Analysis of Average Running Times and Complexity in Root-Finding Algorithms for Polynomial Factorization

Nishi Tadamalla
nishienos12@gmail.com

***Abstract*—This paper investigates the average running times and complexities of various root-finding algorithms, including the Affine Method (ARM), Sparse Root Algorithm (SRA), and Branching Tree Algorithm (BTA), in the context of polynomial factorization over finite fields. The algorithms are analyzed based on their ability to separate the roots of a polynomial into affine spaces of diminishing dimensions. We establish connections to the statistical properties of digital tries, allowing for an estimation of the expected height of the tree structures generated by these algorithms. Our findings reveal that ARM and BTA typically halt at a height related to the logarithm of the number of distinct roots, while SRA operates based on the degree of the polynomial. Additionally, we explore the implications of worst-case scenarios for ARM and propose improvements to its efficiency through partial precomputation techniques. The paper concludes by addressing the computational challenges associated with polynomial factorization, especially in cases where the degree is highly composite, and offers insights into potential optimizations for practical implementations of these algorithms.**

I. Introduction

Let $\mathbb{F}_{q^n}$ be the finite field with q^n elements, and let f be a polynomial of degree d over $\mathbb{F}_{q^n}$. The *root-finding problem* is the problem of computing one, several or all elements $x \in \mathbb{F}_{q^n}$ such that $f(x) = 0$. This problem has many applications, in particular in cryptography and in coding theory [1]. It also has a rich history, with many strategies proposed over the years. In this paper, we review three related algorithms for root finding exploiting the structure of the finite affine geometry of $\mathbb{F}_{q^n}$.

Previous work. As mentioned by von zur Gathen [2], the first method for finding the roots of $f \in \mathbb{F}_p[X]$, with p an odd prime, is due to Legendre (1752-1833) who starts by considering the factorization of the field equation

$$X^p - X = X \cdot (X^{(p-1)/2} - 1) \cdot (X^{(p-1)/2} + 1)\,.$$

He then observes that computing the GCD of f with each of the two latter factors splits the (nonzero) roots of f into two sets, the squares and the non-squares. He then observes that replacing X by $X + r$ for a random r offers other ways of splitting the roots into two sets, which may successively be applied to the factors of f. This probabilistic method is the basis of many modern ones.

For a polynomial $f \in \mathbb{F}_q[X]$, with q a prime power, the factoring method of Berlekamp [3] is based on the so-called *Petr-Berlekamp algebra* of f which consists of the set of polynomials h satisfying

$$h(X)^q = h(X) \bmod f(X)$$

Using $X^q - X = \prod_{r\in\mathbb{F}_q}(X - c)$, Berlekamp deduces the factorization $f(X) = \prod_{r\in\mathbb{F}_q} \gcd(f(X), h(X) - r)$. The method requires computation q GCD's, hence, it cannot be used for arbitrary finite fields. The root-finding method called *Berlekamp Trace Algorithm* (BTA) [4] takes advantage of the factorization

$$X^{q^n} - X = \alpha^{-1} \cdot \prod_{r\in\mathbb{F}_q} (\mathrm{Tr}_{\mathbb{F}_{q^n}:\mathbb{F}_q}(\alpha \cdot X) - r)\,.$$

The roots of f are separated by computing GCDs with the different factors for some random α. This method is recursive and probabilistic and may be applied to large finite fields with small characteristics. It is still one of the most efficient methods today.

Moenck [5] uses resultants to determine the α's and r's, leading to non-trivial GCD's in the above methods. He also introduces the *Subgroup Refinement Algorithm* to pursue further the factorization of the field equation considered by Legendre by considering successive subgroups of $\mathbb{F}_p^*$ for some special primes p. Rabin [6] proposes a method close to Legendre's.

Cantor and Zassenhaus [7] suggest considering random polynomials $h(X)$ instead of $X + r$ in Legendre's decomposition in order to factor a polynomial f with factors of equal degree by means of GCD's. The method is probabilistic and is one of the most efficient methods today.

Berlekamp [8] proposes the *Least Affine Multiple method* (LAM), which consists of computing the least affine multiple of f, then exhaustively searching the roots of f among the roots of this polynomial.

Menezes, van Oorschot and Vanstone [9], [10] combines BTA with the LAM method. Moreover, they [11] generalize Moenck's Subgroup Refinement Algorithm [5] and introduce the *Affine Refinement Method* (ARM) by mirroring the successive refinement of a subgroup of $\mathbb{F}_q^*$ in Moenck's algorithm by refinement of linear subspaces of $\mathrm{AG}(q, n)$, the n-dimensional affine geometry of $\mathbb{F}_q$. As described in their papers, the ARM makes use of the LAM method.

Shoup[12] proposes a deterministic generalization of BTA for the factorization and root-finding problems. Its worst-case running time complexity is similar to the average running time

of the best-known probabilistic methods such as BTA and Cantor-Zassenhaus.

