Fast and Secure Root Finding for Code-based Cryptosystems

Falko Strenzke[1]

[1] fstrenzke@gmx.de **,
[2] Cryptography and Computeralgebra, Department of Computer Science, Technische Universität Darmstadt, Germany

Abstract. In this work we analyze four previously published respectively trivial approaches to the task of finding the roots of the error locator polynomial during the decryption operation of code-based encryption schemes. We compare the performance of these algorithms and show that optimizations concerning finite field element representations play a key role for the speed of software implementations. Furthermore, we point out a number of timing attack vulnerabilities that can arise in root-finding algorithms, some aimed at recovering the message, others at the secret support. We give experimental results of software implementations showing that manifestations of these vulnerabilities are present in straightforward implementations of most of the root-finding variants presented in this work. As a result, we find that one of the variants provides security with respect to all vulnerabilities as well as the fastest computation time for code parameters that minimize the public key size.

Keywords: side channel attack, timing attack, implementation, code-based cryptography

1 Introduction

Implementations of code-based cryptosystems like the McEliece [1] and Niederreiter [2] schemes have received growing interested from researchers in the past years and been analyzed with respect to efficiency on various platforms [3–7]. Furthermore, a growing number of works has investigated the side-channel security of code-based cryptosystems [8–12].

In this work, we will turn to an algorithmic task that arises in the decryption operation of both Niederreiter and McEliece cryptosystems. This is the root-finding algorithm. It deserves attention for two reasons: first of all, as addressed already in previous work, it is in general the most time-consuming part of the decoding algorithm [3, 5]. The second aspect is that of side-channel security of the root-finding and has so far, to the best of our knowledge, only been considered in [13]. We point out basically two types of timing side-channel vulnerabilities that

** To the most part, this work was done in the author's private capacity, a part of the work was done at[2] can arise in the root-finding procedure. One is aimed at recovering the message to a given ciphertext, the other at finding the support that is part of code-based private keys.

Based on these considerations, we chose four different root-finding algorithms, which we describe in Sec. 4 after providing some elementary preliminaries about code-based cryptosystems in Sec. 2. In Sec. 5, we perform a timing side-channel analysis of these algorithms including theoretical discussions and experimental results. Afterwards, in Sec. 6, we give the results of a performance evaluation of the chosen algorithms on modern x86 architectures.

As the main result from our work, we find that the root-finding variant according to [14], which to the best of our knowledge has not been taken into consideration for code-based cryptosystems so far, achieves both the shortest running time and security with respect to all potential timing side-channel vulnerabilities in the root-finding. But we wish to stress that it is not the aim of this work to give a definitive answer to the question which root-finding variant is the best to use in a code-based cryptosystem. This is mainly for the reason that the we are considering pure software implementations here, and the use of crypto coprocessors might change the picture. We will come back to this in the Conclusion in Sec. 7.

As the starting point for the implementation that the experimental results of this work are based on, we used the HyMES open source implementation [15] of the McEliece scheme presented in [3]. We added the countermeasure previously proposed in [11], and removed some fault attack vulnerabilities, the latter is addressed in App. B. Furthermore, we adopted the root-finding algorithm variant given by the Berlekamp Trace Algorithm [16], as it is found in HyMES. However, we performed some straightforward optimizations in the implementation of this algorithm which are mentioned in Sec. 4.2, We added the remaining root-finding variants that are presented in Sec. 4. In this context, we want to emphasize that we do not take any credit for the choice of the Berlekamp Trace Algorithm for the purposes of root-finding in code-based cryptosystems, the implementation of this algorithm used for the results in this work, and its description as given in this work, since all of this is adopted from [3] resp. [15].

2 Preliminaries

In the following, we describe those aspects of the encryption and decryption of code-based cryptosystems as McEliece [1] and Niederreiter [2] schemes that are relevant for the topics of this work. As these only depend on properties common to both types of cryptosystems, it is possible for us to basically omit any distinction between them.

Code-based cryptosystems build on error correcting codes. Specifically, the only codes known to be secure for this use are Goppa Codes [17]. The parameters of such a code are its length n, which is the length of the code words, $n \leqslant 2^m$; the dimension k (with $k < n$), which is the length of the message words; and the error correcting capability t (all lengths refer to lengths of the binary representation).

For such a code, if a message word is encoded into a code word, up to t bit flip errors in the code word may be corrected by a corresponding error correction algorithm, thus allowing to recover the message word. In the following, we will also make use of the expression of "adding errors" to the code word, by which we mean carrying out the "exclusive or" (XOR) operation with the code word $\boldsymbol{c}$ and the error vector $\boldsymbol{e}$, i.e. $\boldsymbol{c} \oplus \boldsymbol{e}$.

In both McEliece and Niederreiter schemes, the encryption involves the creation of an error vector $\boldsymbol{e}$, whose Hamming weight $\mathrm{wt}(\boldsymbol{e})$ is equal to the error correcting capability t of the employed code. The concrete realization of the encryption is different in both schemes, but in either case, it is vital for the privacy of the encrypted message that $\boldsymbol{e}$ remains secret.

In the McEliece and Niederreiter cryptosystems, syndrome decoding through Patterson's Algorithm [18] plays a key role. As the details of this algorithm are irrelevant for purposes of understanding the topics of this work, we give only a brief outline of this algorithm. In the McEliece cryptosystem, the syndrome is computed from the ciphertext by multiplying the ciphertext with the so called parity check matrix, in the Niederreiter scheme, the syndrome is the ciphertext itself. The Patterson decoding algorithm takes as input the so called syndrome vector $\boldsymbol{s}$ and outputs the error vector $\boldsymbol{e}$ that was added to the code word.

