Classic McEliece on the ARM Cortex-M4

Anonymous Submission

Abstract. This paper presents a constant-time implementation of Classic McEliece for ARM Cortex-M4. Specifically, our target platform is `stm32f4-Discovery`, a development board on which the amount of SRAM is not even large enough to hold the public key of the smallest parameter sets of Classic McEliece. For the level-1 parameter sets `mceliece348864` and `mceliece348864f`, our implementation takes 594,247 cycles for encapsulation and 2,672,525 cycles for decapsulation. Compared to the level-1 parameter set of FrodoKEM, our encapsulation time is more than 79 times faster, and our decapsulation time is more than 17 times faster. For the level-3 parameter sets `mceliece460896` and `mceliece460896f`, our implementation takes 1,108,854 cycles for encapsulation and 6,503,329 cycles for decapsulation. In addition, our implementation is also able to carry out key generation for the level-1 parameter sets and decapsulation for level-5 parameter sets on the board.

Keywords: Classic McEliece · Cortex-M4 · Constant-time implementations · NIST PQC standardization

1 Introduction

Since Shor [23] showed that quantum computers are able to break RSA and ECC in polynomial time, the cryptography community has been searching for cryptographic primitives resistant to attacks from large-scale quantum computers. In response to this pursuit, in 2017, the National Institute of Standards and Technology (NIST) of the U.S. announced a call for proposals to standardize primitives of post-quantum cryptography (PQC) [19]. The standardization process consists of several rounds, and only some of the candidates in each round are chosen to enter the next round. In July 2020, NIST announced that, among the key agreement or public key encryption schemes in the 2nd round, 4 schemes are chosen to enter the 3rd round as "finalists", and 5 schemes are chosen to enter the 3rd round as "alternate candidates" [20].

Classic McEliece is one of these 4 finalists. Classic McEliece [1] is a key establishment mechanism (KEM) designed to achieved IND-CCA2 security. Its construction is based on the McEliece cryptosystem [17], the first code-based cryptosystem. The McEliece cryptosystem is well-known for its amazingly stable security record since its introduction in 1978, which makes Classic McEliece a highly confidence-inspiring cryptosystem. In early 2020, the Germany Federal Office for Information Security (BSI) also recommended Classic McEliece and FrodoKEM [2], a 3rd-round alternate candidate, for long-term confidentiality protection [10].

The confidence in Classic McEliece (and the McEliece cryptosystem), however, comes with a price: the public keys are pretty large. Indeed, Classic McEliece has 2 level-1 parameter sets with public key size around 256 KB, 2 level-3 parameter sets with public key size around 512 KB, 4 level-5 parameter sets with public key size around 1 MB, and 2 level-5 parameter sets with public key size around 1.3 MB. For this reason, Classic McEliece is often considered unsuitable for embedded devices.

In 2019, Kannwischer et al. designed a benchmarking and testing framework named `pqm4` for the ARM Cortex-M4 and published their benchmark results for the NIST PQC candidates [14]. However, performance numbers of Classic McEliece were not shown in [14].

In fact, the authors listed Classic McEliece as one of the schemes "arguably unsuited for a microcontroller environment of this size" and commented that

> "Their public key sizes range from 255 KB to 1326 KB. These are too large to fit into the memory of our platform."

Indeed, `stm32f4-Discovery`, the development board used by the `pqm4` team, has only 192 KB of SRAM. It is conceivable that many readers of the comment will draw the following conclusion: "It is impossible for Classic McEliece to run on such a small device, let along running efficiently on it."

1.1 Our Contribution

This paper presents a constant-time implementation of Classic McEliece tailored for `stm32f4-Discovery`. The implementation follows the 3rd-round specification. There is no cache for SRAM on `stm32f4-Discovery`, but our implementation does not take advantage of this: our implementation does not use secret-dependent memory indices, so it is constant-time even on M4 devices with caches for SRAM.

As shown in Table 1, for the level-1 parameter sets `mceliece348864*` (which means `mceliece348864` and `mceliece348864f`), our implementation takes $594,247$ cycles for encapsulation and $2,700,398$ cycles for decapsulation. The encapsulation time is more than 78 times faster, and our decapsulation time is more than 17 times faster than the corresponding numbers of FrodoKEM (see Table 10), which is often considered as the most conservative lattice-based scheme submitted to the NIST Post-quantum Cryptography Standardization Process. For the level-3 parameter sets `mceliece460896*`, our implementation takes $1,108,854$ cycles for encapsulation and $6,587,202$ cycles for decapsulation. We note that the cycle counts in Table 1, along with all other cycles counts for our implementation are measured at the maximum frequency 168 Mhz of the device unless specified otherwise.

In addition to encapsulation and decapsulation for all level-1 and level-3 parameter sets, our implementation is also able to carry out key generation for `mceliece348864*` and `mceliece348864f` on the device, which takes $2,172,369,343$ cycles (12.9 seconds) and $1,572,880,048$ cycles (9.4 seconds), respectively. Our implementation is also able to carry out decapsulation for the level-5 parameter sets `mceliece6688128*` and `mceliece8192128*` on the development board. We have not implemented decapsulation for `mceliece6960119*`, but we do not see any reason why it cannot run on the device.

The reason why we are able to perform encapsulation and key generation (for some parameter sets) is because we are able store the public key in the 1MB of flash memory on

Table 1: Cycle counts for encapsulation and decapsulation in our implementation. We use `*` to mean both the "non-f" parameter set (simply removing `*`) and the corresponding "f" parameter set (replacing `*` by f). Note that a non-f parameter set and the corresponding f parameter set share the same encapsulation and decapsulation algorithms. The cycle counts for encapsulation and key generation are average numbers of 100 measurements. All the cycle counts are measured at the maximum frequency 168 MHz.

parameter set	level	decapsulation	encapsulation	key generation
`mceliece348864f`	1	2,672,525	594,247	$1,572,880,048$
`mceliece348864`	1			$2,172,369,343$
`mceliece460896*`	3	6,503,329	1,108,854	
`mceliece6688128*`	5	7,353,464		
`mceliece8192128*`	5	7,396,245		

the device, and the amount of SRAM required to perform the operations turns out to be much smaller than the size of the public key.

Our current implementation is not able to carry out encapsulation for the level-5 parameter sets and key generation for the level-3 and level-5 parameter sets. As one might have expected, this is due to the size limit of SRAM and flash memory on `stm32f4-Discovery`. However, the reader should be aware that this does not mean that the operations cannot be carried out on all M4 platforms. For example, we believe that all the three operations of all the 10 parameter sets can be carried out on some of the "Giant Gecko" microcontrollers [15] manufactured by Silicon Labs, where 512KB of SRAM and 2MB of flash memory are available. The optimization techniques presented in this paper are expected to be useful for implementing Classic McEliece on such larger devices.

1.2 Previous Works

In 2013, Bernstein, Chou, and Schwabe proposed McBits [6], a constant-time, bitsliced implementation of a cryptosystem based on the Niederreiter variant [18] of the McEliece cryptosystem [17]. They proposed to use non-conventional algorithms such as the Gao–Mateer additive FFT [11], the "transposed" Gao–Mateer additive FFT, and sorting networks for decoding. These algorithms, when combined with bitslicing, allow McBits to achieve a record-breaking *throughput* (but not latency). The parallelism for bitslicing, however, is external: it comes from the assumption that many instances are processed at the same time, which is not a reasonable assumption in all applications.

In 2017, Chou proposed another bitsliced implementation of a cryptosystem based on Niederreiter [8]. Chou's implementation makes use of internal parallelism which lies in the additive FFT, the transposed additive FFT, the Berlekamp-Massey algorithm, and a type of permutation networks called Beněs networks [5] (which replaces the sorting networks). In this way, the assumption used in McBits is no longer required, and a record-breaking latency can be achieved.

In addition to the reference implementation, the third-round submission of Classic McEliece includes three implementations `vec`, `sse`, and `avx`. Encapsulation and decapsulation in the three implementations are all implemented using the strategy of [8]. Our implementation of encapsulation and decapsulation was adapted from the portable C implementation `vec`. The `vec` implementation works on 64-bit words, so we first converted it into an implementation that works on 32-bit words, and then tried to add our M4-specific optimizations to the 32-bit implementation. Many of our optimizations aim for reducing the number of memory accesses, as loads and stores cannot be hidden by (carried out in parallel with) arithmetic instructions on M4. Our encapsulation time and decapsulation time ended up being much faster than `vec`.

Regarding key generation, our implementation makes use of ideas from two implementations. The first implementation is an implementation targeting x86-64 by one of the authors of this paper, which was included in `supercop-20200531`. The other implementation is introduced in [21], which also targets Cortex-M4 but only supports key generation and encapsulation. Our key generation time and encapsulation time are much faster than the implementation of [21]. Both previous implementations make use of LUP decomposition. Details regarding the two implementations are introduced in Section 3.