Niederreiter[13] proposes an algorithm based on the resolution of differential equations over finite fields. As mentioned by several authors [14], this algorithm is highly related to Berlekamp's factoring algorithm.

Finally, Petit[15] recently introduced a method called the *Successive Resultants Algorithm* (SRA), which is also closely related to ARM, as we shall see.

Our contribution. We review the affine refinement method (ARM), the successive resultant algorithm (SRA), and Berlekamp's trace algorithm (BTA) and highlight their inter-relationships.

The affine refinement method was designed at a time when fast arithmetic was mostly a theoretical curiosity. As pointed out in [15], the method is, at best, as good as BTA with fast arithmetic, and it is indeed no longer used in practice. We provide a variant of ARM that does not require the computation of the least affine multiple of f, and is therefore significantly more efficient. Indeed, our variant has the same asymptotic complexity as BTA, up to constant factors and some precomputation (which is also present in the original algorithm); whereas the original method is only interesting for very small degree polynomials. It also has the same complexity as SRA.

With respect to the original paper [15], our variant of the successive resultant algorithm adds a squarefree reduction step in the last loops of the first phase to reduce the degrees of the polynomials involved. This enables early termination of the algorithm, which is especially effective on large-degree polynomials. We also introduce an ad hoc algorithm to perform the special resultant computations required by SRA, and we compute the roots in the second step with multi-evaluation rather than by solving small-degree equations. Put together, our modifications improve slightly the asymptotic complexity, and give an algorithm efficient and easier to implement.

We also provide methods to decrease the precomputation costs of both methods significantly when the field extension degree is composite and a method to reduce it down to the cost of the other steps in the case of ARM.

Finally, we show that SRA and ARM are, in a sense, dual to each other and that our precomputation-free variant of ARM is, in fact, a generalized version of BTA.

Outline. The paper is organized as follows. In Sec 2, we review basic facts about the finite affine geometry $\mathrm{AG}(q,n)$. In Sec,3, we present the three root-finding algorithms based on the structure of $\mathrm{AG}(q,n)$: ARM, SRA, and BTA. In Sec 4, we present some refinements of those algorithms that apply in special cases. In Sec 5 ,we present experimental results.

II. The affine geometry AG(q,n)

The main ingredient of the algorithms we present is the *finite affine geometry* $\mathrm{AG}(q,n)$, i.e., the set of all vector subspaces of $\mathbb{F}_q^n$ and their translates. Fixing a basis of $\mathbb{F}_{q^n}$ over $\mathbb{F}_q$, we can identify the elements of $\mathbb{F}_{q^n}$ with the points of $\mathrm{AG}(q,n)$. The relationship between $\mathrm{AG}(q,n)$ and the problem of root finding was originally highlighted in [16].

Approximating $\mathbb{F}_{q^n}$ by a flag. Let $\{v_1,\dots,v_n\}$ be any basis of $\mathbb{F}_{q^n}/\mathbb{F}_q$. To this basis, we associate the flag of linear subspaces $V_0 \subset V_1 \subset \cdots \subset V_n$ defined by

$$V_i = \mathbf{span}(v_1,\dots,v_i), \tag{1}$$

so that $\dim V_i = i$ and $\#V_i = q^i$. To each of the subspaces we associate its minimal polynomial that we denote by L_i. Then we have the well know relation (see [8, Ch. 11])

$$L_0(X) = X,\;\; L_i(X) = (X^q - L_{i-1}(v_i)^{q-1}X)\circ L_{i-1}(X), \tag{2}$$

and in particular $L_n = X^{q^n} - X$.

Notice that, by definition, each L_i defines a linear map $\mathbb{F}_{q^n} \to \mathbb{F}_{q^n}$, with kernel V_i. It will be convenient to encode this information in an $n \times n$ matrix, thus we define $\gamma_{i,j} = L_i(v_j)$, and

$$\Gamma = \begin{pmatrix} \gamma_{0,1} & \cdots & \gamma_{0,n} \\ \vdots & & \vdots \\ \gamma_{n-1,1} & \cdots & \gamma_{n-1,n} \end{pmatrix}. \tag{3}$$

Observe that by definition $\gamma_{i,j} = 0$ whenever $j \le i$, hence Γ is an upper triangular matrix associated to a linear map $\mathbb{F}_{q^n} \to \mathbb{F}_{q^n}^n$ sending any $\delta \in \mathbb{F}_{q^n}$ to the vector $\big(L_0(\delta),\dots, L_{n-1}(\delta)\big)$.

Now define V_i^* as the image space of L_i, i.e., the subspace generated by the elements of the i-th row of Γ. It is easily verified that $\dim V_i^* = n - i$ and that $\{\gamma_{i,i+1},\dots,\gamma_{i,n}\}$ is a basis. Then define L_i^* as the minimal polynomial of V_i^*. It is shown in [8, Ch. 11] that L_i^* is the unique linearized polynomial such that

$$(L_i^* \circ L_i)(X) = (L_i \circ L_i^*)(X) = X^{q^n} - X\,, \tag{4}$$

where L_i^* is called the *dual* of L_i.

