0, f_1) \leftarrow (gh_1, gh_0)$
privKey= (h_0, h_1), pubKey= (f_0, f_1)
party B
pubKey=(f_0, f_1)
Encaps
$(e_0, e_1) \in \mathcal{R}^2$ with $|e_0| + |e_1| = t$
$m \xleftarrow{\$} \mathcal{R}$
$c = (c_0, c_1) \leftarrow (mf_0 + e_0, mf_1 + e_1)$
$K \leftarrow KDF(e_0, e_1)$
sessionKey=K, c
party A
privKey=(h_0, h_1), c
Decaps
$s \leftarrow c_0h_0 + c_1h_1$
$(e'_0, e'_1) \leftarrow$ decode(s)
If $|(e'_0, e'_1)| \neq t$ OR decode-fail ret $\perp$
$K \leftarrow KDF(e'_0, e'_1)$
sessionKey=K, $\perp$

Fig. 1: BIKE-1 specification

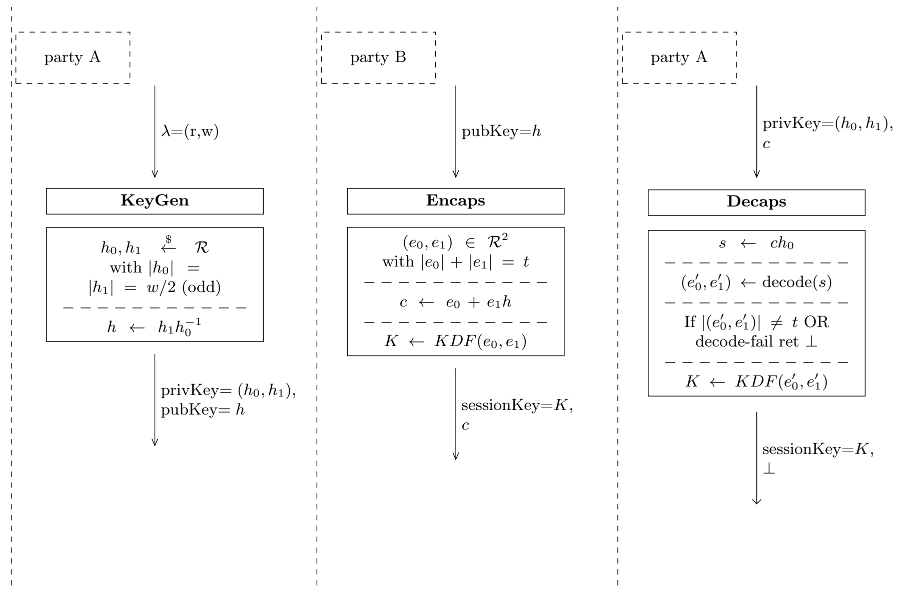
party A
λ=(r,w)
KeyGen
$h_0, h_1 \xleftarrow{\$} \mathcal{R}$ with $|h_0| = |h_1| = w/2$ (odd)
$h \leftarrow h_1h_0^{-1}$
privKey= (h_0, h_1), pubKey= h
party B
pubKey=h
Encaps
$(e_0, e_1) \in \mathcal{R}^2$ with $|e_0| + |e_1| = t$
$c \leftarrow e_0 + e_1h$
$K \leftarrow KDF(e_0, e_1)$
sessionKey=K, c
party A
privKey=(h_0, h_1), c
Decaps
$s \leftarrow ch_0$
$(e'_0, e'_1) \leftarrow$ decode(s)
If $|(e'_0, e'_1)| \neq t$ OR decode-fail ret $\perp$
$K \leftarrow KDF(e'_0, e'_1)$
sessionKey=K, $\perp$