As already mentioned, this work deals with one part of the Patterson Algorithm, this is the finding of the roots, i.e. zeros of the so called error locator polynomial $\sigma(Y) \in \mathbb{F}_{2^m}[Y]$ which is computed in the course of the Patterson Algorithm. In case of $w = \mathrm{wt}(\boldsymbol{e}) \leqslant t$ it holds that

$$\sigma(Y) = \Pi_{i=1}^{w}(\alpha_i - Y), \tag{1}$$

where the ordered set $\Gamma = (\alpha_0, \alpha_1, \ldots, \alpha_{n-1})$, is the so called support formed by pairwise distinct elements of $\mathbb{F}_{2^m}$. A lookup table representing the support is part of the code-based private key. If it becomes known, the whole key is compromised [12]. From the determination of the roots of $\sigma(Y)$, the error positions, i.e. those bits in $\boldsymbol{e}$ that have value one, are found: if α_{E_i} is root of $\sigma(Y)$, then $e_{E_i} = 1$. In the following, we shall the use E_i, $i = 1, \ldots, w$ to denote the indexes of bits having value one in $\boldsymbol{e}$ in arbitrary ordering. If $w > t$, then $\sigma(Y)$ will have degree less or equal than t, where the probability for the latter is very high, but it will not be of the form given in Equ. (1). In Sec. 4, we will present four different concrete algorithms for the task of finding the roots of the error locator polynomial $\sigma(Y)$.

3 Remarks about the $\mathbb{F}_{2^m}$ Operations

Before we start with the descriptions of the root-finding algorithms we compare in this work, we want to point out some details concerning the costs of the basic $\mathbb{F}_{2^m}$ operations that are involved, i.e. addition and multiplication.

While $C_{\mathrm{gf\_add}}$, the cost of an addition in $\mathbb{F}_{2^m}$, is given by a simple XOR operation, the multiplication in $\mathbb{F}_{2^m}$ is much more complex and has a number of variants. An efficient software implementation of finite field arithmetics with characteristic 2 and small extension degrees is realized by the use of one lookup table for the logarithm of each non-zero element to the base of some primitive element, and the corresponding anti-logarithm table.

The standard multiplication, as it is for instance implemented by the "C" macro `gf_mul()` in HyMES [15], which is used throughout their code, takes arguments in the normal representation and outputs the result in normal representation. This type of multiplication, we refer to as *mul_nnn* . Its cost is two conditional branches to check whether the arguments are zero, three table lookups, one arithmetic ADD, and reduction of the result modulo the fields multiplicative order, which in turn consists of several instructions. In the general case, this multiplication is needed, as in most places in the algorithms involved in the decoding with the Patterson Algorithm, multiplication and addition in $\mathbb{F}_{2^m}$ are intermixed, and moreover, operands having value zero cannot be excluded.

However, when operands are known to be non-zero, and multiplications are carried out subsequently, other forms of the multiplication, which have results (a in the algorithm description *mul_abc*) or operands (b and c) in the logarithmic representation, are more efficient:

- *mul_lll* consists only of one arithmetic ADD (a certain number of these multiplications can be carried out before a reduction modulo the multiplicative order becomes necessary to avoid overflowing the register)
- *mul_nln* saves one conditional branch and one table lookup compared with *mul_nnn*

This rough review of the finite field arithmetic implementations in software makes it obvious that it is not sufficient to simply count the instances of multiplication in $\mathbb{F}_{2^m}$, but it has to be considered how the multiplication is embedded into the algorithm and what variant of the multiplication can be used.

4 Variants of Root Finding

In the following subsections we give brief descriptions of the root-finding algorithm variants analyzed in this work.

4.1 Exhaustive Evaluation with and without Division

The most straightforward implementation of the root finding is to simply evaluate the polynomial $\sigma(Y)$ for each element of the code.

The complexity of this algorithm is given as

$$C_{\mathrm{eval-rf}} = nt(C_{\mathrm{gf\_add}} + C_{\mathrm{mul\_nln}})$$

Remember that n is the code length and t is the error correcting capability. Taking a look at the Horner Scheme evaluation used here, we see that when evaluating $\sigma(Y)$, we can transform $x \neq 0$ to the logarithmic representation, avoiding some unnecessary table lookups, i.e. make use of *mul_nln* .

The algorithm can be sped up by dividing the polynomial $\sigma(Y)$ by each root found. Such a division has basically the same complexity as the evaluation of the polynomial for one single element of $\mathbb{F}_{2^m}$. In the following, we will call these two variants *eval-rf* and *eval-div-rf*.

4.2 Berlekamp Trace Algorithm

As stated in the introduction, our implementation is based on the HyMES implementation [3, 15]. There, the root finding is achieved by the so called Berlekamp Trace Algorithm [16]. For completeness, we provide the description of this algorithm as originally given in [3] in Alg. 1. The initial call to this recursive algorithm is given as $\mathrm{BTA}(\sigma(Y), 1)$, which we will refer to as *bta-rf* for the remainder of this work. The trace function is defined as $\mathrm{Tr}(Y) = Y + Y^2 + Y^{2^2} + \ldots + Y^{2^{m-1}}$, and $\{\beta_1, \beta_2, \ldots, \beta_m\}$ is a standard basis of $\mathbb{F}_{2^m}$.

