

# Toward Secure Implementation of McEliece Decryption

Mariya Georgieva<sup>1</sup> and Frédéric de Portzamparc<sup>1,2,3,4</sup>

Gemalto, 6 rue de la Verrerie 92190, Meudon, France<sup>1</sup>  
INRIA, Paris-Rocquencourt Center<sup>2</sup>,  
UPMC Univ Paris 06, UMR 7606, LIP6, F-75005, Paris, France<sup>3</sup>  
CNRS, UMR 7606, LIP6, F-75005, Paris, France<sup>4</sup>  
Mariya.Georgieva@gemalto.com, frederic.urvoy-de-  
portzamparc@polytechnique.org

**Abstract.** We analyse the security regarding timing attacks of implementations of the decryption in McEliece PKC with binary Goppa codes. First, we review and extend the existing attacks, both on the messages and on the keys. We show that, until now, no satisfactory countermeasure could erase all the timing leakages in the Extended Euclidean Algorithm (EEA) step. Then, we describe a version of the EEA never used for McEliece so far. It uses a constant number of operations for given public parameters. In particular, the operation flow does not depend on the input of the decryption, and thus closes all previous timing attacks. We end up with what should become a central tool toward a secure implementation of McEliece decryption.

## 1 Introduction

*Context of this work.* Code-based cryptography relies on the hardness of *decoding*, that is recovering  $\mathbf{m}$  and  $\mathbf{e}$  when given only  $\mathbf{c} = \mathbf{m}\mathbf{G} + \mathbf{e}$  and  $\mathbf{G}$  (for  $\mathbf{m} \in \mathbb{F}_q^k$ ,  $\mathbf{G} \in \mathbb{F}_q^{k \times n}$  and  $\mathbf{e} \in \mathbb{F}_q^n$ ). Indeed, decoding was proved to be exponentially difficult in the Hamming weight of  $\mathbf{e}$  when no structure is known on  $\mathbf{G}$  in [2]. However, the error weight is critical for security for another reason: contrary to the public parameters of the code which are fixed at set by an external entity, the error may vary at each encryption, and may even be chosen by any public user (in some situations).

Therefore, a problem arises in most of the implementations of McEliece proposed (*e.g.* in [5,6,14,13]) because the operation flow of the decryption is strongly influenced by the error vector, but no information is known about the error vector when starting decryption. From an attacker's point of view, this is a favorable situation. It means that the observed or manipulated device may leak information before any detection of the attack. These security aspects were addressed by various authors, who explained that a device implementing an unprotected decryption is prone to attacks on the messages [12,1] and on the key [15,16]. Although countermeasures were proposed against some of the leakages, the situation is still unsatisfactory, as it is noticed in the conclusion of [16]. In particular,

to the best of our knowledge, no decryption algorithm requiring a number of steps independent of the error weight was described. The work of Bernstein et al. in [4] claims to achieve this goal, but some steps of the decryption (including the extended Euclidean algorithm (EEA) in the decoding) are skipped in the description, and no implementation is publicly available.

*Our contributions.* First, we gather the different weaknesses revealed in [12,1,15,16]. In particular, we detail the attacks of Strenzke and show that they can be extended to bypass the countermeasure of [15]. All these attacks targeted only one of the two known methods for decoding a binary Goppa code (namely Patterson algorithm). Along with our presentation, we evaluate how/if those threats transpose to the other decoding method (*ie* the alternant decoder). Our central contribution consists in describing an EEA tailored for the alternant decoder which has a flow of operations independent of the error vectors (Alg. 10). It was inspired by a work of Berlekamp [3]. We explain step-by-step the construction of the algorithm, and provide completeness proofs (which we could not find in the literature).

## 2 McEliece Public-Key Encryption

We recall in Alg. 1 the encryption and decryption in McEliece PKC instantiated with a binary Goppa code, that is  $q = 2$ . The public key is  $\mathbf{G}$  a  $k \times n$  matrix over  $\mathbb{F}_q$  whose rows generate a Goppa code described by the secret elements  $\mathbf{x} \in \mathbb{F}_{q^m}^n$  and  $g(z) \in \mathbb{F}_{q^m}[z]$  of degree  $t$ .

---

### Algorithm 1 McEliece Cryptosystem

---

PARAMETERS : Field size  $q$ , code length  $n$  and dimension  $k$ , parameters  $m, t$  such that  $n - mt \leq 0$ . Plaintext space:  $\mathbb{F}_q^k$ . Ciphertext space:  $\mathbb{F}_q^n$ .

KEYGEN : Pick a support  $\mathbf{x} \in \mathbb{F}_{q^m}^n$ , a polynomial  $g \in \mathbb{F}_{q^m}[x]$  of degree  $t$ ,  $\mathbf{G}$  a generator matrix of  $\mathcal{G}(\mathbf{x}, g)$ . PUBLIC KEY :  $\mathbf{G}_{pub} = \mathbf{SGP}$ ,  $t$  the correction capacity of the code  $\mathcal{G}(\mathbf{x}, g)$ .

PRIVATE KEY :  $T_t$  a  $t$ -decoder for  $\mathcal{G}(\mathbf{x}, g)$ ,  $\mathbf{S}$  a random full rank  $(n - k) \times (n - k)$  matrix,  $\mathbf{P}$  a random  $n \times n$  permutation matrix.

ENCRYPT :

- 1: Input  $\mathbf{m} \in \mathbb{F}_q^k$ .
- 2: Generate random  $\mathbf{e} \in \mathbb{F}_q^n$  with  $w_H(\mathbf{e}) = t$ .
- 3: Output  $\mathbf{c} = \mathbf{mG}_{pub} + \mathbf{e}$ .

DECRYPT :

- 1: Input  $\mathbf{c} \in \mathbb{F}_q^n$ .
  - 2: Compute  $\bar{\mathbf{m}} = T_t(\mathbf{cP}^{-1})$ .
  - 3: If decoding succeeds, output  $\mathbf{S}^{-1}\bar{\mathbf{m}}$ , else output  $\perp$ .
- 

Now we detail the possible decoders  $T_t$  for a binary Goppa code. Suppose one wants to decode an encoded message  $\mathbf{m} \in \mathbb{F}_q^k$  with errors  $\mathbf{e}$ :  $\mathbf{c} = \mathbf{mG} + \mathbf{e}$ , where the Hamming weight of  $\mathbf{e}$  (denoted in the rest of this article by  $w_H(\mathbf{e})$ ) satisfies  $w_H(\mathbf{e}) \leq t$ . We write  $\mathbf{e} = (\dots, 0, e_{i_1}, 0, \dots, 0, e_{i_w}, 0, \dots)$ . Two methods

exist for decoding. The one uses the fact that Goppa codes belong to the larger class of alternant codes, so we refer to this method as the Alternant Decoder. The other one, called Patterson Algorithm, is specific to binary Goppa codes. For both, the main steps are :

1. Compute the **polynomial syndrome**  $S(z)$ , a univariate polynomial deduced from  $\mathbf{c}$ , but depending only on  $\mathbf{e}$ .
2. Solve the **key equation**, which is an equation whose unknowns are univariate polynomials, using an EEA. The solutions give access to the **error locator polynomial**  $\sigma_{\mathbf{e}}(z)$ , whose roots are related to the support elements  $x_{i_j}$  in the error positions  $i_j$ . It also yields the **error evaluator polynomial**  $\omega_{\mathbf{e}}(z)$  (helpful to find the values of the errors).
3. Find the roots of  $\sigma_{\mathbf{e}}(z)$ . Here  $\mathbf{e} \in \mathbb{F}_2^n$ , so  $e_{i_j} \neq 0$  implies that  $e_{i_j} = 1$ .

The polynomial syndromes, key equations and their resolutions are specific to each method. We briefly sum them up.

**Polynomial syndrome**

$$S_{Alt,\mathbf{e}}(z) = \sum_{\ell=0}^{2t-1} \left( \sum_{i=0}^{n-1} c_i g(x_i)^{-2} x_i^\ell \right) z^\ell.$$

**Polynomials to be recovered**

$$\sigma_{inv,\mathbf{e}}(z) = \prod_{j=1}^w (1 - zx_{i_j}),$$

$$\omega_{inv,\mathbf{e}}(z) = \sum_{j=1}^w e_{i_j} g(x_{i_j})^{-1} \prod_{\substack{s=1 \\ s \neq j}}^w (1 - zx_{i_s}).$$

**Key equation**

$(\sigma_{inv,\mathbf{e}}, \omega_{inv,\mathbf{e}})$  unique solution of

$$\begin{cases} \omega_{inv,\mathbf{e}}(z) = \sigma_{inv}(z) S_{Alt,\mathbf{e}}(z) \pmod{z^{2t}}, \\ \deg(\sigma_{inv}) \leq \lfloor t/2 \rfloor, \deg(\omega_{inv}) < \lfloor t/2 \rfloor. \end{cases}$$

**Resolution**

EEA( $z^{2t}, S_{Alt,\mathbf{e}}, t$ ) outputs  
 $(\mu\sigma_{inv}, (-1)^N \mu\omega_{inv}), \mu \in \mathbb{F}_{q^m}^*, N \geq 0$ .

**Error recovery**

$\sigma_{\mathbf{e}}(z) = z^w \sigma_{inv}(1/z)$ .  
 Find the roots of  $\sigma_{\mathbf{e}}$ .

Fig. 1: Alternant Decoder

**Polynomial syndrome**

$$S_{Gop,\mathbf{e}}(z) = \sum_{i=0}^{n-1} \frac{c_i}{z-x_i} \pmod{g(z)}.$$

**Polynomials to be recovered**

$$\sigma_{\mathbf{e}}(z) = \prod_{j=1}^w (z - x_{i_j}),$$

$$\omega_{\mathbf{e}}(z) = \sum_{j=1}^w \prod_{\substack{s=1 \\ s \neq j}}^w (z - x_{i_s}).$$

**Key equation**

$(\sigma_1, \sigma_2)$  unique solution of

$$\begin{cases} \tau(z)\sigma_2(z) = \sigma_1(z) \pmod{g(z)}, \\ \deg(\sigma_1) \leq \lfloor t/2 \rfloor, \deg(\sigma_2) < \lfloor t/2 \rfloor, \\ \tau(z) = \sqrt{S_{Gop,\mathbf{e}}(z)^{-1} + z} \pmod{g(z)}. \end{cases}$$

**Resolution**

1. EEA( $g(z), S_{Gop,\mathbf{e}}(z), 0$ )  
 outputs  $(S_{Gop,\mathbf{e}}^{-1} \pmod{g}, 0)$   
 2. EEA( $g(z), \tau, \lfloor t/2 \rfloor$ )  
 outputs  $(\sigma_1, \sigma_2)$ .