1.3 Source Code

The source code of our implementation has been included in the supplementary material. We plan to put our code in the public domain once the final version of this paper is ready.

1.4 Organization

Section 2 introduce the parameter sets of Classic McEliece and the ARM Cortex-M4. Section 3 shows how we optimize key generation. Section 4 shows how we optimize encapsulation. Section 5 shows how we optimize decapsulation. Section 6 shows some numbers regarding stack usage and comparisons with the implementation of [21] and implementations of other NIST candidates.

2 Preliminaries

This section introduces the parameter sets of Classic McEliece and the ARM Cortex-M4.

2.1 Classic McEliece: Parameter Sets

Classic McEliece is based on the Niederreiter variant of the McEliece cryptosystem. Each key pair of Classic McEliece thus defines a length-n binary Goppa code

$$\Gamma_2(g, \alpha_1, \ldots, \alpha_n) = \{c \in \mathbb{F}_2^n \mid \sum_i c_i/(x - \alpha_i) \equiv 0 \bmod g\},$$

where $g \in \mathbb{F}_{2^m}[x]$ is a degree-t monic irreducible polynomial, and $\alpha_1, \ldots, \alpha_n$ is a list of distinct elements in $\mathbb{F}_{2^m}$ with $g(\alpha_i) \neq 0$ for all i. The dimension of $\Gamma_2(g, \alpha_1, \ldots, \alpha_n)$ is given by $k = n - mt$, and the code is designed to be able to correct t errors. The following table shows parameters m, n, t and the sizes of public keys, secret keys, and ciphertexts in bytes for all the 10 parameter sets.

	m	n	t	level	public key	secret key	ciphertext
`mceliece348864*`	12	3488	64	1	261120	6492	128
`mceliece460896*`	13	4608	96	3	524160	13608	188
`mceliece6688128*`	13	6688	128	5	1044992	13932	240
`mceliece6960119*`	13	6960	119	5	1047319	13948	226
`mceliece8192128*`	13	8192	128	5	1357824	14120	240

For the purpose of this paper, we decided not to define key generation, encapsulation, and decapsulation in this section. Instead, we only introduce the relevant parts of each operation at the beginning of Section 3, 4, and 5. Readers who are interested in the details of the three operations should refer to the 3rd-round specification.

2.2 The ARM Cortex-M4

The ARM Cortex-M4 is a 32-bit RISC processor that implements the `ARMv7E-M` architecture. It provides 13 32-bit general-purpose (GP) registers and a floating-point (FP) unit with 32 32-bit FP registers. One can push the "link register", which stores the return address of a function, into the stack so that 14 general-purpose registers are available. Compared to the `ARMv7-M` architecture of Cortex-M3, one nice feature of `ARMv7E-M` is its support of DSP instructions.

An M4 development board is typically equipped with SRAM and flash memory. Programmers usually use the SRAM as the stack space and store the program and constant data in the flash. The sizes of SRAM and flash memory depends on the vendor of the development boards. For example, the ST Microelectronics equips its `stm32f4` [24] series with a flash of size 64KB to 1536KB and an SRAM of size 32KB to 320KB.

The development board we choose is `stm32f4-Discovery`. `stm32f4-Discovery` is a widely used and cheaply available development board. Its maximum operating frequency

is 168 MHz. `stm32f4-Discovery` is equipped with 192 KB of SRAM and 1 MB of flash memory. The 192 KB of SRAM consists of two pieces, one of size 128 KB and one of size 64 KB, and the memory addresses of the two pieces are disconnected.

Instructions on Cortex-M4. Cortex-M4 supports basic instructions that can be used to manipulate the GP registers. These instructions include 32-bit addition, subtraction, multiplication, and logical instructions. One powerful DSP instruction is `umaal`. `umaal` computes the 64-bit product (of integer multiplication) of two GP registers and add the higher 32 bits to a GP registers and the lower 32 bits to another. Instructions of Cortex-M4 also supports free shift on one of the two input registers. For example, the instruction

```
eor Rd, Rn, Rm
```

performs an XOR between two registers `Rn` and `Rm` and store the result in `Rd`. But we can also use

```
eor Rd, Rn, Rm, lsl #3
```

to shift `Rm` to the left by 3 bits before the XOR. The instruction `vmov` allows us to move data between a GP register and an FP register. Each of the instructions above has a latency of 1 cycle.

There are also instructions for loading 32-bit words from memory such as `ldr` and instructions for storing 32-bit words into memory such as `str`. For the purpose of this paper, one can simply assume that loading n 32-bit words will take $n+1$ cycles and storing n 32-bit words will take n cycles. For a detailed specification of instructions supported by Cortex-M4, see [4].

Accessing Flash Memory. In a typical embedded programming process on a development board, a developer loads his code and constants into the board's flash memory with a flashing process. It is then followed by a running process that runs the code and writes data to SRAM or other IO devices. The flash memory can also be used as a storage device which has its own address in the processor's memory map for read and write while running the user's code.

Data in the flash memory can be read by using memory load instructions directly. On the other hand, writing data to flash memory is more complicated. From the perspective of hardware, the writing process usually starts with an erase operation which erasing a full sector of flash. The erase operation sets all bits of the sector to 1. Then, before the next erase, a user can keep writing to the sector under the condition that we only change some 1's to 0. The flash memory also has a finite number of program-erase cycles (P/E cycles): typically 100,000 P/E cycles. Due to these restrictions, the flash memory is usually used as a long-term storage for long-existing and constant data.

The detailed electrical properties of flash operation is highly dependent on the vendor. For the flash memory in `stm32f4-Discovery`, it contains 16, 64, and 128KB sectors, and has a typical double-word programming time of 16 microseconds and an erase time of 230 milliseconds for the 16 KB sectors. These writing time can change depend on the operating temperature and the voltage applied. We refer to [25] for more details about the flash memory on `stm32f4-Discovery`.

From the perspective of software, the operation of writing involves a lot of settings on the flash registers. See Section 3: Embedded Flash memory interface in [24]. In this work, we rely on the hardware abstraction layer (HAL) library provided in `pqm4` framework to simplify the register operations to high-level function calls. It also generalizes the code for various boards by abstracting the flash registers' detailed control on particular devices.

Benchmarking with `pqm4`. We benchmark our implementation with the framework of `pqm4` [14]. By default, `pqm4` benchmarks implementations at 24 MHz for a zero wait states when accessing code or data in the flash memory. Unlike many other implementations of NIST PQC candidates, our implementation reads/stores the public key from/to the flash memory. In order to capture the latency to read and store the public key, we changed the setting of `pqm4` so that it benchmarks our implementation at the maximum frequency 168 MHz.

3 Key Generation

Each Classic McEliece secret key defines a binary Goppa code. To generate the public key for a non-f parameter set, the key generation algorithm first generates a parity-check matrix

$$\hat{H} = (M \mid \hat{T}) \in \mathbb{F}_2^{(n-k)\times n}$$

of the code, where $M \in \mathbb{F}_2^{(n-k)\times(n-k)}$ and $\hat{T} \in \mathbb{F}_2^{(n-k)\times k}$. Then, the algorithm tries to reduce $\hat{H}$ to systematic form

$$(I_{n-k} \mid T) = M^{-1}\hat{H}$$

using Gaussian elimination. If $\hat{H}$ is successfully reduced to $(I_{n-k} \mid T)$, which means M is invertible, the public key is simply the row-major representation of $T \in \mathbb{F}_2^{(n-k)\times k}$. Otherwise, public key generation fails and a new secret key is generated. As mentioned in the supporting documentation of Classic McEliece, approximately 29% of $\hat{H}$ can be reduced to $(I_{n-k} \mid T)$. This means that on average 3.4 attempts are required to generate a key pair for each non-f parameter set.

To generate the public key for each f parameter set, the algorithm again starts with generating a parity-check matrix $\hat{H}$ and performs a Gaussian elimination on $\hat{H}$. However, the public key generation is considered successful even when the result is not in systematic form: the result of Gaussian elimination only needs to satisfy the following conditions.

- The first $n-k-\mu$ pivots are in the first $n-k-\mu$ columns.
- The last μ pivots are in the next ν columns.

The specification of Classic McEliece defines $\mu = 32$ and $\nu = 64$ for all f parameter sets. If the two conditions are satisfied, a specific column permutation defined by the column indices of the last μ pivots is performed to obtain systematic form $(I_{n-k} \mid T)$, and the public key is again the row-major representation of T. The column permutation only permutes the ν columns where the last μ pivots are allows to be. If any of the two conditions is not satisfied, a new secret key is generated. As opposed to the non-f parameter sets, public key generation for f parameter sets fails with a very low probability ($< 2^{-30}$ according to the supporting documentation), so 1 attempt is almost always enough to generate a key pair.