Algorithm 1 The recursive Berlekamp Trace Algorithm $\mathrm{BTA}(\sigma(Y), i)$.

Require: the error locator polynomial $\sigma(Y)$
Ensure: the roots $\mathcal{E}'$ of $\sigma(Y)$

```
if deg (σ(Y) ⩽ 1) then
    return root of σ(Y)
end if
σ_0(Y) ← gcd(σ(Y), Tr(β_i, z))
σ_1(Y) ← gcd(σ(Y), 1 + Tr(β_i, z))
return BTA(σ_0(Y), i + 1) ∪ BTA(σ_1(Y), i + 1)
```

In [19], the complexity of the BTA is given as $\mathcal{O}(mt^2)$. The concrete cost in terms of $\mathbb{F}_{2^m}$ operations is not given there and cannot be easily derived.

In order to make fair comparison between in terms of performance , we optimized the existing implementation of the algorithm by applying the more cost-efficient versions of multiplication in $\mathbb{F}_{2^m}$ as discussed in Sec. 3 where possible. As a result, the running time was reduced by about 10%.

4.3 Root Finding with linearized Polynomials

In this section, we explain a root-finding method based on decomposing a polynomial in $\mathbb{F}_{2^m}[Y]$ into linearized polynomials [14]. The idea of this approach is based on the fact that the exhaustive evaluation of a linearized polynomial can be done with much less computational complexity than for general polynomials.

Definition 1. *A polynomial $L(Y)$ over $\mathbb{F}_{2^m}$ is called a linearized polynomial if $L(Y) = \sum_i L_i Y^{2^i}$, where $L_i \in \mathbb{F}_{2^m}$.*

As shown in [14], an affine polynomial of the form $A(Y) = L(Y) + \beta$ with $\beta \in \mathbb{F}_{2^m}$ can be evaluated for the value $Y = x_i$ as

$$A(x_i) = A(x_{i-1}) + L(\Delta_i), \Delta_i = x_i - x_{i-1} = \alpha^{\delta(x_i, x_{i-1})}, \tag{2}$$

where $\{\alpha^0, \alpha^1, \ldots, \alpha^{m-1}\}$ is a standard basis of $\mathbb{F}_{2^m}$ and $\text{wt}\,(x_i \oplus x_{i-1}) = 1$, i.e. their Hamming distance is 1. A generic decomposition of a polynomial $f(Y) = \sum_{i=0}^{t} f_i Y^i$, also given in [14], is

$$f(Y) = f_3 Y^3 + \sum_{i=0}^{\lceil (t-4)/5 \rceil} Y^{5i} A_i(Y), \qquad (3)$$

where

$$A_i(Y) = f_{5i} + \sum_{j=0}^{3} f_{5i+2^j} Y^{2^j}. \qquad (4)$$

The evaluation of each $A_i(x_i)$ is done efficiently according to Equ. (2). To this end, the exhaustive evaluation of Equ. (3) is done with the x_i being in Gray-Code ordering, i.e. for all i we have that x_i and x_{i+1} differ only in one single bit. Specifically, we use the Gray Code generated by $x_i = (\text{i} >> 1) \oplus i$, where ">>" denotes logical right shift. The actual computation cost is given by the sum of the precomputations, i.e. the computation of the $A_i(Y)$. This cost is given in [14], it is however negligible for secure code parameters. The dominating cost is that of computing $f(Y)$ for all n code elements:

$$C_{\text{dcmp-rf}} = (n-1)\left(2C_{\text{look}} + C_{\text{mul\_nll}} + \lceil (t+1)/5 \rceil \left(2C_{\text{gf\_add}} + C_{\text{mul\_lll}} + C_{\text{mul\_nln}}\right)\right)$$

where C_{look} refers to the cost of looking up $\log(x^3)$ resp. $\log(x^5)$ from precomputed tables. This optimization we use to avoid the computation of $\log(x^3)$ and $\log(x^5)$ has only a small benefit in speed, as the computation of these values through subsequent *mul_lll* operations is also fast.

5 Security Aspects of the Root Finding in Code-based Cryptosystems

In this section, we show that a dependency of the root finding algorithm's running time on the number of the roots of the Error Locator Polynomial $\sigma(Y)$ introduces vulnerabilities to timing attacks against the cleartext, and that other effects threaten the secrecy of the secret support Γ.

In order to be able to judge the relevance of the timing results provided in this section, it is important to know that the syndrome decoding with the Patterson Algorithm, which we consider to include the root-finding, is at least for the McEliece cryptosystem the only source of variable running time, under some assumptions: multiplication of the ciphertext with the parity check matrix is either constant time of linear in the ciphertext's Hamming weight (these are the straightforward implementation choices) and the CCA2 conversion (necessary for both the McEliece and the Niederreiter cryptosystem, see for instance [21]) is constant time (see for instance [20]).

5.1 Security against Message-aimed Attacks

The first important fact to know is that $\sigma(Y)$ output by Patterson's Algorithm has $\mathrm{wt}(\boldsymbol{e})$ roots in case $\mathrm{wt}(\boldsymbol{e}) \leqslant t$, and only a fraction of t roots if $\mathrm{wt}(\boldsymbol{e}) > t$ (refer to Equ. (1)). For instance, for $n = 2048$ and $t = 50$, $\sigma(Y)$ typically has less than five roots in the latter case.

