Classic McEliece Implementation with Low Memory Footprint*

Johannes Roth[1], Evangelos Karatsiolis[1], and Juliane Krämer[2]

[1] MTG AG, jroth@mtg.de, ekaratsiolis@mtg.de
[2] Technische Universität Darmstadt, juliane@qpc.tu-darmstadt.de

Abstract. The Classic McEliece cryptosystem is one of the most trusted quantum-resistant cryptographic schemes. Deploying it in practical applications, however, is challenging due to the size of its public key. In this work, we bridge this gap. We present an implementation of Classic McEliece on an ARM Cortex-M4 processor, optimized to overcome memory constraints. To this end, we present an algorithm to retrieve the public key ad-hoc. This reduces memory and storage requirements and enables the generation of larger key pairs on the device. To further improve the implementation, we perform the public key operation by streaming the key to avoid storing it as a whole. This additionally reduces the risk of denial of service attacks. Finally, we use these results to implement and run TLS on the embedded device.

Keywords: Post-Quantum Cryptography, Classic McEliece, Low Memory Footprint, Embedded Implementation, TLS

1 Introduction

Code-based cryptographic schemes are often named when researchers in the field of post-quantum cryptography are asked which cryptosystem they recommend until trusted standards exist, e.g., [1]. Since the code-based cryptosystem McEliece [16] and its variant, the Niederreiter cryptosystem [18], date back to the 1970s and 1980s, respectively, their security is widely trusted. The direct successor of these schemes is the key encapsulation mechanism Classic McEliece [4], which was selected as finalist in the ongoing PQC standardization process of the US-American National Institute of Standards and Technology (NIST)[3].

The prime reason that code-based cryptography - contrary to RSA and Diffie-Hellman, which are comparably old -, has not gained general acceptance so far, is the size of the public key. For instance, a McEliece public key of about one megabyte and an RSA key of few kilobytes achieve the same pre-quantum security level. Many works address this problem by replacing binary Goppa codes with families of codes that lead to smaller public keys, e.g., [3,6,11,12,17].

*The conference version of this work is published at CARDIS 2020

[3] https://groups.google.com/a/list.nist.gov/forum/#!topic/pqc-forum/0ieuPB-b8eg

That, however, is no option when working with Classic McEliece: Since other families of codes already led to security breaches, e.g., [2, 8], or are not yet well-studied (which might lead to future security breaches), Classic McEliece focuses on binary Goppa codes since the scheme aims for conservative and well-studied cryptography.

When it comes to using cryptography in practical applications, a huge public key is especially problematic on memory-constrained devices, e.g., on embedded systems. On the other hand, there is a great demand for (post-quantum) cryptography for embedded devices, since on these devices, often sensitive data such as medical data and other personal information is processed. Hence, to obtain efficient and secure post-quantum cryptography for embedded devices by using Classic McEliece, we need to solve the problem that the public key is too big.

Contribution. In this work, we present memory-optimized algorithms for key generation and encapsulation of the Classic McEliece cryptosystem. We base our work on the Classic McEliece reference implementation. In accordance with NIST's request on recommended hardware[4], we use the ARM Cortex-M4 processor as our target platform. We emphasize that our goals are memory rather than speed optimizations. We demonstrate the practicability of our implementation by providing an operational instance of a Classic McEliece-based TLS on an embedded device.

Related Work. To the best of our knowledge, in the field of code-based cryptography there is not much related work addressing memory optimizations for handling the public key on embedded devices. In [7, 10], a seed is used to generate the inverse of the scramble matrix that is multiplied by the public parity-check matrix. This considerably reduces the memory requirements for storing the private key. However, it does not address issues with storing the public key. The public key instead is stored on flash or external memory. In [20] it is demonstrated in the context of public key infrastructures that the McEliece encryption operation can be performed on a smart card, even though the public key is too large to be held in memory. We adapt this approach in Section 4. In Section 4.1 we briefly compare an aspect of our results with the McTiny protocol [5].

Organization. This paper is organized as follows: In Section 2 we provide basic background information on the notation and on the Classic McEliece cryptosystem. Section 3 presents a memory-efficient algorithm to retrieve the public key from the private key, thereby reducing storage requirements. Further, an adapted key generation algorithm, that in combination with the public key retrieval algorithm greatly reduces memory requirements, is presented. Section 4 describes a memory-efficient variant of the encapsulation operation. In Section 5 we provide details on our implementation. Finally, Section 6 demonstrates the practical relevance of our results in a proof of concept TLS implementation.

[4] https://groups.google.com/a/list.nist.gov/forum/#!topic/pqc-forum/cJxMq0_90gU

2 Background

Notation To be consistent with the Classic McEliece notation, elements of $\mathbb{F}_2^n$ are viewed as column vectors. All identity matrices in this work are of dimension $(n-k)\times(n-k)$, which is why we omit subscripts and denote the $(n-k)\times(n-k)$ identity matrix by I. The (Hamming) weight of a vector $v \in \mathbb{F}_2^n$ is denoted by $\mathrm{wt}(v)$. To represent the ith row of a matrix A, we write A_i, while $A_{\bullet i}$ refers to the ith column.