Fig. 2: BIKE-2 specification

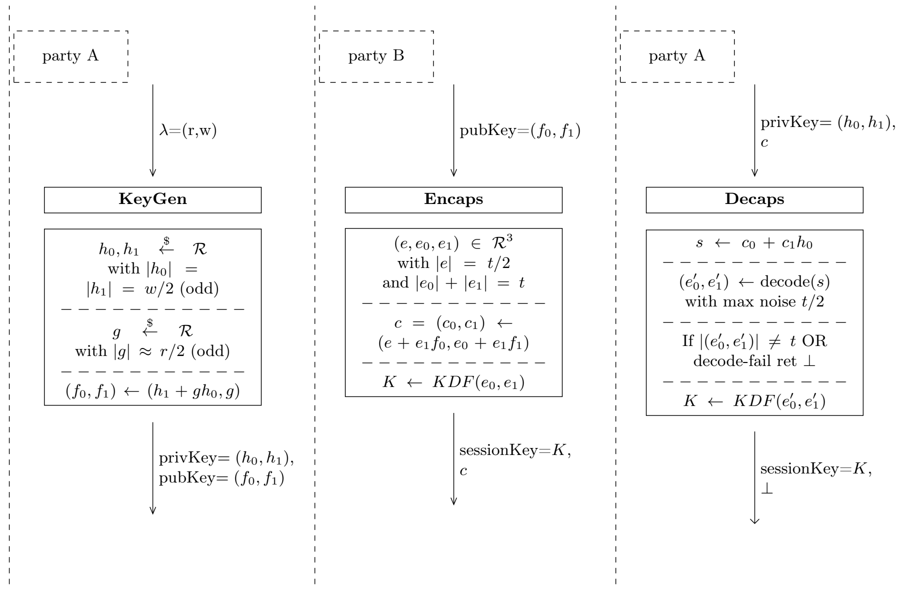
party A
λ=(r,w)
KeyGen
$h_0, h_1 \xleftarrow{\$} \mathcal{R}$ with $|h_0| = |h_1| = w/2$ (odd)
$g \xleftarrow{\$} \mathcal{R}$ with $|g| \approx r/2$ (odd)
$(f_0, f_1) \leftarrow (h_1 + gh_0, g)$
privKey= (h_0, h_1), pubKey= (f_0, f_1)
party B
pubKey=(f_0, f_1)
Encaps
$(e, e_0, e_1) \in \mathcal{R}^3$ with $|e| = t/2$ and $|e_0| + |e_1| = t$
$c = (c_0, c_1) \leftarrow (e + e_1 f_0, e_0 + e_1 f_1)$
$K \leftarrow KDF(e_0, e_1)$
sessionKey=K, c
party A
privKey= (h_0, h_1), c
Decaps
$s \leftarrow c_0 + c_1 h_0$
$(e'_0, e'_1) \leftarrow$ decode(s) with max noise $t/2$
If $|(e'_0, e'_1)| \neq t$ OR decode-fail ret $\perp$
$K \leftarrow KDF(e'_0, e'_1)$
sessionKey=K, $\perp$

Fig. 3: BIKE-3 specification

Unsatisfied parity check equations & threshold Algorithm 1.1 calculates the number of unsatisfied parity check equations per parity check matrix column h_j for all n columns of H and compares it to a given threshold $\tau|h_j|$ depending on the currently processed column h_j. h_j is the j-th column of the parity check matrix H interpreted as a row vector. $h_j \star s'$ is the component-wise product of two vectors and $|h_j \star s'|$ is the number of unchecked parity equations involving h_j. If the number of unsatisfied equations is higher than $\tau|h_j|$, the corresponding bit in the resulting error e_j is flipped. After all columns have been processed, the temporary syndrome s' gets updated $s' = s - eH^T$ with the current error pattern and the next round starts.

The stopping condition $|s'| > u$ specifies whether the correct error has been found. If $u = 0$ is used then $s' = s - eH^T = s - s = 0$. This also means that s is the exact syndrome of e. This is called *noiseless* syndrome decoding. If $u > 0$, the decoding is called *noisy*. In this case, the syndrome includes an additional error e' (or noise) and is not the exact syndrome of e.

3 Portable Implementation

The BIKE reference implementation relies on the NTL library [24] to perform arithmetic operations in finite fields, as well as OpenSSL for the AES-based pseudorandom number generators and the key derivation hash function SHA384. These external dependencies are the main reason, why there has not been a microcontroller implementation of BIKE yet [9]. The first step towards a microcontroller implementation is therefore to make the code portable. The source

Algorithm 1.1: Bit flipping algorithm

```
Input: $H \in \mathbb{F}_2^{(n-k)\times n}, s \in \mathbb{F}_2^{(n-k)}, u \geq 0$
Output: $e$.
begin
    $e \leftarrow 0$;
    $s' \leftarrow s$;
    while $|s'| > u$ do
        $\tau \leftarrow Threshold \in [1, 0]$;
        for $j \leftarrow 0$ to $\mathrm{n} - 1$ do
            if $|h_j \star s'| \geq \tau |h_j|$ then $e_j \leftarrow e_j + 1 \bmod 2$;
        end
        $s' \leftarrow s - eH^T$;
    end
    return $e$
end
```

code of our portable implementation will also be made available online with the publication of this work.