Lemma 1: The coefficients of the matrix Γ can be computed using $O(n^2 \log q)$ operations over $\mathbb{F}_{q^n}$.

By the recursive definition of L_i, it is clear that

$$\gamma_{i,j} = \begin{cases} v_j & \text{for } i = 0, \\ \gamma_{i-1,j}^q - \gamma_{i-1,i}^{q-1}\gamma_{i-1,j} & \text{for } i > 0. \end{cases} \tag{5}$$

Thus, each $\gamma_{i,j}$ can be computed from the previous ones using $O(\log q)$ operations, and there is a total $O(n^2)$ of them to compute.

The eigenvalues of Γ play a special role in our algorithms, thus we define $\beta_i = \gamma_{i-1,i}$ and $\alpha_i = \beta_i^{q-1}$ for any $1 \le i \le n$. We deduce a decomposition

$$X^{q^n} - X = (X^q - \alpha_n X)\circ\cdots\circ(X^q - \alpha_1 X). \tag{6}$$

The affine geometry of $\mathbb{F}_{q^n}$. Let now $\rho \in \mathbb{F}_{q^n}$, such that $\rho = \sum_j r_j v_j$. For any V_i we define the affine space (also called an i-flat)

$$V_{i,\rho} = V_i + \rho. \tag{7}$$

By construction, the reunion of all i-flats for any i is isomorphic to $\mathrm{AG}(q,n)$, and we call it the *affine geometry of* $\mathbb{F}_{q^n}$. We

also define the polynomial $M_{i,\rho}$ as the minimal polynomial of $V_{i,\rho}$, hence

$$M_{i,\rho}(X) = L_i(X-\rho) = L_i(X) - \sum_{j>i} r_j \gamma_{i,j}. \quad (8)$$

Observe that $M_{i,\rho} = M_{i,\rho'}$, and thus $V_{i,\rho} = V_{i,\rho'}$, if and only if $\rho - \rho' \in V_i$. Hence any $V_{i,\rho}$ can be represented canonically by taking ρ of the form $\rho = \sum_{j>i} r_j \upsilon_j$. Hence, there are exactly q^{n-i} distinct i-flats, each of size q^i.

By definition,n we have

$$V_{i,\rho} = \bigcup_{c\in\mathbb{F}_q} V_{i-1,\rho+c\upsilon_i}, \quad (9)$$

hence

$$M_{i,\rho}(X) = \prod_{c\in\mathbb{F}_q} M_{i-1,\rho+c\upsilon_i}(X). \quad (10)$$

This defines a decomposition of $X^{q^n} - X$ into a product tree, where each node at level i is a $M_{i,\rho}$, defined by a $\rho = \sum_{j>n-i} r_j \upsilon_j$, and has q children, defined by the elements $c\upsilon_{n-i} + \rho$ for $c \in \mathbb{F}_q$. In particular, the leaves are the linear polynomials $X - \rho$ for all points $\rho \in \mathbb{F}_{q^n}$.

III. Root finding using AG(q,n)

We are interested in algorithms for the following problem: given a degree d polynomial $f(X)$ with coefficients in a finite field $\mathbb{F}_{q^n}$, find all the roots $\rho \in \mathbb{F}_{q^n}$ such that $f(\rho) = 0$. In this section, we present algorithms based on the structure of the affine geometry described previously.

A. Preliminaries

Complexity notations. In this paper, we use an *algebraic complexity model*, where the running time of an algorithm is measured in terms of the number of operations ($+$, $\times$, $\div$) in $\mathbb{F}_{q^n}$, and its space requirements in terms of number of elements of $\mathbb{F}_{q^n}$ stored. As customary, we use the O-notation to neglect constant factors. All asymptotic complexities will be expressed in terms of the parameters q, n and d.

We denote by $\mathsf{M} : \mathbb{N} \to \mathbb{N}$ a function such that polynomials in $\mathbb{F}_{q^n}[X]$ of degree at most m can be multiplied in $\mathsf{M}(m)$ operations ($+$, $\times$) in $\mathbb{F}_{q^n}$, and we make the usual super-linearity assumptions on M [17, Chapter 8]. Using FFT multiplication, we can take $\mathsf{M}(m) \in O(m \log m \operatorname{loglog} m)$.

Fundamental algorithms. We recall now the complexities of the basic subroutines we shall need next. The greatest common divisor of two polynomials of degree d can be computed in $O(\mathsf{M}(d)\log d)$ operations in $\mathbb{F}_{q^n}$ using a Schönhage-type algorithm [18], [19]. Interpolating a polynomial of degree d from $d+1$ points can be done in $O(\mathsf{M}(d)\log d)$ operations using the algorithm described in [17, Ch. 10]. Similarly, evaluating a polynomial of degree d at $d+1$ points can also be done in $O(\mathsf{M}(d)\log d)$ operations using the multi-point evaluation algorithm of [17, Ch. 10].