**Error recovery**

$\sigma_{\mathbf{e}}(z) = \sigma_1(z)^2 + z\sigma_2(z)^2$ ,  
 $\omega_{\mathbf{e}} = \sigma_{\mathbf{e}} S_{\mathbf{e}} \pmod{g}$ .  
 Find the roots of  $\sigma_{\mathbf{e}}$ .

Fig. 2: Patterson Algorithm

Completeness proofs are classic coding theory literature. For details, see for instance [7][Ch. 12 §9] for the Alternant Decoder and [18,9] for Patterson Algorithm.

The extended Euclidean Algorithm which is used in all the available implementations (see [5,6,14,13]) consists in successive Euclidean divisions as in Alg. 2. Its complexity is in  $O(\deg(a)^2)$  field multiplications. It is to be noticed that asymptotically better algorithms exist, generally referred to as Fast EEA or HGCD (for Half-GCD), with complexity  $O(\deg(a) \log \deg(a))$ . The reason not to use them here is that constants are hidden in the  $O$  (see for details [19]). Consequently, for the degrees at stake in McEliece decryption ( $\leq 200$ ), those are not more efficient than Alg. 2.

---

**Algorithm 2** Extended Euclidean Algorithm (EEA)

---

**Input:**  $a(z), b(z), \deg(a) \geq \deg(b), d_{fin}$

**Output:**  $u(z), r(z)$  with  $b(z)u(z) = r(z) \pmod{a(z)}$  and  $\deg(r) \leq d$

```

1:  $r_{-1}(z) \leftarrow a(z), r_0(z) \leftarrow b(z), u_{-1}(z) \leftarrow 1, u_0(z) \leftarrow 0,$ 
2:  $i \leftarrow 0$ 
3: while  $\deg(r_i(z)) > d_{fin}$  do
4:    $i \leftarrow i + 1$ 
5:    $q_i \leftarrow r_{i-2}(z)/r_{i-1}(z)$  (quotient of the Euclidean division of  $r_{i-2}(z)$  by  $r_{i-1}(z)$ )
6:    $r_i \leftarrow r_{i-2}(z) - q_i(z)r_{i-1}(z)$  (rest of the Euclidean division of  $r_{i-2}(z)$  by  $r_{i-1}(z)$ )

7:    $u_i \leftarrow u_{i-2}(z) - q_i(z)u_{i-1}(z)$ 
8: end while
9:  $N \leftarrow i$ 
10: return  $u_N(z), r_N(z)$ 

```

---

For the EEA executions solving the key equations (EEA in Alternant decoder, second EEA in Patterson algorithm), the paragraph §5 of [18] gives refined complexities of respectively  $7.5tw_H(\mathbf{e})$  and  $3.5tw_H(\mathbf{e})$ . For the syndrome polynomial inversion in the Patterson decoding (first EEA), a quick analysis shows that, the number of field multiplications is bounded by  $2t^2$ . We obtain with an error vector of weight  $t$  a cost in field multiplications of  $7.5t^2$  for the Alternant decoder and  $5.5t^2$  for Patterson algorithm. This is why Patterson algorithm is generally preferred.

### 3 Decryption oracle attacks

The attacker has a ciphertext  $\mathbf{c}$ . He also has access to a decryption oracle. He can request decryption of any message  $\mathbf{c}' \neq \mathbf{c}$  and observe the execution of the decryption.

### 3.1 Plaintext-recovery attacks

In [12,1,15], the authors described attacks using the same framework. They exploit a decryption oracle to recover the plaintext from an encrypted message  $\mathbf{c}$ . Those focus only on Patterson method for decoding (Fig. 2). Here, we apply the different attacks proposed to both decoding algorithms in order to determine which one is the most resistant.

*Generic attack scenario.* The common framework is summed up in Alg. 3.

---

**Algorithm 3** Framework for message-recovery attacks on a decryption device.

---

INPUT: A valid ciphertext  $\mathbf{c} = \mathbf{m}\mathbf{G}_{pub} + \mathbf{e}$ , a decryption device  $\mathcal{D}$ .

OUTPUT: The error vector  $\mathbf{e}$  and plaintext  $\mathbf{m}$ .

---

- 1: **for**  $i = 0, \dots, n - 1$  **do**
  - 2:     Modify  $\mathbf{c}$  into  $\mathbf{c}^{*i} = \mathbf{c} + (0, \dots, 0, \underbrace{1}_{i\text{-th bit}}, 0, \dots)$ .
  - 3:     Request decryption  $\mathcal{D}(\mathbf{c}^{*i})$ .
  - 4:     Deduce by timing analysis or power consumption of  $\mathcal{D}$  whether  $e_i = 0$  or  $e_i = 1$ .
  - 5: **end for**
  - 6: Solve the linear system with unknown  $\mathbf{m}$ :  $\mathbf{m}\mathbf{G}_{pub} = \mathbf{c} + \mathbf{e}$ .
  - 7: **return** Plaintext  $\mathbf{m}$ .
- 

Alg. 3 describes an attack against a McEliece encryption scheme. The same framework is applicable against Niederreiter encryption when a public encryption key  $\mathbf{H}_{pub}$  is known. It suffices to replace Step 2 by

- 2: Modify  $\mathbf{c}$  into  $\mathbf{c}^{*i} = \mathbf{c} + \mathbf{H}_{pub}^{(i)}$ , where  $\mathbf{H}_{pub}^{(i)}$  denotes the  $i$ -th row of  $\mathbf{H}_{pub}$ .

In CCA2 conversions, a condition is imposed to the error. Ciphertext manipulation is detected thanks to a test on the validity of the error and the algorithm outputs a failure: the attacker only checking the output does not recover information except that the ciphertext was not valid.

Historically, the first method was explained in [17] and refined in [1]. It dates back to 2008 and focuses on the determination of the roots of the error locator polynomial  $\sigma_{\mathbf{e}}(z)$  (Figure 2). The authors of [1] and [17] propose convincing countermeasures against this weakness (see [1][Alg. 4]). Indeed, when starting the root-finding step, errors of weight lower than  $t$  can be detected thanks to the low degree of the error locator polynomial. This makes this step rather simple to protect. Unfortunately, this is not sufficient to discard attacks following the framework of Alg. 3. The reason is that, as first noticed in [12], the EEA determining  $\sigma_{\mathbf{e}}$  also has an execution time depending on the error weight. Now we will focus on the EEA step which is still problematic.

**Exploitation of EEA leakages** The authors of [12] focused on Patterson algorithm (Fig. 2). In the second EEA, implemented as in Alg. 2, the number of iterations in the **while** loop depends on the weight of the error. To do so, they show that the output  $\sigma_1(z), \sigma_2(z)$  satisfy the relations:

$$\deg(\sigma_2(z)) = \sum_{i=1}^N \deg(q_i) \quad \deg(\sigma_1(z)^2 + z\sigma_2(z)^2) = w_H(\mathbf{e})$$

Thanks to relations (1) and (2), they deduce that, assuming that all the Euclidean divisions in the second EEA have a quotient of degree one (which happens with probability  $(1-2^{-m})^N$ ), the number  $N$  of iterations in the **while** loop varies as in Table 1. These variations of  $N$  can be detected either by measuring time execution or counting the number of patterns on a power consumption trace. Thus, they provide a successful tool to perform Alg. 3 and recover a plaintext.

*Countermeasure.* To protect against the previous attack, the authors of [12] propose to check the degrees of the polynomials processed by the EEA during its execution instead of checking only the output. We recall that, in Patterson's decoding, the error locator polynomial is split into even and odd part:  $\sigma_e(z) = \sigma_1(z)^2 + z\sigma_2(z)^2$ . [12] gives a precise description of the degrees to be expected for  $\sigma_1(z)$  and  $\sigma_2(z)$  according to the weight of the error, recalled in Table 1.

	$t = 2t'$			$t = 2t' + 1$		
	$N$	$\deg(\sigma_1)$	$\deg(\sigma_2)$	$N$	$\deg(\sigma_1)$	$\deg(\sigma_2)$
$w_H(\mathbf{e}) = t + 1$	$\leq t' - 1$	$= t'$	$\leq t' - 1$	$\leq t'$	$\leq t'$	$= t'$
$w_H(\mathbf{e}) = t$	$\leq t' - 1$	$= t'$	$\leq t' - 1$	$= t'$	$\leq t' - 1$	$= t'$
$w_H(\mathbf{e}) = t - 1$	$= t' - 1$	$\leq t' - 1$	$= t' - 1$	$\leq t' - 1$	$= t' - 1$	$\leq t' - 1$
$w_H(\mathbf{e}) = t - 2$	$\leq t' - 2$	$= t' - 1$	$\leq t' - 2$	$= t' - 1$	$\leq t' - 2$	$\leq t' - 1$

Table 1: Degrees of the output polynomials of  $\text{EEA}(g(z), \tau(z), \lfloor t/2 \rfloor)$ .

They deduce Alg. 4, a modified EEA which detects the ciphertext manipulation and completes the computation to take same execution time both on the ciphertext  $\mathbf{c}$  and on the twisted  $\mathbf{c}^{*i}$ . With this version, the authors claim to obtain a decryption time taking a constant number of clock cycles. This would discard a timing attack. However, no power consumption trace is analyzed. It remains unclear whether the extra manipulation in Step 9 of Alg. 4, whose execution implies that  $e_i = 1$ , is undetectable. In particular, each **while** iteration will not have the same execution time. Fakingly executing those steps at each iteration would have a serious impact on the performances.

*EEA leakages in the Alternant Decoder.* We adapt the framework of Alg. 3 to the alternant decoder. The alternant decoder, as Patterson one, resorts to an EEA

---

**Algorithm 4** Protected EEA for Patterson decoding

---

Beginning as is Alg. 2.

7:  $u_i \leftarrow u_{i-2}(z) - q_i(z)u_{i-1}(z)$

CASE  $t = 2t'$ .

8: **if**  $\deg(r_i) < t'$  **then**

9: Manipulate  $r_i$  so that  $\deg(r_i) = \deg(r_{i-1}) - 1$ .