3.1 Replacing NTL Modules

Multiplication in GF2 has the most crucial impact on the performance of BIKE and is done with the help of the NTL library in the reference implementation of BIKE. The gf2x-library [5] is the result of extensive research [4] for efficient multiplication in finite fields. The gf2x-library has been integrated into NTL, after NTL was significantly outperformed by gf2x in the area of finite field operations. Another big advantage of gf2x is that almost everything is written in plain C (except very few C++ parts) and does not rely on any external dependencies, therefore we integrated the relevant software modules from the gf2x-library into our portable code.

3.2 Replacing OpenSSL Modules

To remove the platform dependent OpenSSL AES and SHA384 modules, we chose to adapt two alternatives [7, 10] that are publicly available. According to the authors, both of these replacements for AES and SHA2 are validated against NIST test vectors and were designed in respect to portability, compactness and efficiency. This offered ideal conditions to be used for our portable BIKE implementation.

4 Cortex-M4 Implementation

After the development of a portable BIKE implementation, the next step is to develop an efficient Cortex-M4 implementation. Our evaluation platform is the

STM32F4-DISCOVERY board. It runs with a clock frequency of up to 168 MHz. The board offers 192 kB of RAM as well as 1 MB of flash memory. Furthermore, it features a true random number generator (TRNG) based on analog circuitry and a floating-point unit (FPU). NIST recommends to use the Cortex-M4F as target platform for microcontroller evaluations of post-quantum standardization candidates [13].

4.1 Bit Flipping in the BIKE Reference Implementation

BIKE-1 and BIKE-2 use *noiseless* syndrome decoding and BIKE-3 uses *noisy* syndrome decoding. BIKE operates with (2,1)-quasi-cyclic MDPC codes. Its decode-function, has the signature `decode(e, s, h0, h1, u)`. The inputs are the syndrome s and the first two rows h_0 and h_1 of the two quasi-cyclic code blocks of the parity check matrix H. The integer u distinguishes between noiseless ($u = 0$) and noisy ($u > 0$) decoding. Decode outputs $e = e_0|e_1$ for which holds $|e_0h_0 + e_1h_1 + s| \leq u$, i.e. the *noise* of the syndrome must not be larger than u for decode to work. The secret key is of the form $sk = (h_0, h_1)$ for all BIKE variants, so the same decode function can be applied. The authors of BIKE suggest a one-round bit flipping algorithm [2] similar to the one illustrated in Algorithm 1.1.

4.2 Memory Requirements for Decoding

When the one round bit flipping algorithm is applied, the decoding becomes a major memory bottleneck. Table 2 shows the memory consumption of the largest data structures in the reference implementation of the `decode` operation for all BIKE variants. Clearly, this simple decoding mechanism can not be used on the microcontroller, since it requires way more memory than the 192 kB that are available, and needs to be substituted with a more adequate solution.

Table 2: Dynamic memory requirements of arrays in decode call for all BIKE variants in number of kB using the reference implementation.

	BIKE-1/2			**BIKE-3**		
Data structure	**Level 1**	**Level 3**	**Level 5**	**Level 1**	**Level 3**	**Level 5**
`unsat_counter2[]`	41	79	131	44	87	145
`errorPos[]`	41	79	131	44	87	145
`unsat_counter[]`	20	40	65	22	43	72
`J[][]`	813	1,271	2,620	882	1,735	2,890
`e[]`	20	40	65	22	43	72
`syndrome`	10	20	33	11	22	36
Overall	**945**	**1,529**	**3,046**	**1,026**	**2,017**	**3,360**

4.3 Decoding on Embedded Platforms

Extensive research on how to efficiently implement QC-MDPC McEliece on embedded devices has been done by von Maurich [26, Ch. 4]. Von Maurich evaluated and compared various different bit flipping decoders targeting constrained devices. In the following, we will shortly review the decoders that we adapted for our microcontroller implementation.

