Solutions for the Storage Problem of McEliece Public and Private Keys on Memory-constrained Platforms

Falko Strenzke[1]

[1] FlexSecure GmbH, Germany**,
`strenzke@flexsecure.de`
[2] Cryptography and Computeralgebra, Department of Computer Science, Technische Universität Darmstadt, Germany

Abstract. While it is generally believed that due to their large public and private key sizes code based public key schemes like the McEliece PKC cannot be conveniently implemented on memory-constrained devices, we demonstrate otherwise. We show that for the public key we face rather a transmission problem than a storage problem: we propose an approach for Public Key Infrastructure (PKI) scenarios which totally eliminates the need to store public keys of communication partners. Instead, all the necessary computation steps are performed during the transmission of the key. We show the feasibility of the approach through an example implementation and give arguments that it will be possible for a smart card controller to carry out the associated computations fast enough to sustain the transmission rates of possible future high speed contactless interfaces. Concerning the McEliece private key, we demonstrate, contrasting to previously published implementations, that the parity check matrix, which is by far the largest part of this key, is not necessary to achieve fast decryption on embedded systems.

Key words: post-quantum cryptography, code-based cryptography, public key encryption scheme, efficient implementation, embedded devices

1 Introduction

Code-based cryptography, i.e. the class of cryptographic schemes built on error correcting codes, encompasses public key encryption schemes [1,2] as well as signature schemes [3,4] and an identification scheme [5]. The main advantage of code-based cryptographic schemes over currently used schemes that are based on the factoring or discrete logarithm problem is their believed security in the presence of quantum computers [6], but at least the encryption schemes' operations can also be implemented comparatively fast [7].

However, the large public key size in these schemes are considered a tremendous disadvantage. For this reason, a number of attempts have been made to

** A part of the work of F. Strenzke was done at[2] reduce the public key size by using special codes [8,9,10]. But some of these attempts have already been shown to result in insecure cryptosystems [11,12,13]. All recent proposals that reduce the key size using other codes than in the original McEliece scheme will have to prevail for some time until they can be granted the same trust as the original scheme, employing classical binary Goppa codes [14], the security of which is still unquestioned after more than 30 years – however, with the exception of certain choices of code parameters [15], which are only of relevance for code-based signature schemes [3].

Accordingly, in the first part of the work, we address the problem of performing the public operations, i.e. encryption or signature verification, of conventional code-based public key cryptosystems with large public keys on devices with limited memory resources, like for instance smart cards. Typically, smart cards have less than 20 KB of RAM, while the available amount of non-volatile memory (NVM), e.g. flash-memory, can be as large as 512 KB [16,17]. If a public key of a communication partner shall be temporarily stored on the device for the purpose of performing e.g. an encryption, it would have to be stored in the NVM since it exceeds the size of the RAM many times over. Specifically, the public keys will be at least 100 KB large for reasonable security parameters, as we will see in Section 2.2. For instance, the works [18,19,20] all describe implementations of code-based encryption schemes on embedded devices, where the public key is stored in the devices NVM. The drawbacks of storing such an amount of data in the device's NVM are first of all the cost of keeping such a large amount of memory available for this purpose and also the much slower writing speed compared to RAM access. In order to circumvent these problems, we show in this work that the public operations can be executed by only storing very small parts of the public keys at any given time during the operation. Our approach also considers that these operations are always carried out in a PKI context, which implies the verification of user public key certificates against issuer certificates. We point out that this approach is equally usable for the Niederreiter PKC as well as certain code-based signature schemes.

The second part of this work shows a solution for the McEliece private key, which in previously published implementations [21,18,19,22] of the McEliece PKC features the parity check matrix, the size of which is similar to that of the McEliece public key. The benefit of the parity check matrix, which is not an essential part of the private key, but a precomputation that allows faster computation of the so called syndrome vector. We show that an optimized implementation of the syndrome computation allows practical decryption times without this matrix, reducing the private key size dramatically.

2 Preliminaries

2.1 Public Key Cryptography

In a public key infrastructure, the trustworthiness of a public key is always verified against a trust anchor. From the trust anchor, which is usually a certification authority (CA) certificate, to the user certificate, there is a certificate chain involved. The trustworthiness of a certificate lower in the chain is guaranteed by its authentic digital signature created by the respective issuer, verifiable via the corresponding public key contained in the issuer certificate.

For the case of public key encryption, it means that a user A's public key intended for encryption is contained in the user certificate. A user B willing to encrypt a message for A thus goes through the following steps:

1. retrieve A's public encryption certificate Enc-Cert_A (for example by accessing a database or asking A directly)
2. verify the authenticity of Enc-Cert_A by checking the signature on the certificate against the trust anchor (CA certificate)
3. encrypt the secret message using Enc-Cert_A and send it to A

Since in this work we will address problems and solutions for embedded devices such as smart cards, we wish to point out why it is necessary to be able to carry out not only the private operations of a public key scheme (i.e. decryption or signature generation) but also the public operations on such devices. One application are key exchange schemes. Key exchange schemes based on public key cryptography are used for instance in the context of the German ePassport [23]. There, an elliptic curve based key agreement scheme is realized [24]. In order to replace this scheme with a quantum computer secure solution, one would have to combine a public key encryption scheme with a public key signature scheme that both have this property. Then, one party sends the signed and encrypted symmetric key to the other party. In the mentioned context this means that eventually the ePassport's chip has to carry out the encryption operation.

2.2 Code-based Encryption Schemes

In the following, we explain two code-based encryption schemes, where we focus on the encryption operation and those parts of the decryption operation that are relevant for the understanding of this work.

Both code-based encryption schemes employ irreducible binary Goppa Codes [14] as error correcting codes.