This section presents our implementation for key generation of the level-1 parameter sets `mceliece348864` and `mceliece348864f`.

3.1 Using LUP Decomposition for Public Key Generation

In late 2020, Roth, Karatsiolis, and Krämer [21] proposed to apply LUP decomposition on M for public key generation of the non-f parameter sets. If M is invertible, the LUP decomposition computes a lower-triangular matrix $L \in \mathbb{F}_2^{(n-k)\times(n-k)}$, an upper-triangular matrix $U \in \mathbb{F}_2^{(n-k)\times(n-k)}$, and a permutation matrix $P \in \mathbb{F}_2^{(n-k)\times(n-k)}$ with $PM = LU$. The algorithm for LUP decomposition is based on the "kij-variant" of the outer-product formulation of Gaussian elimination [26, Section 3.2.9] and is shown in Algorithm 1. One nice feature about the LUP decomposition is that it is almost in-place: L and U are

Algorithm 1 The LUP decomposition used in [21].

Input: $\hat{A} \in \mathbb{F}_2^{(n-k)\times w}$, $w \geq n-k$. The first $n-k$ columns form a matrix $A \in \mathbb{F}_2^{(n-k)\times(n-k)}$.
Output: $L, U, P \in \mathbb{F}_2^{(n-k)\times(n-k)}$ with $PA = LU$ or $\perp$.

$B \leftarrow \hat{A}$
for $\ell = 0$ to $n-k-1$ **do**
    $r_\ell \leftarrow \ell$
end for
for $\ell = 0$ to $n-k-1$ **do**
    for $i = \ell+1, \ldots, n-k-1$ **do**
        if $B_{i,\ell} = 1$ **then**
            swap row i with row ℓ
            swap r_i with r_ℓ
        end if
    end for
    if $B_{\ell,\ell} \neq 1$ **then**
        return $\perp$
    end if
    for $i = \ell+1, \ldots, n-k-1$ **do**
        for $j = \ell+1, \ldots, w-1$ **do**
            $B_{i,j} = B_{i,j} - B_{i,\ell} \cdot B_{\ell,j}$
        end for
    end for
end for
$L \leftarrow$ the lower triangular part of the first $n-k$ columns of B
$U \leftarrow$ the upper triangular part of the first $n-k$ columns of B
$P \leftarrow$ the permutation matrix represented by $r_0, \ldots, r_{n-k-1}$
return L, U, P

computed and stored in the space of M, and P is stored as an array of $n-k$ indices $r_0, \ldots, r_{n-k-1}$, such that entry r_i of row i is 1 for all i. The memory requirement for the LUP decomposition is thus close to $(n-k)^2$ bits.

In order to compute the public key, Roth, Karatsiolis, and Krämer proposed to compute L^{-1} from L in an in-place fashion, compute U^{-1} from U in an in-place fashion, compute $L^{-1}U^{-1}$ in an in-place fashion, and multiply $L^{-1}U^{-1}$ with P by "permuting the columns of $L^{-1}U^{-1}$" to obtain M^{-1}. Finally, M^{-1} is multiplied with $\hat{T}$ to obtain the public key. We note that it is not explained in [21] how the column permutation is carried out exactly. Carrying out the column permutation in a naive way can lead to usage of secret-dependent memory indices.

In fact, Roth, Karatsiolis, and Krämer are not the first ones to use LUP decomposition for key generation. In early 2020 (during the second round), one of the authors of this paper wrote an implementation for the key generation process, which already makes use of (a variant of) LUP decomposition to accelerate public key generation. For ease of presentation, we call the author "author A" in the remainder of this section. Author A's implementation was included in `supercop-20200531` under `crypto_kem/mceliece*/avx` and `crypto_kem/mceliece*/sse`, but has not been published in any conference/journal papers. As opposed to [21], author A's implementation covers all the 10 parameter sets. The `avx` and `sse` implementations in the 3rd-round submission of Classic McEliece are adapted from author A's implementation and use the same algorithm for LUP decomposition and the same steps afterwards to obtain T.

To generate the public key for a non-f parameter set, author A's implementation performs an LUP decomposition on M, such that as long as M is invertible, we obtain a

lower-triangular matrix $L^{-1} \in \mathbb{F}_2^{(n-k)\times(n-k)}$, an upper-triangular matrix $U \in \mathbb{F}_2^{(n-k)\times(n-k)}$, and a permutation matrix $P \in \mathbb{F}_2^{(n-k)\times(n-k)}$ with $PM = LU$. The pseudocode for the LUP decomposition is shown in Algorithm 3. One can understand Algorithm 3 as an algorithm that is essentially the same as Algorithm 1 but with some extra operations to convert L into L^{-1}. It is easy to see that author A's LUP decomposition again takes about $(n-k)^2$ bits.

After the LUP decomposition, $\hat{T}$ is first multiplied (one the left) by P. The multiplication is carried out by a sorting network to avoid usage of secret-dependent memory indices, which is explained in detail in Section 3.4. Then, the result is multiplied on the left by by L^{-1} and U^{-1} sequentially. The multiplication by L^{-1} and U^{-1} are carried out as two sequences of fundamental row operations. Note that the row operations in the two sequences are explicitly shown in the entries of L^{-1} and U, which means there is no need to compute U^{-1} from U. For example, suppose

$$B = \begin{pmatrix} 1 & 0 & 0 \\ b_0 & 1 & 0 \\ b_1 & b_2 & 1 \end{pmatrix},$$

then multiplication by B on the left can be carried out by applying the row operations

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & b_2 & 1 \end{pmatrix}, \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ b_1 & 0 & 1 \end{pmatrix}, \begin{pmatrix} 1 & 0 & 0 \\ b_0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix},$$

sequentially. Likewise, multiplication by B^{-1} on the left can be carried out by applying the row operations in the reverse order.

To generate the public key for an f parameter set, author A's implementation applies Algorithm 3 on the leftmost $(n-k) \times (n-k+\nu-\mu)$ submatrix M' of $\hat{H}$. The first $n-k-\mu$ iterations of the loop starting from line 5 of Algorithm 3 is applied to obtain the first $n-k-\mu$ pivots in the first $n-k-\mu$ columns. Then, a Gaussian elimination is applied on the intersection of the next ν columns and the last μ rows. If the row echelon form of the $\mu \times \nu$ matrix does not have μ pivots, which means it is not full rank, public key generation fails. Otherwise, the specific column permutation introduced by the column indices of the μ pivots is performed on B (which can use the space of M'). Let the matrix formed by first $n-k$ columns of the column-permuted $\hat{H}$ be M, and the matrix formed by the last k columns of the column-permuted $\hat{H}$ be $\hat{T}$. Finally, the last μ iterations of Algorithm 3 are carried out to obtain L, U and P, and $T = (U^{-1}L^{-1}P)\hat{T} = M^{-1}\hat{T}$ is computed in the same way as for the non-f parameter sets. The same implementation strategy can be used with Algorithm 1. The memory demand of applying such an "LUP decomposition" on M' is close to $(n-k)(n-k+\nu-\mu)$ bits.

3.2 Reducing the Memory Demand of For Computing T

Even though the two versions of LUP decomposition both have a memory demand close to $(n-k)^2$ bits for the non-f parameter sets or $(n-k)(n-k+\nu-\mu)$ for the f parameter sets, we still have to multiply M^{-1} or P, L^{-1}, U^{-1} with $\hat{T}$. $\hat{T}$ takes $(n-k) \times k$ bits, e.g., 261120 bytes for `mceliece348864*`, which is apparently larger than the amount of SRAM on our target platform. In order to reduce the memory demand, author A's implementation divides $\hat{T}$ into a few roughly equal-size column blocks. Each block is generated on-demand and multiplied with P, L^{-1}, U^{-1} to generate a part of the public key. [21] uses essentially the same approach, except that $\hat{T}$ is divided into blocks of 8 columns only.

3.3 Our Implementations for Public Key Generation

The discussion above shows several implementation choices.

1. There is a choice between the two versions of LUP decomposition.
2. We may follow [21] to first compute M^{-1} and then multiply it with $\hat{T}$. However, we may also follow author A to apply P, L^{-1}, and U^{-1} to $\hat{T}$ sequentially.
3. When we need to multiply a matrix on the left by L^{-1} or U^{-1}, assuming that only L or U is available, we may follow [21] to compute L^{-1} from L or U^{-1} from U first and then apply the inverse matrix. However, we may also apply the matrix directly as in author A's implementation.
4. We can choose how $\hat{T}$ is decomposed into column blocks.

Algorithm 1 clearly takes fewer operations than Algorithm 3. On the other hand, regarding the steps taken after the LUP decomposition, author A's implementation appears to be faster according to our experiments. We ended up with 2 implementations for (public) key generation.