10: **end if**

CASE	$t$	$=$	$2t'$	$+$
1				
8: <b>if</b>	$\deg(r_i) \leq t'$	<b>and</b>	$\deg(v_i) < t'$	<b>then</b>
9:	Manipulate $r_i$ so that $\deg(r_i) = \deg(r_{i-1}) - 1$ .			
10: <b>end if</b>				

End as is Alg. 2.

---

(Fig. 1) prone to leak information when the error weight varies. The equivalent of Relations (1) and (2) in this context is

$$\deg(\sigma_{inv}(z)) = \sum_{i=1}^N \deg(q_i) = \begin{cases} w_H(\mathbf{e}) & \text{if } 0 \notin \mathbf{x} \\ w_H(\mathbf{e}) - 1 & \text{if } 0 \in \mathbf{x}. \end{cases} \quad (3)$$

Looking at the degree of the output polynomial of the EEA in Alg. 1, we have the following link with the weight of the error vector ( $\alpha$  denotes the position of the support such that  $x_\alpha = 0$ ): The error locator polynomial is deduced

	deg( $\sigma_{inv}$ ) if $e_\alpha = 0$	deg( $\sigma_{inv}$ ) if $e_\alpha = 1$
$w_H(\mathbf{e}) = t$	$t$	$t - 1$
$w_H(\mathbf{e}) = t + 1(e_i = 0)$	$t$	$t - 1$
$w_H(\mathbf{e}) = t - 1(e_i = 1)$	$t - 1$	$t - 2$

Table 2: Degrees of the output polynomials of EEA( $z^t, S_e(z), \lfloor t/2 \rfloor$ ).

from  $\sigma_{inv}(z)$  when the weight of the error vector is known. Indeed, when 0 is a possible error position, one cannot determine with certainty the degree of  $\sigma_e(z)$  from the degree of its reciprocal polynomial  $\sigma_{inv}(z)$ . The reason is that for any polynomial  $p(z) \in \mathbb{F}_q[z]$ ,  $p(z)$  and  $zp(z)$  have same reciprocal polynomial. After computing a polynomial  $\sigma_{inv}(z)$  of degree  $d$ , if 0 belongs to the support, there are two possibilities, either the index  $\alpha$  such that  $x_\alpha = 0$  is not an error position,  $\sigma_e$  is not divisible by  $z$ , then  $\deg(\sigma_e) = \deg(\sigma_{inv})$  and  $\sigma_e(z)$  is equal to  $z^{\deg(\sigma_{inv})}\sigma_{inv}(z^{-1})$ , or  $\alpha$  is an error position, and  $\sigma_e(z) = z^{\deg(\sigma_{inv})+1}\sigma_{inv}(z^{-1})$ .

Therefore, in this case, looking at the degree of  $\sigma_{inv}$  does not distinguish manipulated ciphertexts from correct ones, and the EEA cannot be correctly protected by this method.

*Countermeasure.* Building up on the countermeasure for Patterson decoding described in [12], we propose the following adaptation (Alg. 5) to the alternant

decoder. It always detects ciphertext manipulation provided that 0 is not an element of the support, and somehow restores a usual behavior of the EEA (that is, that of a valid ciphertext). The final output will not be the correct plaintext, but this is not a problem as long as the attacker cannot extract information from this result. However, we note that this protection has the same drawbacks as its Patterson equivalent: each **while** execution does not have same execution time.

---

**Algorithm 5** Protected EEA for Alternant decoder

---

Beginning as is Alg. 2.

7:  $v_i \leftarrow v_{i-2}(z) - q_i(z)v_{i-1}(z)$

8: **if**  $\deg(r_i) < t$  **then**

9:     Manipulate  $r_i$  so that  $\deg(r_i) = \deg(r_{i-1}) - 1$

10: **end if**

End as is Alg. 2.

---

### 3.2 Secret decryption key recovery attacks

We address a different kind of physical attack initiated by Strenzke in [15,16] against McEliece encryption using Patterson decoding. It aims at recovering the secret key.

*Generic attack scenario.* The attack scenario is the following. The attacker has access to a decryption device  $\mathcal{D}$  on which he can perform physical measurements. He also knows a public encryption key, so that he can generate codewords with errors of his choice. By observing the decryption phase (more precisely, the EEA execution), Strenzke shows that one can deduce information on the support elements corresponding to the error positions. Roughly, the reason is that when a polynomial condition on those elements is satisfied, the number of iterations of the **while** loop in Alg. 2 is reduced compared to the average number of iterations necessary to perform the EEA for error vectors of same weight. The attack consists in scanning a lot of error positions and collect sufficiently many polynomial relations so that the algebraic system obtained can be solved.

We sum up in Alg. 6 the global attack framework arising from [16].

The polynomials  $P_w$  will be elementary symmetric polynomials of the form, for an error  $\mathbf{e} = (0, \dots, e_{i_1}, \dots, e_{i_w}, \dots, 0)$  with  $w_H(\mathbf{e}) = w$  and  $j \geq 0$ :

$$\omega_j(\mathbf{e}) = \sum_{1 \leq \ell_1 < \dots < \ell_j \leq w} x_{i_{\ell_1}} \dots x_{i_{\ell_j}}.$$

That is,  $\omega_j(\mathbf{e})$  is the evaluation of the  $j^{\text{th}}$  elementary symmetric polynomial in  $w$  variables in  $(x_{i_1}, \dots, x_{i_w})$ .



---

**Algorithm 6** Framework for key-recovery attacks on a decryption device.

---

INPUT: A decryption device  $\mathcal{D}$ , public encryption key  $\mathbf{G}_{pub}$ .

OUTPUT: The secret support  $\mathbf{x}$ .

- 1: **for**  $w$  well-chosen error weights **do**
  - 2:   **for**  $(i_1, \dots, i_w)$  subset of  $\{0, \dots, n-1\}$  **do**
  - 3:     Pick a low-weight error vector  $\mathbf{e} = (0, \dots, e_{i_1}, \dots, e_{i_w}, \dots, 0)$  with  $w_H(\mathbf{e}) = w$ .
  
  - 4:     Request decryption  $\mathcal{D}(\mathbf{e})$ .
  - 5:     Perform timing or power consumption analysis of  $\mathcal{D}(\mathbf{e})$ .
  - 6:     **if** EEA execution is faster than average (more precise conditions in this Section) **then**
  - 7:       Deduce a polynomial condition on  $x_{i_1}, \dots, x_{i_w}$  ( $P_w$  is a polynomial depending only on  $w$ ):
 
$$P_w(x_{i_1}, \dots, x_{i_w}) = 0 \tag{4}$$
  - 8:     **end if**
  - 9:   **end for**
  - 10: **end for**
  - 11: Solve the non-linear system of all the collected equations (4).
  - 12: **return** Secret support  $\mathbf{x} = (x_0, \dots, x_{n-1})$ .
- 

*State-of-the-art.* More precisely, Strenzke uses errors of weights  $w = 1, w = 4$  and  $w = 6$ . For  $w = 6$ , errors such that Eq. (4) is satisfied are harder to find than for  $w = 4$ . For this reason, his strategy consists in collecting as many Eq. (4) with  $w = 1$  and  $w = 4$  as possible. He obtains a linear system of rank  $n - m$  (in some cases  $n - m - 1$ ) in the  $n$  elements of the support. Then, he selects subsets of errors of weight  $w = 6$  to look for Eq. (4). These subsets are chosen so as help the polynomial system solving. According to Strenzke, for an encryption scheme with parameters  $m = 10, n = 2^m, t = 40$ , it takes about 15,000,000 decryption queries to collect enough equations and 28 hours to solve the algebraic system. Eventually, the full secret support  $\mathbf{x}$  is recovered by the attacker, and then the Goppa polynomial is easy to find. Indeed, it is well explained in [8][p. 125] how, given the public key, it is possible to recover one from the other in polynomial time.

**First example of leakage exploitable by Framework 6.** The first attack resorting to the method of Alg. 6 was proposed by Strenzke in [15]. It focuses on the second EEA of Patterson alg. with errors of weight  $w = 4$ . In this case,  $S_{\mathbf{e}}(z) = \sum_{j=1}^4 \frac{1}{z-x_{i_j}} = \frac{\omega_{\mathbf{e}}(z)}{\sigma_{\mathbf{e}}(z)}$ , and

$$\omega_{\mathbf{e}}(z) = \underbrace{(x_{i_1} + x_{i_2} + x_{i_3} + x_{i_4})}_{\omega_1(\mathbf{e})} z^2 + \underbrace{x_{i_1}x_{i_2}x_{i_3} + x_{i_1}x_{i_2}x_{i_4} + x_{i_1}x_{i_3}x_{i_4} + x_{i_2}x_{i_3}x_{i_4}}_{\omega_3(\mathbf{e})}.$$

If  $\omega_1(\mathbf{e}) = 0$ , then  $S_{\mathbf{e}}(z) = \frac{\omega_3(\mathbf{e})}{\sigma_{\mathbf{e}}(z)}$ , and  $S_{\mathbf{e}}^{-1} \bmod g = \omega_3(\mathbf{e})^{-1}\sigma_{\mathbf{e}}(z)$  therefore  $\tau(z) = \sqrt{S_{\mathbf{e}}^{-1}(z) + z} \bmod g(z) = \sqrt{\omega_3(\mathbf{e})^{-1}\sigma_{\mathbf{e}}(z) + z}$  and  $\tau(z)$  has degree lower than  $\lfloor t/2 \rfloor$  (for  $w = 4$  we have  $\deg(\tau(z)) = 2$ ). As a consequence, the **while** test in  $\text{EEA}(g(z), \tau(z), \lfloor t/2 \rfloor)$  is never fulfilled and the number of iterations  $N$  is equal to 0. When  $\omega_1(\mathbf{e}) \neq 0$ ,  $\deg(\tau(z)) > \lfloor t/2 \rfloor$  with overwhelming probability ( $\tau(z)$  is a reduction modulo a polynomial of degree  $t$ ), so that  $N > 0$ . This allows to collect many equations of the form

$$x_{i_1} + x_{i_2} + x_{i_3} + x_{i_4} = 0.$$

As Strenzke explains, the final system's rank never exceeds  $n - m$ . So it is not sufficient in practice to recover the private key and one has to assume that some other parts of private elements are required. Still, he proposes a counter-measure to avoid this information leakage.

*Countermeasure to protect Second EEA by Strenzke.* Strenzke proposes to detect the polynomials  $\tau(z)$  leading to this leakage by checking if  $\deg(\tau(z)) < \lfloor t/2 \rfloor$ . This can be done just after the determination of  $\tau(z)$ . If so, manipulate  $\tau(z)$  so that it has degree  $t - 1$ .