Definition 1. *Let the polynomial $g(Y) = \sum_{i=0}^{t} g_i Y^i \in \mathbb{F}_{2^m}[Y]$ be monic and irreducible over $\mathbb{F}_{2^m}[Y]$, and let m, t be positive integers. Then $g(Y)$ is called a* Goppa polynomial *(for an irreducible binary Goppa code).*

Then an irreducible binary Goppa code is defined as $\mathcal{C}(\Gamma, g(Y)) = \{\boldsymbol{c} \in \mathbb{F}_2^n | S_{\boldsymbol{c}}(Y) := \sum_{i=0}^{n-1} \frac{c_i}{Y-\alpha_i} = 0 \mod g(Y)\}$, where $n \leqslant 2^m$, $S_{\boldsymbol{c}}(Y)$ is the syndrome of $\boldsymbol{c}$, $\Gamma = \{\alpha_i | i = 0, \ldots, n-1\}$, the support *of the code, where the α_i are pairwise distinct elements of $\mathbb{F}_{2^m}$, and c_i are the entries of the vector $\boldsymbol{c}$.*

The code defined in such way has length n (i.e. the length of the code words), dimension $k \geqslant n - mt$ (i.e. the length of the message words) and can correct up to t bit flip errors.

As for any linear error correcting code, for a Goppa code there exists a generator matrix $G \in \mathbb{F}_2^{n \times k}$ and a parity check matrix $H \in \mathbb{F}_2^{mt \times n}$ [25]. Given these matrices, a message $\boldsymbol{v} \in \mathbb{F}_2^k$ can be encoded into a codeword $\boldsymbol{c}$ of the code by computing $\boldsymbol{c} = \boldsymbol{v}G$, and the syndrome $\boldsymbol{s} \in \mathbb{F}_2^{mt}$ of a (potentially distorted) codeword can be computed as $\boldsymbol{s} = \boldsymbol{c}H^T$. Here, we do not give the formulas for the computation of these matrices as they are of no importance for the understanding of the topics of this work. The interested reader, however, is referred to [25].

The first encryption scheme we present is the McEliece [1] scheme.

Overview of the McEliece PKC. In this section we give a brief overview of the McEliece PKC.

Algorithm 1 The McEliece encryption Operation

```
Require: the McEliece public key $G_p \in \mathbb{F}_2^{k \times n}$ and the message $m \in \mathbb{F}_2^k$,
Ensure: the ciphertext $z \in \mathbb{F}_2^n$
 1: create a random binary vector $e \in \mathbb{F}_2^n$ with Hamming weight $\mathrm{wt}(e) = t$
 2: $z \leftarrow mG_p \oplus e$
```

The McEliece *secret key* consists of the Goppa polynomial $g(Y)$ of degree t and the support Γ, together they define the secret code $\mathcal{C}$.

The *public key* is given by the public $n \times k$ generator matrix $G_p = TG$ over $\mathbb{F}_2$, where G is a generator matrix of the secret code $\mathcal{C}$ and T is a non-singular $k \times k$ matrix over $\mathbb{F}_2$, the purpose of which is to bring G_p into reduced row echelon form, i.e. $G_p = [\mathbb{I}|G_2]$, which results in a more compact public key [7]. The *encryption* operation (Algorithm 1) allows messages $\boldsymbol{v} \in \mathbb{F}_2^k$. A random vector $\boldsymbol{e} \in \mathbb{F}_2^n$ with hamming weight $\mathrm{wt}(\boldsymbol{e}) = t$ has to be created. Then the ciphertext is computed as $\boldsymbol{z} = \boldsymbol{v}G_p + \boldsymbol{e}$. Note that due to the reduced row echelon form of G_p the first k bits of $\boldsymbol{v}$ are reproduced in $\boldsymbol{v}G_p$. However, since the McEliece PKC, as well as the Niederreiter PKC, which is introduced shortly, needs to be wrapped in a so-called CCA2 conversion [26], this is not a problem [7].

McEliece *decryption* is performed by applying error correction to the ciphertext $\boldsymbol{z}$, which can only be done with the knowledge of the secret key, i.e. the code $\mathcal{C}$. With the exception of the first step, which is the computation of the syndrome polynomial $S(Y)$, which will be the topic of Section 4 and will be explained there, the further details of the decryption procedure are irrelevant for the understanding of the topics of this work.

The McEliece parameters are given by the code parameters n, k and t. An example parameter set giving about 100 bits of security with respect to the attacks given in [27] would be be $n = 2048$, $k = 1498$ and $t = 50$, yielding a public key size of about 100 KB.

The other encryption scheme is the Niederreiter [2] scheme. Here, the public key consists of the public parity check matrix $H_p = TH_s$, where H_s is the parity check matrix of the private code and $H_p \in \mathbb{F}_2^{(n-k) \times n}$, and T is chosen equivalently to its counterpart in the McEliece scheme. Furthermore, as in the McEliece scheme, H_p can be put in systematic form. Then the public key will be of the same size as for the McEliece cryptosystem. The Niederreiter encryption is depicted in Algorithm 2. The message is encoded into an error vector of weight t and the ciphertext is the corresponding syndrome, which can only be decoded by the holder of the private key.

```
Algorithm 2 The Niederreiter encryption Operation
Require: the Niederreiter public key $H \in \mathbb{F}_2^{(n-k)\times n}$ and the message $m$
Ensure: the ciphertext $z \in \mathbb{F}_2^{n-k}$
1: encode the message $m$ into $e \in \mathbb{F}_2^n$, where $\mathrm{wt}(e) = t$, using an appropriate algorithm
   (“constant-weight-word encoding”)
2: $z \leftarrow eH$
```

3 Online Public Operation

In this section, we explain the main idea of the paper, namely how to implement the public operations of code-based schemes without storing full public keys on the device. In a naive approach, the public operation, which we here assume to be an encryption operation, would be realized by first retrieving the public key (embedded into a public key certificate containing also a signature) of the communication partner, storing it on the device, computing the hash value of the certificates to-be-signed (TBS) data (which includes the code-based public key), verifying the signature, and finally encrypting the designated message using the certificate's public key. With the proposed approach however, no storage of the whole public key is required. Instead, only a comparatively small amount of RAM memory will be used. The basic idea is to use the computation time that is available to the devices CPU in the time interval between the receival of two bytes via the serial interface. During this interval both the encryption algorithm and the hash algorithm are advanced by one small step. Hence we call this approach “on-line public operation”.