A dependence of the running time on the number of roots thus potentially creates the following problem: if the case $w > t$ can be inferred from the running time, an attack similar to that described in [9] is possible. In such an attack, the attacker flips a bit in ciphertext he wishes to decrypt, observes the decryption, and from the running time tries to guess whether $t + 1$ or $t - 1$ errors resulted from his bit flip. This clearly gives him information about the error positions piece by piece.

Note that the case of $w < t$ is covered by the countermeasures proposed in [11]. In the presence of these countermeasures, the decryption of a ciphertext with $\mathrm{wt}(e) < t$ also results in $\sigma(Y)$ with degree t and very few roots. But it is important to be aware that even if $w < t$ and $w > t$ are not distinguishable based on the timings, but $w = t$ can be distinguished from $w \neq t$, an attack is still possible: by flipping two bits in a ciphertext and trying to find those cases where $w = t$, the attacker will learn whenever he flipped one non-error and one error position.

In the Patterson Algorithm of our implementation, the countermeasure proposed in [11] is included, so that for $w < t$ we still have $\deg(\sigma(Y)) = t$, but the number of roots of $\sigma(Y)$ is much less than t, as already mentioned. This happens automatically for $w > t$ in Patterson's Algorithm, so that we can expect to find major differences in the running time of the root-finding algorithm only for the cases $w \neq t$ and $w = t$.

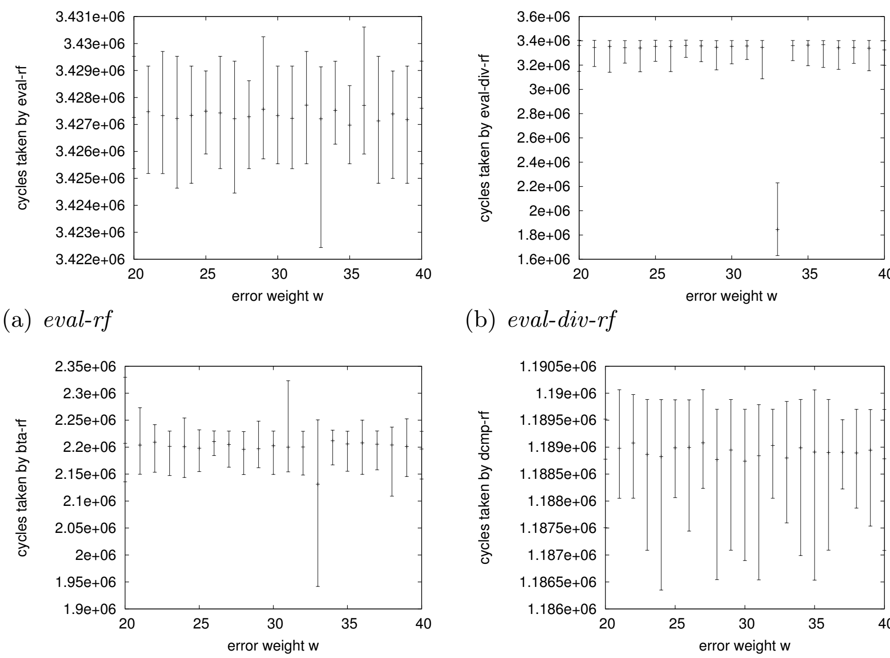
In Figures 1(b), 1(a), 1(c) and 1(d), we give plots of the running time of the root-finding for the four different algorithms. The timings were taken on an Atmel ATMega1284P Microcontroller. We chose this platform, as it provides far more deterministic cycle counts than a modern x86 CPU, and thus is more suited to identify possible timing vulnerabilities. We used a Goppa Code with parameters $n = 512$ and $t = 33$ for the syndrome decoding performed on the microcontroller and created 30 different syndromes for each value of w between 20 and 40. The cycle counts apply to the running time of the respective root-finding algorithm. For each value of w the center mark indicates the mean of the set of the 30 different syndromes, and the bar shows the minimal and maximal values from this set.

The *eval-rf* algorithm's running time, depicted in Fig. 1(a), shows the mean running time of $w = t$ in line with the those of $w \neq t$. However, there seem to be cases of $w = t$ with considerably lower running time than for $w \neq t$, as can be seen by the depicted minimal value. Neither did we find the reason for this, nor did we analyze whether this effect can be used for actual attacks. We justify these omissions by the fact that, as already apparent from the results given in this section, *eval-rf* is not a competitive candidate for root-finding in code-based cryptosystems.

Fig. 1(b) shows clearly the speedup by a factor of two by *eval-div-rf* compared to *eval-rf*. However, also the inherent timing vulnerability [13] of this algorithm cannot be overlooked: the case $w = t$, where the benefit of the divisions has its real impact, is almost twice as fast as for $w \neq t$. This renders it an insecure choice.

The timing results of *bta-rf*, as implemented in HyMES, are shown in Fig. 1(c). Here, we can realize that already the mean of the running times for $w = t$ is below most of the minimal values of sets for $w \neq t$, clearly indicating a vulnerability. We did however not find the reason for this effect.

Finally, Fig. 1(d) shows the results for *dcmp-rf*. There is no apparent difference between the cases $w = t$ and $w \neq t$.



(a) *eval-rf* (b) *eval-div-rf* (c) *bta-rf* (d) *dcmp-rf*

Fig. 1: Cycle counts taken on an ATMega1284P for the various root-finding algorithm variants with parameters $n = 512$ and $t = 33$.