---

**Algorithm 7** Patterson decoding with Strenzke's countermeasure against weight 4 errors leakages

---

INPUT:  $n - k$ -bit syndrome  $\mathbf{s} = \mathbf{H}\mathbf{e}^T$  with  $w_H(\mathbf{e}) \leq \lfloor t \rfloor$ , private key  $(\mathbf{x}, g(z))$

OUTPUT: The error vector  $\mathbf{e}$ .

- 1: POLYNOMIAL SYNDROME DETERMINATION: Idem Fig. 2.  
COMPUTATION OF  $\tau(z)$ :
  - 2: Find  $f(z)$  such that  $f(z)S_{\mathbf{e}}(z) = 1 \bmod g(z)$  by  $\cdot$ ,  $f(z) = \text{EEA}(g(z), S_{\mathbf{e}}(z), 0)$
  - 3: Set  $\tau(z) = \sqrt{f(z) + z}$ .
  - 4: **if**  $\deg(\tau(z)) < \lfloor t/2 \rfloor$  **then**
  - 5:   Manipulate  $\tau(z)$  so that  $\deg(\tau(z)) = t - 1$ .
  - 6: **end if**
  - 7: ERROR LOCATOR POLYNOMIAL DETERMINATION & ERROR VECTOR DETERMINATION  
Idem Fig. 2.
- 

This countermeasure avoids leaking information only in the second EEA, only when decoding errors of weight 4. Exploitable leakages remain, as shown in the next paragraph.

**Leakage in the first EEA of Patterson Decoding.** In order to complete the attack initiated in [15], Strenzke proposed in [16] to apply Alg. 6 by focusing on time leakages in both EEA's of Patterson decoding. In [16][Corollary 1], he gives the number of iterations of the **while** loop in the first EEA. We recall it

here, and complete it with the analogous result for the second EEA (which we could not find in [16]).

**Lemma 1.** *Let  $\mathcal{C} = \mathcal{G}(\mathbf{x}, g(z))$  be a binary Goppa code and  $S_{\mathbf{e}}(z)$  the polynomial syndrome associated to an error  $\mathbf{e}$  with  $w_H(\mathbf{e}) \leq \deg(g)/2 - 1$ . Write  $S_{\mathbf{e}}(z) = \frac{\omega_{\mathbf{e}}(z)}{\sigma_{\mathbf{e}}(z)} \pmod{g(z)}$ . Let  $N_I$  and  $N_K$  be the number of iterations of the **while** loop respectively in  $\text{EEA}(g(z), S_{\mathbf{e}}(z), 0)$  and  $\text{EEA}(g(z), \tau(z), \lfloor t/2 \rfloor)$ . Then*

$$N_I \leq \deg(\omega_{\mathbf{e}}(z)) + \deg(\sigma_{\mathbf{e}}(z)) \text{ and } N_K \leq \deg(\omega_{\mathbf{e}}(z))/2. \quad (5)$$

*Proof.* The result on  $N_I$  is proved in [16][Corollary 1]. Regarding  $N_K$ , observe that  $v_0$  has degree 0 and  $v_{N_K} = \sigma_2(z)$  has degree  $\deg(\omega_{\mathbf{e}})/2$  (since by derivating the relation  $\sigma = \sigma_1^2 + z\sigma_2^2$  we obtain  $\omega_{\mathbf{e}} = \sigma_2^2$ ). As the degrees are raised at least by one at each iteration, we obtain  $N_K \leq \deg(\omega_{\mathbf{e}})/2$ .

Let's apply this small weight error vectors. Note that, for any error  $\mathbf{e}$ ,  $\deg(\sigma_{\mathbf{e}}) = w_H(\mathbf{e})$ .

For an error weight  $w = 4$ . Pick  $\mathbf{e} = (0, \dots, e_{i_1}, \dots, e_{i_4}, \dots, 0)$ . We know that  $\omega_{\mathbf{e}}(z) = \omega_1(\mathbf{e})z^2 + \omega_3(\mathbf{e})$ . Lemma 1 gives the number  $N_I$  of iterations in  $\text{EEA}(g(z), S_{\mathbf{e}}(z), 0)$  satisfies

$$\begin{aligned} x_{i_1} + x_{i_2} + x_{i_3} + x_{i_4} \neq 0 &\implies N_I = 6, \\ x_{i_1} + x_{i_2} + x_{i_3} + x_{i_4} = 0 &\implies N_I = 4. \end{aligned}$$

Therefore, even if the second EEA has been protected with Strenzke's counter-measure, errors of weight  $w = 4$  leak the same information in the first EEA. This does not give enough equations. To complete the algebraic system, the idea is to use error weights  $w = 6$ .

For an error weight  $w = 6$ . For  $\mathbf{e} = (0, \dots, e_{i_1}, \dots, e_{i_6}, \dots, 0)$ , we develop  $S_{Gop, \mathbf{e}}(z)$  :

$$S_{Gop, \mathbf{e}}(z) = \frac{\omega_1(\mathbf{e})z^4 + \omega_3(\mathbf{e})z^2 + \omega_5(\mathbf{e})}{\sigma_{\mathbf{e}}(z)}.$$

Strenzke's purpose is to detect for which  $\mathbf{e}$  is holds that  $\omega_3(\mathbf{e}) = \omega_1(\mathbf{e}) = 0$ . These cases are exactly those with  $S_{\mathbf{e}}(z)^{-1} = \omega_5(\mathbf{e})^{-1}\sigma_{\mathbf{e}}(z)$  and hence  $\deg(\tau(z)) < \lfloor t/2 \rfloor$ , so that the number of iterations in the second EEA is 0 **provided that Strenzke's counter-measure is not applied**. This is a somehow surprising proposition, since this criterion can be rendered useless by a counter-measure already proposed by the same author.

**Combination of first and second EEA.** When using error weights  $w \geq 6$ , the attacker will encounter problems due to the fact that all the values given in 3 are only bounds (except in the cases  $N \leq 0$ ). Indeed, it may happen that one of the Euclidean divisions entails a degree fall greater than 1 independantly of

		EEA( $g, S_{\mathbf{e}}, 0$ )	EEA( $g, \tau, \lfloor t/2 \rfloor$ )
$w_H(\mathbf{e}) = 4$	$\omega_1(\mathbf{e}) \neq 0$	$N_I \leq 6$	$N_K \leq 1$
	$\omega_1(\mathbf{e}) = 0$	$N_I \leq 4^*$	$N_K = 0^+ \text{ CM deg}(\tau) < \lfloor t/2 \rfloor^+$
$w_H(\mathbf{e}) = 6$	$\omega_1(\mathbf{e}) \neq 0, \omega_3(\mathbf{e}) \neq 0$	$N_I \leq 10$	$N_K \leq 2$
	$\omega_1(\mathbf{e}) = 0, \omega_3(\mathbf{e}) \neq 0$	$N_I \leq 8$	$N_K \leq 1$
	$\omega_1(\mathbf{e}) = 0, \omega_3(\mathbf{e}) = 0$	$N_I \leq 6$	$N_K = 0^* \text{ CM deg}(\tau) < \lfloor t/2 \rfloor$
$w_H(\mathbf{e}) = 2w'$	$\omega_1(\mathbf{e}) \neq 0, \omega_3(\mathbf{e}) \neq 0$	$N_I \leq 4w' - 2$	$N_K \leq w' - 1$
	$\omega_1(\mathbf{e}) = 0, \omega_3(\mathbf{e}) \neq 0$	$N_I \leq 4w' - 4$	$N_K \leq w' - 2$
	$\omega_1(\mathbf{e}) = 0, \omega_3(\mathbf{e}) = 0$	$N_I \leq 4w' - 6$	$N_K \leq w' - 3$

Table 3: Overview of small- error-weight message attacks. Cases marked with a  $*$  or a  $+$  are proposed resp. in [15] and [16].

the degree of  $\omega_{\mathbf{e}}$ . For example, with  $w = 6$ , the attacker may observe  $N_K = 1$  whereas  $\omega_1(\mathbf{e})$  is not zero. This remark leads Strenzke to discard those cases for an attack as long as no way of distinguishing those cases is found. We propose such distinguisher, by using  $N_I$  to determine if  $\omega_1(\mathbf{e})$  is zero, as  $\omega_1(\mathbf{e}) = 0$  implies  $N_I \leq 8$ . Indeed, an attacker observing the errors  $\mathbf{e}$  with  $(N_I, N_K) = (10, 1)$  can conclude that  $\omega_1(\mathbf{e}) \neq 0$  (cf Table 3). We may have  $(N_I, N_K) = (8, 1)$  when  $\omega_1(\mathbf{e}) \neq 0$  if three cancellations occur in the 12 intermediate polynomials, which has probability  $p_3 = \binom{12}{3} 2^{-3m} (1 - 2^{-m})^9 \approx 2.10^{-7}$  for  $m = 10$  (we model the leading coefficients as random elements of  $\mathbb{F}_{2^m}$ ). When sampling  $x$  error vectors, we expect to find  $p_3 x$  such misleading cases. With the numbers of samples from [16][Table 2], the probability to find one is not negligible. If at least one wrong equation is deduced, the system to solve has no solution and the attack fails. We propose to avoid this problem by using errors with  $w \geq 8$ .