This approach works because both the computation of the hash value of the public key and the matrix-vector product only depend on a small part of the whole public key at any given point in time: while the hash function acts on blocks of multiple bytes (for instance 64 bytes for SHA-256), the matrix multiplication could in principle be carried out bit-wise.

3.1 Description of the Online Public Operation

In Figure 1, the complete process of the on-line public operation approach is depicted. On the left hand side, the processing of the certificate containing the code-based public key to be used in the public operation is shown. Here, we assume that the public key is contained in an X.509 public key certificate [28]. Such a certificate is constituted by the sequence of the TBS data, followed by a field containing information about the signature algorithm (not shown in the figure) and finally the signature. The signature ensures the authenticity of the TBS data, and is calculated based on their hash value, using a hash algorithm as specified in the preceding information field. Please note that the signature algorithm used to sign the user certificate needs not to be code-based (in which case the trust anchor certificate would contain a large code-based key itself). Instead, a hash based signature scheme [29] could be used. These schemes are also quantum computer resistant and feature extremely small public keys.

In Step 1a the part of the TBS data that precedes the public key is received by the device and processed in the normal manner, which includes the computation of the hash value of the received data. Once the transmission of the public key, i.e. the public matrix M, begins (Step 2a), the computation of the product vM begins, where v is a binary vector whose meaning depends on the type of the code-based scheme. In an encryption scheme like McEliece or Niederreiter, v represents a message. The hash computation is also continued. After the whole public matrix has been received, the remaining TBS data is again processed in the normal manner (Step 3a). Finally, when the TBS data have been completely received the hash value of the TBS data is ready. It is then used to verify the certificate's signature with the help of the certificate of the issuer I which is stored on the device as the trust anchor (Step 4).

The public operation of the code-based scheme is potentially composed of computations before the matrix-vector product is needed (Step 1b). These computations can be done before the public matrix transmission begins, e.g. they could be carried out before and/or during the receival of the TBS data preceding the public key. Once the public key matrix has been fully received and processed (i.e. after Step 2a), the remaining computations of the public operation are carried out (Step 3b), e.g. the addition of the error vector e in the McEliece scheme. The result is either a ciphertext (in case of an encryption scheme) or a Boolean value (in case of a signature verification). But whether this result is output respectively further processed by the device (Step 5a) depends on the result of the signature verification (Step 4). If the verification fails, the device will output an error answer (Step 5b).

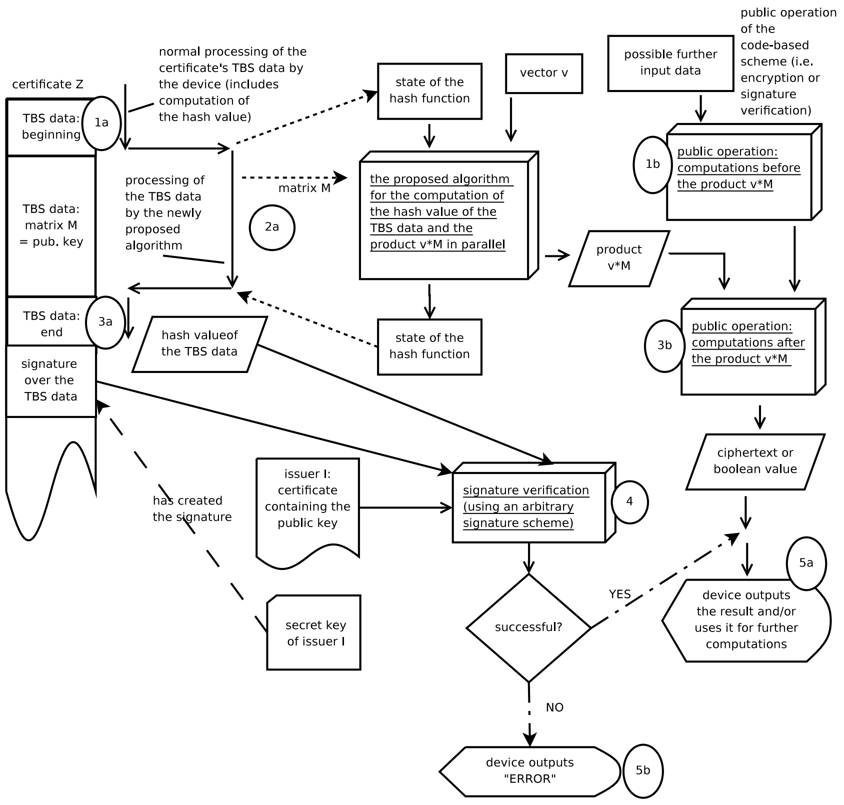
3.2 Transmission Rates

In this section, we give an overview of transmission rates available for embedded systems, especially smart card microcontrollers.

For instance a SLE66CLX360PE [17] smart card platform from Infineon Technologies AG features an ISO/IEC 14443 compliant contactless interface which can transmit up to 106 KB/s. This allows the transmission of a McEliece public key of size 100 KB for the parameters given in Section 2.2 in about 1s, which can be considered at least acceptable for certain applications.