1. The implementation starts with with Algorithm 1. Then, each column block of $\hat{T}$ is multiplied with P using a sorting network, and L^{-1} and U^{-1} are applied to the result sequentially to obtain a column block of T, without explicitly computing L^{-1} and U^{-1} from L and U. $\hat{T}$ is decomposed into 4 column blocks of 640 columns and 1 column block of 160 columns.
2. This implementation is the same as the 3rd implementation, except that $\hat{T}$ is decomposed into 85 column blocks of 32 columns only to save memory.

Clearly, the amount of SRAM on our target platform is not big enough to hold the public key. In order to deal with this problem, we store the public key in the flash: each time a column block is multiplied by P, U^{-1}, and L^{-1}, the partial public key is written into the flash. We note that the implementation of [21] instead streams out the partial public key.

3.4 Optimizing Matrix Multiplications

Below we describe some optimization techniques that we use for the 2 implementations introduced in the previous section.

Applying P with a Sorting Network Let P be a permutation matrix and consider the task of computing PA for some matrix A. We store P as an array of indices $r_0, \ldots, r_{n-k-1}$, such that r_i is the index of the nonzero entry in row i. In other words, row i of PA is A_{r_i}, where A_i means row i of A. Constructing PA by copying $A_{r_0}, \ldots, A_{r_{n-k-1}}$ is easy, but this allows attackers to obtain information of P via cache-timing attacks. In order to avoid cache-timing attacks, we make use of sorting networks.

In order to apply P to A, we first use a sorting network to sort the list

$$(0, r_0),\ (1, r_1),\ (2, r_2),\ \ldots,\ (n-k-1, r_{n-k-1}),$$

based on the values of the second entries. Let the sorted list be

$$(r'_0, 0),\ (r'_1, 1),\ (r'_2, 2),\ \ldots,\ (r'_{n-k-1}, n-k-1).$$

Then, for each matrix A that needs to be multiplied with P, we sort

$$(r'_0, A_0),\ (r'_1, A_1),\ (r'_2, A_2),\ \ldots,\ (r'_{n-k-1}, A_{n-k-1})$$

based on the values of the first entries to obtain

$$(0, A_{r_0}),\ (1, A_{r_1}),\ (2, A_{r_2}),\ \ldots,\ (n-k-1, A_{r_{n-k-1}}).$$

```
static inline
void matrix_madd_32x4(uint32_t *c, const uint32_t *a, const uint32_t *b)
{
  uint32_t c0=c[0];
  uint32_t c1=c[1];
  uint32_t c2=c[2];
  uint32_t c3=c[3];
  uint32_t a0=a[0];
  uint32_t a1=a[1];
  uint32_t a2=a[2];
  uint32_t a3=a[3];
  for(int i=0;i<32;i++) {
    uint32_t bi=b[i];
    c0 ^= bi*((a0>>i)&1);
    c1 ^= bi*((a1>>i)&1);
    c2 ^= bi*((a2>>i)&1);
    c3 ^= bi*((a3>>i)&1);
  }
  c[0] = c0;
  c[1] = c1;
  c[2] = c2;
  c[3] = c3;
}
void matrix_madd_32x32(uint32_t *c, const uint32_t *a, const uint32_t *b)
{
  for(int i=0;i<32;i+=4)
    matrix_madd_32x4(c+i,a+i,b);
}
```

Figure 1: The function for multiply-and-add of 32×32 submatrices.

Combining the second entries gives PA. For completeness, we note that the sorting network we use is Bather's odd-even mergesort. Bather's odd-even mergesort has a complexity $O(n(\log n)^2)$, where n is the number of elements being sorted.

Applying L^{-1} and U^{-1} with blocking For the two implementations presented in the previous subsection, we need to apply the row operations represented by L^{-1} and U^{-1} to each of the column blocks. In our implementations, we decompose L, U, and each column block T' of $\hat{T}$ into 32×32 submatrices. Then, assuming that we would like to compute $L \cdot T'$, a function is used to multiply a submatrix in L by a submatrix in T' and add the product to another submatrix in T'. As explained in Section 3.1 by the examples of 3×3 matrices, with the right order of submatrix multiplications, the resulting operation will be L^{-1}. The multiply-and-add function is illustrated in in Figure 1. The function allows us to keep 8 32-bit words in registers and keep using them whenever possible.

Table 2: Cycle counts for generating the control bits, the LUP decomposition, computing T from L, U, P, and storing each column block of T into flash memory.

control bits	LUP decomp.	$L, U, P \rightarrow T$	storing T	imple.
615,072,683	187,851,545	354,789,588	378,159,046	a
		617,974,872	366,401,614	b

Table 3: Average (of 100 runs) cycle counts and the corresponding time in seconds for key generation of `mceliece348864*`.

	`mceliece348864`	`mceliece348864f`
impl. a	2,172,369,343 (12.9 s)	1,572,880,048 (9.4 s)
impl. b	2,476,419,846 (14.7 s)	1,824,307,900 (10.9 s)

3.5 Experiment Results

The performance numbers for implementation a and b are shown in Table 2 and Table 3. As shown in Table 3, the key generation time is about 15 seconds for `mceliece3488864` when implementation b is used. Although this is not fast, since Classic McEliece is a scheme with IND-CCA2 security, the user can safely reuse a key pair if no stronger security notion is required. In applications where Classic McEliece is used to achieve forward secrecy, the user can also update the key pair periodically (say, every 10 minutes) such that the key generation time would not be a problem. In addition, the 3rd-round supporting documentation suggests that the user can use a truncated format of secret keys, such that the control bits are not included. This saves key generation time and reduces secret key size, with the cost of slower decapsulation.

One interesting thing we found is that the time to write data into the flash seems to be dependent of the data itself. We are not sure if this is caused by the libraries we use for accessing the flash, or it is actually caused by the hardware. In any case, as we only write the public key T into the flash, the variance in running time caused by accessing the flash does not affect the claim that our implementation is constant-time.

4 Encapsulation

Given an error vector $e \in \mathbb{F}_2^n$ of weight t which serves as the plaintext and a parity-check matrix $H \in \mathbb{F}_2^{(n-k)\times n}$ which serves as the public key, the Niederreiter cryptosystem encrypts e as He. The main component of encapsulation in Classic McEliece is essentially Niederreiter encryption. The encapsulation process starts with generating a uniform random error vector e and then computes He. Recall that Classic McEliece uses H in systematic form, i.e., H of the form $(I_{n-k} \mid T)$, where $T \in \mathbb{F}_2^{(n-k)\times k}$. Our implementation computes He as $e^{(0)} + Te^{(1)}$, where $e^{(0)}$ consists of the first $n-k$ entries of e and $e^{(1)}$ consists of the last k entries of e. Below we explain how the error vector e is generated and how $Te^{(1)}$ is computed in our implementation.

4.1 Generation of the Error Vector

To generate the error vector, the `vec` implementation first generates a list of t random indices that indicate the positions of 1's in e. The indices are then checked for repetition. To check whether there is repetition in the t indices, the `vec` implementation uses a sorting network to sort the indices and then compares every two consecutive indices. The sorting network guarantees that nothing about the list of indices is leaked through timing during sorting. If there is repetition, a new list of t indices will be generated. Otherwise, the

Table 4: Average cycle counts to generate the error vector.

parameter set	quicksort	sorting network
`mceliece348864*`	76,759	115,743
`mceliece460896*`	175,135	269,984

```
#include "stdint.h"
void T_mul_e1(uint32_t *c, const uint32_t T[][ T_NCOLS/32 ], const uint32_t *e1)
{
    for (int i = 0; i < T_NROWS; i++)
    {
        uint32_t b = 0;
        for (int j = 0; j < T_NCOLS/32; j++)
            b ^= T[i][j] & e1[j];

        b ^= b >> 16;
        b ^= b >> 8;
        b ^= b >> 4;
        b ^= b >> 2;
        b ^= b >> 1;
        b &= 1;

        c[ i/32 ] ^= (b << (i%32));
    }
}
```

Figure 2: A simple C function for computing $Te^{(1)}$.

error vector is generated using the indices in constant time. Note that the 3rd-round specification defines how the t indices are generated and regenerated exactly.

Our implementation follows the implementation strategy of the `vec` implementation, except that we use a different sorting algorithm. We observed that there is no need to hide everything related to the list of indices from the attacker, as information of e only lies in the corresponding *set* of t indices. Their order in the list is independent of e. Therefore, one can use a sorting algorithm that leaks the order through timing, as long as nothing else about the indices is leaked through timing. In fact, one can use any comparison-based sorting algorithm. In our implementation, we use quicksort instead of a sorting network to sort the indices. This simple change gives a noticeable speedup in error-vector generation, as shown in Table 4.

We note that non-comparison-based sorting algorithms, however, can leak information about the error vector through timing. An example of this is bucket sort, which uses memory indices that depends on the values being sorted.