*Example  $w = 8$ .* We sampled randomly 10,000,000 errors  $\mathbf{e}$  of weight 8 and collected the couples  $(N_I, N_K)$  in Table 4. When  $w_H(\mathbf{e}) = 8$ , there are more possibilities than with  $w = 6$ . Samples with  $(N_I \leq 12, N_K \leq 2)$  do not necessarily have  $\omega_1(\mathbf{e}) = 0$ : this happens with probability  $p'_3 = \binom{17}{3} 2^{-3m} (1 - 2^{-m})^{14} \approx 6.10^{-7}$  for  $m = 10$  (we found 3). In particular, the case marked with a  $*$  in Table 4 would make the attacker to think erroneously that the corresponding error vector satisfies  $\omega_1(\mathbf{e}) = 0$ . However, the number of parasitic cancellations necessary to provide values  $(N_I, N_K)$  compatible with  $(\omega_1(\mathbf{e}), \omega_3(\mathbf{e})) = (0, 0)$  is 6, which happens with probability  $p'_3 = \binom{17}{6} 2^{-6m} (1 - 2^{-m})^{11} \approx 10^{-14}$  for  $m = 10$ . If  $\omega_1(\mathbf{e}) = 0$  but  $\omega_3(\mathbf{e}) \neq 0$ , then a couple  $(10, 1)$  is found if 3 cancellations occur. This has probability  $2^{-m} p'_3 \approx 6.10^{-10}$  (as  $\omega_1$  takes all the values of  $\mathbb{F}_{2^m}$  with same probability). Therefore, we are able to say without ambiguity when  $(\omega_1(\mathbf{e}), \omega_3(\mathbf{e})) = (0, 0)$  on a considerable amount of samples. We deduce from our samples 10 equations  $\omega_1(\mathbf{e}) = 0$  which are correct with proba.  $(1 - 10^{-7})$  and 10 equations  $\omega_3(\mathbf{e}) = 0$  correct with proba.  $(1 - 10^{-3})$ . To conclude, although our method requires more samples than the previous one (around  $10^9$  to collect some thousands equations with  $\omega_1$ , and dozens with  $\omega_3$ ), we showed that it is possible

	No parasitic cancellation	1 parasitic cancellation	2 parasitic cancellations	3 parasitic cancellations
$\omega_1(\mathbf{e}) \neq 0$ $\omega_3(\mathbf{e}) \neq 0$	(14,3): 9855087	(13,3): 115439 (14,2): 18916	(12,3): 614 (13,2): 248 (14,1): 8	(12,2): 1 * (11,3): 2
$\omega_1(\mathbf{e}) = 0$ $\omega_3(\mathbf{e}) \neq 0$	(12,2): 9570	(11,2): 96 (12,1): 8	(10,2): 0 (11,1): 0	
$\omega_1(\mathbf{e}) = 0$ $\omega_3(\mathbf{e}) = 0$	(10,1): 10	(9,1): 0	(8,1): 0	

Table 4: Number of samples for each  $(N_I, N_K)$  for 10,000,000 error vectors with  $w = 8$ . Code parameters:  $m = 10, n = 2^m, t = 40$ . See text for explanation on \*.

to recover information on the support even if the countermeasure  $\deg(\tau) < \lfloor t/2 \rfloor$  is implemented.

**Small weight error messages in Alternant decoder** We determine if an attacker can retrieve any information by applying Alg. 6 if the Alternant decoder is implemented. Lemma 2, which is analogous to Lemma 1, analyses the impact of small weight error messages on the EEA.

**Lemma 2.** *Let  $\mathbf{e}$  be an error with  $w_H(\mathbf{e}) \leq t$ . Then  $S_{Alt,\mathbf{e}}(z) = \frac{\omega_{inv,\mathbf{e}}(z)}{\sigma_{inv,\mathbf{e}}(z)} \pmod{z^{2t}}$  and the number of iterations  $N$  of the **while** loop of the Alternant decoder in the EEA satisfies*

$$N \leq N_{max} = \min(\deg(\sigma_{inv,\mathbf{e}}), \deg(S_{Alt,\mathbf{e}}) - \deg(\omega_{inv})) \tag{6}$$

*Specific case of weight 1 errors.* If  $w = 1$ , we always have  $\deg(\omega_{inv}) = 0$  and  $\deg(\sigma_{inv}) = 1$  **except** if the error is positioned in the zero element of the support. Indeed, in this case, the polynomial syndrome is a constant:  $S_{\mathbf{e}}(z) = \frac{1}{g(0)^2}$  and the **while** loop is never executed.

*Error weights  $w > 1$ .* We suppose that no error occurred in the zero element of the support so that  $\deg(\sigma_{inv}) = w_H(\mathbf{e})$  always holds (the coefficient of  $z^w$  in  $\sigma_{inv}$  is  $x_{i_1} \dots x_{i_w}$ ). Therefore, faster decryptions indicate the cancellation of a leading coefficient in the intermediate values, but in the alternant decoder we found no way of determining which intermediate value was concerned. If by any chance a power analysis can ensure that it is the first intermediate polynomial (that is, the syndrome polynomial  $S_{Alt,\mathbf{e}}(z)$ ) that has a degree smaller than expected, then the information recovered would be:

$$\sum_{j=1}^w g(x_{i_j})^{-2} \sum_{j=1}^w x_{i_j}^{2t-1} = 0. \tag{7}$$

We observe that the equations written thanks to this method are more complex than with Patterson algorithm, at least for two reasons. First, they are not directly polynomial, and the degrees implied are much higher. Second, as both  $\mathbf{x}$  and  $g$  have to be unknown ([8][p. 125]), additive unknowns are necessary: either  $t + 1$  to describe the secret polynomial's coefficients, or  $n$  if we introduce new equations  $y_i = g(x_i)^{-2}$ . We conclude that the alternant decoder is intrinsically more resistant to Strenzke's attacks. However, the overall security is still not clear due to the uncertainty on the countermeasure (described in Alg. 5) against Alg. 3.

## 4 Extended euclidean Algorithm with constant flow

We expose a way of implementing the EEA algorithm unused so far for McEliece decryption. It has the very interesting property of requiring a number of operations depending only on the Goppa polynomial degree  $t$  and **not** on the weight of the error introduced in the ciphertext. Therefore, the attacks of 3.1 and 3.2 are not possible.

It is inspired by Berlekamp's work in [3] (which as followed by other works of optimization in the VLSI community, amongst many others [10,11]). We could find no reference to it in any paper related to McEliece. On the contrary, designing such an algorithm is desirable goal according to the conclusion of [16]. The reason may be that [3] has a very limited access, and we could find no completeness proofs of the algorithm proposed. Here, we transform smoothly the original EEA (Alg. 2) into successive version gaining in regularity (Algorithms 8 and 9). We end up with Alg. 10, which is simpler and more regular than all the previous ones. At each step, we give and prove (in Appendix) the form of the outputs and intermediate values. Finally, each execution of Alg. 10 costs, in field multiplications, exactly  $16t^2$  ( $2t$  times a loop costing  $4 \times 2t$ ).

In the rest of this article we will set  $N$  be the number of Euclidean divisions performed during  $\text{EEA}(z^{2t}, S_{Alt}(z), t)$  in Alg. 2,  $d_i = \deg(r_i(z))$ , and  $\delta_i = \deg(q_i(z)) = \deg(r_{i-2}) - \deg(r_{i-1})$ .

For any polynomial  $P(z) \in \mathbb{F}_{q^m}[z]$ , we denote its coefficients by  $P_j$  even for  $j > \deg(P)$  (in which case  $P_j = 0$ ), so that

$$P(z) = \sum_{j=0}^{+\infty} P_j z^j = P_{\deg(P)} z^{\deg(P)} + \dots + P_0.$$

Regarding the  $\delta_i$ 's, we prove the following result which will be very useful to design an algorithm with regular pattern:

**Lemma 3.** *Let the  $\delta_i$ 's be defined as previously during the execution of  $\text{EEA}(z^{2t}, S_{\mathbf{e}}(z), t)$ , then it holds that*

$$\sum_{i=1}^N \delta_i = \deg(u_N(z)) = \deg(\omega_{inv, \mathbf{e}}) = w_H(\mathbf{e}) - 1.$$

We propose several intermediate versions of algorithms computing  $\sigma_{inv, \mathbf{e}}(z)$ .

**Unrolling Euclidean divisions** In Alg. 8, we decompose each Euclidean division into a number of polynomial subtractions depending only on  $\delta_i$  the degrees of the quotients. We explicit the intermediate values of the Euclidean division of  $R_{i-2}(z)$  by  $R_{i-1}(z)$ , that we denote by  $R_i^{(0)}(z), \dots, R_i^{(\delta_i+1)}(z)$ . To do so, we eliminate in each  $R_i^{(j)}(z)$  (for  $0 \leq j \leq \delta_i + 1$ ) the term  $z^{d_i-2-j}$ , whether the associated coefficient is zero or not. This is why we perform the Euclidean divisions in a way to avoid the divisions by a field elements (Steps 7 to 11 of Alg. 8). Consequently, the outputs are multiple of the outputs of Alg. 2 with the same inputs.

**Proposition 1 (Comparison of Algorithms 2 and 8).** *Let  $a(z)$  and  $b(z)$  be two polynomials with  $\deg(a(z)) \geq \deg(b(z))$ , and  $d$  a non-negative integer.  $u_i(z), v_i(z), r_i(z), q_i(z)$  are the intermediate values in Alg. 2, and  $U_i(z), V_i(z), R_i(z)$  are the intermediate values in Algorithm 8. It holds that, for all  $i = -1, \dots, N$ , there exists  $\lambda_i \in \mathbb{F}_{q^m}^*$  such that:*

$$\begin{aligned} R_i(z) &= \lambda_i r_i(z), \\ U_i(z) &= \lambda_i u_i(z). \end{aligned}$$

As a consequence,  $\Delta_i = \deg(R_{i-2}) - \deg(R_{i-1}) = \deg(r_{i-2}) - \deg(r_{i-1}) = \delta_i$  for all  $i$ .

There are two problems with Alg. 8: the first one is that the inner **for** loop ((Steps 7 to 11) has a variable length, and contains a multiplication  $z^{\delta_i-(j-1)} R_i(z)$  which depends on the iteration, which will produce a recognizable pattern and the second problem is that the **while** loop leads to a variable number of operations according to the input. Alg. 9 is a first step towards the resolution of the second problem. It is not realistic (it requires that Alg. 8 has already been executed and observed), but it eases the proofs of completeness of Alg. 10, which solves both issues.

**Regular polynomial shift pattern.** In Alg. 9, we perform the Euclidean division in such a way that we only multiply the operand by  $z$  at each **for** iteration. This can be done by splitting in two phases each Euclidean divisions. The first phase (Steps 4 to 7) "re-aligns" the operands  $\tilde{R}_{i-2}$  and  $\tilde{R}_{i-1}$  so that they both have same degree  $d = \deg(R_{-1}(z)) (= 2t)$ . Doing so, the second phase (Steps 8 to 12) compute the polynomial subtractions (corresponding to Steps 9-10 of Alg. 8) and perform a shift "re-aligning" the operands. A consequence is that the polynomials  $\tilde{R}_i(z)$  are of the form  $z^{k_i} R_i(z)$  and the degrees  $d_i$  are lost. This problem will be solved in Alg. 10.