In the future, contactless transmission rates may be about 837,500 bytes/s [30], i.e. about 8 times higher than the rate considered above[3]. In the following

[3] In the referenced work, this transmission rate is actually only achieved in the direction from the card to the reader. However, we want to use it merely as an orientation for the transmission rates achievable in the near future.



certificate Z
TBS data: beginning
1a
normal processing of the certificate's TBS data by the device (includes computation of the hash value)
state of the hash function
vector v
possible further input data
public operation of the code-based scheme (i.e. encryption or signature verification)
TBS data: matrix M = pub. key
processing of the TBS data by the newly proposed algorithm
matrix M
2a
the proposed algorithm for the computation of the hash value of the TBS data and the product v*M in parallel
1b
public operation: computations before the product v*M
product v*M
TBS data: end
3a
hash valueof the TBS data
state of the hash function
3b
public operation: computations after the product v*M
signature over the TBS data
ciphertext or boolean value
has created the signature
issuer I: certificate containing the public key
signature verification (using an arbitrary signature scheme)
4
5a
YES
device outputs the result and/or uses it for further computations
secret key of issuer I
successful?
NO
device outputs "ERROR"
5b

Fig. 1. Overview of the complete process of the on-line public operation.

section, we will show that it is still feasible to sustain such a high transmission rate at typical smart card CPU speeds of about 30 MHz if adequate hardware support is available on the device. Note that in this case there are still about 35 CPU cycles available between the receival of two bytes.

3.3 Example Implementation

We implemented the proposed approach in the C programming language on an ATUC3A1512 32-bit microcontroller from Atmel's AVR32 family. We chose an embedded 32-bit platform basically because SHA-256 is designed for 32-bit platforms. There also exist 32-bit smart card controllers [31], thus our evaluations are significant for this type of platform.

The personal computer (PC) communicates with the AVR32 over a serial line. For the implementation of the serial communication, on the AVR32, we used the API for the device's Universal Asynchronous Receiver Transmitter (UART) provided by Atmel. On the side of the personal computer, we used the API to the serial port of the Linux operating system. The PC can send commands to the AVR32, which are formed by a six byte header and optional payload data, the length of which is encoded the last four header bytes. The first header byte is zero for all commands, and the second byte determines one of the following commands:

- set the vector to multiply
- carry out the on-line multiplication (starts an interactive protocol for the matrix transmission described below)
- get the multiplication result from the AVR32
- get the hash result from the AVR32

The AVR32 responds to these commands by sending a two byte status code and optional data payload preceding the status code, or in the case of the on-line multiplication command, by starting an interactive protocol.

This protocol is depicted in Figure 2. As a precondition, the vector to multiply has to be set in the device through the corresponding command. After the receival of the on-line multiplication command (which does not carry payload data), the AVR32 sets up two buffers $B1$ and $B2$ which are of an equal predefined size. It sends a two byte acknowledgement (ACK) code to the PC as the answer to the command. Then the PC sends the first matrix part which is of equal size as the buffers $B1$ and $B2$. The receival of a single byte over the UART interface of the AVR32 triggers an interrupt which is serviced by an Interrupt Service Routine (ISR) which writes the byte to the next free position in $B1$. After the first block has been received completely, the AVR32 sends another ACK code to the PC, who in turn reacts by sending the next part. At this point the AVR32 exchanges the role of the buffers $B1$ and $B2$: the data is now received to $B2$ (which did not play any role while receiving the first part), and $B1$, containing the first matrix part, is fed into the SHA-256 computation and the matrix multiplication. Both, the hashing and matrix multiplication are implemented as objects which can be updated by calling routines that take arbitrary amounts of data as an argument.

For hash functions, this is the standard implementation technique. Because demanded by our approach, we adopted this technique for the matrix multiplication. In our implementation, the matrix-vector multiplication is carried out column-wise. The advantages and disadvantages of this approach in contrast to row-wise multiplication is discussed in Section 3.4. The multiplication object knows the number of rows and columns of the matrix and has the source vector set. As the matrix data is fed column-wise it keeps track of the current row and column position. It processes the current column by carrying out the logical AND (multiplication in $\mathbb{F}_2$) between the matrix column and the vector 32-bit word-wise, and computes the XOR (addition in $\mathbb{F}_2$) with a 32-bit accumulator. When a column is finished, the parity (i.e. sum of all the word's bits in $\mathbb{F}_2$) of the accumulator is written to the corresponding result bit.

Non-interactive Version of the Protocol It turned out that the interactive protocol incurs significant delay in the communication which most probably results from the fact that our PC program is running in user space and thus sending and receiving data via the serial interface is delayed. If the protocol were implemented in a card terminal, which could be the case in a real world implementation of the on-line multiplication such issues would not arise. To show the efficiency of the approach, we modified the protocol depicted in Figure 2: the AVR32 does not send any ACK answers beyond the very first one. Consequently, the matrix data is sent as a continuous stream after the AVR32 has sent the initial ACK. In this way, the protocol looses the feature that it works independently of the ratio of transmission speed and computation speed: in this non-interactive setting, it must be guaranteed that the hash and multiplication computation of the processed buffer has finished before the receive buffer has been completely filled. With this approach the performance could be improved by a factor of roughly 1.3 compared to the interactive variant of the protocol. The concrete results are discussed shortly.