In root-finding literature, it is common to assume that all the irreducible factors of the polynomial are linear and distinct. This can be enforced by replacing $f(X)$ with $\gcd(X^{q^n} - X, f(X))$. The main cost of this GCD is the computation of $X^{q^n} \bmod f(X)$. This can be achieved with a standard square-and-multiply algorithm in $O(n\mathsf{M}(d)\log q)$, or (when n is large) via modular composition (see [20], [21]). Another fundamental operation on polynomials is square-free factorization: given a polynomial f compute the product of all the irreducible factors of f, each taken with multiplicity one. This can be done in $O(\mathsf{M}(d)\log d)$ operations using the algorithm of [17, Ch. 14.6].

Recall that, given polynomials f and g, the resultant

$$\operatorname{Res}_X(g(X) - Y, f(X))$$

has the same degree as f, and its roots are the elements $g(\rho)$ for each root ρ of f, taken with multiplicity. We shall need an algorithm to compute a special resultant of this kind:

$$\operatorname{Res}_X(X^q - \beta^{q-1}X - Y, f(X))$$

for some $\beta \in \mathbb{F}_{q^n}$. This is described in Algorithm 2, and relies upon the variant of multi-point evaluation described in Algorithm 1.

Algorithm 1 Polynomial evaluation at special points

Input: A polynomial $f \in \mathbb{F}_{q^n}[X]$ of degree d, $\delta_0, \ldots, \delta_d \in \mathbb{F}_{q^n}$ and $\beta \in \mathbb{F}_{q^n}$.
Output: $f(\delta_i + c\beta)$ for all $0 \le i \le d$ and $c \in \mathbb{F}_q$.

1: Let $\alpha = \beta^{q-1}$;
2: Compute $\bar{f}(X,Y) = f \mod X^q - \alpha X - Y$;
3: Let $\bar{f}(X,Y) = \sum_j \bar{f}_j(Y) X^j$;
4: Compute $\Delta_i = \delta_i^q - \alpha\delta_i$ for $0 \le i \le d$;
5: **for** $0 \le j < q$ **do**
6: Compute $\eta_{i,j} = \bar{f}_j(\Delta_i)$ for $0 \le i \le d$;
7: Let $f_i(X) = \sum_j \eta_{i,j} X^j = \bar{f}(X, \Delta_i)$;
8: **for** $0 \le i \le d$ **do**
9: Compute $\varepsilon_{i,c} = f_i(\delta_i + c\beta)$ for all $c \in \mathbb{F}_q$;
10: **return** $\varepsilon_{i,c}$ for $0 \le i \le d$ and $c \in \mathbb{F}_q$.

Lemma 2: Algorithm 1 is correct. On input a polynomial of degree $d \gg q$ and $O(d)$ evaluation points, it computes its output using $O(\mathsf{M}(d)\log d + q^2\mathsf{M}(d/q)\log d + q^2 d)$ operations over $\mathbb{F}_{q^n}$.

Observe that $\prod_{c\in\mathbb{F}_q}(X - \delta_i - c\beta) = X^q - \alpha X - \Delta_i$, with Δ_i defined as in step 4. Hence

$$f_i(X) = f(X) \mod \prod_{c\in\mathbb{F}_q}(X - \delta_i - c\beta)$$

Correctness follows immediately.

We now analyze the complexity. In step 2, we reduce f so that $\bar{f}$ has degree $< q$ in X. This reduction can be computed in $O(\mathsf{M}(d)\log d)$ using a divide-and-conquer approach.

By construction, the polynomials $\bar{f}_j(Y)$ have degree $O(d/q)$, hence step 6 can be computed in $O(q\mathsf{M}(d/q)\log d)$ splitting the points Δ_i in q batches, and using q classical multipoint evaluations. Hence, the overall cost of the for loop is $O(q^2\mathsf{M}(d/q)\log d)$.

Finally, step 9 can be computed again by multipoint evaluation in degree $O(q)$. However, we assume that q is too small

to apply an asymptotically fast algorithm. Thus we account $O(q^2)$ operations for this step, giving $O(q^2 d)$ operations for the whole loop. All other steps have negligible cost.

Algorithm 2 Resultant with a special polynomial

Input: A polynomial $f \in \mathbb{F}_{q^n}[X]$ of degree $d < q^{n-1}$, an element $\beta \in \mathbb{F}_{q^n}$.
Output: The resultant $\mathrm{Res}_X\big(X^q - \beta^{q-1}X - Y, f(X)\big)$.
1: Select points $\delta_0, \ldots, \delta_d$ s.t. $\beta \notin \mathbf{span}(\delta_0, \ldots, \delta_d)$;
2: Compute $\varepsilon_{i,c} = f(\delta_i + c\beta)$ for $0 \le i \le d$ and $c \in \mathbb{F}_q$;
3: **return** g such that $g\left(\prod_c \delta_i + c\beta\right) = (-1)^{qd} \prod_c \varepsilon_{i,c}$.