Classic McEliece We do not detail the Classic McEliece algorithms here, but refer to the round-2 submission to NIST's PQC standardization process [4]. For readers not familiar with coding theory, we refer to [15].

We use symbols consistently with the Classic McEliece notation. Noteworthy symbols that are not explicitly introduced in this work are the Classic McEliece parameters $n, t, m, k = n - mt, q = 2^m$, the public key $T \in \mathbb{F}_2^{(n-k)\times k}$, the parity-check matrix $H \in \mathbb{F}_2^{(n-k)\times n}$, and its precursors $\hat{H} \in \mathbb{F}_2^{(n-k)\times n}$ and $\tilde{H} \in \mathbb{F}_q^{t\times n}$.

Note that we omit the *mceliece6960119* parameter set in this paper. These parameters produce bit sequences of length that are not multiples of 8. However, modern platforms operate on bytes. The trailing bits need an extra handling which our current implementation does not consider. This has no influence on the results.

3 Memory-Optimized Storing and Generation of the Key Pair

The public key of Classic McEliece is a parity-check matrix for the binary Goppa code that is chosen as private key. In McEliece and Niederreiter schemes, the public key is usually scrambled by multiplying suitable matrices. Instead of scrambling the rows by multiplying the parity-check matrix with a non-singular random matrix, Classic McEliece applies Gaussian elimination. This serves the same purpose and additionally transforms the parity-check matrix to systematic form, i.e., $H = (I \mid T)$. Choosing the systematic form has the benefit of reducing the public key size since the identity matrix I can be omitted.

For embedded devices, holding the parity-check matrix in memory and performing Gaussian elimination to obtain the resulting public key is challenging due to the size. To address this issue, we present an algorithm to generate a more compact form of the key pair (Section 3.1). We further present an algorithm to compute the public key ad-hoc in small chunks from the compact form (Section 3.2). The combination of both algorithms enables us to stream the public key to a peer without the need to hold it in memory in its entirety. This reduces both, memory and storage requirements for handling Classic McEliece key pairs.

3.1 Extended Private Key Generation

Classic McEliece defines the public key as $K_{pub} = T$ and the private key as $K_{priv} = (\Gamma, s)$. Here, $\Gamma = (g(x), (\alpha_1, ..., \alpha_n))$ where $g(x)$ is a polynomial in

$\mathbb{F}_q[x]$, $(\alpha_1, ..., \alpha_n)$ are finite field elements in $\mathbb{F}_q$, and s is a uniformly random generated n-bit string. Our proposal is to omit the public key T altogether, and instead store a smaller matrix $S \in \mathbb{F}_2^{(n-k)\times(n-k)}$ in the private key. We call the new private key *extended private key* and define it as $K_{priv\_ext} = (\Gamma, s, S)$. We define S^{-1} as the leftmost $(n-k) \times (n-k)$ submatrix of $\hat{H}$, and S as its inverse, for which the relationship $S\hat{H} = H = (I \mid T)$ holds. The x86-optimized SSE and AVX code variants that have been submitted as additional implementations to NIST already make use of this relationship. The authors use it to speed-up the Gaussian elimination in the key generation operation, whereas we aim to reduce memory requirements instead.

With Algorithm 1, we present an algorithm to generate the extended private key. In comparison to the original Classic McEliece key generation [4, Section 2.4], no public key is computed. Further, Algorithm 1 does not operate on the complete matrix H and its precursors $\hat{H}$ and $\tilde{H}$. Instead, only the first $n-k$ columns are computed to produce S^{-1}, which in line 5 is inverted to obtain S.

Algorithm 1: Extended Private Key Generation

Parameter: $n, t, m, q = 2^m, k = n - mt$
Output: $K_{priv\_ext} = ((g(x), (\alpha_1, ..., \alpha_n), s, S)$

1 Generate a uniform random irreducible polynomial $g(x) \in \mathbb{F}_q[x]$ of degree t.
2 Select a uniform random sequence $(\alpha_1, \alpha_2, ..., \alpha_n)$ of n distinct elements of $\mathbb{F}_q$.
3 Compute the $t \times (n-k)$ matrix $\tilde{S}^{-1}$ over $\mathbb{F}_q$ by letting $(\tilde{S}^{-1})_{i,j} = h_{i,j}$, where $h_{i,j} = \alpha_j^{i-1}/g(\alpha_j)$ for $i = 1, ..., t$ and $j = 1, ..., n-k$.
4 Form an $(n-k) \times (n-k)$ matrix S^{-1} over $\mathbb{F}_2$ by replacing each entry $c_0 + c_1 z + ... + c_{m-1} z^{m-1}$ of $\tilde{S}^{-1}$ with a column of m bits $c_0, c_1, ..., c_{m-1}$.
5 Compute S as the inverse of S^{-1}.
6 **if** *the previous step fails (i.e., S^{-1} is singular)* **then**
7 | go back to line 1.
8 **end**
9 Generate a uniform random n-bit string s.