4.2 Computation of $Te^{(1)}$

In Classic McEliece, the public key is defined as the row-major representation of T. A simple way to obtain $Te^{(1)}$ is to compute the inner product between the first row of T and $e^{(1)}$, compute the inner product between the second row of T and $e^{(1)}$, and so on. To compute each inner product, one may prepare a 32-bit variable which is set to zero. Then, AND the first 32-bit words of the row and $e^{(1)}$, XOR the result into the variable, AND the second 32-bit words of the row and $e^{(1)}$, XOR the result into the variable, and so one. Finally, compute the parity of the 32-bit variable using a sequence of shifts and XORs. Then the inner product is equal to the parity. Figure 2 shows a C function that implements this strategy. Essentially the same strategy is used in the `vec` implementation, except that `vec` uses 64-bit words.

The code in Figure 2 loads all 32-bit words in $e^{(1)}$ for each i. In other words, $e^{(1)}$ is loaded $n-k$ times in the matrix-vector multiplication. A simple way to reduce the number of load instructions is to store the whole $e^{(1)}$ in registers. However, this is not possible as $e^{(1)}$ would take $(3488 - 12 \cdot 64)/32 = 85$ registers to store even for `mceliece348864*`. Instead,

each time we load a 32-bit word in $e^{(1)}$, we always perform `AND`'s with 4 corresponding 32-bit words from 4 different rows. In this way, the number of loads for $e^{(1)}$ can be reduced by a factor of 4. Also, we load 3 consecutive 32-bit words from $e^{(1)}$ at once whenever possible and perform the corresponding `AND`s and `XOR`s with the corresponding 32-bit words from 4 rows. In this way, the number of memory accesses for the destination array can also be reduced by a factor of 3. In conclusion, each iteration of the inner loop in our implementation handles computation for a 4×96 submatrix of T, which is different from handling only a 1×32 submatrix of T in each iteration as in Figure 2.

In order to process 4 rows at the same time, we need to maintain 4 32-bit variables b_0, b_1, b_2, b_3, one for each row. Instead of reducing b_i's one-by-one, we use the following inline function to reduce b_0, b_1, b_2, b_3 in parallel to obtain 4 inner products.

```
static inline uint32_t parallel_reduce(uint32_t b0, uint32_t b1,
                                       uint32_t b2, uint32_t b3)
{
    b0 ^= b0 >> 2; b1 ^= b1 >> 2; b2 ^= b2 >> 2; b3 ^= b3 >> 2;
    b0 ^= b0 >> 1; b1 ^= b1 >> 1; b2 ^= b2 >> 1; b3 ^= b3 >> 1;
    b0 &= 0x11111111; b1 &= 0x11111111; b2 &= 0x11111111; b3 &= 0x11111111;
    b0 ^= b1<<1; b0 ^= b2<<2; b0 ^= b3<<3;
    b0 ^= b0>>4; b0 ^= b0>>8; b0 ^= b0>>16;
    return b0 & 0xF;
}
```

The function consists of 19 instructions, while reducing b_i's one-by-one takes $4 \cdot 6 = 24$ instructions (see Figure 2). We note that our function for $T \cdot e^{(1)}$, which includes the code for `parallel_reduce`, is written in assembly. The numbers of cycles for the two versions of matrix-vector multiplication are shown in Table 5.

5 Decapsulation

Given the secret key and a ciphertext of the form $c = He$, the decryption of the Niederreiter cryptosystem recovers e using a decoding algorithm. The main component of decapsulation in Classic McEliece is Niederreiter decryption. In this section, we show how we optimize the Berlekamp decoder, which is used in [6], [8], and all the four implementations of the Classic McEliece team.

5.1 Representations of Field Elements

Field arithmetics in $\mathbb{F}_{2^m}$ (recall that $m \in \{12, 13\}$), and in particular field multiplications, are critical building blocks in the decoding algorithm. In the specification, $\mathbb{F}_{2^{12}}$ is constructed as $\mathbb{F}_2[x]/(x^{12} + x^3 + 1)$ and $\mathbb{F}_{2^{13}}$ is constructed as $\mathbb{F}_2[x]/(x^{13} + x^4 + x^3 + x + 1)$. Our implementation, uses two representations for field elements:

- The bitsliced representation, which is used in the additive FFTs, the transposed additive FFTs.
- The "radix-16" representation, which is only used in the Berlekamp-Massey algorithm.

Below we show how field multiplications and inversions are optimized when the two representations are used.

Table 5: Cycle counts for computing $T \cdot e^{(1)}$.

parameter set	our code	Figure 2
`mceliece348864*`	389,032	720,592
`mceliece460896*`	779,246	1,525,850

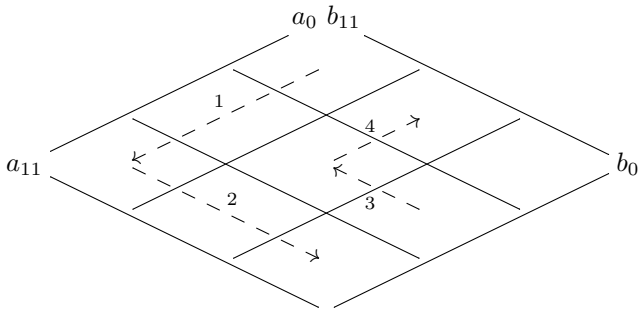

Figure 3: The order of computing $(a_0 + \cdots + a_{11}x^{11}) \cdot (b_0 + \cdots + b_{11}x^{11})$

5.1.1 The Bitsliced Representation

The bitsliced field multiplication used in the `vec` implementation has been explained in [8]. The algorithm takes $2m$ 64-bit words $a_0, \ldots, a_{m-1}, b_0, \ldots, b_{m-1}$ as inputs and outputs m 64-bit words $c_0, \ldots, c_{m-1}$. The first phase of the algorithm, which we call the polynomial multiplication phase, consists of 64 polynomial multiplications with schoolbook multiplication. This is achieved by using m^2 AND instructions and $(m-1)^2$ XOR instructions. The second phase of the algorithm, which we call the reduction phase, reduces the 64 products modulo the irreducible polynomial. On M4, we can easily build a 32-bit version of the same function, such that 32 field multiplications are carried out at the same time. However, a naive implementation in C will leads to many register spills.

In order to reduce the number of memory accesses, we follow [12] divide the polynomial multiplication phase into small pieces so that each piece only requires a small set of a_i's and b_i's. Figure 3 shows how we divide the polynomial multiplication phase for $\mathbb{F}_{2^{12}}$. As shown in the figure, the whole polynomial multiplication phase is divided into several rhombuses, where each rhombus involves only 4 consecutive a_i's and 4 consecutive b_i's. For the first rhombus, we load a_0, a_1, a_2, a_3 and b_8, b_9, b_{10}, b_{11}'s from memory. Then, each time we move on to the next rhombus, we only need to load either 4 consecutive a_i's or 4 consecutive b_i's from memory.

When each rhombus is processed, 7 c_i's have to be updated. One approach is to load the c_i's from memory, update them, and then store them back to memory. However, a slightly cheaper approach is to store c_i's in the FP registers. Indeed, moving data between with a GP register and an FP register with `vmov` takes 1 cycle, while loading a word from SRAM would take more than 1 cycle on average (see Section 2). Note that there is always an overlap between the set of c_i's used in the current rhombus and the set of c_i's used in the next rhombus: the intersection is always a set of 3 c_i's. We always keep the 3 c_i's in registers without storing them to FP registers when we move on to the next rhombus. In this way, we can save some `vmov` instructions.

Bitsliced multiplications for $\mathbb{F}_{2^{13}}$ is implemented using essentially the same strategy. Table 6 shows the cycle counts for performing 32 field multiplications in a bitsliced fashion.