Simulation of higher Transmission Rates On the chosen AVR32 platform, the maximal transmission speed is given by a baud rate of 460,800. In the RS232 transmission format each data byte is encoded in 10 bits, yielding a net transmission rate of 46,080 byte/s. In order to demonstrate the computation speed that would be possible beyond this limitation, we implemented a means of simulating higher transmission speeds. This is achieved by creating a matrix whose rows have repetitive entries, i.e. the values of 8-bit chunks repeats r times. An example of the beginning of a row for $r = 4$ would be

```
0x1D, 0x1D, 0x1D, 0x1D, 0xA3, 0xA3, 0xA3, 0xA3, 0x22, ...
```

In this setting, on the PC side such a repetitive matrix is generated. When the matrix is transmitted, however, each repeated element is sent only once. On the receiving side, the repetition value r is also known and each received byte is appended to the buffer r times. In this way, we simulate a transmission rate $B_{\text{sim}} = rB_{\text{real}}$, where B_{real} is the actual UART transmission rate.

Table 1 shows the measurement results for the non-interactive version described in the previous paragraph. Here, we used a matrix with 1000 rows and 800 columns, i.e. yielding a size of 100,000 bytes. This is approximately the size of McEliece public keys with 100 bit security [19]. In all our measurements the CPU speed of the AVR32 was set to 33 MHz, since also todays contactless smart card platforms run at approximately this speed, for instance the Infineon Technologies SLE76 [16] smart card controller. The SLE76 CPU only runs at 30MHz, using this in our implementation showed that at this CPU speed the (simulated) transmission rate given in the rightmost column of Table 1 could not be supported in the experiment.

Furthermore, we measured the random error vector creation as the second part of the encryption operation for parameters $n = 2048$ and $t = 50$ to be less than 4 ms at a CPU speed of 33MHz, the addition (XOR) of the error vector to the intermediate vector is certainly even much less complex and thus completely negligible for the timings considered here.

The transmission speed of 386,640 bytes/s, that can be sustained in our test setup, is approximately half of that of the research implementation presented in

[30] already mentioned in Section 3.2. Thus our results show that even without dedicated hardware, todays embedded platforms already enable computation speeds for the hash computation and matrix multiplication not too far from the associated transmission rates that can be expected to be supported by contactless devices in the near future. This makes it feasible that with adequate hardware support the full 837,500 Byte/s rate given in [30] can be supported by the throughput of the computational operations.

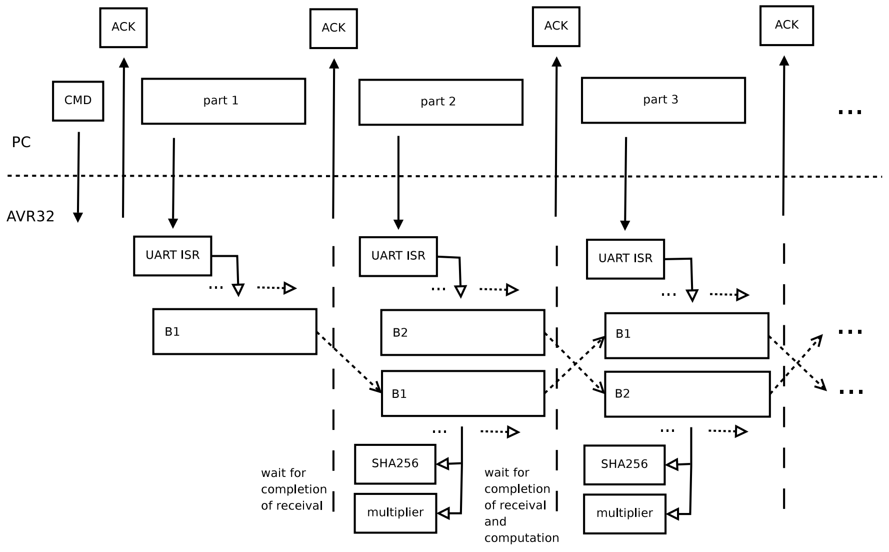


Fig. 2. Schematic overview of the interrupt based implementation of the on-line multiplication.

The hash implementation is based on the open source implementation [32]. The C source code allows for complete unrolling of the SHA-256 compression function through a macro definition. Activating loop unrolling resulted in a performance gain of 1.6 for the hash function computation. All further performance data is based on this implementation choice.

3.4 Column-wise vs. Row-wise Matrix-Vector Multiplication

The row-wise computation of the matrix-vector multiplication is an alternative to the column-wise approach. In this case the computation of the result is according to $b = \sum_i M_i a_i$, where M_i is the vector represented by the i-th row of M and a_i it the i-th entry of the vector a. This means that a row M_i is added to the result if the corresponding bit a_i is one, otherwise nothing has to be done. In the normal case, where the whole matrix is available instantly, this approach has a significant advantage over the column-wise approach since on average half of

	based on computation throughput	**experimental result - w/o ACK**
cycles/byte	measured: 55.6 for SHA-256, 4.2 for mult. yields: **59.8**	92
time at 33MHz CPU for 100,000 Bytes	181ms	**279ms**
transmission rate in bytes/s	551,839	$B_{\mathrm{sim}} = 368,640$ ($r = 8$)

Table 1. Performance of the SHA-256 and binary matrix multiplication on the AT32UC3A1 platform. The results in the first column are based on the throughput benchmarking results for the two computational tasks. The following two entries in this column, that give the resulting time for the on-line multiplication and the transmission rate necessary to support the throughput of both computational tasks, are theoretically derived from the former. In the second column, the time of the whole on-line matrix multiplication with the given transmission rate $B_{\mathrm{sim}} = 8 \cdot 46,080$ byte/s was measured on the ATUC3A1512 platform and the computational throughput given in the first row is the effective throughput corresponding to the measured running time. Here a receive buffer size of 1536 bytes was used.

vector a's bits have value zero. But in the case of the on-line public operation, this advantage disappears since the matrix-vector multiplication's running time is determined by the transmission time alone (under the assumption of sufficient computational power of the device as analyzed in Section 3.2). The row-wise approach would only have an advantage if the saved computational effort could be used to perform other tasks, which can be assumed to be rather unlikely or at least of minor relevance in the context of embedded devices such as smart cards.