Lemma 3: Algorithm 2 is correct. On input, a polynomial of degree $d < q^{n-1}$, it computes its output using $O(\mathsf{M}(d)\log d + q^2\mathsf{M}(d/q)\log d + q^2 d)$ operations over $\mathbb{F}_{q^n}$.

It is well known that the resultant commutes with evaluation, i.e., if

$$g(Y) = \mathrm{Res}_X\big(X^q - \beta^{q-1}X - Y, f(X)\big), \tag{11}$$

then

$$g(\Delta) = \mathrm{Res}_X\big(X^q - \beta^{q-1}X - \Delta, f(X)\big) \tag{12}$$

for any $\Delta \in \mathbb{F}_{q^n}$. Our algorithm is just a specialization of the classical evaluation-interpolation approach to compute bivariate resultants. We know that the polynomial g has degree d, thus we only need to evaluate at $d+1$ points.

Let now $\Delta_i = \delta_i^q - \beta^{q-1}\delta_i$. By linearity we have

$$\begin{aligned}\mathrm{Res}_X\big(X^q - \beta^{q-1}X - \Delta_i, f(X)\big) = \\ \mathrm{Res}_X\Big(\prod_{c\in\mathbb{F}_{q^n}} (X - \delta_i - c\beta), f(X)\Big) = \\ \prod_{c\in\mathbb{F}_{q^n}} \mathrm{Res}_X\big(X - \delta_i - c\beta, f(X)\big) = \\ (-1)^{qd} \prod_{c\in\mathbb{F}_{q^n}} f(\delta_i + c\beta),\end{aligned} \tag{13}$$

where the second and third equalities come from the multiplicativity of the resultant. Observe that $\Delta_0, \ldots, \Delta_d$ are all distinct otherwise one would have $\beta^{q-1} = (\delta_i - \delta_j)^{q-1}$ for some i and j, contraddicting $\beta \notin \mathbf{span}(\delta_0, \ldots \delta_d)$. The correctness of the algorithm then follows immediately.

We now go to the complexity. To construct the δ_i's, we simply enumerate vectors in a span not containing β. This is possible as long as $d < q^{n-1}$, and costs d operations in $\mathbb{F}_{q^n}$. Step 2 is computed using Algorithm 1, at the cost given in Lemma 2. Finally, step 3 is computed using a classical fast interpolation algorithm at a cost $O(\mathsf{M}(d)\log d)$.

If $d \ge q^{n-1}$, we can't find enough δ_i's for step 1 of Algorithm 1. In this case, Eq. (13) cannot be applied anymore, however, the evaluation-interpolation approach is still possible at a slightly worse cost. Note that this is not a concern for root finding, as exhaustive search is more efficient in this case.

B. Two variants of ARM and SRA

We now present our variants of two root-finding algorithms based on the structure of $\mathrm{AG}(q,n)$. We start with the Affine Refinement Method (ARM) of Menezes, van Oorschot and Vanstone [11], then go to the Successive Resultants Algorithm (SRA) of Petit [15]. For both, we present variants that are more efficient than the original ones. Our presentation highlights a link between the two algorithms that had previously gone unnoticed: in both an abstract and a computational sense, ARM and SRA can be seen as *dual* to each other.

ARM works by *intersecting* the roots of the input polynomial with the subspaces $V_{i,\rho}$. The successive intersections are computed by means of GCDs. SRA works by *projecting* the roots onto the spaces V_i^*. The successive projections are computed by means of resultants.

At an algorithmic level, the duality is embodied in the well-known property

$$\gcd\big(f(X), g(X)\big) \neq 1 \;\Leftrightarrow\; \mathrm{Res}_X\big(f(X), g(X)\big) = 0\,, \tag{14}$$

which implies that ARM isolates a root of f whenever SRA does.

Pre-computation. Both methods take as input a basis $\upsilon_1 \ldots, \upsilon_n$, and use it to define the affine space $\mathrm{AG}(q,n)$ as in Section II. To this end, they compute the matrix Γ of Eq. (3) at a cost of $O(n^2 \log q)$ operations. Note that this phase depends only on the field and may be re-used for the root-finding of several polynomials defined on the same field.

ARM. The method consists in *intersecting* the roots of f with the i-flats $V_{i,\rho}$, i.e, with the fibers by L_i of each element $\ell_{i,\rho} \in V_i^*$. This is achieved by computing the non-constant GCDs among the set

$$\gcd(L_i(X) - \ell_{i,\rho}, f(X)) \text{ for any } \ell_{i,\rho} \in V_i^*\,, \tag{15}$$

for i starting from n and going down to 0. It ultimately leads to the roots of f because the GCD's satisfying

$$\gcd(L_0(X) - \ell_{0,\rho}, f(X)) = \gcd(X - \ell_{0,\rho}, f(X)) \neq 1$$

yield the linear factors of f. Note that

$$\gcd(L_i(X) - \ell_{i,\rho}, f(X)) \neq 1 \tag{16}$$

implies that there is a root of f in $V_{i,\rho}$. In practice, these GCDs are computed in a recursive way. Let us first define