**Proposition 2 (Comparison of Algorithms 8 and 9).** *For each  $i = 1, \dots, N$ , after Step 13 of Alg. 9, it holds that*

$$\begin{aligned} (\tilde{R}_{i-1}(z), \tilde{R}_i(z)) &= (z^{d-d_{i-1}} R_{i-1}(z), z^{d-d_{i-1}+1} R_i(z)), \\ (\tilde{U}_{i-1}(z), \tilde{U}_i(z)) &= (z^{d-d_{i-1}} U_{i-1}(z), z^{d-d_{i-1}+1} U_i(z)). \end{aligned}$$

---

**Algorithm 8** EEA with unrolled Euclidean Division
 

---

**Input:**  $a(z) = z^{2t}, b(z) = S_e(z)$ .

**Output:**  $U(z) = \lambda_N \sigma_e(z), R(z) = \lambda_N \omega_e(z)$  (for some  $\lambda_N \in \mathbb{F}_{q^m}^*$ ).

- 1:  $R_{-1}(z) \leftarrow a(z), R_0(z) \leftarrow b(z), U_{-1}(z) \leftarrow 1, U_0(z) \leftarrow 0, i \leftarrow 0$ .
  - 2: **while**  $\deg(R_i(z)) > t$  **do**
  - 3:    $i \leftarrow i + 1$
  - 4:    $R_{i-2}^{(0)}(z) \leftarrow R_{i-2}(z), U_{i-2}^{(0)}(z) \leftarrow U_{i-2}(z)$
  - 5:    $\Delta_i \leftarrow \deg(R_{i-2}) - \deg(R_{i-1})$
  - 6:    $\beta_i \leftarrow \text{LC}(R_{i-1}(z))$ .
  - 7:   **for**  $j = 0, \dots, \Delta_i$  **do**
  - 8:      $\alpha_{i,j} \leftarrow R_{i, d_{i-2}-j}^{(j)}$ ,
  - 9:      $R_{i-2}^{(j+1)}(z) \leftarrow \beta_i R_{i-2}^{(j)}(z) - \alpha_{i,j} z^{\Delta_i-j} R_{i-1}(z)$
  - 10:     $U_{i-2}^{(j+1)}(z) \leftarrow \beta_i U_{i-2}^{(j)}(z) - \alpha_{i,j} z^{\Delta_i-j} U_{i-1}(z)$
  - 11:   **end for**
  - 12:    $R_i(z) \leftarrow R_{i-2}^{(\Delta_i+1)}(z), U_i(z) \leftarrow U_{i-2}^{(\Delta_i+1)}(z)$
  - 13: **end while**
  - 14:  $N \leftarrow i$ .
  - 15: **return**  $U_N(z), R_N(z)$
- 

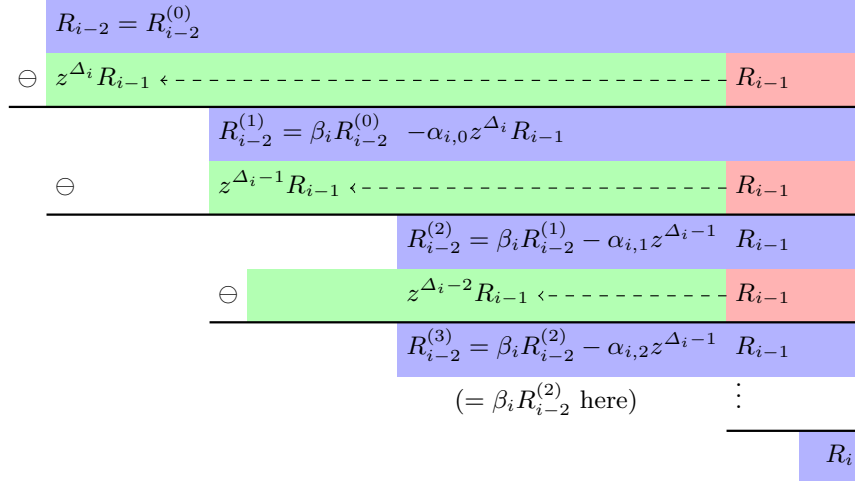


Fig. 3: Euclidean division of  $R_{i-2}$  by  $R_{i-1}$  in exactly  $\Delta_i + 1$  polynomial subtractions (with  $\Delta_i = \deg(R_{i-2}) - \deg(R_{i-1})$ ). In this example,  $\alpha_{i,2} = 0$



---

**Algorithm 9** Toy EEA with regular shift pattern

---

**Input:**  $a(z) = z^{2t}, b(z) = S_e(z), d = 2t$

**Output:**  $\tilde{U}_N(z) = z^{d-d_{N-1}+1}\sigma_e(z), \tilde{R}_N(z) = z^{d-d_{N-1}+1}\omega_e(z).$

- 1:  $\tilde{R}_{-1}(z) \leftarrow a(z), \tilde{R}_0(z) \leftarrow zb(z), \tilde{U}_{-1}(z) \leftarrow 1, \tilde{U}_0(z) \leftarrow 0.$
  - 2: **for**  $i = 1, \dots, N$  **do**
  - 3:    $\tilde{R}_{i-2}^{(0)}(z) \leftarrow \tilde{R}_{i-2}(z), \tilde{U}_{i-2}^{(0)}(z) \leftarrow \tilde{U}_{i-2}(z)$
  - 4:   **for**  $j = 1, \dots, \Delta_i - 1$  **do**
  - 5:      $\tilde{R}_{i-1}(z) \leftarrow z\tilde{R}_{i-1}(z)$  }  $L_1$
  - 6:      $\tilde{U}_{i-1}(z) \leftarrow z\tilde{U}_{i-1}(z)$  }
  - 7:   **end for**
  - 8:   **for**  $j = 0, \dots, \Delta_i$  **do**
  - 9:      $\tilde{\alpha}_{i,j} \leftarrow \tilde{R}_{i,d}^{(j)}, \tilde{\beta}_i \leftarrow \tilde{R}_{i-1,d}.$
  - 10:     $\tilde{R}_{i-2}^{(j+1)}(z) \leftarrow z \left( \tilde{\beta}_i \tilde{R}_{i-2}^{(j)}(z) - \tilde{\alpha}_{i,j} \tilde{R}_{i-1}(z) \right)$  }  $L_2$
  - 11:     $\tilde{U}_{i-2}^{(j+1)}(z) \leftarrow z \left( \tilde{\beta}_i \tilde{U}_{i-2}^{(j)}(z) - \tilde{\alpha}_{i,j} \tilde{U}_{i-1}(z) \right)$  }
  - 12:   **end for**
  - 13:    $\tilde{R}_i(z) \leftarrow \tilde{R}_{i-2}^{(\Delta_i+1)}(z), \tilde{U}_i(z) \leftarrow \tilde{U}_{i-2}^{(\Delta_i+1)}(z)$
  - 14: **end for**
  - 15: **return**  $\tilde{U}_N(z), \tilde{R}_N(z)$
- 

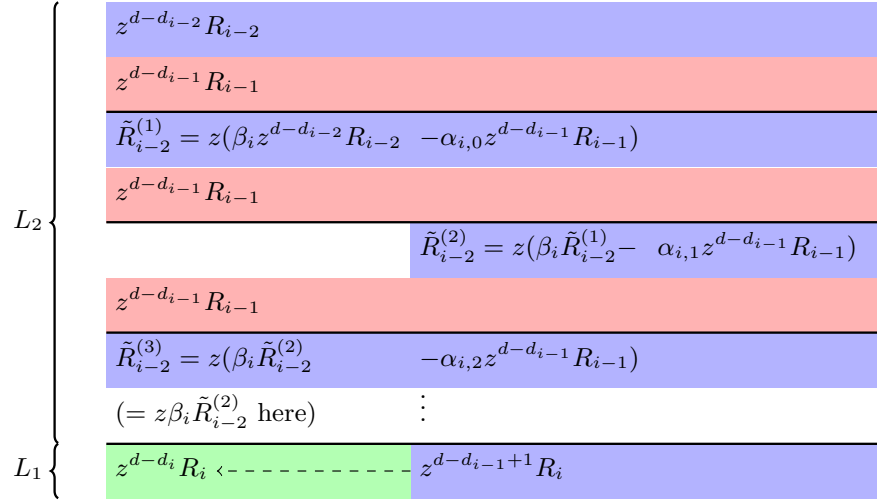


Fig. 4: "Re-aligned" Euclidean division of  $R_{i-2}$  by  $R_{i-1}$ : inputs and outputs are degree- $d$  polynomials  $z^{d-d_{i-2}}R_{i-2}$ ,  $z^{d-d_{i-1}}R_{i-1}$ , and  $z^{d-d_i}R_i$ .

**Complete Regular Flow EEA.** To design a real constant flow algorithm, we merge the loops  $L_1$  and  $L_2$  in a common pattern so as to be indistinguishable (Steps 5 to 7 of Alg.10). They differentiate by the assignments which are performed in Steps 14- 15 and 18-19. To know when polynomials substractions have to be stopped, we collect in a counter  $\delta$  the number of shifts necessary to re-align the operands. Finally, when the polynomials  $\sigma_{inv}$  and  $\omega_{inv}$  have been computed, the extra executions of the main loop (Steps 4 to 22) consist in shifting the operands. therefore, the number of iterations can be safely set to the maximum value (ie  $2t$  to decode the errors with  $w_H(\mathbf{e}) = t$ ), and the **while** loop is replaced by **for**.

**Proposition 3 (Comparison of Algorithms 8 and 10.).** *For each  $i = 1, \dots, N$ , after steps 21, it holds that:*

$$\begin{aligned}\hat{R}_{2(\delta_1+\dots+\delta_i)}(z) &= z^{d-d_{i-1}+1}R_i(z), \\ \hat{U}_{2(\delta_1+\dots+\delta_i)}(z) &= z^{d-d_{i-1}+1}U_i(z).\end{aligned}$$

The outputs of Alg. 10 are, for some  $\mu \in \mathbb{F}_{q^m}^*$ :

$$\begin{aligned}\hat{R}_d(z) &= z^{d-w_H(\mathbf{e})+1}R_N(z) = \mu z^{d-w_H(\mathbf{e})+1}\omega_{inv}(z), \\ \hat{U}_d(z) &= z^{d-w_H(\mathbf{e})+1}U_N(z) = \mu z^{d-w_H(\mathbf{e})+1}\sigma_{inv}(z).\end{aligned}$$

Therefore, provided 0 is not an element of  $\mathbf{x}$ ,  $\hat{U}_d(z)$  allows to recover the error positions without ambiguity. Transposing this result to Patterson decoding requires to adapt both EEA's. The adaptation of the second one is straightforward. For the first one (syndrome inversion), a problem arises: the analogous of Proposition 3 would yield  $\hat{U}_{N_i}(z) = \mu z^{k_i}(S_{Gop,\mathbf{e}}^{-1} \bmod g)$  for some  $k_i > 0$ , and we found no way of determining when  $z$  is a factor of  $S_{Gop,\mathbf{e}}^{-1} \bmod g$ . However, we can protect the second EEA to avoid the attack of 3.2.