On the other hand, the disadvantage of the row-wise multiplication lies in its potential side-channel vulnerability. Specifically, if an attacker is able to find out whether the currently transmitted row is added or ignored, for instance by analyzing the power trace [33], he can deduce the value of the secret bit a_i. Of course, countermeasures can be implemented. A certain randomization could for instance be introduced by keeping a number of received rows in a buffer and processing them in a randomized order. However, whether the questionable computational advantage of this method is worth such efforts must be decided in a concrete implementation scenario.

In any case, once the X.509 key format for a code-based scheme is defined, the choice for one of the two methods is taken. While it then would still be possible to transmit the matrix in the other orientation in order to carry out the multiplication, the on-line hash computation only works if the correct orientation is used.

3.5 Code-based Signature Schemes

A number of code-based signature schemes have been proposed. In the following, we will address two of these schemes very briefly with the goal of showing that the proposed approach for the on-line public operation is applicable to both of them.

In [3], the McEliece scheme is inverted in the sense that the signer proves his ability to decode a binary vector related to the message using a certain code. Thus, the signature verification basically consists of a matrix-vector multiplication just like for the encryption schemes described in Section 2.2. For security considerations concerning this scheme please refer to [34,35].

A signature scheme involving two binary matrices as the public key is presented in [4]. In the verification operation, both matrices have to be multiplied by a vector. Thus the on-line public operation can be carried out by transmitting them one after another. Note, however, that the originally proposed parameters for this scheme are insecure [36].

4 McEliece Decryption without the Parity Check Matrix

In the McEliece scheme, the first step of the decryption operation, which for the sake of brevity we will not not fully explain here, is, as already mentioned in Section 2.2, the computation of the syndrome polynomial $S(Y)$ as $S(Y) \equiv \sum_{i=1}^{n} \frac{c_i}{Y \oplus \alpha_i} \mod g(Y)$, where $g(Y)$ is the Goppa Polynomial, c_i is the i-th ciphertext bit and the α_i is the i-th support element.

In the implementations [21,18,19,22], the syndrome computation is done with the help of the parity check matrix H of the code. This matrix is in fact nothing else than a list of all the n different polynomials $\frac{1}{Y \oplus \alpha_i} \mod g(Y)$ which yields the syndrome vector when multiplied with the ciphertext as a bit vector.

code parameters		$n = 2048$, $t = 50$		$n = 2960$, $t = 56$	
security level		100 bit		> 122 bit	
		cycles	t @ 33 MHz	cycles	t @ 33 MHz
with par. ch. mat.	cyc. whole decr.	$2.00 \cdot 10^6$	61 ms	$3.12 \cdot 10^6$	95 ms
	cyc. only syndr. comp.	$0.26 \cdot 10^6$	8 ms	$0.39 \cdot 10^6$	12 ms
	private key bytes	158,140		277,328	
w/o par. ch. mat.	cyc. whole decr.	$4.42 \cdot 10^6$	134 ms	$7.39 \cdot 10^6$	224 ms
	cyc. only synd. comp.	$2.65 \cdot 10^6$	80 ms	$4,71 \cdot 10^6$	143 ms
	private key bytes	17,340		28,688	

Table 2. Private key sizes, cycle counts and corresponding timings taken for the McEliece decryption operation and its suboperation, the syndrome computation, on an Atmel AT32 AP7000 CPU. Each cycle count was obtained by carrying out the operation ten times and taking the mean of the results.

The syndrome computation without the parity check matrix is in principle achieved by invoking an Extended Euclidean Algorithm (EEA) with $g(Y)$ and $Y \oplus \alpha_i$ as the initial remainders. This EEA executes in a single iteration. Accordingly, in an implementation of the syndrome computation a number of optimizations are possible. The resulting Algorithm is given in Algorithm 3. There, $z[i]$ denotes the i-th ciphertext bit and B_j the coefficient to Y^j of $B(Y)$, etc. Its average complexity (i.e. for a ciphertext with Hamming weight $n/2$), expressed in the terms of additions, multiplication and inversions in $\mathbb{F}_{2^m}$, is $C_{\text{syndr}} = nt(C_{\text{mult}} + C_{\text{add}}) + \frac{n}{2}C_{\text{inv}}$.

We implemented this algorithm in a McEliece PKC implementation based on the open source implementation [21] presented in [7]. Table 2 shows the timing results measured on an Atmel AT32 AP7000 CPU, a CPU similar to the AT32UC3A1. The CPU runs at 150 MHz, however, we give the according running time for the typical smart card CPU speed of 33 MHz, which was already employed in Section 3.3. The smaller parameter set has already been introduced in Section 2.2, the larger one is is proposed in [27] for 128-bit security though there the authors assume addition of $t + 1$ errors, which is possible through the employment of list-decoding [37], which is not supported by our implementation. Accordingly the security level here is only approximate, but the reduction of security of our implementation only using t errors, however, can easily be bounded by understanding that an attacker can get from a ciphertext with $t+1$ errors to t errors by guessing one error position correctly, the success probability of which is $(t + 1)/n = 0.02$. Accordingly, the security of the scheme with t errors cannot be smaller than $128 - \log_2(1/0.02) > 122$ bits.