$$f_{i,\rho}(X) = \gcd(L_i(X) - L_i(\rho), f(X))$$

where $\rho \in \mathbb{F}_{q^n}$. Hence $f_{i,\rho}(X) = f_{i,\rho'}(X)$ if and only if $\rho - \rho' \in V_i$. Using Eq. (15) and the fact that $V_n^* = \{0\}$, $f_{n,0}(X)$ is first computed, i.e.,

$$f_{n,0}(X) = \gcd(L_n(X), f(X)) = \gcd(X^{q^n} - X, f(X))\,.$$

The recursive step is based on Eq. (10) and works as follows: for any $L_i(\rho) \in V_i^*$ such that $f_{i,\rho}(X)$ is neither a constant polynomial nor a linear factor,

$$\begin{aligned} f_{i,\rho}(X) &= \textstyle\prod_{c\in\mathbb{F}_q} \gcd(M_{i-1,\rho+c\cdot\upsilon_i}(X), f_{i,\rho}(X)) \\ &= \textstyle\prod_{c\in\mathbb{F}_q} \gcd(L_{i-1}(X) - L_{i-1}(\rho + c\cdot\upsilon_i), f_{i,\rho}(X)) \\ &= \textstyle\prod_{c\in\mathbb{F}_q} f_{i-1,\rho+c\cdot\upsilon_i}(X) \end{aligned}$$

This procedure builds a tree where, at each level, only the nodes containing a polynomial of degree strictly greater than 1 have children. Said otherwise, the leaves of the tree are either constant polynomials or the distinct linear factors of f (without their algebraic multiplicity). ARM is summarized in Algorithm 3.

Algorithm 3 Affine Refinement Method

Input: A polynomial $f \in \mathbb{F}_{q^n}[X]$ of degree d,
A basis $v_1, \ldots, v_n$ of $\mathbb{F}_{q^n}$,
The matrix $\Gamma = (\gamma_{i,j})$ defined in (3).
Output: The roots of f in $\mathbb{F}_{q^n}$.
1: Set $L_0 = X$;
2: **for** $1 \leq i \leq n$ **do**
3: Compute $L_i = L_{i-1}^q - \gamma_{i-1,i}^{q-1} L_{i-1} \mod f$;
4: Initialize tree with root $f_{n,0} = f \mod L_n$;
5: **for** $n \geq i \geq 1$ **do**
6: **for** any node $f_{k,\rho}$ of degree > 1 in the tree **do**
7: Compute $\ell_{i-1,\rho} = L_{i-1}(\rho) = \sum_j r_j \gamma_{i-1,j}$;
8: **for** any leaf in the tree $f_{i,\rho}$ of degree > 1 **do**
9: Compute $\bar{L}_\rho = L_{i-1} \mod f_{i,\rho}$;
10: **for** any $c \in \mathbb{F}_q$ **do**
11: $f_{i-1,\rho+c\cdot v_i} = \gcd(\bar{L}_\rho - \ell_{i-1,\rho} - c\gamma_{i-1,i}, f_{i,\rho})$;
12: **if** all leaves have degree ≤ 1 **then**
13: **return** The root of each leaf of degree 1.

Lemma 4: Algorithm 3 (ARM) is correct. If the root tree for the input f and $v_1, \ldots, v_n$ has height h, it computes its output using $O(n\mathsf{M}(d)\log q + h^2 d + hq\mathsf{M}(d)\log d)$ operations over $\mathbb{F}_{q^n}$.

Correctness has been discussed above. Concerning complexity, the computation of the polynomials L_i in step 3 is done in $O(\mathsf{M}(d)\log q)$ for each polynomial, contributing $O(n\mathsf{M}(d)\log q)$ to the total.

Now we treat the steps inside loop 5. Each of the complexities below is multiplied by h in the total count.

In step 7, we need to compute $L_{i-1}(\rho)$ for each root approximation ρ in the tree. If ρ is the approximation associated to a node, and ρ' is the approximation associated to its parent, then $\rho - \rho' = cv_j$ for some $c \in \mathbb{F}_q$ and some basis vector v_j. Hence, knowing the elements $\gamma_{i-1,j}$, the value $L_{i-1}(\rho)$ can be computed from $L_{i-1}(\rho')$ using only one field operation. At any iteration, there are at most hd nodes in the tree, hence a cost of $O(hd)$.

Step 9 can be computed putting all $f_{i,\rho}$ together in a subproduct tree, as in the classical multi-point evaluation algorithm. Reducing modulo the subproduct tree costs then $O(\mathsf{M}(d)\log d)$.

Finally, in step 11 the degrees of the polynomials $f_{i,\rho}$ sum up to d. For each of these, we compute q GCDs with polynomials of similar degree. Each of these polynomials can be computed in $O(\mathsf{M}(d_\rho)\log d_\rho)$, where d_ρ is the degree of $f_{i,\rho}$. Summing over all polynomials and using the superlinearity of the function M, we bound this by $O(q\mathsf{M}(d)\log d)$.