Note that entries of $\hat{H}$ (and $\hat{S}$) can be computed "on-the-fly" as the binary representation of entries of $\tilde{H}$: we obtain $\tilde{H}_{i,j}$ by $\tilde{H}_{i,j} = \alpha_j^{i-1}/g(\alpha_j)$. To efficiently access all entries of $\hat{H}$ and $\tilde{H}$, respectively, it is beneficial to produce the elements top-down. For $\tilde{H}_{i,j}$ with $i > 1$, it holds that $\tilde{H}_{i,j} = \alpha_j \tilde{H}_{i-1,j}$, i.e., computing $\tilde{H}_{i,j}$ from $\tilde{H}_{i-1,j}$ only encompasses one modular multiplication and preserves the previous work of computing the modular exponentiation and inversion.

3.2 Public Key Retrieval

In the following, we describe an algorithm that obtains chunks of the public key ad-hoc from the extended private key. We propose to call the process of obtaining the public key from the extended private key *retrieving the public key*[5].

[5]This wording avoids calling it *generating the public key* since the public key is already uniquely defined after the private key is chosen. The term "generate" might be misleading and imply that randomness is introduced into this process.

The original Classic McEliece key generation algorithm already comprises a possibility to retrieve the public key. That is, for a valid private key, in lines 3-5 [4, Section 2.4], the corresponding public key is computed. This however entails holding the $(n-k) \times n$ matrix $\hat{H}$ in memory and subsequently performing Gaussian elimination. The matrix $\hat{H}$ is larger than the public key itself.

We therefore propose Algorithm 2 which operates on smaller matrices only. The algorithm has the additional property that it does not retrieve the complete public key in one large chunk, but instead can be used to retrieve single columns at a time. The public key column retrieval algorithm can therefore be used to stream the public key to a peer by ad-hoc retrieving single columns of it.

Algorithm 2: Public Key Column Retrieval

Input: $K_{priv\_ext} = (I, s, S)$, Column c (Integer)
Parameter: $n, t, m, q = 2^m, k = n - mt$
Output: cth column of the public key: $T_{\bullet c}$

1 Compute $\tilde{H}_{\bullet c}$ as the t-dimensional vector over $\mathbb{F}_q$ with $\tilde{H}_{j,c} = \alpha_c^{j-1}/g(\alpha_c)$ for $j = 1, 2, ..., t$.
2 Compute $\hat{H}_{\bullet c}$ as the mt-dimensional vector over $\mathbb{F}_2$ that results by replacing each entry $c_0 + c_1 z + ... + c_{m-1} z^{m-1}$ of $\tilde{H}_{\bullet c}$ with a column of m bits $c_0, c_1, ..., c_{m-1}$.
3 Compute $H_{\bullet c} = S\hat{H}_{\bullet c}$.

4 Streaming Encapsulation

In the previous Section 3 we specified the memory-efficient handling and the ad-hoc generation of the public key on a device that holds the (extended) private key. Now, we discuss the memory-efficient handling of the encapsulation operation [4, Section 2.7] that takes place on the device of a peer that, naturally, does not have access to the private key. The goal is to minimize the memory footprint of the operation. This is achieved by not processing the complete public key at once, but in smaller chunks. That is, we compute the encapsulation operation while the public key is streamed from a peer, without buffering it completely. A similar approach is already described for the McEliece PKE scheme in [20]. We adapt this approach for Classic McEliece.

In Classic McEliece, the encapsulation operation calls the encoding subroutine that is used to compute the syndrome of a weight-t error vector. The computation of the syndrome is the only operation in the encapsulation algorithm that makes use of the public key. We therefore need to address the syndrome computation. The syndrome C_0 of e is computed as $C_0 = He$, where H is the parity check matrix that is formed by $H = (I \mid T)$.

A naive implementation will form a buffer that contains the complete parity-check matrix H and then perform the matrix-vector-multiplication. However, it

is easy to see that for obtaining the matrix-vector-product all calculations are independent of each other and each entry of H is only used once. This means that it is not necessary to buffer more than a single byte of the public key at a time. Furthermore, it does not matter in which order the public key is processed as long as the order is defined.

With Algorithm 3 we introduce an encoding subroutine that operates on single columns of the public key, in accordance with the public key column retrieval algorithm (Algorithm 2). In lines 1 to 5, the implicit identity matrix that is part of the parity-check matrix is handled. In line 10 the syndrome computation is updated with the column that is received in line 9. Note that the ith column, i.e. $T_{\cdot i}$, can be discarded after each iteration of the while-loop.