5.2 Security with respect to Attacks aiming at the secret Support

We now show that other vulnerabilities can arise in the root-finding algorithm, which allow attacks against the secret support of the code-based scheme. This is for instance the case, when the running time of the root-finding algorithm depends on the values of the roots found. To understand that this is a vulnerability one has to consider that an attacker can create ciphertexts with $\boldsymbol{e}$ known to him. Then, according to Equ. (1) any information about the roots is information about the support Γ.

One possible vulnerability arises if in *eval-div-rf*, the evaluation of $\sigma(Y)$ is done with Y being substituted in lexicographical order; in this case the found roots are later mapped to the corresponding E_i values by using a table for Γ^{-1}: Fig. 2(a) and 2(b) in given in App. A show running times on the AVR platform of the syndrome decoding with *eval-div-rf* for $n-(t-1)$ error vectors created in the following way: a random error pattern of weight $t-1$ was fixed, and the position of the last error, E_t was varied over the remaining free positions, resulting in an error vector with Hamming weight t. On the x-axis, E_t is shown. We will refer to this type of plot as "support scan" henceforth. The result is a relatively clear linear ascend, which is not surprising when considering the *eval-div-rf* algorithm: Starting evaluation at $Y = 0$, the earlier a root is found, the more beneficial is the reduction of the degree of $\sigma(Y)$ by one through the subsequent division. Thus, the task for an attacker amounts to bringing the measured timings into an ascending ordering, giving him Γ. Obviously, there is some distortion of this ordering in Fig. 2(a), which stems from other operations of variable duration in the syndrome decoding. We leave it open whether in this manner the support Γ becomes known to the attacker in its entirety, it is however clearly obvious, that a large amount of information about Γ becomes available.

This vulnerability can be avoided by performing the evaluation of $\sigma(Y)$ with Y being substituted in the order $\alpha_0, \alpha_1, \ldots, \alpha_{n-1}$. Note however that the vulnerable version is slightly faster, since there only t table lookups in Γ^{-1} for the found roots are done, whereas in the secure version n such lookups in Γ are necessary. Thus the described problem is realistic.

We also wish to point out that an attack exploiting this vulnerability, in contrast to other previously published timing attacks [11, 10], cannot be detected: The ciphertext carries t errors and will pass the CCA2 integrity test. This is important, because the other attacks, which cannot be carried out in a clandestine manner in this sense, can be thwarted by countermeasures which detect the irregularity of the ciphertext, and for instance add an enormous delay or enforce constant running time if possible on the respective platform. In the presence of the threat of power analysis attacks, however, such countermeasures would in most cases not suffice as adding delays after the actual computation will most likely be detectable in the power trace.

We also analyzed the *dcmp-rf* and *bta-rf* algorithms with respect to these vulnerabilities. As one should expect from an algorithm that performs an exhaustive search, *dcmp-rf* does not exhibit any dependency of the running time on the root values, except for a single pitfall that has to be avoided. This is discussed in some detail in App. A. Though for *bta-rf* no concrete attack could be derived, the question of its security with respect to key-aimed attacks remains unclear, we give the analysis also in App. A.

6 Performance of the Root-finding Variants

In this section, we give a comparison of the performance of *eval-div-rf*, *bta-rf*, and *dcmp-rf*. We omit *eval-rf* as it is a hopeless candidate in terms of running time as we have already seen in Sec. 5. The code was compiled with GCC version 4.5.2 with the optimization options `-finline-functions -O3 -fomit-frame-pointer -march=i686 -mtune=i686` and run on a Intel(R) Core(TM)2 Duo CPU U7600 CPU.

Tab. 1 summarizes the results for two parameter sets based on the propositions given in [22], which are based on code parameter choices aiming at the minimization of the public key size, which is known to be the most problematic feature of code-based cryptosystems. The only deviation of our parameter choices are with respect to the number of errors added during encryption: in [22], List Decoding [23] which allows for the correction of more than t errors is assumed during decryption. For the smaller parameter set, they choose $t+1$ errors and for the larger $t+2$ errors. The reduction of security of our implementation only using only 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. An according calculation for the larger parameter set gives a lower bound of 244 bits.

It is noteworthy that these parameter sets optimized for minimal public key size for a given security level use codes with $n < 2^m$, and that this has different effects for our four candidate algorithms. *eval-rf* and *eval-div-rf* will both be faster for $n < 2^m$ in contrast to $n = 2^m$, however for the latter the speedup is less than for the former, as there the roots found at the end of the support cause less effort. *dcmp-rf* also benefits from $n < 2^m$, since also then support can be build from a Gray Code. The *bta-rf* however has the same running time no matter whether $n < 2^m$ or $n = 2^m$.

For these parameters, we clearly see that *dcmp-rf* is clearly the fastest choice. *bta-rf* is second, for the smaller parameter set closely followed by *eval-rf*. For the larger parameter set, the spread between them is larger.

However, it must be pointed out, that for parameter choices that realize a given security level by choosing t as small as possible and $m = 2^n$, *bta-rf* clearly wins against *dcmp-rf*. In Tab. 2, we give the root-finding running times for these two algorithms for some parameter sets taken from the results given in [3]. Note that there lower security levels are realized, the declaration of which in [3] are deprecated by [22]. The drawback of such a parameter choice is a large public key size.