## 5 Conclusion

We proposed an algorithm determining the error-locator polynomial costing always  $16t^2$  field multiplications on any input. It contains a test depending on secret data, followed by two balanced branches. The indistinguishability of those branches by an attacker is crucial for the security of the decryption, and depends on the architecture of the implementation.

**Acknowledgements** We are thankful to Jean-Charles Faugère, Aline Gouget, Ludovic Perret and anonymous reviewers for their comments in the preparation of this paper.

## References

1. R. Avanzi, S. Hoerder, D. Page, and M. Tunstall. Side-channel attacks on the McEliece and Niederreiter public-key cryptosystems. *J. Cryptographic Engineering*, 1(4):271–281, 2011.

---

**Algorithm 10** EEA with regular flow

---

**Input:**  $a(z) = z^{2t}, b(z) = S_e(z), d = 2t$

**Output:**  $\hat{U}_d(z) = \mu z^{d-w_H(e)+1} \sigma_{inv}(z), \hat{R}_d(z) = \mu z^{d-w_H(e)+1} \omega_e(z)$  for some  $\mu \in \mathbb{F}_q^*$ .

```

1:  $\hat{R}_{-1}(z) \leftarrow a(z), \hat{R}_0(z) \leftarrow zb(z),$ 
2:  $\hat{U}_{-1}(z) \leftarrow 1, \hat{U}_0(z) \leftarrow 0,$ 
3:  $\delta \leftarrow -1.$ 
4: for  $j = 1, \dots, d$  do
5:    $\alpha_j \leftarrow \hat{R}_{j-1,d}, \beta_j \leftarrow \hat{R}_{j-2,d}.$ 
6:    $temp_R(z) \leftarrow z \left( \alpha_j \hat{R}_{j-2}(z) - \beta_j \hat{R}_{j-1}(z) \right).$ 
7:    $temp_U(z) \leftarrow z \left( \alpha_j \hat{U}_{j-2}(z) - \beta_j \hat{U}_{j-1}(z) \right).$ 
8:   if  $\alpha_j = 0$  (ie  $\deg(\hat{R}_{j-1}) < \deg(\hat{R}_{j-2})$ ) then
9:      $\delta \leftarrow \delta + 1.$ 
10:  else
11:     $\delta \leftarrow \delta - 1.$ 
12:  end if
13:  if  $\delta < 0$  then
14:     $(\hat{R}_j(z), \hat{R}_{j-1}(z)) \leftarrow (\hat{R}_{j-1}(z), temp_R)$ 
15:     $(\hat{U}_j(z), \hat{U}_{j-1}(z)) \leftarrow (\hat{U}_{j-1}(z), temp_U)$ 
16:     $\delta \leftarrow 0.$ 
17:  else
18:     $(\hat{R}_j(z), \hat{R}_{j-1}(z)) \leftarrow (temp_R, \hat{R}_{j-2}(z))$ 
19:     $(\hat{U}_j(z), \hat{U}_{j-1}(z)) \leftarrow (temp_U, \hat{U}_{j-2}(z))$ 
20:     $\delta \leftarrow \delta.$ 
21:  end if
22: end for
23: return  $\hat{U}_d(z), \hat{R}_d(z)$ 

```

} L

---

2. E. Berlekamp, R. McEliece, and H. van Tilborg. On the inherent intractability of certain coding problems. *Information Theory, IEEE Transactions on*, 24(3):384–386, May 1978.
3. E. Berlekamp, G. Seroussi, and T. Po. A hypersystolic Reed–Solomon decoder. In V. K. Bhargava, S. B. Wicker, IEEE Communications Society, and IEEE Information Theory Society, editors, *Reed-Solomon codes and their applications*, pages 205–241. IEEE Press Piscataway, NJ, 1994.
4. D. J. Bernstein, T. Chou, and P. Schwabe. McBits: fast constant-time code-based cryptography. In G. Bertoni and J.-S. Coron, editors, *Cryptographic Hardware and Embedded Systems – CHES 2013*, volume 8086 of *Lecture Notes in Computer Science*. Springer-Verlag Berlin Heidelberg, 2013. Document ID: e801a97c500b3ac879d77bcecf054ce5, <http://cryptojedi.org/papers/#mcbits>.
5. B. Biswas. *Aspects de mise en oeuvre de la cryptographie basée sur les codes*. These, Ecole Polytechnique X, Oct. 2010.
6. S. Heyse. Implementation of McEliece Based on Quasi-dyadic Goppa Codes for Embedded Devices. In B.-Y. Yang, editor, *PQCrypto*, volume 7071 of *Lecture Notes in Computer Science*, pages 143–162. Springer, 2011.
7. F. J. MacWilliams and N. J. A. Sloane. *The Theory of Error-Correcting Codes*. North-Holland, Amsterdam, fifth edition, 1986.

8. R. Overbeck and N. Sendrier. Code-based cryptography. In D. Bernstein, J. Buchmann, and E. Dahmen, editors, *Post-Quantum Cryptography*, pages 95–145. Springer Berlin Heidelberg, 2009.
9. N. Patterson. The algebraic decoding of Goppa codes. *IEEE Transactions on Information Theory*, 21(2):203–207, 1975.
10. D. Sarwate and N. Shanbhag. High-speed architectures for reed-solomon decoders. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 9(5):641–655, Oct 2001.
11. D. Sarwate and Z. Yan. Modified euclidean algorithms for decoding reed-solomon codes. In *Information Theory, 2009. ISIT 2009. IEEE International Symposium on*, pages 1398–1402, June 2009.
12. A. Shoufan, F. Strenzke, H. G. Molter, and M. Stöttinger. A Timing Attack against Patterson Algorithm in the McEliece PKC. In D. Lee and S. Hong, editors, *ICISC*, volume 5984 of *Lecture Notes in Computer Science*, pages 161–175. Springer, 2009.
13. A. Shoufan, T. Wink, H. Molter, S. Huss, and E. Kohnert. A Novel Cryptoprocessor Architecture for the McEliece Public-Key Cryptosystem. *Computers, IEEE Transactions on*, 59(11):1533–1546, Nov 2010.
14. F. Strenzke. A Smart Card Implementation of the McEliece PKC. In P. Samarati, M. Tunstall, J. Posegga, K. Markantonakis, and D. Sauveron, editors, *Information Security Theory and Practices. Security and Privacy of Pervasive Systems and Smart Devices*, volume 6033 of *Lecture Notes in Computer Science*, pages 47–59. Springer Berlin Heidelberg, 2010.
15. F. Strenzke. A Timing Attack Against the Secret Permutation in the McEliece PKC. In *Proceedings of the Third International Conference on Post-Quantum Cryptography*, PQCrypto’10, pages 95–107, Berlin, Heidelberg, 2010. Springer-Verlag.
16. F. Strenzke. Timing Attacks against the Syndrome Inversion in Code-Based Cryptosystems. In P. Gaborit, editor, *PQCrypto*, volume 7932 of *Lecture Notes in Computer Science*, pages 217–230. Springer, 2013.
17. F. Strenzke, E. Tews, H. G. Molter, R. Overbeck, and A. Shoufan. Side Channels in the McEliece PKC. In *Proceedings of the 2Nd International Workshop on Post-Quantum Cryptography*, PQCrypto ’08, pages 216–229, Berlin, Heidelberg, 2008. Springer-Verlag.
18. Y. Sugiyama, M. Kasahara, S. Hirasawa, and T. Namekawa. A method for solving key equation for decoding Goppa codes. *Information and Control*, 27(1):87 – 99, 1975.
19. K. Thull and C. Yap. A Unified Approach to HGCD Algorithms for polynomials and integers., 1990. Manuscript. Available from <http://cs.nyu.edu/cs/faculty/yap/allpapers.html/>.

## A Proofs of completeness

**Proof of Proposition 1.** We recall of the  $R_i(z)$ ’s are computed: set  $R_{-1}(z) = r_{-1}(z) = a(z)$  and  $R_0(z) = r_0(z) = b(z)$ . Then, for  $1 \leq i \leq N$ , set  $R_{i-2}^{(0)}(z) = R_{i-2}(z)$  and recursively

$$R_{i-2}^{(j+1)}(z) = \beta_i R_{i-2}^{(j)}(z) - \alpha_{i,j} z^{\delta_i - j} R_{i-1}(z) \quad (8)$$

with  $\alpha_{i,j} = R_{i,d_i-2-j}^{(j)}$ ,  $\beta_i = \text{LC}(R_{i-1}(z))$ .  $R_i(z)$  is defined by  $R_i(z) = R_{i-2}^{(\delta_i+1)}(z)$ .