Table 6: Average cycle counts for one field multiplications when using the bitsliced representation. The actual functions carry out 32 multiplications in parallel.

	our code	naive `C`
$\mathbb{F}_{2^{12}}$	22.8	44.1
$\mathbb{F}_{2^{13}}$	26.9	52.5

```
uint32_t gfu32_2_12_mul( uint32_t a, uint32_t a, uint32_t b, uint32_t b) {
    const uint32_t mask = 0x11111111;
    uint32_t a0 = a&mask;     uint32_t a1 = (a>>1)&mask;
    uint32_t b0 = b&mask;     uint32_t b1 = (b>>1)&mask;
    union{ uint64_t v64; uint32_t v32[2]; } t0, t1;
    uint32_t t2;
    // multiplication
    t0.v64  = ((uint64_t)a0) * ((uint64_t)b0);
    t2  = a1 * b1;
    t1.v32[0] = t0.v32[1];
    t1.v32[1] = t2;
    t1.v64  += ((uint64_t)a1) * ((uint64_t)b0);
    t1.v64  &= 0x1111111111111111ULL;
    t1.v64  += ((uint64_t)a0) * ((uint64_t)b1);
    t0.v32[1] = t1.v32[0];
    t2        = t1.v32[1];
    // reduction
    const uint32_t rd_x12 = 0x1001;     // x^12 = x^3 + 1
    const uint32_t rd_x16 = rd_x12<<16; // x^16 = x^7 + x^4
    t0.v64   += ((uint64_t)(t2&0x11111111)) * ((uint64_t)(rd_x16));
    t0.v64   &= 0x1111111111111111ULL;
    t0.v64   += ( ((uint64_t)rd_x12) * ((uint64_t)(t0.v32[1]>>16)) );
    // output
    t0.v32[0] &= mask;
    t0.v32[1] &= (mask>>16);  // 12 terms only
    return t0.v32[0]|(t0.v32[1]<<1);
}
```

Figure 4: Multiplication of $\mathbb{F}_{2^{12}}$ for data in raidx-16 format

5.1.2 The Radix-16 Representation

The bitliced field multiplication is efficient regarding the number of cycles per multiplication. However, 32 multiplications have to be carried out simultaneously to make full use of it. Individual field multiplications in $\mathbb{F}_{2^m}$ are easy to implement on platforms with instructions for carryless multiplications, such as `pclmulqdq`. On M4, there is no native instruction for carryless multiplications, but we found that carryless multiplications can still be carried out using instructions for integer multiplication.

To carry out carryless multiplications using integer multiplications, consider each polynomial $a = a_0 + a_1x + a_2x^2 + \cdots + a_7x^7 \in \mathbb{F}_2[x]$ as a 32-bit integer $a_0 + a_1 2^4 + a_2 2^8 + \cdots + a_7 2^{28}$. Then, the product c of $a = \sum_{i=0}^{7} a_i x^i$ and $b = \sum_{i=0}^{7} b_i x^i$ can be computed by multiplying the corresponding 32-bit integers. Indeed, the result of the integer multiplication is

$$\begin{aligned}&(a_0 \cdot 2^0 + a_1 \cdot 2^4 + a_2 \cdot 2^8 + \cdots + a_7 \cdot 2^{28}) \cdot (b_0 \cdot 2^0 + b_1 \cdot 2^4 + b_2 \cdot 2^8 + \cdots + b_7 \cdot 2^{28})\\ =&a_0 \cdot b_0 \cdot 2^0 + (a_1 \cdot b_0 + a_0 \cdot b_1) \cdot 2^4 + (a_2 \cdot b_0 + a_1 \cdot b_1 + a_0 \cdot b_2) \cdot 2^8 + \cdots + (a_7 \cdot b_7) \cdot 2^{56},\end{aligned}$$

where the bit of index $4i$ is exactly c_i. We thus use the `umaal` instruction to perform a small carryless multiplication. Our implementation of field multiplication use `umaal` a few times to perform the polynomial multiplications and reductions modulo the irreducible polynomial.

Figure 4 shows a multiplication function for $\mathbb{F}_{2^{12}}$. The function takes 4 `umaal` for the polynomial multiplication phase and 2 `umaal` for the reduction phase. Our multiplication function for $\mathbb{F}_{2^{13}}$ also takes 4 `umaal` for the polynomial multiplication phase and 2 `umaal` for the reduction phase, but 1 `mla` is used in the reduction phase to perform a smaller carryless multiplication. We note that the actual code we use is written in assembly.

To compute the multiplicative inverse of an element in $\mathbb{F}_{2^m}$, we raise the element to

Algorithm 2 Massey's version of the Berlekamp-Massey algorithm

Input : a sequence of $2t$ elements $(S_0, \ldots, S_{2t-1})$ in $\mathbb{F}_{2^m}$.
Output: a minimal polynomial $\sigma(x)$ generating the input sequence.

$\sigma(x) \leftarrow 1 \in \mathbb{F}_{2^m}[x]$
$\beta(x) \leftarrow x \in \mathbb{F}_{2^m}[x]$
$\delta = 1 \in \mathbb{F}_{2^m}$
$\ell \leftarrow 0 \in \mathbb{Z}$
for $k \leftarrow 0$ to $2t-1$ **do**
  $d \leftarrow \sum_{i=0}^{t} \sigma_i \cdot S_{k-i}$ ▷ $\sigma(x) = \sigma_0 + \sigma_1 x + \sigma_2 x^2 + \cdots$
  if $d = 0$ or $k < 2\ell$ **then**
    $(\sigma(x), \beta(x), \ell, \delta) \leftarrow (\sigma(x) - \frac{d}{\delta}\beta(x), \beta(x), \ell, \delta)$
  else
    $(\sigma(x), \beta(x), \ell, \delta) \leftarrow (\sigma(x) - \frac{d}{\delta}\beta(x), \sigma(x), k - \ell + 1, d)$
  end if
  $\beta(x) \leftarrow x\beta(x)$
end for
return $\sigma(x)$

the power of $2^m - 2$. It costs 11 squares and 5 multiplication for $\mathbb{F}_{2^{12}}$ and 12 squares and 4 multiplications for $\mathbb{F}_{2^{13}}$. The cycle counts for one field multiplication and one inversion are listed in Table 7.

5.2 Optimizing the Berlekamp-Massey algorithm

The `vec` implementation follows [8] to use one version of the Berlekamp-Massey algorithm (BM) by Xu [27]. Xu's algorithm was adapted from the version by Massey [16], of which the pseudocode is is shown in Algorithm 2. The main difference between the two versions is that, while Massey's version requires to compute a field inversion in each of the $2t$ iterations, in Xu's version the inversion is replaced by multiplications: in Xu's version $\sigma(x)$ is updated to $\delta\sigma(x) - d\beta(x)$.

Note that Massey's version is expected to be faster. To see this why this is the case, observe that the maximal degrees of polynomials $\sigma(x)$ and $\beta(x)$ grow by 1 in each iteration. This means that in a constant-time implementation the numbers of coefficients that we maintain for $\sigma(x)$ and $\beta(x)$ also have to grow by 1 in each of the $2t$ iterations, even though the polynomials might actually have fewer coefficients. Recall that a field inversion takes 390 and 375 cycles for $\mathbb{F}_{2^{12}}$ and $\mathbb{F}_{2^{13}}$, which is fewer than the cycle count for 18 multiplications, no matter whether the bitsliced or the radix-16 data representation is used (see Table 6 and 7). As the average lengths for $\sigma(x)$ and $\beta(x)$ are more than 18, we concluded that Massey's version should outperform Xu's version.

In addition to the choice between Massey's version and Xu's version, we also have the choice between using the bitsliced representation or the radix-16 representation for $\alpha(x)$ and $\beta(x)$. Using the bitsliced representation gives a better cycle count per multiplication.

Table 7: Cycle counts for one multiplication and one inversion with the radix-16 representation. For the cycle counts for multiplication, we perform 128 multiplications in our assembly function and divide the resulting cycle counts by 128 to avoid the function call overhead.

	multiplication	inversion
$\mathbb{F}_{2^{12}}$	26.2	390
$\mathbb{F}_{2^{13}}$	27.2	375

Table 8: Cycle counts of BM with different implementation choices. Note that the parameter sets `mceliece348864*` use $(m, t) = (12, 64)$; `mceliece460896*` use $(m, t) = (13, 96)$; `mceliece6688128*` and `mceliece8192128*` use $(m, t) = (13, 128)$.

m	t	representation	Massey's	Xu's
12	64	radix-16	464,017	570,827
		bitsliced	486,289	602,694
13	96	radix-16	1,051,988	1,307,557
		bitsliced	1,124,585	1,447,112
13	128	radix-16	1,800,708	2,290,832
		bitsliced	1,854,254	2,450,188

However, we found that using the radix-16 representation also has some advantages. Below we show the main difference in implementing BM using the two representations.

1. As discussed above, there is a need to increase the numbers of coefficient for $\sigma(x)$ and $\beta(x)$ during BM. When using the radix-16 representation for $\sigma(x)$ and $\beta(x)$, we can simply add one new coefficient in each iteration. On the other hands, with the bitsliced representation, the best we can do is to add 32 new coefficients every 32 iterations. In other words, the radix-16 representation offers a better granularity regarding the numbers of coefficients, and this allows the radix-16 representation to avoid many dummy operations compared to the bitsliced representation.