The respective private key sizes without the parity check matrix given in Table 2 are formed by the Goppa Polynomial $g(Y)$, the support Γ, a matrix for computing the square root modulo $g(Y)$, which is needed to speed up the decryption, and the logarithm and anti-logarithm tables for $\mathbb{F}_{2^m}$, each of size $\lceil \log_2 n \rceil$ elements, i.e. a total of 8192 resp. 16384 bytes for either parameter set (each element occupies 2 bytes). These tables need not necessarily be stored in the key, instead they can be created in RAM before the decryption operation, if allowed by the memory constraints of the given platform.

From this example implementation, that does not use any hardware support for the $\mathbb{F}_{2^m}$ operations or DSP instructions, we see that the decryption time approximately doubles when the parity check matrix is not stored as part of the key, but due to the general speed advantage of the McEliece scheme over RSA or Elliptic curve based schemes [7,18] these timings are still highly competitive.

5 Conclusion

In this work, on the one hand, we have shown an approach for implementing the operations involving code-based public keys on memory-constrained devices like smart cards, that covers the matrix-vector multiplication as well as the hash computation for the verification of the user certificate. The solution is applicable to basically all code-based encryption and signature schemes that

Algorithm 3 The Syndrome computation without parity check matrix

```
Require: the ciphertext $\boldsymbol{z} \in \mathbb{F}_2^n$, and the Goppa Polynomial $g(Y) \in \mathbb{F}_{2^m}[Y]$ of degree $t$
Ensure: the syndrome polynomial $S(Y) \in \mathbb{F}_{2^m}[Y]$ of degree $\leqslant t-1$
  $S(Y) \leftarrow 0$
  for $i \leftarrow 0$ up to $n-1$ do
    if $\boldsymbol{z}[i] = 1$ then
      $B(Y) \leftarrow 0$
      $b \leftarrow g_t$
      for $j \leftarrow t-1$ down to 0 do
        $B_j \leftarrow b$
        $b \leftarrow b \cdot \alpha_i \oplus g_j$
      end for
      $f \leftarrow b^{-1}$
      for $j \leftarrow 0$ up to $\deg(B(Y))$ do
        $S_j \leftarrow S_j \oplus f \cdot B_j$
      end for
    end if
  end for
```

have been proposed so far. Thus we are confident that this work improves on the applicability of this class of cryptographic schemes by reducing the impact of the large public key sizes for memory-constrained devices.

Furthermore, we also showed that the McEliece private key size can be dramatically reduced by excluding the parity check matrix while still allowing for practical decryption timings on memory constrained devices. As a result, the McEliece scheme in our opinion gains superiority over the Niederreiter scheme: while the situation for the public key as discussed in this work is equal for both schemes, only the McEliece scheme allows for the reduction of the private key size as proposed in this work. The reason is simply that the Niederreiter private key size is mainly determined by the size of the scrambler matrix T, which cannot be excluded in this scheme. In [20], the matrix T is implemented as a pseudorandom sequence of bits, which effectively reduces the Niederreiter private size, but this comes at the expense of public key size: because of the pseudorandom nature of T, the public key matrix cannot be in reduced row echelon form, resulting in an otherwise unnecessary increase of the public key size. In view of the results of the first part of this work, this directly affects the time taken by the encryption operation on an embedded device (at least in a PKI context).

References

1. R. J. McEliece: A public key cryptosystem based on algebraic coding theory. DSN progress report **42–44** (1978) 114–116
2. Niederreiter, H.: Knapsack-type cryptosystems and algebraic coding theory. In: Problems Control Inform. Theory. Volume Vol. 15, number 2. (1986) 159–166

3. Courtois, N., Finiasz, M., Sendrier, N.: How to Achieve a McEliece-Based Digital Signature Scheme. In Boyd, C., ed.: Advances in Cryptology - ASIACRYPT 2001. Volume 2248 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2001) 157–174
4. Kabatianskii, G., Krouk, E., Smeets, B.: A digital signature scheme based on random error-correcting codes. In Darnell, M., ed.: Crytography and Coding. Volume 1355 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (1997) 161–167
5. Stern, J.: A new identification scheme based on syndrome decoding. In: CRYPTO '93: Proceedings of the 13th annual international cryptology conference on Advances in cryptology, New York, NY, USA, Springer-Verlag New York, Inc. (1994) 13–21
6. Bernstein, D.J., Buchmann, J., Dahmen, E.: Post Quantum Cryptography. Springer Publishing Company, Incorporated (2008)
7. Biswas, B., Sendrier, N.: McEliece Cryptosystem Implementation: Theory and Practice. In: PQCrypto. (2008) 47–62
8. Berger, T.P., Cayrel, P.L., Gaborit, P., Otmani, A.: Reducing Key Length of the McEliece Cryptosystem. In: AFRICACRYPT '09: Proceedings of the 2nd International Conference on Cryptology in Africa, Berlin, Heidelberg, Springer-Verlag (2009) 77–97
9. Berger, T.P., Loidreau, P.: How to Mask the Structure of Codes for a Cryptographic Use. Designs, Codes and Cryptography **35** (2005) 63–79 10.1007/s10623-003-6151-2.
10. Misoczki, R., Barreto, P.: Compact McEliece Keys from Goppa Codes. In Jacobson, M., Rijmen, V., Safavi-Naini, R., eds.: Selected Areas in Cryptography. Volume 5867 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2009) 376–392
11. Otmani, A., Tillich, J.P., Dallot, L.: Cryptanalysis of Two McEliece Cryptosystems Based on Quasi-Cyclic Codes. Mathematics in Computer Science **3** (2010) 129–140
12. Faugère, J.C., Otmani, A., Perret, L., Tillich, J.P.: Algebraic Cryptanalysis of McEliece Variants with Compact Keys. In: In Proceedings of Eurocrypt 2010. (2010)
13. Umana, V.G., Leander, G.: Practical key recovery attacks on two McEliece variants. In: SCC 2010: Proceedings of the Second International Conference on Symbolic Computation and Cryptography. Royal Holloway, University of London, Egham, UK, 2325, Carlos Cid, Jean-Charles Faugere (editors) (2010) 27–44
14. Goppa, V.D.: A new class of linear correcting codes. Problems of Information Transmission **6** (1970) 207–212
15. Faugère, J.C., Otmani, A., Perret, L., Tillich, J.P.: A distinguisher for high rate mceliece cryptosystems. In: Information Theory Workshop (ITW), 2011 IEEE, IEEE (2011) 282–286
16. Infineon Technologies AG: SLE76 Product Data Sheet `http://www.infineon.com/cms/de/product/channel.html?channel=db3a3043156fd57301161520ab8b1c4c`.
17. Infineon Technologies AG: SLE 66CLX360PE(M) Family Data Sheet `http://www.infineon.com/dgdl/SPI_SLE66CLX360PE_1106.pdf?folderId=db3a304412b407950112b408e8c90004&fileId=db3a304412b407950112b4099d6c030a&location=Search.SPI_SLE66CLX360PE_1106.pdf`.
18. Eisenbarth, T., Güneysu, T., Heyse, S., Paar, C.: MicroEliece: McEliece for Embedded Devices. In: CHES '09: Proceedings of the 11th International Workshop