The primary idea [26, Ch. 4.4.1] is based on a decoding mechanism for LDPC codes first introduced by Gallager [6]. Similar to Algorithm 1.1, this strategy calculates the number of unsatisfied parity check equations for each bit of the received message and flips the corresponding bit, if a specific threshold b is reached. Thresholds of a given parameter set are pre-calculated for each iteration b_i of decoding, by calculating the probability P_i for a bit to be in error after iteration i. Decoding approaches using such types of thresholds are also called *hard decision bit flipping based* and mainly operate following these basic steps [26, p. 40]:

1. Calculate the syndrome of the inputted message and pass it to the decoder
2. Calculate the number of unsatisfied parity check equations for each bit of the input message
3. Flip those bits that violate at least b_i equations in iteration i
4. Recalculate the syndrome based on the updated message (since the message changed, the syndrome also changes)

A corresponding decoder $\mathcal{D}_1$ working the mentioned way is shown in Algorithm 1.2 in Appendix A. For BIKE-3, Line 2 in Algorithm 1.2 needs to check for the syndrome's hamming weight to be larger than some noise u, rather than 0. In case of BIKE-1/2, u will just be 0. Decoder $\mathcal{D}_1$ implements the following optimizations:

1. As proposed by Gallger [6], $\mathcal{D}_1$ uses precomputed thresholds b_i (Algorithm 1.2, Line 14), eliminating the need to calculate the maximum number of unsatisfied parity check equations for every invocation (like other decoders do).
2. The syndrome changes as follows if the threshold is reached: $s_{new}{=}s_{old} \oplus h_j$ [26, p. 41]. This crucial observation shows that it is not necessary to completely recalculate the syndrome every time it is updated. It can rather be computed by just adding the current h_j (Algorithm 1.2, Line 16), significantly speeding up syndrome recomputation.
3. Von Maurich points out [26, p. 46] that if about 4-6 iterations of the decoding algorithm have passed, it is extremely rare for it to still succeed without any adjustments to the threshold. So he suggests an ITERATIONS_MAX of 5 iterations for his QC-MDPC McEliece instance.

The decoding failure rate can be further reduced [26, p. 42-43] if in case of failure the thresholds b_i are increased by one, until a *threshold_delta_max* of 5 is reached. This concept is called decoder $\mathcal{D}_2$ shown in Algorithm 1.3 in Appendix A. $\mathcal{D}_2$ essentially just acts as a wrapper around $\mathcal{D}_1$, which upon failure increases the *threshold_delta* and starts decoding over again.

4.4 Microcontroller Optimizations

Instead of the simple decoding algorithm from the reference implementation, we use the proposed decoders $\mathcal{D}_1/\mathcal{D}_2$ [26, p. 62] for the microcontroller implementation which perform most of its computations in-place, omitting the necessity to store a lot of data at the cost of losing performance. This combination of decoders provides further failure rate reduction.

We can apply a time-memory tradeoff strategy [26, p. 61] to the sequential algorithm, turning it into a somewhat parallel method. *Parallel* in this sense means processing h_0 and h_1 in the same turn, using separate variables and counters. The parallel decoding method for decoder $\mathcal{D}_1$ is shown in Algorithm 1.4 in Appendix A. The same adjustments for the noise u as explained for the sequential algorithm are required for this method to work with BIKE. Using two variables `current0` and `current1`, the parallel way counts the unsatisfied parity check equations for both h_0 and h_1 in `violated0` and `violated1` in the same loop iteration (Algorithm 1.4, Lines 10-15). Since this processes two vectors in the same turn, it has to finish the whole loop before doing the syndrome update outside the inner loop. If not stated otherwise, we will always refer to the parallel decoder for the remainder of this paper as it provides a superior performance.

The gf2x library proposes optimization-suggestions for ARMv7 (and other) platforms to speed up its implementation. Besides suggesting slightly different Karatsuba/Toom thresholds, the changes basically consist of a few optimized multiplication base cases `gf2x_mul3`, `gf2x_mul5` and `gf2x_mul6`. In `gf2x_mul3`, we get an optimized version with 6 multiplications instead of 7. `gf2x_mul5` now uses the formula of Peter Montgomery [11] allowing to multiply two 5-term polynomials with only 13 multiplications instead of 17. `gf2x_mul6` implements the K_3 formula by Weimerskirch and Paar [30] which needs only 6 calls to `gf2x_mul2`, resulting in a total of 18 multiplications instead of 21. We furthermore replaced the AES implementation from [10] that we use as PRNG with the Cortex-M-optimized constant-time implementation from [22]. To seed the PRNG, we generate a random sample from the on-board TRNG.