7 Conclusion and Outlook

In this work we have evaluated four different root-finding algorithms with respect to their performance and timing side-channel security in code-based cryptosystems. We have shown that timing vulnerabilities can be present in all of these

parameters	security level	root-finding algorithm	running time / 10^5 cycles
$n = 2960$, $t = 56$	≈ 128 bit	*eval-rf*	16.22
		eval-div-rf	10.68
		bta-rf	8.89
		dcmp-rf	6.45
$n = 6624$, $t = 115$	≈ 256 bit	*eval-rf*	141.86
		eval-div-rf	71.48
		bta-rf	34.91
		dcmp-rf	25.55

Table 1: Comparison of the average root-finding algorithm performance on an x86 Intel Intel(R) Core(TM)2 Duo CPU U7600 for code parameters as suggested in [22]. All given values are the average of 10 decryptions.

parameters	root-finding algorithm	running time / 10^5 cycles
$n = 4096$, $t = 21$	*bta-rf*	1.47
	dcmp-rf	5.34
$n = 8192$, $t = 18$	*bta-rf*	1.28
	dcmp-rf	10.10

Table 2: Comparison of the root-finding algorithm performance on an x86 Intel Intel(R) Core(TM)2 Duo CPU U7600 for code parameters with large m and small t.

variants. The variant *eval-rf* and *eval-div-rf* can be ruled out as they are both not competitive in terms of computation speed (which is a well known fact). The latter, which has at least considerable performance advantages over the former, exhibits a timing side-channel vulnerability with respect to message-aimed attacks, which is beyond repair.

Considering the remaining two candidates, we find that for code parameters that minimize the public key size, *dcmp-rf* is clearly faster than the *bta-rf*, however the latter can achieve much faster results for code parameters with small t resulting large public keys. Since timing side-channel security of *bta-rf* is problematic at least with respect to message-aimed attacks, and the fact that the public key size is the much greater challenge in code-based encryption schemes than the computation times, *dcmp-rf* can be seen as the winner of this evaluation.

But as we stated in the introduction, we do not want to postulate this as the definitive answer concerning the choice of root-finding algorithms in code-based cryptosystems. If one would achieve a timing side-channel secure variant of the *bta-rf*, running time advantages could be achieved at the expense of public key size, which might be desirable in certain applications. Furthermore, the most important task certainly is the implementation of code-based cryptosystems on smart cards and related platforms. To achieve competitive performance on such resource constrained platforms, hardware support certainly would have to be present, as it is the case for RSA and elliptic curve based algorithms today. Thus, the real question is that of an optimal choice of algorithms and hardware support, achieving both good performance and side-channel security on these platforms. In this context, among other aspects, it will become relevant how easily an algorithm can be parallelized. Note that *eval-rf*, *eval-div-rf*, and *dcmp-rf* can easily be parallelized by starting independent evaluations at 2^x different equally distant offsets into $\mathbb{F}_{2^m}$ (in the Gray-Code order for *dcmp-rf*). However, the circuitry for any single instance of an *eval-rf* evaluator would be considerably simpler than for *dcmp-rf*. The parallelization of *bta-rf* seems the most complicated, it would have to be applied to the recursive structure of the algorithm. In view of these open questions we encourage future research investigating implementations with efficient hardware support on resource constrained platforms. Pure software implementations on embedded systems, however, would in the case of a widespread adoption of code-based encryption schemes also remain of great importance, as it is the case for RSA today. Thus the results of this work clearly suggest the superiority of *dcmp-rf* at least in this 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. Biswas, B., Sendrier, N.: McEliece Cryptosystem Implementation: Theory and Practice. In: PQCrypto. (2008) 47–62

4. 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
5. 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
6. 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
7. 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
8. Molter, H.G., Stöttinger, M., Shoufan, A., Strenzke, F.: A Simple Power Analysis Attack on a McEliece Cryptoprocessor. Journal of Cryptographic Engineering (2011)
9. Strenzke, F., Tews, E., Molter, H., Overbeck, R., Shoufan, A.: Side Channels in the McEliece PKC. In Buchmann, J., Ding, J., eds.: Post-Quantum Cryptography. Volume 5299 of Lecture Notes in Computer Science., Springer Berlin / Heidelberg (2008) 216–229
10. Strenzke, F.: A Timing Attack against the secret Permutation in the McEliece PKC. In: The third international Workshop on Post-Quantum Cryptography PQCRYPTO 2010, Lecture Notes in Computer Science
11. Shoufan, A., Strenzke, F., Molter, H., Stöttinger, M.: A Timing Attack against Patterson Algorithm in the McEliece PKC. In Lee, D., Hong, S., eds.: Information, Security and Cryptology - ICISC 2009. Volume 5984 of Lecture Notes in Computer Science., Springer Berlin / Heidelberg (2009) 161–175
12. Heyse, S., Moradi, A., Paar, C.: Practical power analysis attacks on software implementations of mceliece. In Sendrier, N., ed.: PQCrypto. Volume 6061 of Lecture Notes in Computer Science., Springer (2010) 108–125
13. Strenzke, F.: Message-aimed Side Channel and Fault Attacks against Public Key Cryptosystems with homomorphic Properties. Journal of cryptographic Engineering (2011) DOI: 10.1007/s13389-011-0020-0; a preliminary version appeared at COSADE 2011 .
14. Federenko, S., Trifonov, P.: Finding Roots of Polynomials over Finite Fields. IEEE Transactions on Communications **20** (2002) 1709–1711
15. 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`.
16. E. R. Berlekamp: Factoring polynomials over large finite fields. Mathematics of Computation **24(111)** (1970) 713–715
17. Goppa, V.D.: A new class of linear correcting codes. Problems of Information Transmission **6** (1970) 207–212
18. Patterson, N.: Algebraic decoding of Goppa codes. IEEE Trans. Info.Theory **21** (1975) 203–207
19. Biswas, B., Herbert, V.: Efficient Root Finding of Polynomials over Fields of Characteristic 2. WEWoRK (2009) `hal.inria.fr/hal-00626997/PDF/tbz.pdf`.