2. Computation of $d = \sum_{i=0}^{t} \sigma_i \cdot S_{k-i}$ in line 6 allows "lazy reduction": one can compute the results of the polynomial multiplication phase for each $\sigma_i \cdot S_{k-i}$, compute the sum of the results, and perform only one reduction phase to obtain d. To implement lazy reduction, when using the radix-16 representation, we can store the sum of all $\sigma_i \cdot S_{k-i}$ that we have computed so far in GP registers and use the remaining GP registers to compute the next $\sigma_i \cdot S_{k-i}$. When using the bitsliced representation, we can first compute 32 $\sigma_i \cdot S_{k-i}$ and store the results of the polynomial multiplication phase in 23 or 25 FP registers. Then, for each 32 $\sigma_i \cdot S_{k-i}$ that have not been computed, we add the results of the polynomial multiplication phase to the FP registers. After all $\sigma_i \cdot S_{k-i}$ are processed, we perform a reduction phase and add the 32 values to obtain the d.

3. In line 12, $\beta(x)$ is updated to $x \cdot \beta(x)$. When using the radix-16 representation, we can use a pointer to access each coefficient β_i, and the update is carried out by simply decreasing the value of the pointer by 1. On the other hand, when using the bitsliced representation, we need to perform a sequence of shifts and ORs on 32-bit words.

The discussion above shows that there are 4 ways to implement BM due to the choice between Massey's version and Xu's version, and the choice between the bitsliced and the radix-16 representations. To see which one gives the most efficient implementation, we actually implemented BM in four different ways. The optimization techniques above such lazy reduction are used in our implementation. The results are summarized in Table 8. As shown in the table, using Massey's version with the radix-16 representation gives the best performance, so our final implementation takes this approach.

One interesting thing we found is that the bitsliced representation gives a better performance when the frequency is reduced to 24 MHz, the default frequency used in `pqm4`. We show the cycle counts for the four implementations when the frequency is set to 24 MHz in Table 12.

5.3 Optimizing the Beneš Network

[8] uses a Beneš network to permute bit arrays of length n. The permutations are defined by $\alpha_1, \ldots, \alpha_n$. The structure of the Beneš network is depicted in [8, Figure 3]. We note that the implementations of the Classic McEliece team use an equivalent view, such that the first and last stages consist of conditional swaps between consecutive elements. The naive approach to carry out the network is to deal with the layers sequentially. Following the strategy of [8, Section 3], we can deal with each stage with the following building block.

- Load 2 32-bit words.
- Perform the corresponding 32 conditional swaps.
- Store the 2 32-bit elements.

The building block is used $2^{m-1}/32$ times for each stage. This means we need to load and store all the 32-bit words in every layer. In order to reduce the number of memory accesses, we carry out conditional swaps in two layers at the same time, such that the building block in our implementation consists of the following operations.

- Load 4 32-bit words.
- Perform the corresponding $2 \cdot 2 \cdot 32 = 128$ conditional swaps.
- Store the 4 32-bit elements.

In this way, the number of memory accesses is reduced by a factor of 2.

5.4 Cycles Count For Each Component in Decryption

Table 9 shows the cycle counts for the whole Niederreiter decryption and the components of decryption. We use the terms in [8] in the table: "key eq" stands for the Berlekamp-Massey algorithm, "root" stands for the additive FFT for root finding, "synd" stands for the transposed additive FFT for syndrome computation, and "perm" stands for the Beneš network to perform the permutation (or the inverse of it) introduced by the support. The cycle count for decryption, as explained in [8], is expected to be close to

$$\text{“perm”} \times 2 + \text{“synd”} \times 2 + \text{“key eq”} + \text{“root”} \times 2.$$

Note that one "synd" is used for re-encryption, which is an important step to achieve CCA security. The numbers of "synd", "root", and "perm" for `mceliece460896*` and `mceliece6688128*` are expected to be close to the numbers for `mceliece8192128*`, so we decided not to show the numbers in the table.

6 Comparisons and Memory Usage

This section shows memory usage of our implementation and comparisons with the Classic McEliece implementation of [21] and implementations of other schemes.

Table 9: Cycle counts for each component of decryption.

	decryption	key eq	synd	root	perm	reference
`mceliece348864*`	2,672,525	464,017	456,336	328,899	90,849	our code
	5,788,346	1,428,397	875,579	662,720	271,849	`vec`
`mceliece8192128*`	7,396,245	1,800,708	1,142,521	996,623	212,038	our code
	17,239,538	2,184,084	1,933,911	1,933,911	411,694	`vec`

6.1 Comparison with the Implementation by Roth et al.

Roth, Karatsiolis, and Krämer [21] reported that it takes 1,938,512,183 cycles to generate the "extended secret key" for `mceliece348864`. The extended secret key does not include the control bits of the Beneš network, but it include M^{-1}. It is also reported in [21] that 667,392,425 cycles are required to obtain T from M^{-1}, so in total $2,605,904,608$ is required to obtain T.

For comparison, if the time for computing the control bits is excluded from the key generation time, implementation a would take $2,172,369,343 - 615,072,683 = 1,557,296,660$ cycles and implementation b would take $2,476,419,846 - 615,072,683 = 1,861,347,163$ cycles. These numbers are both much smaller than $2,605,904,608$, even though they actually include the time to write T into flash memory.

The implementation of [21] is able to carry out encapsulation for `mcelice348864` in 3,106,183 cycles and encapsulation for `mcelice460896` in 5,868,529 cycles, but it seems that the numbers do not include the time to generate the error vector. The numbers in Table 1 show that our implementation is much faster.

6.2 Comparison with Other NIST Post-quantum Candidates

Table 10 shows the performance numbers of the third-round candidates FrodoKEM [2], KYBER [22], Saber [9], NTRU [7], and SIKE [13]. The numbers are measured at 24 MHz, which means the numbers for 168 MHz are expected to be larger. Compared to the level-1 parameter set of FrodoKEM, our encapsulation time for `mceliece348864*` is more than 79 times faster and our decapsulation time for `mceliece348864*` is more than 17 times faster. Compared to other lattice-based schemes, our encapsulation time and decapsulation time are not as fast but are still reasonably efficient.

The reader might wonder why we did not include performance numbers of other 3rd-round code-based schemes. We did not include the numbers because we are not aware of any implementation of the schemes that are both constant-time and tailored for Cortex-M4. The BIKE [3] team, for example, does have an implementation named `portable` which can run on Cortex-M4, but it is neither constant-time nor tailored for Cortex-M4.

6.3 Memory Usage

We use the system tool `size` for the size of required ROM in our implementation, and it reports 618,796/2488/68 for the `.text/.data/.bss` sections of the objective file. With the 1 MB flash, `stm32f4-discovery` is capable of storing the program code on the device. Table 11 reports the stack usage of key generation, encapsulation, and decapsulation. Since

Table 10: Performance numbers of the 3rd-round KEMs from `pqm4` commit-6841a6b.

scheme (implementation)	level	key generation	encapsulation	decapsulation
`frodokem640aes` (m4)	1	48,348,105	47,130,922	46,594,383
`kyber512` (m4)	1	463,343	566,744	525,141
`kyber768` (m4)	3	763,979	923,856	862,176
`lightsaber` (m4f)	1	361,687	513,581	498,590
`saber` (m4f)	3	654,407	862,856	835,122
`ntruhps2048509` (m4f)	1	79,658,656	564,411	537,473
`ntruhps2048677` (m4f)	3	143,734,184	821,524	815,516
`sikep434` (m4)	1	48,264,129	78,911,465	84,276,911
`sikep610` (m4)	3	119,480,622	219,632,058	221,029,700

Table 11: Stack usage for the three operations.

	key gen. (impl. a)	key gen. (impl. b)	encap.	decap.
`mceliece348864f`	$121,744 + 64,512$	$121,744 + 6,144$	1,420	16,164
`mceliece348864`	$121,712 + 64,512$	$121,656 + 6,144$		
`mceliece460896*`			2,004	34,964
`mceliece6688128*`				35,724
`mceliece8192128*`				36,100

the `stm32f4-discovery` provides two stack spaces (128KB and 64KB), we report the stack usage of the two spaces separately.

The stack usage is measured with a modified version of the stack tool in `pqm4`. We need to modify the tool so that it knows that the public key is in flash memory. In addition, the tool only measures the amount of stack memory within larger piece (128 KB) of SRAM, and there is a limit on the amount it is able to measure. We modified the tool so that it does not have the limit. For the stack memory within the smaller piece (64 KB) of SRAM, we count manully.

References

[1] Martin Albrecht, Daniel J. Bernstein, Tung Chou, Carlos Cid, Jan Gilcher, Tanja Lange, Varun Maram, Ingo von Maurich, Rafael Misoczki, Ruben Niederhagen, Kenneth G. Paterson, Edoardo Persichetti, Christiane Peters, Peter Schwabe, Nicolas Sendrier, Jakub Szefer, Cen Jung Tjhai, Martin Tomlinson, and Wen Wang. Classic Mceliece, 2017. https://classic.mceliece.org/.

[2] E. Alkim, J.W. Bos, L. Ducas, P. Longa, I. Mironov, M. Naehrig, C. Peikert V. Nikolaenko, and D. Stebila A. Ragunathan. FrodoKEM: Practical quantum-secure key encapsulation from generic lattices, 2019. https://frodokem.org.