SRA. The method consists in *projecting* the roots of f onto the spaces V_i^* for i starting from n and going down to 0. It is achieved by the computation of the values $\ell_{i,\rho} \in V_i^*$ annihilating

$$\mathrm{Res}_X(L_i(X) - \ell_{i,\rho}, f(X)). \tag{17}$$

It ultimately leads to the roots of f because

$$\mathrm{Res}_X(L_0(X) - \ell_{0,\rho}, f(X)) = \mathrm{Res}_X(X - \ell_{0,\rho}, f(X)) = 0$$

means that $\ell_{0,\rho}$ is a root of f. Note that

$$\mathrm{Res}_X(L_i(X) - \ell_{i,\rho}, f(X)) = 0 \tag{18}$$

means that there is a root of f in $V_{i,\rho}$. Comparing this with Eq. (16), the duality between ARM and SRA becomes apparent. Indeed, $\gcd(L_i(X) - \ell_{i,\rho}, f(X)) \neq 1$ if and only if $\mathrm{Res}_X(L_i(X) - \ell_{i,\rho}, f(X)) = 0$.

Like ARM, SRA is a two-swipe algorithm. We first compute the polynomials $f^{(i)}$ having as roots the projections onto V_i^* of the roots of f, i.e.,

$$f^{(i)}(Y_i) = \mathrm{Res}_X(L_i(X) - Y_i, f(X)) \text{ for } 0 \leq i \leq n. \tag{19}$$

The polynomial $f^{(i)}$ is computed from $f^{(i-1)}$ by projecting its roots onto V_i^*. By the definition of L_i, we have $f^{(0)} = f$ and

$$f^{(i)}(Y_i) = \mathrm{Res}_{X_{i-1}}(X_{i-1}^q - \alpha_i X_{i-1} - Y_i, f^{(i-1)}(X_{i-1})).$$

At each iteration, some roots of $f^{(i-1)}$ may be projected to the same root of $f^{(i)}$. We apply a square-free factorization algorithm to only keep one representative of those roots. Doing this allows us to stop the descent before $f^{(n)}$ is computed. Indeed, we can verify that $f^{(i)} = L_i^*$ simply by checking the degree of $f^{(i)}$, then we know that the roots of $f^{(i)}$ are all the elements of V_i^*. In the worst case, this will happen at the last step, when $f^{(n)}(X) = X = L_n^*(X)$, and the only root is 0. In practice, the square-free factorization is unlikely to significantly reduce the degree of f before the last $\log d$ iterations, it is thus advisable to skip it in the first ones.

After the $f^{(i)}$'s are computed, SRA iteratively computes their roots starting from the last. The roots of $f^{(i-1)}$ are deduced from those of $f^{(i)}$ using Eq. (2). For any root ℓ_i of $g^{(i)}$ belonging to V_i^*, we check whether $g^{(i-1)}(\ell_{i-1}) = 0$ for any $\ell_{i-1} \in V_{i-1}^*$ such that $\ell_{i-1}^q - \alpha_i \cdot \ell_{i-1} = \ell_i$. This procedure builds a tree where, at level i the nodes are the roots of $g^{(i)}$. The leaves of the tree at level 0 are the roots of f.

Lemma 5: Algorithm 4 (SRA) is correct. If the root tree for the input f and $v_1, \ldots, v_n$ has height h, it computes its output using $O\left(h(\mathsf{M}(d)\log d + q^2\mathsf{M}(d/q)\log d + q^2 d + hd)\right)$ operations over $\mathbb{F}_{q^n}$, or more compactly $O(hq\mathsf{M}(d)\log d + h^2 d)$.

Correctness has been discussed above. Concerning complexity, we remark that any step in loop 2 and loop 9 is repeated exactly h times, thus the overall complexity is multiplied by h.

Step 3 can be computed using Algorithm 2. According to Lemma 3, this costs $O(\mathsf{M}(d)\log d + q^2\mathsf{M}(d/q)\log d + q^2 d)$.

Step 4 is computed in $O(\mathsf{M}(d)\log d)$ using a square-free factorization algorithm.

Algorithm 4 Successive Resultant Algorithm

Input: A polynomial $f \in \mathbb{F}_{q^n}[X]$ of degree d,
A basis $\upsilon_1, \ldots, \upsilon_n$ of $\mathbb{F}_{q^n}$,
The matrix $\Gamma = (\gamma_{i,j})$ defined in (3).
Output: The roots of f in $\mathbb{F}_{q^n}$.