The private key can be compressed by only storing the indices of the $\frac{w}{2}$ ones per vector, ending up with a $w * \lceil log_2(r) \rceil$-bit long representation [2]. We can then rotate the vector by just increasing all positions by one instead of shifting the whole vector by one bit. We take care of overflows by just setting a corresponding position to zero, if it was equal to r. The positions are reordered in case of an overflow, to keep them in an ordered fashion. This way, only the last position needs to be checked for overflow. We adapted the sparse representation [26, p. 108] to our code.

4.5 Hardening the Implementation against Timing Attacks

We also investigated measures to secure the bit flipping decoding algorithm against timing attacks. Thorough research on how to secure QC-MDPC decoding has been done by von Maurich, Güneysu and Oder [26, 29]. We adapted and extended the techniques from [26] for constant-time implementation of several components, like private key rotation, threshold comparison, syndrome and error update, and counting of unsatisfied parity check equations. Our tests showed that the overwhelming majority of decodings succeed after 2-4 iterations independent from the chosen BIKE variant. We hence fixed the number of iterations of the decoder to be always 5. That means that the decoder will always run 5 times, even if the decoding was successful in earlier runs already. However, the multiplication routine of the gf2x library is *not* constant-time. Our implementation therefore provides some resistance against timing-attacks but is not fully protected.

5 Results and Comparison

As environment, we used the Eclipse IDE for C/C++ developers, Version Photon Release 4.8.0 and Build id 20180619-1200 combined with the OpenSTM32 System Workbench for STM32 - C/C++ Embedded Development Tools for MCU 2.5.0.201807130628. For debugging purposes, we configured the internal eclipse debugger to use gdb via OpenOCD connecting to the STLink embedded debugger of the board. We counted clock cycles using the data watchpoint and trace unit (DWT) of the Cortex-M4 for performance measurements at 168 MHz.

5.1 Performance and Memory Evaluation

In Table 3, we show the cycle counts for all 9 implementations of BIKE. We also include the cycle counts for the case that the countermeasures against timing side-channels as described in Section 4.5 are applied to the decoding. The unusual high cost of the key generation in BIKE-2 is due to the expensive finite field inversion required in BIKE-2 only. Except for BIKE-2, the Decaps operation is the clear bottleneck regarding performance. In Table 4, we also show the dynamic memory consumption (heap and stack) of our implementations. The memory requirements range from 21 to 74 kB and therefore comfortably fit the memory of our evaluation platform. Encaps and Decaps have similar memory requirements and the increase in memory consumption is linear in the security level.

We compare our implementation with the evaluation results of several other NIST post-quantum candidates from the `pqm4` library [9] in Table 5. In direct comparison with lattice-based schemes, the performance of BIKE is inferior. The only lattice-based scheme that performs worse is the Frodo KEM. Frodo is however a very conservative scheme. For a comparison with schemes with similar trust in the underlying security assumption, QC-MDPC-based schemes (like BIKE) should rather be compared to ideal lattice-based schemes (like NewHope). However, compared to isogeny-based cryptography, BIKE clearly has the superior performance and can therefore be seen as a backup alternative to lattice-based schemes.

Table 3: Final performance of KEM phases for all BIKE variants using parallel decoding, including timing attack countermeasures [TP]. Cycle counts are given in million cycles.

		KeyGen	**Encaps**	**Decaps**
BIKE-1	Level 1	6	7	89
	[TP]			305
	Level 3	15	17	228
	[TP]			773
	Level 5	28	30	569
	[TP]			1,685
BIKE-2	Level 1	918	4	87
	[TP]			303
	Level 3	3,345	9	222
	[TP]			766
	Level 5	8,763	15	557
	[TP]			1,673
BIKE-3	Level 1	4	7	86
	[TP]			309
	Level 3	10	18	234
	[TP]			795
	Level 5	20	37	774
	[TP]			1,819

Table 4: Final dynamic memory requirements of KEM phases for all BIKE variants in kB consisting of stack and heap memory. The maximum memory consumption of an implementation is highlighted in bold font.