In essence we have to prove that  $R_{i-2}^{(\delta_i+1)}(z)$  is the rest of the Euclidian division of  $R_{i-2}(z)$  by  $R_{i-1}(z)$ . First, we prove by induction on  $j$  that, for all  $1 \leq i \leq N$  and  $0 \leq j \leq \delta_i + 1$ , there exists a polynomial  $f_{i,j}(z) \in \mathbb{F}_{q^m}[z]$  and a non-zero scalar  $\mu_{i,j}$  such that

$$R_{i-2}^{(j)}(z) = \mu_{i,j}R_{i-2}^{(0)}(z) - f_{i,j}(z)R_{i-1}(z) \text{ and } \deg(R_{i-2}^{(j)}(z)) \leq d_{i-2} - j. \quad (9)$$

For  $j = 0$ , pick  $f_{i,0}(z) = 0, \mu_{i,0} = 1$  and notice that  $\deg(R_{i-2}^{(0)}) = d_{i-2}$ . Now at step  $j > 0$ , suppose  $R_{i-2}^{(j)}(z) = \mu_{i,j}R_{i-2}^{(0)}(z) - f_{i,j}(z)R_{i-1}(z)$ , then

$$\begin{aligned} R_{i-2}^{(j+1)}(z) &= \beta_i R_{i-2}^{(j)}(z) - \alpha_{i,j} z^{\delta_i-j} R_{i-1}(z) \\ &= \underbrace{\mu_{i,j} \beta_i}_{\mu_{i,j+1}} R_{i-2}^{(0)}(z) - \underbrace{(f_{i,j}(z) + \alpha_{i,j} z^{\delta_i-j})}_{f_{i,j+1}(z)} R_{i-1}(z). \end{aligned}$$

Regarding the degree, if  $\deg(R_{i-2}^{(j)}) < d_{i-2} - j$ , then  $R_{i-2, d_{i-2}-j}^{(j)} = 0$ , so  $\alpha_{i,j} = 0$  and  $\deg(R_{i-2}^{(j+1)}) = \deg(R_{i-2}^{(j)}) \leq d_{i-2} - (j+1)$ . If  $\deg(R_{i-2}^{(j)}) = d_{i-2} - j$ , then observe that  $LC(\beta_i R_{i-2}^{(j)}) = LC(R_{i-1}(z))R_{i, d_{i-2}-j}$  and  $LC(\alpha_{i,j} z^{\delta_i-j} R_{i-1}(z)) = R_{i, d_{i-2}-j} LC(R_{i-1}(z))$ , so that the leading monomials cancel, and therefore

$$\deg(R_{i-2}^{(j+1)}) \leq d_{i-2} - (j+1).$$

Thanks to Eq. (9), we know that at step  $j = \delta_i + 1$  :

$$R_{i-2}(z) = f_{i, \delta_i+1}(z)R_{i-1}(z) + \mu_{i, \delta_i+1}R_{i-2}^{(\delta_i+1)}(z) \text{ and } \deg(R_{i-2}^{(\delta_i+1)}) \leq d_{i-1} - 1.$$

Thanks to the induction hypothesis  $R_{i-2}(z) = \lambda_{i-2}r_{i-2}(z)$  and  $R_{i-1}(z) = \lambda_{i-1}r_{i-1}(z)$ , so we have

$$r_{i-2}(z) = \lambda_{i-1}\lambda_{i-2}^{-1}f_{i, \delta_i+1}(z)r_{i-1}(z) + \lambda_{i-2}^{-1}\mu_{i, \delta_i+1}R_{i-2}^{(\delta_i+1)}(z).$$

By unicity in the Euclidean division, we set  $\lambda_i = \lambda_{i-2}\mu_{i, \delta_i+1}^{-1}$  it follows that:

$$R_{i-2}^{(\delta_i+1)}(z) = \lambda_i r_i(z), \quad \lambda_{i-1}\lambda_{i-2}^{-1}f_{i, d_{i-2}-d_{i-1}+1}(z) = q_i(z). \quad (10)$$

For  $(U_{i-2}^{(j)}(z))$ , we would prove the same way that  $U_{i-2}^{(j)}(z) = \mu_{i,j}U_{i-2}^{(0)}(z) - f_{i,j}(z)U_{i-1}(z)$  with the same  $\mu_{i,j}$  and  $f_{i,j}(z)$ . Therefore, thanks to the induction hypothesis  $(U_{i-2}(z) = \lambda_{i-2}u_{i-2}(z)$  and  $U_{i-1}(z) = \lambda_{i-1}u_{i-1}(z))$ ,

$$u_{i-2}(z) = \underbrace{\lambda_{i-1}\lambda_{i-2}^{-1}f_{i, \delta_i+1}(z)}_{q_i(z)(10)} u_{i-1}(z) + \underbrace{\lambda_{i-2}^{-1}\mu_{i, \delta_i+1}}_{\lambda_i^{-1}(10)} U_{i-2}^{(\delta_i+1)}(z).$$

With Eq. (10), we see that  $U_{i-2}^{(\delta_i+1)}(z) = \lambda_i u_{i-2}(z) - \lambda_i q_i(z) u_{i-1}(z) = \lambda_i u_i(z)$ .  $\square$

**Proof of Proposition 2.** The operands at entering the main **for** loop (Steps 2-14 of Alg. 9) are by induction hypothesis

$$(\tilde{R}_{i-2}, \tilde{R}_{i-1}) = z^{d-d_{i-2}} R_{i-2}(z), z^{d-d_{i-2}+1} R_{i-1}(z),$$

(for  $i = 1$ , we have indeed  $(\tilde{R}_{-1}, \tilde{R}_0) = (R_{-1}(z), zR_0(z))$ ).

The main loop is decomposed into two phases:  $L_1$  (Steps 4-7) and  $L_2$  (Steps 8-12).  $L_1$  consists in shifting  $z^{d-d_{i-2}+1} R_i(z)$  up-wise until it has a non-zero  $d$ -th coefficient (ie it has degree  $d$ ), that is exactly  $\Delta_i - 1 = d_{i-2} - d_{i-1} - 1$  times, so that after Step 7  $\tilde{R}_{i-1}(z) = z^{d-d_{i-1}} R_{i-1}(z)$ .

$L_2$  (Steps 8-12) performs a twisted Euclidean division of  $R_{i-2}(z)$  by  $R_{i-1}(z)$  on the shifted values  $\tilde{R}_{i-2}(z) z^{d-d_{i-2}} R_{i-2}(z)$  and  $\tilde{R}_{i-1}(z) = z^{d-d_{i-1}} R_{i-1}(z)$ . We prove that, after Step 12,  $\tilde{R}_{i-2}^{(\delta_i+1)}(z) = z^{d-d_{i-1}+1} R_i(z)$ . To do so, we show the following link between the intermediate results of Algorithms 8 ( $R_{i-2}^{(j)}(z)$ ) and 9 ( $\tilde{R}_{i-2}^{(j)}(z)$ ): for  $0 \leq j \leq \Delta_i + 1$ ,

$$\tilde{R}_{i-2}^{(j)}(z) = z^{d-d_{i-2}+j} R_{i-2}^{(j)}(z). \quad (11)$$

This is done by induction on  $j$ . For  $j = 0$ , we have  $\tilde{R}_{i-2}^{(0)}(z) = z^{d-d_{i-2}} R_{i-2}(z) = z^{d-d_{i-2}} R_{i-2}^{(0)}(z)$ . Then, suppose  $\tilde{R}_{i-2}^{(j)}(z) = z^{d-d_{i-2}+j} R_{i-2}^{(j)}(z)$ . So, it holds that  $\tilde{R}_{i-2,d}^{(j)} = R_{i-2,d-d_{i-2}+j}^{(j)}$ , and  $\text{LC}(\tilde{R}_{i-1}(z)) = \text{LC}(R_{i-1}(z))$ , so that  $\tilde{\alpha}_{i,j} = \alpha_{i,j}$  and  $\tilde{\beta}_i = \beta_i$ . Now  $\tilde{R}_{i-2}^{(j+1)}(z)$  is given by:

$$\begin{aligned} \tilde{R}_{i-2}^{(j+1)}(z) &= z \left( \tilde{\beta}_i \tilde{R}_{i-2}^{(j)}(z) - \tilde{\alpha}_{i,j} \tilde{R}_{i-1}(z) \right) \\ &= z \left( z^{d-d_{i-2}+j} \beta_i R_{i-2}^{(j)}(z) - \alpha_{i,j} z^{d-d_{i-1}} R_{i-1}(z) \right) \\ &= z^{d-d_{i-2}+j} \left( \beta_i R_{i-2}^{(j)}(z) - \alpha_{i,j} z^{\delta_i-(j)} \tilde{R}_{i-1}(z) \right) \\ &\stackrel{\text{Eq (8)}}{=} z^{d-d_{i-2}+j} R_{i-2}^{(j)}(z). \end{aligned}$$

So (11) is proved by induction. With  $j = \delta_i + 1$ , we obtain  $\tilde{R}_{i-2}^{(\delta_i+1)}(z) = z^{d-d_{i-1}+1} R_{i-2}^{(\delta_i+1)}(z) = z^{d-d_{i-1}+1} R_i(z)$  (thanks to Eq. (10)): the announced result holds for  $\tilde{R}_i(z)$ . The proof is exactly the same for  $\tilde{U}_i$  by substituting  $R$  by  $U$  in the previous proof.

**Proof of Proposition 3.** By induction, we suppose that  $\hat{R}_{2(\delta_1+\dots+\delta_{i-1})}(z) = z^{d-d_{i-2}+1} R_{i-1}(z)$ . As in the proof of Proposition 2, we distinguish several phases when iterating the main **for** loop  $L$  (Steps 4-22). First, for  $j = 0, \dots, \delta_i - 1$ ,  $\hat{R}_{2(\delta_1+\dots+\delta_{i-1})+j} = z^{d-d_{i-2}+1} R_{i-1}(z)$  has degree lower than  $d$  and  $L$  consists in shifting  $\hat{R}$  up-wise (up to a multiplicative constant) as  $L_1$  in Alg 9. In addition, we update the counter  $\delta$  to record the degree difference between  $R_{i-2}$  and  $R_{i-1}$ . Indeed, this phase stops when  $z^{d-d_{i-2}+1+j} R_{i-1}(z)$  reaches degree  $d$ , that is when  $j = d_{i-2} - d_{i-1} - 1 = \delta_i - 1$ . Second, for  $j = \delta_i, \dots, 2\delta_i - 1$ ,  $L$  processes the

Euclidean division of  $R_{i-2}(z)$  by  $R_{i-1}(z)$  as  $L_2$  in Alg 9. The intermediate results are exactly those detailed in Proposition 2. The difference here is that, prior to computing the last step of the division,  $\delta$  is equal to  $-1$ , so that the intermediate values are swapped, and the announced result for  $\hat{R}_{2(\delta_1+\dots+\delta_i)}(z)$  holds.

We deduce that  $\hat{R}_{2(\delta_1+\dots+\delta_N)}(z) = z^{d-d_{N-1}+1}R_N(z) = \lambda_N z^{d-d_{N-1}+1}\sigma_{inv}(z)$  and the same proof shows that

$$\hat{U}_{2(\delta_1+\dots+\delta_N)}(z) = z^{d-d_{N-1}+1}U_N(z) = \lambda_N z^{d-d_{N-1}+1}\omega_{inv}(z).$$

The following iterations ( $j > 2w_H(\mathbf{e})$ ) consists only in shifts. Indeed, as

$$\deg(z^{d-d_{N-1}+1}R_N(z)) = w_H(\mathbf{e}) + 1 + \deg(\omega_{\mathbf{e}}(z)) \leq 2w_H(\mathbf{e}),$$

and after  $2t - 2w_H(\mathbf{e})$  up-wise shifts the output still has degree  $\leq d = 2t$ .  $\square$