Algorithm 3: Single-Column Encoding Subroutine

Input: weight-t Vector $e \in \mathbb{F}_2^n$
Parameter: $n, t, m, q = 2^m, k = n - mt$
Output: C_0

1 Initialize C_0 as an $n-k$-dimensional zero-vector.
2 Set $i := 1$.
3 **while** $i \leq n - k$ **do**
4 | Set $C_{0i} := e_i$.
5 | Set $i := i + 1$.
6 **end**
7 Set $i := 1$.
8 **while** $i \leq k$ **do**
9 | Read the ith column of the public key T from some location into $T_{\cdot i}$.
10 | Set $C_0 := C_0 + e_i \cdot T_{\cdot i}$.
11 | Set $i := i + 1$.
12 **end**

4.1 Mitigation of the Risk of Denial-of-Service Attacks

In a scenario where clients connect to a server and send Classic McEliece public keys, a concern is that the server is vulnerable to denial-of-service attacks. If the server accepts multiple connections and buffers public keys of the size of a megabyte or more, an attacker can abuse this to exhaust the server's memory. This prevents any new connection to be made without dropping connections that currently are in progress. It might even cause the server to behave erratically if it does not properly handle the case of failed memory allocations.

Such concerns have already been addressed in [5] where the *McTiny protocol* is described. In the McTiny protocol the server does not need to keep any per-client state at all. The per-client state is instead stored encrypted (by the server) on the client-side, thereby preventing such attacks.

We argue that in protocols like TLS, there is already a per-client state of significant size. E.g. the TLS standard implies the use of input/output-buffers that can store record plaintext sizes of 2^{14} bytes. By using the streaming encapsulation approach, the additional per-client state that results from performing the streaming encapsulation is the size of the syndrome C_0. For the largest parameter set, *mceliece8192128*, this amounts to 208 bytes. The input buffer that temporarily holds public key data can be fully consumed each time data is received as outlined previously.

To clarify, we do not suggest that the streaming encapsulation replaces the McTiny protocol, as there is still a per-client state. However, with protocols such as TLS, the risk of such an attack is greatly reduced by using the streaming encapsulation as described in this paper.

5 Implementation

We implemented the algorithms presented in Sections 3 and 4 on an STM32 Nucleo-144 development board with an STM32F429ZI MCU. It features an ARM Cortex-M4 with 256 KiB RAM and 2 MiB flash memory. The RAM is separated into two blocks: 192 KiB and 64 KiB of core coupled memory. The clock speed is set to 168 MHz. We use FreeRTOS v10.0.1 as operating system.

In the following, the network setup for the network-related measurements is detailed. We report our measurements for completeness, but we emphasize that we did not optimize the implementation for speed. This includes the Classic McEliece reference code, the SPHINCS$^+$ reference code, as well as the network stack. The lwIP library v2.0.3 is used to implement the TCP/IP stack. The development board is connected via ethernet to a PC that features an Intel i5-8400 CPU. The throughput of the TCP connection that we measured depends on the sending direction and the size of the packets that are sent. For receiving data on the board, we measured speeds between 3.04 MiB/s and 3.60 MiB/s. For sending data from the board, we measured speeds between 53.10 KiB/s and 2.07 MiB/s. The throughput for sending data from the board seems to drop to low speeds when exceeding the TCP MSS (Maximum Segment Size). This is the case when streaming the Classic McEliece public key in our current implementation. We further measured the average round-trip time of the connection as 0.39 ms. No package-loss has been observed. Measurements are generally rounded to two decimal places.

5.1 Memory-Efficient Matrix Inversion

The algorithms presented in Section 3 reduce the sizes of the involved matrices from $(n-k) \times n$ for H and $(n-k) \times k$ for T to $(n-k) \times (n-k)$ for S. Therefore, the total memory requirements now largely depend on the size of S

as the dominating factor.[6] Furthermore, the temporary memory that is required to obtain S from S^{-1} determines how memory-efficiently Algorithm 1 can be implemented. For example, to obtain the inverse of S^{-1}, it suggests itself to use the Gaussian elimination algorithm. However, this induces the memory overhead of transforming an $(n-k) \times (n-k)$-identity matrix. We propose a variant that can be implemented more memory-efficiently, i.e. almost in-place with $2(n-k)$ bytes of non-constant overhead. We outline the steps in Algorithm 4.

Algorithm 4: LU-Decomposition-based Matrix Inversion

Input: $S^{-1} \in \mathbb{F}_2^{(n-k)\times(n-k)}$
Output: S
1 Find the LU decomposition of S^{-1}, i.e. $PS^{-1} = LU$ where $P, L, U \in \mathbb{F}_2{}^{(n-k)\times(n-k)}$ and P is a permutation matrix and L and U are lower and upper triangular matrices.
2 Invert L and U.
3 Compute the product $U^{-1}L^{-1}$.
4 Undo the permutation to obtain $S = U^{-1}L^{-1}P$.

The correctness of this approach follows by verifying $PS^{-1} = LU \iff S^{-1} = P^{-1}LU \iff S = U^{-1}L^{-1}P$. We now outline how these steps can be implemented almost-in-place.