		KeyGen	**Encaps**	**Decaps**
BIKE-1	Level 1	13.95	**21.59**	20.84
	Level 3	26.17	**41.07**	39.37
	Level 5	42.24	**66.83**	63.79
BIKE-2	Level 1	15.62	19.05	**20.84**
	Level 3	30.18	36.10	**39.37**
	Level 5	49.51	58.63	**63.79**
BIKE-3	Level 1	15.01	**23.30**	22.41
	Level 3	28.42	**44.71**	42.74
	Level 5	46.61	**73.75**	70.24

Table 5: Comparison of cycle counts of our implementations with results from pqm4 [9].

Scheme	**Security**	**PQ Family**	**KeyGen**/10^3	**Encaps**/10^3	**Decaps**/10^3
BIKE-1 (our)	Level 1	Codes	6,437	6,867	89,131
NewHope-512	Level 1	Lattices	628	915	163
Kyber-512	Level 1	Lattices	514	653	621
Frodo-640	Level 1	Lattices	47,051	45,883	45,366
SIKEp434	Level 1	Isogenies	650,735	1,065,631	1,136,703
BIKE-1 (our)	Level 3	Codes	15,309	16,641	228,244
Kyber-768	Level 3	Lattices	977	1,147	1,095
SIKEp610	Level 3	Isogenies	1,819,652	3,348,669	3,368,114
BIKE-1 (our)	Level 5	Codes	27,605	29,797	568,517
NewHope-1024	Level 5	Lattices	1,035	1,495	206
Kyber-1024	Level 5	Lattices	1,575	1,779	1,709
SIKEp751	Level 5	Isogenies	3,296,225	5,347,056	5,742,522

6 Conclusion

In this work, we developed an efficient and secure implementation of the post-quantum standardization candidate BIKE. Our baseline was the reference implementation that depended on platform-specific third-party software libraries. A portable implementation was achieved by incorporating substitutions for the OpenSSL and NTL dependencies. Our analysis further revealed that the BIKE bit flipping decoding, as done in the reference implementation, is not feasible on the microcontroller and we provided adaptations of more adequate decoders that were able to satisfy the limiting memory constraints of the development board. We furthermore added countermeasures against timing attacks to our implementation. Our final implementation offers reasonable results in comparison with other Cortex-M4 post-quantum cryptography KEMs in terms of efficiency, security and memory requirements. Our work has successfully demonstrated that the BIKE standardization candidate can be implemented and run on an embedded microcontroller like the Cortex-M4. This result lives up to the expectation of NIST that the proposed algorithms are required to be implementable in a wide range of hardware and software platforms.

The finite field inversion resembles a heavy bottleneck for BIKE-2 key generation performance. As documented in the BIKE specification [2], batch key generation can reduce some performance loss generated by the inversion, in exchange for occupying more memory. We expect the benefit from batch key generation will be limited on embedded platforms due to memory restrictions but nonetheless, the exact limitations of these assumptions remained unknown and should still be verified in future work.

Acknowledgement. This work was supported in part through DFG Excellence Strategy grant 39078197 (EXC 2092, CASA), and by the Federal Ministry of Education and Research of Germany through the QuantumRISC project (16KIS1038).