[3] Nicolás Aragón, Paulo S. L. M. Barreto, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Shay Gueron, Tim Güneysu, Carlos Aguilar Melchor, Rafael Misoczki, Edoardo Persichetti, Nicolas Sendrier, Jean-Pierre Tillich, and Gilles Zémor. BIKE – bit flipping key encapsulation, 2020. https://bikesuite.org.

[4] ARM. ARM Cortex-M4 processor technical reference manual revision r0p1, 2009. https://developer.arm.com/docs/100166/0001.

[5] Václav E Beneš. *Mathematical theory of connecting networks and telephone traffic*. Academic press, 1965.

[6] Daniel J. Bernstein, Tung Chou, and Peter Schwabe. Mcbits: Fast constant-time code-based cryptography. In Guido Bertoni and Jean-Sébastien Coron, editors, *Cryptographic Hardware and Embedded Systems - CHES 2013 - 15th International Workshop, Santa Barbara, CA, USA, August 20-23, 2013. Proceedings*, volume 8086 of *Lecture Notes in Computer Science*, pages 250–272. Springer, 2013. doi:10.1007/978-3-642-40349-1_15.

[7] Cong Chen, Oussama Danba, Jeffrey Hoffstein, Andreas Hülsing, Joost Rijneveld, Tsunekazu Saito, John M. Schanck, Peter Schwabe, William Whyte, Keita Xagawa, Takashi Yamakawa, and Zhenfei Zhang. NTRU: A submission to the nist post-quantum standardization effort, 2020. https://ntru.org/index.shtml.

[8] Tung Chou. Mcbits revisited. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, volume 10529 of *Lecture Notes in Computer Science*, pages 213–231. Springer, 2017. doi:10.1007/978-3-319-66787-4_11.

[9] Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, and Frederik Vercauteren. SABER, MLWR-based KEM, 2019. https://www.esat.kuleuven.be/cosic/pqcrypto/saber/.

[10] Federal Office for Information Security. Technical guideline – cryptographic algorithms and key lengths. 2020. https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/TechGuidelines/TG02102/BSI-TR-02102-1.pdf?__blob=publicationFile&v=10.

[11] Shuhong Gao and Todd Mateer. Additive fast fourier transforms over finite fields. *IEEE Transactions on Information Theory*, 56(12):6265–6272, 2010. doi:10.1109/TIT.2010.2079016.

[12] Michael Hutter and Erich Wenger. Fast multi-precision multiplication for public-key cryptography on embedded microprocessors. In Bart Preneel and Tsuyoshi Takagi, editors, *Cryptographic Hardware and Embedded Systems - CHES 2011 - 13th International Workshop, Nara, Japan, September 28 - October 1, 2011. Proceedings*, volume 6917 of *Lecture Notes in Computer Science*, pages 459–474. Springer, 2011. doi:10.1007/978-3-642-23951-9_30.

[13] David Jao, Reza Azarderakhsh, Matthew Campagna, Craig Costello, Luca De Feo, Basil Hess, Aaron Hutchinson, Amir Jalali, Koray Karabina, Brian Koziel, Brian LaMacchia, Patrick Longa, Michael Naehrig, Geovandro Pereira, Joost Renes, Vladimir Soukharev, and David Urbanik. Sike — supersingular isogeny key encapsulation, 2020. https://sike.org/.

[14] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. pqm4: Testing and benchmarking NIST PQC on ARM Cortex-M4. 2019. https://eprint.iacr.org/2019/844.

[15] Silicon Labs. EFM32 giant gecko series 1 32-bit microcontrollers (MCUs), 2020. https://www.silabs.com/mcu/32-bit/efm32-giant-gecko-gg11.

[16] James Massey. Shift-register synthesis and bch decoding. *IEEE transactions on Information Theory*, 15(1):122–127, 1969. http://crypto.stanford.edu/~mironov/cs359/massey.pdf.

[17] Robert J. McEliece. A public key cryptosystem based on algebraic coding theory. *JPL DSN Progress Report*, 1978. https://tmo.jpl.nasa.gov/progress_report2/42-44/44N.PDF.

[18] Harald Niederreiter. Knapsack-type cryptosystems and algebraic coding theory. *Problems of Control and Information Theory*, 15(2):159–166, 1986.

[19] NIST. Post-quantum cryptography: Call for proposals, 2017. https://csrc.nist.gov/Projects/post-quantum-cryptography/Post-Quantum-Cryptography-Standardization/Call-for-Proposals.

[20] NIST. PQC standardization process: Third round candidate announcement, July 2020. https://csrc.nist.gov/News/2020/pqc-third-round-candidate-announcement.

[21] Johannes Roth, Evangelos Karatsiolis, and Juliane Krämer. Classic Mceliece implementation with low memory footprint. In Pierre-Yvan Liardet and Nele Mentens, editors, *Smart Card Research and Advanced Applications, 19th International Conference, CARDIS 2020, Lübeck, Germany, November 18–20, 2020, Revised Selected Papers*, volume 12609 of *Lecture Notes in Computer Science*. Springer, 2021. To appear. https://cardis2020.its.uni-luebeck.de/files/CARDIS2020_Roth_ClassicMcElieceImplementation_paper.pdf.

[22] Peter Schwabe, Roberto Avanzi, Joppe Bos, Leo Ducas, Eike Kiltz, Tancrede Lepoint, Vadim Lyubashevsky, John M. Schanck, Gregor Seiler, and Damien Stehle. Crystals-kyber, 2019. https://pq-crystals.org/kyber/index.shtml.

[23] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509, October 1997. doi:10.1137/s0097539795293172.

[24] STMicroelectronics. RM0090 reference manual, 2019. https://www.st.com/resource/en/reference_manual/dm00031020-stm32f405-415-stm32f407-417-stm32f427-437-and-stm32f429-439-advanced-arm-based-32-bit-mcus-stmicroelectronics.pdf.

[25] STMicroelectronics. Stm32f405xx, stm32f407xx, 2020. https://www.st.com/resource/en/datasheet/dm00037051.pdf.

[26] Charles F. Van Loan and Gene H. Golub. *Matrix computations*. Johns Hopkins University Press Baltimore, 1983.

[27] Youzhi Xu. Implementation of berlekamp-massey algorithm without inversion. *IEE Proceedings I-Communications, Speech and Vision*, 138(3):138–140, 1991. doi:10.1049/ip-i-2.1991.0018.

A Cycle Counts for the Berlekamp-Massey Algorithm at 24 MHz

Table 12: Cycle counts of BM with different implementation choices when the frequency is set to 24 MHz. Note that the parameter sets `mceliece348864*` use $(m, t) = (12, 64)$; `mceliece460896*` use $(m, t) = (13, 96)$; `mceliece6688128*` and `mceliece8192128*` use $(m, t) = (13, 128)$.

m	t	representation	Massey's	Xu's
12	64	radix-16	438,802	570,250
		bitsliced	397,124	489,916
13	96	radix-16	1,018,576	1,306,895
		bitsliced	922,033	1,164,257
13	128	radix-16	1,756,409	2,290,297
		bitsliced	1,523,876	1,960,530

B The LUP Decomposition Algorithm Used in Author A's Implementation

Algorithm 3 The LUP decomposition used in author A's implementation.

Input: $\hat{A} \in \mathbb{F}_2^{(n-k)\times w}$, $w \geq n-k$. The first $n-k$ columns form a matrix $A \in \mathbb{F}_2^{(n-k)\times(n-k)}$.
Output: $L^{-1}, U, P \in \mathbb{F}_2^{(n-k)\times(n-k)}$ with $PA = LU$ or $\perp$.

$B \leftarrow \hat{A}$
for $\ell = 0$ to $n-k-1$ **do**
    $r_\ell \leftarrow \ell$
end for
for $\ell = 0$ to $n-k-1$ **do**
    for $i = \ell+1, \ldots, n-k-1$ **do**
        if $B_{i,\ell} = 1$ and $B_{\ell,\ell} = 0$ **then**
            swap row i of B with row ℓ of B
            swap r_i with r_ℓ
        end if
    end for
    if $B_{\ell,\ell} \neq 1$ **then**
        return $\perp$
    end if
    for $i = \ell+1, \ldots, n-k-1$ **do**
        for $j = 0, \ldots, w-1$ **do**
            if $j \neq \ell$ **then**
                $B_{i,j} = B_{i,j} - B_{i,\ell} \cdot B_{\ell,j}$
            end if
        end for
    end for
end for
$L^{-1} \leftarrow$ the lower triangular part of the first $n-k$ columns of B
$U \leftarrow$ the upper triangular part of the first $n-k$ columns of B
$P \leftarrow$ the permutation matrix represented by $r_0, \ldots, r_{n-k-1}$
return L^{-1}, U, P