Step 1 LU Decomposition. We implement the LU decomposition as in Algorithm 5. This is essentially the "kij-variant" of the outer-product formulation of Gaussian elimination [9, Section 3.2.9]. In the binary case, however, it becomes a bit simpler since there is no need to divide by the diagonal elements. The algorithm can be implemented to replace the buffer of S^{-1} by L as its lower triangular matrix and U as its upper triangular matrix in a straightforward manner. That is, the memory access pattern shown in line 10 of Algorithm 5 already leads to an in-place implementation. However, the permutation matrix P cannot be stored in-place. Thus, the algorithm does not entirely run in-place. Since P is a sparse matrix with exactly $n-k$ non-zero entries, we implement it as a $2(n-k)$-byte vector that keeps track of which rows are swapped. Two bytes per entry are enough to handle the indices of matrices with up to $2^{16} = 65{,}536$ rows, enough to handle all Classic McEliece parameter sets. Additional care has to be taken to implement the pivoting in constant-time. For instance, this is achieved by not explicitly branching at the two distinct cases in line 3 of Algorithm 5, but masking the operations with all-zero or all-one masks, respectively. Further, the

[6] While there are other significant, temporary memory-overheads in the Classic McEliece code, none of them is as big as that for S. Furthermore, for the extended private key generation, temporary buffers can often be placed in the buffer where S is written into at the end of the key generation process. This results in a decreased overall memory consumption.

Algorithm 5: LU Decomposition

Input: $A \in \mathbb{F}_2^{(n-k)\times(n-k)}$
Output: $P, L, U \in \mathbb{F}_2^{(n-k)\times(n-k)}$, s.t. $PA = LU$. Return $\perp$ (error) if A is singular.

```
1  for k := 1 to n − 1 do
2      for i := k + 1 to n do
3          Swap row i with row k if the ith row has a non-zero entry at the kth
             column and update P accordingly (partial-pivoting).
4      end
5      if Pivoting fails (i.e. A is singular) then
6          return ⊥
7      end
8      for i := k + 1 to n do
9          for j := k + 1 to n do
10             A(i, j) := A(i, j) − A(i, k) · A(k, j)
11         end
12     end
13 end
```

return command in line 6 can safely abort the key generation. Doing so possibly leaks at which step the key generation failed. However, as the Classic McEliece authors already argue, doing so does not leak any information on the actually generated key since separate random numbers are used for each attempt [4].

Step 2 Inversion of U and L. The inversion of U and L amounts to backwards and forwards substitution respectively. Implementing this in-place is straightforward as well as to achieve the constant-time property because no code path depends on secret values.

Step 3 Multiplication of $U^{-1}L^{-1}$. U^{-1} and L^{-1} are both stored in the memory where S^{-1} used to be – as upper and lower triangular matrix, respectively. To obtain an algorithm that multiplies both matrices in-place, we utilize the triangular structure of U^{-1} and L^{-1}. First, let us give a formula with which an element of the product can be computed. For convenience, we define $\bar{A}$ as the matrix that contains L^{-1} and U^{-1} as a lower and an upper triangular matrix and A as $A := U^{-1}L^{-1}$. Each entry $A(i,j)$ can then be written as $A(i,j) = \sum_{k=\max(i,j)+1}^{n} \bar{A}(k,j) \cdot \bar{A}(i,k)$. By appropriately ordering the computations of entries of A, we prevent overriding values that are needed for future computations. More precisely, our solution is to first compute the element in the top-left corner, i.e. the first diagonal element $A(1,1)$. Then, the three remaining elements in the top-left 2×2-matrix can be computed in any order. Continuing like this, i.e. computing the remaining five elements in the top-left 3×3-matrix in any order, and so on, all elements of A can be computed. Each evaluation of the given formula only depends on values of $\bar{A}$ that have not been overwritten by elements of A yet. Therefore, the outlined approach can be implemented

in-place. Implementing this as constant-time is again straightforward since no computation path depends on secret values.

Step 4 Undo Permutation. Finally, the permutation needs to be undone. Since P is now multiplied from the right, this amounts to swapping the columns that are indicated by P.

5.2 Extended Private Key Generation

In Table 1 we list the performance of the extended private key generation (Algorithm 1) for the *mceliece348864* parameter set for different numbers of attempts that are made. Each time the key generation fails because S^{-1} is singular, the number of attempts is increased by one. One can see that the key generation is linear in the number of attempts. Each failed attempt roughly adds 1.77 s to the key generation time. On average, the key generation succeeds after 3.4 attempts [4]. We extrapolate the runtime for 3.4 attempts from our measurements as 1,938,512,183 cycles or 11.54 seconds.

# Attempts	Algorithm 1 Cycles	Algorithm 1 s
1	1,226,192,185	7.30
2	1,522,914,956	9.06
3	1,819,628,971	10.83
4	2,116,353,011	12.60
⋮	⋮	⋮

Table 1. Timings for the private key generation for the *mceliece348864* parameter set on our development board. The key generation time depends on the number of attempts that have to be made until a non-singular matrix is found.

In Table 2 we specify the size of the extended private key for different parameter sets and compare it to the size of the Classic McEliece key pair. We note that the private key size is implementation-dependent. In contrast to the reference implementation, we store the field elements $(\alpha_1, \alpha_2, ..., \alpha_n)$ as two-byte values. The reference implementation stores these elements by generating the control bits of a Beneš network. The benefits of the Beneš network are not utilized in the reference implementation, but only play a role in x86-optimized implementations that have been submitted as additional implementations. Since the implementation that generates the control bits has a large memory footprint and our implementation does not benefit from it, we omit this step.