References

1. C. Aguilar-Melchor, N. Aragon, S. Bettaieb, L. Bidoux, O. Blazy, J.-C. Deneuville, P. Gaborit, E. Persichetti, and G. Zémor. Hamming Quasi-Cyclic (HQC), Nov. 2017. Submission to the NIST post quantum standardization process. 2017.
2. N. Aragon, P. S. L. M. Barreto, S. Bettaieb, L. Bidoux, O. Blazy, J.-C. Deneuville, P. Gaborit, S. Gueron, T. Güneysu, C. A. Melchor, R. Misoczki, E. Persichetti, N. Sendrier, J.-P. Tillich, V. Vasseur, and G. Zemor. Bike: Bit Flipping Key Encapsulation, 2018. `http://bikesuite.org/files/BIKE.pdf`, as of November 18, 2019.
3. D. J. Bernstein, T. Chou, T. Lange, I. von Maurich, R. Misoczki, R. Niederhagen, E. Persichetti, C. Peters, P. Schwabe, N. Sendrier, J. Szefer, and W. F. Wang. Classic McEliece : conservative code-based cryptography. 2017. Submission to the NIST post quantum standardization process. 2017.
4. R. P. Brent, P. Gaudry, E. Thomé, and P. Zimmermann. Faster multiplication in gf(2)[x]. In *Algorithmic Number Theory, 8th International Symposium, ANTS-VIII, Banff, Canada, May 17-22, 2008, Proceedings*, pages 153–166, 2008.
5. R. P. Brent, P. Gaudry, E. Thomé, and P. Zimmermann. Inriaforge: gf2x: Project home, 2018. `https://gforge.inria.fr/projects/gf2x/`, as of November 18, 2019.
6. R. G. Gallager. Low-density parity-check codes. *IRE Trans. Information Theory*, 8(1):21–28, 1962.
7. O. Gay. Fast software implementation in C of the FIPS 180-2 hash algorithms SHA-224, SHA-256, SHA-384 and SHA-512, 2018. `https://github.com/ogay/sha2`, as of November 18, 2019.
8. R. J. McEliece. A public-key cryptosystem based on algebraic coding theory. *JPL DSN Progress Report*, 44, 05 1978.
9. M. J. Kannwischer, J. Rijneveld, P. Schwabe, and K. Stoffelen. pqm4: Testing and benchmarking NIST PQC on ARM Cortex-M4. Cryptology ePrint Archive, Report 2019/844, 2019. `https://eprint.iacr.org/2019/844`.
10. kokke. Small portable AES128/192/256 in c, 2018. `https://github.com/kokke/tiny-AES-c`, as of November 18, 2019.
11. P. L. Montgomery. Five, six, and seven-term Karatsuba-like formulae. *IEEE Trans. Computers*, 54(3):362–369, 2005.
12. D. Moody. The ship has sailed: the NIST post-quantum cryptography "competition". In *Invited talk at ASIACRYPT 2017, Hongkong*, 2017.
13. D. Moody. Round 2 of NIST PQC competition. *Invited talk at PQCrypto 2019, Chongqing, China*, 2019.
14. T. Moses. Quantum Computing and Cryptography - Their impact on cryptographic practice. Technical report, Entrust, Inc., 2009. `https://www.entrust.com/wp-content/uploads/2013/05/WP_QuantumCrypto_Jan09.pdf`.
15. H. Niederreiter. Knapsack type cryptosystems and algebraic coding theory. *Problems of Control and Information Theory*, 15, 01 1986.
16. NIST. Post-Quantum Cryptography: NIST's Plan for the Future. Technical report, NIST, 2016. `https://csrc.nist.gov/csrc/media/projects/post-quantum-cryptography/documents/pqcrypto-2016-presentation.pdf`.
17. NIST. Call for Proposals - Post-Quantum Cryptography | CSRC. Technical report, NIST, 2017. `https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Post-Quantum-Cryptography-Standardization/Call-for-Proposals`.