on Cryptographic Hardware and Embedded Systems, Berlin, Heidelberg, Springer-Verlag (2009) 49–64
19. Strenzke, F.: A Smart Card Implementation of the McEliece PKC. In: Information Security Theory and Practices. Security and Privacy of Pervasive Systems and Smart Devices. Volume 6033 of Lecture Notes in Computer Science., Springer Berlin / Heidelberg (2010) 47–59
20. Heyse, S.: Low-Reiter: Niederreiter Encryption Scheme for Embedded Microcontrollers. In Sendrier, N., ed.: Post-Quantum Cryptography. Volume 6061 of Lecture Notes in Computer Science., Springer Berlin / Heidelberg (2010) 165–181
21. Biswas, B., Sendrier, N.: HyMES - an open source implementation of the McEliece cryptosystem (2008) `http://www-rocq.inria.fr/secret/CBCrypto/index.php?pg=hymes`.
22. Shoufan, A., Wink, T., Molter, G., Huss, S., Strenzke, F.: A Novel Processor Architecture for McEliece Cryptosystem and FPGA Platforms. In: ASAP '09: Proceedings of the 2009 20th IEEE International Conference on Application-specific Systems, Architectures and Processors, Washington, DC, USA, IEEE Computer Society (2009) 98–105
23. German Federal Bureau of Information Security (BSI): Technical Guideline TR-03110: Advanced Security Mechanisms for Machine Readable Travel Documents, Version 2.02 (2009)
24. German Federal Bureau of Information Security (BSI): Technical Guideline TR-03111: Elliptic Curve Cryptography, Version 1.11 (2009)
25. F. J. MacWilliams and N. J. A. Sloane: The theory of error correcting codes. North Holland (1997)
26. Kobara, K., Imai, H.: Semantically secure McEliece public-key cryptosystems - conversions for McEliece PKC. Practice and Theory in Public Key Cryptography - PKC '01 Proceedings (2001)
27. Bernstein, D.J., Lange, T., Peters, C.: Attacking and defending the McEliece cryptosystem. Post-Quantum Cryptography, LNCS **5299** (2008) 31–46
28. Cooper et al.: RFC 5280 `http://tools.ietf.org/html/rfc5280`.
29. Coronado, L.C., Buchmann, J., Carlos, L., Garcia, C., Dahmen, E., Klintsevich, E., Darmstadt, T.U.: CMSS – An Improved Merkle Signature Scheme Johannes Buchmann (2006) `www.cdc.informatik.tu-darmstadt.de/~dahmen/papers/BCDDK06.pdf`.
30. Witschnig, H., Patauner, C., Maier, A., Leitgeb, E., Rinner, D.: High speed RFID lab-scaled prototype at the frequency of 13.56 MHz. e & i Elektrotechnik und Informationstechnik **124** (2007) 376–383 10.1007/s00502-007-0485-9.
31. Infineon Technologies AG: SLE78 Product Data Sheet `http://www.infineon.com/cms/en/product/channel.html?channel=db3a30431ce5fb52011d47b166342af0`.
32. Olivier Gay: `http://www.ouah.org/ogay/sha2/`.
33. Kocher, P., Jaff, J., Jun, B.: Differential Power Analysis. Advances in Cryptology-CRYPTO'99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings **1666** (1999) 388–397
34. Overbeck, R., Sendrier, N.: Code-based cryptography. In Bernstein, D., Buchmann, J., Dahmen, E., eds.: Post-Quantum Cryptography. Springer (2009) 95–145
35. Finiasz, M.: Parallel-CFS: Strengthening the CFS McEliece-based signature scheme. In Biryukov, A., Gong, G., Stinson, D., eds.: Selected Areas in Cryptography. Volume 6544 of LNCS., Springer (2010) 159–170
36. Otmani, A., Tillich, J.P.: An Efficient Attack on All Concrete KKS Proposals. In Yang, B.Y., ed.: PQCrypto 2011. Volume 7071 of LNCS., Springer (2011) 98–116

37. Bernstein, D.J.: List Decoding for binary Goppa Codes. In: Coding and cryptology – third international workshop, IWCC 2011, Qingdao, China, May 30–June 3, 2011, Yeow Meng Chee, Zhenbo Guo, San Ling, Fengjing Shao, Yuansheng Tang, Huaxiong Wang, and Chaoping Xing (editors), Lecture Notes in Computer Science 6639, Springer, 2011 (2011) 62–80