Table 3 depicts the reduction of the memory footprint when performing the extended private key generation instead of the Classic McEliece key generation. For the Gaussian elimination we assume the required memory as the size of the matrix $\hat{H}$. For the inversion of S we assume that this is done in-place with the

Parameter Set	Key Pair Bytes	Extended Private Key Bytes	Difference Bytes	Ratio
mceliece348864	267,572	81,268	186,304	0.30
mceliece460896	537,728	204,672	333,056	0.38
mceliece6688128	1,058,884	360,580	698,304	0.34
mceliece8192128	1,371,904	363,776	1,008,128	0.27

Table 2. Size of the extended private key compared to the Classic McEliece key pair. Note that for the extended private key, it is intended to omit storing the public key.

Parameter Set	m	n	t	Gaussian Elimination of $\hat{H}$	Matrix Inversion of S^{-1}	Difference	Ratio
mceliece348864	12	3488	64	334,848	75,264	259,584	0.22
mceliece460896	13	4608	96	718,848	197,184	521,664	0.27
mceliece6688128	13	6688	128	1,391,104	349,440	1,041,664	0.25
mceliece8192128	13	8192	128	1,703,936	349,440	1,354,496	0.21

Table 3. Comparison of the memory footprint (in bytes) for producing the public key T or S^{-1}, respectively, from Γ. The original Classic McEliece algorithm applies Gaussian elimination to the $(n-k) \times n$ matrix $\hat{H}$. We propose to (almost) in-place invert the $(n-k) \times (n-k)$ matrix S^{-1} instead (Algorithm 1 and Section 5.1).

addition of storing $2(n-k)$ bytes for the pivoting (see Section 5.1). The reduction in the memory footprint is considerable, since the explicit representation of the public key is not stored. This amounts to over a megabyte of memory that is saved for the largest parameter set.

5.3 Public Key Column Retrieval

In Table 4 we list the timings for the public key column retrieval algorithm (Algorithm 2) for consecutively retrieving the complete public key. The numbers are the raw computation time, i.e. the cycle count does not include sending the resulting columns over a network to a peer. As an implementation detail, we do not operate on chunks of single columns but on eight columns at a time. This is easier to implement and also faster since bytes (i.e. eight bits) usually are the smallest addressable memory unit. Working on eight columns at a time therefore saves unnecessary operations that are needed to access single bits within a byte. This is true at least, when the matrices are stored in a row-major order. The size of each generated chunk is therefore $n-k$ bytes instead of $(n-k)/8$ bytes. Our implementation has a memory overhead of $17/8(n-k)$ bytes, including the output buffer and intermediary results (but excluding control variables). Table 4 also illustrates the memory footprint for each parameter set.

5.4 Streaming Encapsulation

We implemented a variant of the encapsulation algorithm by adapting it to use the single-column encoding subroutine (Algorithm 3). For the same reason as

Parameter Set	Algorithm 2		k mat-vec-muls		Memory Overhead
	Cycles	s	Cycles	s	bytes
mceliece348864	667,392,425	3.97	623,672,694	3.71	1632
mceliece460896	2,250,917,383	13.40	1,965,249,172	11.70	2652
mceliece6688128	5,820,127,974	34.64	5,152,221,701	30.67	3536
mceliece8192128	7,558,882,087	44.99	6,694,439,348	39.85	3536

Table 4. Timings for retrieving the public key with Algorithm 2. We include the biggest contributor to its runtime, the k matrix-vector-multiplications with the $(n-k)\times(n-k)$ matrix S. We also list the total memory footprint of our implementation.

before, we work on chunks of eight columns instead of single columns, again leading to chunks of $n-k$ bytes instead of $(n-k)/8$ bytes. Table 5 lists the runtime and the memory footprint of our implementation for the encoding subroutine. The memory footprint includes the error vector and the chunk of eight columns as the input buffers and the syndrome as the output buffer and amounts to $n/8+9/8(n-k)$ bytes. We note that processing the public key in row-major order can be implemented faster than processing it in column-major order. In comparison to the reference implementation that processes the public key in row-major order, our implementation is around twenty to fifty percent slower (depending on the parameter set).

Parameter Set Pubkey origin	Algorithm 3 local buffer	Algorithm 3 local buffer	Algorithm 3 network	Memory Overhead –
	Cycles	ms	ms	bytes
mceliece348864	3,106,183	18.49	92.37	1300
mceliece460896	5,868,529	34.93	183.14	1980
mceliece6688128	11,464,900	68.24	358.67	2708
mceliece8192128	14,696,239	87.48	463.64	2896

Table 5. Measured speed of the streaming encapsulation operation. Columns 2 and 3 give the timings when operating on a local buffer in the RAM of the device. This is given as a reference point for the speed of the operation itself. Column 4 depicts the timing when streaming the public key to the board over a TCP/IP connection.