20. Overbeck, R.: An Analysis of Side Channels in the McEliece PKC (2008) available at `https://www.cosic.esat.kuleuven.be/nato_arw/slides_participants/Overbeck_slides_nato08.pdf` .
21. 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)
22. Bernstein, D.J., Lange, T., Peters, C.: Attacking and defending the McEliece cryptosystem. Post-Quantum Cryptography, LNCS **5299** (2008) 31–46
23. Bernstein, D.J.: List Decoding for binary Goppa Codes (2008)

A Further Results to the running Time Dependencies on the Root Values

Fig. 2 shows the plots of the dependencies of the running time of the root-finding on the position of a single error bit. See Sec. 5.2 for the discussion of the results for *eval-div-rf*.

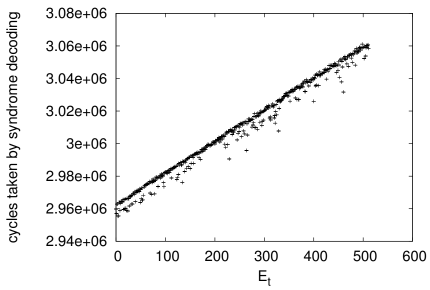
For *bta-rf*, we see some "clouding" effect in the running times, which is also apparent for timings of the whole syndrome decoding, as shown in Fig. 2(c). It is obvious, that these running times are neither constant nor random. There seems to be a tendency to build "clouds"; by which we mean that it seems that an attacker should be able to build hypotheses that if for two different values of E_1 and E_2 the timings are close to each other, then also α_{E_1} and α_{E_2} have close values in their lexicographical interpretation as numbers.

Note for instance the values of E_t below 100 in 2(c), which have consequently lower timings than $3.04 \cdot 10^6$. Though such a dramatic effect was not obvious in all support scans we conducted, it corroborates the notion of "clouding" effects in the timings for *bta-rf*. Thus we strongly suggest that the running time properties of the *bta-rf* be subject to thorough analysis before considering its use in real world implementations of code-based schemes.

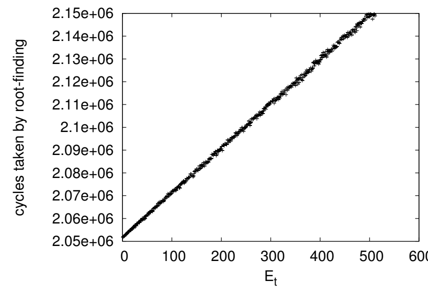
The pitfall concerning the implementation of *dcmp-rf* mentioned in Sec. 5.2 is given through the multiplication by f_3, i.e. σ_3, in Equ. (3). In our implementation, we precompute the logarithmic representation of σ_3 to subsequently use *mul_nll* for the computation $\sigma_3 Y^3$. In the unprotected variant of our implementation we cover the case $\sigma_3 = 0$ by a conditional branch that bypasses this multiplication. However, in this case, the timings clearly allow identification of a syndrome decoding where $\sigma_3 = 0$. This is shown in Fig. 3(a) and 3(b).The information gained by such an observation is, according to Equ. (1):

$$0 = \sigma_3 = \alpha_{E_1}\alpha_{E_2}\dots\alpha_{E_{w-3}} \oplus \alpha_{E_1}\alpha_{E_2}\dots\alpha_{E_{w-4}}\alpha_{E_w} \oplus \dots,$$

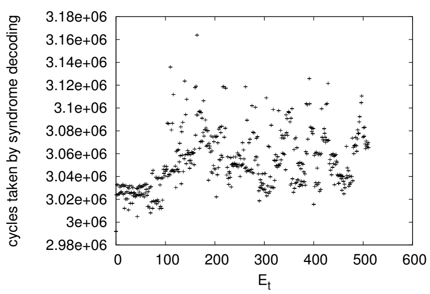
i.e. the sum of products of all possible combinations of $w-3$ different support elements associated with the respective error positions, where w is the error vector's Hamming Weight, usually $w = t$. It is certainly not trivial to exploit this information, however, in combination with other vulnerabilities it might be useful to provide a means of verifying guesses for Γ. The countermeasure to protect against this vulnerability is trivial and comes at a low computational cost, it is described in the following.



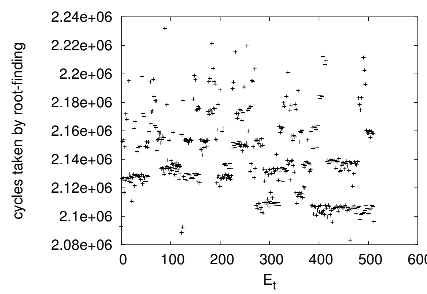
(a) Timing for the syndrome decoding with *eval-div-rf*.



(b) Timing for the root-finding with *eval-div-rf*.