18. NIST. Post-Quantum Cryptography - Workshops and Timeline. Technical report, NIST, 2017. https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Workshops-and-Timeline.
19. NIST. PQC Standardization Process: Second Round Candidate Announcement. Technical report, NIST, 2019. https://csrc.nist.gov/News/2019/pqc-standardization-process-2nd-round-candidates.
20. NIST. Status Report on the First Round of the NIST Post-Quantum Cryptography Standardization Process. Technical report, NIST, 2019. https://csrc.nist.gov/publications/detail/nistir/8240/final.
21. S. Ouzan and Y. Be'ery. Moderate-density parity-check codes. *CoRR*, abs/0911.3262, 2009.
22. P. Schwabe and K. Stoffelen. All the AES you need on cortex-m3 and M4. In R. Avanzi and H. M. Heys, editors, *Selected Areas in Cryptography - SAC 2016 - 23rd International Conference, St. John's, NL, Canada, August 10-12, 2016, Revised Selected Papers*, volume 10532 of *Lecture Notes in Computer Science*, pages 180–194. Springer, 2016.
23. P. Shor. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. Technical report, AT&T Research, 1996. https://arxiv.org/abs/quant-ph/9508027.
24. V. Shoup. NTL: A library for doing number theory, 2018. https://www.shoup.net/ntl/, as of November 18, 2019.
25. D. Stehlé and P. Zimmermann. A binary recursive gcd algorithm. In *Algorithmic Number Theory, 6th International Symposium, ANTS-VI, Burlington, VT, USA, June 13-18, 2004, Proceedings*, pages 411–425, 2004.
26. I. von Maurich. *Efficient implementation of code- and hash-based cryptography*. PhD thesis, Ruhr University Bochum, Germany, 2017.
27. I. von Maurich and T. Güneysu. Lightweight code-based cryptography: QC-MDPC McEliece encryption on reconfigurable devices. In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2014, Dresden, Germany, March 24-28, 2014*, pages 1–6, 2014.
28. I. von Maurich and T. Güneysu. Towards side-channel resistant implementations of QC-MDPC McEliece encryption on constrained devices. In *Post-Quantum Cryptography - 6th International Workshop, PQCrypto 2014, Waterloo, ON, Canada, October 1-3, 2014. Proceedings*, pages 266–282, 2014.
29. I. von Maurich, T. Oder, and T. Güneysu. Implementing QC-MDPC McEliece encryption. *ACM Trans. Embedded Comput. Syst.*, 14(3):44:1–44:27, 2015.
30. A. Weimerskirch and C. Paar. Generalizations of the Karatsuba algorithm for efficient implementations. *IACR Cryptology ePrint Archive*, 2006:224, 2006.

A Decoding Algorithms

Algorithm 1.2: Decoder $\mathcal{D}_1$ (sequential)

```
Input: syndrome $s$, private key $h_0$,$h_1$ and noise $u$
Output: error $e$
begin
    while ($|s| > u$) & ($iterations{+}{+} < ITERATIONS\_MAX$) do
        for $i \leftarrow 0$ to 1 do
            if $!i$ then
                current ← $h_0$;
            else
                current ← $h_1$;
            end
            for $j \leftarrow 0$ to R_BITS-1 do
                violated ← 0;
                for $k \leftarrow 0$ to R_BITS-1 do
                    if getBit(current,k) & getBit(syndrome,k) then
                        violated ← violated + 1;
                        if violated $\geq b_i$ then
                            setBit(e,(i*R_BITS+j));
                            syndrome ← syndrome $\oplus$ current;
                            break;
                        end
                    end
                end
                rotate(current);
            end
        end
    end
end
```

Algorithm 1.3: DECODER $\mathcal{D}_2$

```
Input: syndrome $s$, private key $h_0$,$h_1$ and noise $u$
Output: error $e$
begin
    for $threshold\_delta \leftarrow 0$ to $threshold\_delta\_max$ do
        // from here starts $\mathcal{D}_1$
        ...;
        // threshold needs to be considered in following line
        if $violated \geq b_i + threshold\_delta$ then
            ...;
        // check if decoding is successful and if so, leave function
    end
end
```

Algorithm 1.4: Decoder $\mathcal{D}_1$ (parallel)

```
Input: syndrome $s$, private key $h_0$,$h_1$ and noise $u$
Output: error $e$
begin
    while ($|s| > u$) & ($iterations{+}{+} < ITERATIONS\_MAX$) do
        for $i \leftarrow 0$ to $1$ do
            current0 $\leftarrow h_0$;
            current1 $\leftarrow h_1$;
            for $j \leftarrow 0$ to $R\_BITS\text{-}1$ do
                violated0 $\leftarrow$ 0;
                violated1 $\leftarrow$ 0;
                for $k \leftarrow 0$ to $R\_BITS\text{-}1$ do
                    if getBit(current0,k) & getBit(syndrome,k) then
                        violated0 $\leftarrow$ violated0 + 1;
                    end
                    if getBit(current1,k) & getBit(syndrome,k) then
                        violated1 $\leftarrow$ violated1 + 1;
                    end
                end
                if $violated0 \geq b_i$ then
                    setBit(e,(i*R_BITS+j));
                    syndrome $\leftarrow$ syndrome $\oplus$ current0;
                end
                if $violated1 \geq b_i$ then
                    setBit(e,(i*R_BITS+j));
                    syndrome $\leftarrow$ syndrome $\oplus$ current1;
                end
            end
            rotate(current0);
            rotate(current1);
        end
    end
end
```