6 TLS 1.2 Implementation

We implemented the algorithms in Sections 3 and 4 into the mbedTLS library[7] by defining a new cipher suite for TLS 1.2. The current version of mbedTLS supports only TLS 1.2 and a prototype implementation of TLS 1.3 is work in progress. Our algorithms function both for TLS 1.2 and TLS 1.3 because the changes between these two versions do not affect them. In addition TLS 1.3 also does not have accepted standards for PQC yet. Later in this section we

[7] https://tls.mbed.org/

discuss the relevant changes for a TLS 1.3 implementation. We do not describe our implementation in full detail but outline our prototype that serves as a proof of concept. The cipher suite uses Classic McEliece as a key exchange algorithm and a server certificate with a SPHINCS$^+$ key [14].

In our cipher suite, the server generates an ephemeral extended private key for Classic McEliece, using Algorithm 1. In the *server key exchange* message the public key is streamed to the client by utilizing Algorithm 2. A SPHINCS$^+$ signature is appended to the key. The client performs the streaming encapsulation operation (Algorithm 3) and verifies the signature. The premaster secret is then generated analogously to RSA cipher suites by turning the Classic McEliece scheme into a PKE scheme through a KEM-DEM conversion [19, Section 5]. That is, the key from the encapsulation is used as an AES-256 key which is then used to encrypt a 48-byte premaster secret, which is defined and used analogously to the RSA-encrypted premaster secret. The client sends the encrypted premaster secret in the *client key exchange* message. If the server can successfully decrypt the premaster secret, both parties form the same master secret.

The parameter sets that we use in our cipher suite are *mceliece348864* and *SPHINCS$^+$-256f*. For Classic McEliece, the size of the public key is 261,120 B and the size of the required memory for the matrix inversion during the key generation is 73,728 B. A *SPHINCS$^+$-256f* signature amounts to 49,216 B. The client verifies two of these signatures: One in the server certificate, since we chose to generate a root CA with a *SPHINCS$^+$-256f* key in order to have a full post-quantum handshake, and one in the *server key exchange* message. While SPHINCS$^+$ signatures can in principle be processed in a streaming fashion [13], our implementation stores the signature in a buffer on the device's memory.

Board as Server					
	KeyGen	Decapsulation	Sign	2x Verify	send CERT+PK+SIG
Server	10.83 s	0.99 s	109.71 s	–	4.29 s
Client	–	–	–	0.01 s	–
Total Handshake Time 126.30 s					

Board as Client					
	KeyGen	Decapsulation	Sign	2x Verify	send CERT+PK+SIG
Server	0.14 s	0.00 s	0.11 s	–	0.34 s
Client	–	–	–	5.18 s	–
Total Handshake Time 5.83 s					

Table 6. Total handshake time (client hello to second finished message) including noteworthy sub-operations. Sign and Verify refer to the SPHINCS$^+$ sign and verify operation. The client verifies two signatures, the server certificate signature and the signed public key. The other listed operations are performed by the server. The last column depicts the measured time to send the server certificate (containing a SPHINCS$^+$ signature), as well as the signed Classic McEliece public key (the public key retrieval algorithm is used). Results are averaged and we chose seeds for the key generation that result in three key generation attempts.

We report the timings for the handshake in Table 6. The SPHINCS$^+$ operations on the board take about 86.86 % and 88.85 % of the total handshake time. Since we focus on a memory-efficient implementation of Classic McEliece, we do not optimize the SPHINCS$^+$ operations. We chose to set the number of key generation attempts fixed to three by choosing appropriate seeds. This approximates the mean of 3.4 key generation attempts but avoids the need to measure a vast amount of handshakes only to average the variance in key generation. The number of round-trips is the same as in common TLS 1.2 connections, i.e., two full round-trips before application data is sent.

With regard to using our approach in TLS 1.3, the following differences have to be considered: First, in TLS 1.3, to maintain the 1-RTT benefit over TLS 1.2, the client would have to generate the key pair. This might be unwanted for embedded-client scenarios. Second, in TLS 1.3 sending a Classic McEliece public key is not straightforward. The natural place to convey the public key and the ciphertext is the *key_exchange*-field in the *KeyShareEntry* struct which is part of the *key_share*-extension. However, the field only holds keys of size up to $2^{16}-1$ B. Classic McEliece keys exceed this limit. Therefore, an implementer would have to consider a strategy to circumvent this limit. Other than that, we see only minor changes for the sake of employing our proof of concept implementation in TLS 1.3.

A complete handshake has been performed with the outlined cipher suite on our development board. Both, the server side and the client side can be executed on the development board that features only 256 KiB RAM. Our proof of concept implementation demonstrates that our results can be applied in the real world and leave enough room to handle large signatures, as well as the memory overhead in the TLS and TCP stack.

Acknowledgements

This work was partially funded by the German Federal Ministry of Education and Research (BMBF) under the project “QuantumRISC” (ID 16KIS1037 and ID 16KIS1039). Moreover, JK was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – SFB 1119 – 236615297. The authors thank Stathis Deligeorgopoulos for his preliminary work on TLS.

The final authenticated publication is available online at https://doi.org/10.1007/978-3-030-68487-7_3.