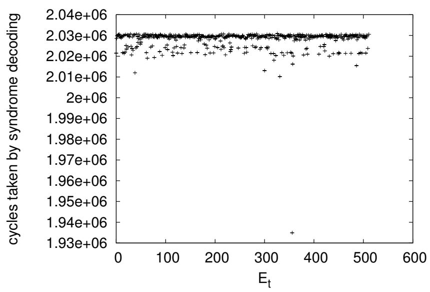
(c) Timings for syndrome decoding with *bta-rf*.



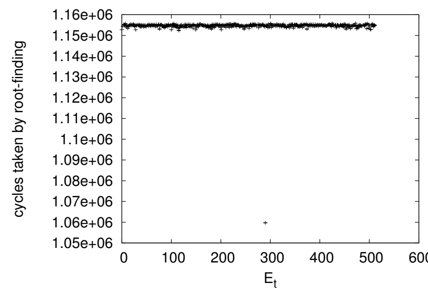
(d) Timings for root-finding with *bta-rf*.

Fig. 2: Running times of *eval-div-rf* and *bta-rf* for $n-(t-1)$ ciphertexts, where $t-1$ error positions are fixed and the $t-th$ position varies, with code parameters $n = 512$ and $t = 33$.

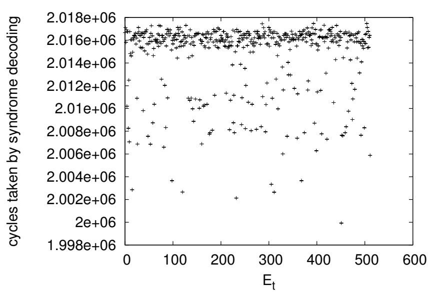
The countermeasure is realized by assigning the precomputed value of the logarithm of σ_3 a dummy value during the initialization phase of *dcmp-rf* if $\sigma_3 = 0$, and carrying out the multiplication $Y^3\sigma_3$ with both operands in logarithmic representation regardless of the value of σ_3. Afterwards, a logical AND operation is performed on the result with a mask having all bits set in case of $\sigma_3 \neq 0$ and having value 0 otherwise. The timings for support scans in these cases are given in Fig. 3(c) and 3(d). The latter, showing the timings of only the root-finding operation, shows a multi-level structure that we have not analyzed further, but must be assumed to result from the precomputation phase of *dcmp-rf*. Whether additional countermeasures are necessary to remove these timing differences remains an open question.



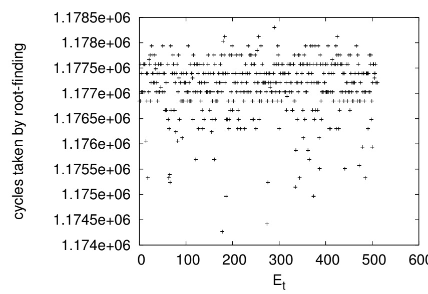
(a) Running times of the syndrome decoding without countermeasures to hide $\sigma_3 = 0$ in *dcmp-rf*. The outlier with smallest running time shows such a case.



(b) Running times of the root-finding with *dcmp-rf* without countermeasures to hide $\sigma_3 = 0$. The outlier with smallest running time shows such a case.



(c) Running times of the syndrome decoding with countermeasures to hide $\sigma_3 = 0$ in *dcmp-rf*.



(d) Running times of the root-finding with *dcmp-rf* with countermeasures to hide $\sigma_3 = 0$.

Fig. 3: Running times of the syndrome decoding with *dcmp-rf* for $n - (t - 1)$ ciphertexts, where $t - 1$ error positions are fixed and the t-th position E'_t varies.

B Further Vulnerabilities in HyMES Syndrome Decoding Implementation

While working with the HyMES implementation [15], we encountered a number of vulnerabilities, potentially enabling both timing and fault attacks. We list them here, because this is good example showing what problems can arise when the syndrome decoding is implemented without implementation security in mind (which was not in the scope of that work).

All code relevant to the syndrome decoding in HyMES is found in the file `decrypt.c`, all line numbers given in the following refer to this file. In line 270, when $\deg(\sigma(Y)) \neq t$, decryption is aborted with an error. This is only a problem if the countermeasures proposed in [11] are not implemented, in this case it allows message-aimed fault-attacks of the type explained in Sec. 5.1 (highly likely that $w > t$ leads to $\deg(\sigma(Y)) = t$, and always that $w < t$ leads to $\deg(\sigma(Y)) = w$).

In line 276, if the root-finding did not return t roots decryption is also aborted with an error. This is clearly allowing message-aimed fault attacks with the two-bit-flip attack described earlier in Sec. 5.1, such a check must not be present in a secure implementation.

In line 285, a Quick Sort algorithm is applied to the set of roots to sort the array. Though we did not seek for attacks against this algorithm, it is certainly clear that the running time of Quick Sort depends on the number of roots, and in general also on their positions. As far as understood by the author, the sorting is needed in HyMES for the constant weight word encoding[3]. We want to point out that a real world implementation must ensure that such sorting is done without giving information about the roots positions. The number of roots, however, can be, as a countermeasure, artificially increased to t before the algorithm is applied, as a deviating number of roots indicates an irregular ciphertext anyway.

[3] The constant weight word encoding (CWE) is needed in the implementation [15] to encode additional information in the error vector; in this aspect their scheme deviates from the original McEliece scheme [1]. Note, however, that in most CCA2 conversions proposed for the McEliece scheme, CWE is used [21]. Furthermore, CWE is needed in the Niederreiter scheme [2].