References

1. Augot, D., Batina, L., Bernstein, D.J., Bos, J.W., Buchmann, J., Castryck, W., Dunkelman, O., Güneysu, T., Gueron, S., Hülsing, A., Lange, T., Rechberger, C., Schwabe, P., Sendrier, N., Vercauteren, F., Yang, B.: Initial recommendations of long-term secure post-quantum systems (2015)
2. Baldi, M., Bodrato, M., Chiaraluce, F.: A New Analysis of the McEliece Cryptosystem Based on QC-LDPC Codes. In: Ostrovsky, R., De Prisco, R., Visconti,

I. (eds.) Security and Cryptography for Networks. pp. 246–262. Springer Berlin Heidelberg (2008)
3. Baldi, M., Santini, P., Chiaraluce, F.: Soft McEliece: MDPC code-based McEliece cryptosystems with very compact keys through real-valued intentional errors. In: Proc. IEEE International Symposium on Information Theory (ISIT 2016). pp. 795–799 (Jul 2016). https://doi.org/10.1109/ISIT.2016.7541408
4. Bernstein, D., Chou, T., Lange, T., von Maurich, I., Misoczki, R., Niederhagen, R., Persichetti, E., Peters, C., Schwabe, P., Sendrier, N., Szefer, J., Wang, W.: Classic McEliece Supporting Documentation (2019)
5. Bernstein, D.J., Lange, T.: McTiny: fast high-confidence post-quantum key erasure for tiny network servers. Cryptology ePrint Archive, Report 2019/1395 (2019), https://eprint.iacr.org/2019/1395
6. Cayrel, P.L., Hoffmann, G., Persichetti, E.: Efficient Implementation of a CCA2-Secure Variant of McEliece Using Generalized Srivastava Codes. In: Fischlin, M., Buchmann, J., Manulis, M. (eds.) Public Key Cryptography – PKC 2012. pp. 138–155. Springer Berlin Heidelberg (2012)
7. Eisenbarth, T., Güneysu, T., Heyse, S., Paar, C.: MicroEliece: McEliece for Embedded Devices. In: Clavier, C., Gaj, K. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2009. pp. 49–64. Springer Berlin Heidelberg (2009)
8. Faugère, J.C., Otmani, A., Perret, L., de Portzamparc, F., Tillich, J.P.: Structural cryptanalysis of McEliece schemes with compact keys. Designs, Codes and Cryptography **79**(1), 87–112 (Apr 2016). https://doi.org/10.1007/s10623-015-0036-z, https://doi.org/10.1007/s10623-015-0036-z
9. Golub, G.H., van Loan, C.F.: Matrix Computations. JHU Press, fourth edn. (2013), http://www.cs.cornell.edu/cv/GVL4/golubandvanloan.htm
10. Heyse, S.: Low-Reiter: Niederreiter Encryption Scheme for Embedded Microcontrollers. In: Sendrier, N. (ed.) Post-Quantum Cryptography. pp. 165–181. Springer Berlin Heidelberg (2010)
11. Heyse, S.: Implementation of McEliece Based on Quasi-dyadic Goppa Codes for Embedded Devices. In: Yang, B.Y. (ed.) Post-Quantum Cryptography. pp. 143–162. Springer Berlin Heidelberg (2011)
12. Heyse, S., von Maurich, I., Güneysu, T.: Smaller Keys for Code-Based Cryptography: QC-MDPC McEliece Implementations on Embedded Devices. In: Bertoni, G., Coron, J.S. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2013. pp. 273–292. Springer Berlin Heidelberg (2013)
13. Hülsing, A., Rijneveld, J., Schwabe, P.: ARMed SPHINCS, pp. 446–470 (2016). https://doi.org/10.1007/978-3-662-49384-7_17
14. Hülsing, A., et al.: Sphincs+, https://sphincs.org/
15. van Lint, J.H.: Introduction to Coding Theory. Springer-Verlag, Berlin, Heidelberg, 3rd edn. (1998)
16. McEliece, R. J.: A Public-Key Cryptosystem Based on Algebraic Coding Theory. Deep Space Network Progress Report **42**(44), 114–116 (1978)
17. Misoczki, R., Barreto, P.S.L.M.: Compact McEliece Keys from Goppa Codes. In: Jacobson, M.J., Rijmen, V., Safavi-Naini, R. (eds.) Selected Areas in Cryptography. pp. 376–392. Springer Berlin Heidelberg (2009)
18. Niederreiter, H.: Knapsack-type cryptosystems and algebraic coding theory. In: Problems of Control and Information Theory 15. pp. 159–166 (1986)
19. Shoup, V.: A proposal for an iso standard for public key encryption (version 2.1) (Jan 2002), https://www.shoup.net/papers/iso-2_1.pdf

20. Strenzke, F.: Solutions for the Storage Problem of McEliece Public and Private Keys on Memory-Constrained Platforms. In: Gollmann, D., Freiling, F.C. (eds.) Information Security. pp. 120–135. Springer Berlin Heidelberg (2012)