1: Let $f^{(0)} = f$;
2: **for** $1 \leq i \leq n$ **do**
3: $\quad \hat{f}^{(i)}(Y) = \mathrm{Res}_X(X^q - \gamma_{i-1,i}^{q-1} X - Y, f^{(i-1)}(X))$;
4: $\quad$ Compute $f^{(i)}$, the square-free part of $\hat{f}^{(i)}$;
5: $\quad$ **if** $\deg f^{(i)} = q^{n-i}$ **then**
6: $\quad\quad$ Set $h = i$;
7: $\quad\quad$ **break** out of the **for** loop;
8: Let $A_h = (\rho_0, \ldots, \rho_{q^{n-h}})$ such that $V_h^* = \big(L_h(\rho_j)\big)_j$;
9: **for** $h > i \geq 0$ **do**
10: $\quad$ **for** any $\rho \in A_i$ **do**
11: $\quad\quad$ Compute $\ell_{i,\rho} = L_i(\rho) = \sum_j r_j \gamma_{i,j}$;
12: $\quad E_i = \big(f^{(i)}(\ell_{i,\rho} + c\gamma_{i,i+1})$ for any $\rho \in A_{i+1}$ and $c \in \mathbb{F}_q\big)$;
13: $\quad$ **for** any $\rho \in A_i$ **do**
14: $\quad\quad$ Add $\rho + c\upsilon_{i+1}$ to A_i only if $f^{(i)}(\rho + c\gamma_{i,i+1}) = 0$.
15: **return** A_0.

The elements $L_i(\rho)$ in step 11 are computed using the tree structure of the roots, like in ARM, thus requiring only one field operation per value. There are at most hd approximations at each iteration thus this step contributes $O(hd)$ operations.

Finally, step 12 can be computed using the multi-point evaluation algorithm 1, at the same cost of step 3.

Using the two previous lemmas, we can now easily state the worst-case the complexity of both ARM and SRA, indeed for both algorithms, the height h equals n in this case. Unsurprisingly, the two algorithms have very similar complexities.

Theorem 1: In the worst case, both ARM and SRA compute their output using $O(nq\mathsf{M}(d)\log d + n^2 d)$.

C. BTA

For completeness, we review here Berlekamp's trace algorithm (BTA), and recall how it is related to ARM. BTA attempts to decompose f into smaller degree factors using the product equality

$$f(X) = \prod_{r \in \mathbb{F}_q} \gcd\big(\mathrm{Tr}_{\mathbb{F}_{q^n}:\mathbb{F}_q}(\alpha X) - r, f(X)\big) \qquad (20)$$

for randomly chosen $\alpha \in \mathbb{F}_{q^n}$.

For any basis of $\mathbb{F}_{q^n}$ over $\mathbb{F}_q$, the equality (20) leads to a non-trivial factorization of f for at least one basis element α [4]. The polynomial $\mathrm{Tr}_{\mathbb{F}_{q^n}:\mathbb{F}_q}(\alpha X) \mod f$ is computed iteratively using $O(n\mathsf{M}(d)\log q)$ operations. If the result is not a constant, the q GCDs are computed in $O(q\mathsf{M}(d)\log d)$. Otherwise, the algorithm must be restarted with a different α. The process is then recursively applied to the factors. If the α are carefully managed, it can be shown that the worst-case complexity is $O((n\mathsf{M}(d)\log q + q\mathsf{M}(d)\log d)n)$ operations (see [11]).

Compared to ARM, BTA does not decompose the polynomial $X^{q^n} - X$ completely but instead uses the partial decomposition

$$X^{q^n} - X = \alpha^{-1}(X^q - X) \circ \mathrm{Tr}_{\mathbb{F}_{q^n}:\mathbb{F}_q}(\alpha X),$$

or

$$(X^{q^n} - X) = (X^q - \alpha^{q^{n-1}-1} X) \circ \alpha^{-q^{n-1}} \mathrm{Tr}_{\mathbb{F}_{q^n}:\mathbb{F}_q}(\alpha X).$$

If in ARM we choose a basis $\{\upsilon_1, \ldots, \upsilon_n\}$ such that υ_i/α has trace 0 for all $i < n$ and υ_n is such that $\mathrm{Tr}_{\mathbb{F}_{q^n}:\mathbb{F}_q}(\alpha\upsilon_n) = 1$, then we exactly have $L_{n-1}(X) = \alpha^{-q^{n-1}} \mathrm{Tr}_{\mathbb{F}_{q^n}:\mathbb{F}_q}(\alpha X)$ and

$$M_{n-1,r}(X) = \alpha^{-q^{n-1}} \left(\mathrm{Tr}_{\mathbb{F}_{q^n}:\mathbb{F}_q}(\alpha X) - r\right).$$

Up to the constant factors $\alpha^{-q^{n-1}}$, the polynomials used in BTA and in the last level of ARM are, therefore, identical.

From a geometric point of view, each trace computed by BTA separates the roots of f into q different affine sets of size q^{n-1}, corresponding to the roots of $\mathrm{Tr}(\alpha X) - r$ for all $r \in \mathbb{F}_q$. After repeating this separation process on the factors for $O(\log_q d)$ randomly chosen α which are linearly independent, the intersections of these affine sets are likely to contain one root of f, leading to its linear factors.

Note the similarity with ARM and SRA. All algorithms separate the roots of f into affine spaces $V_{i,\rho}$ of smaller and smaller dimensions, such that $V_{i,\rho} \subset V_{i+1,\rho}$. While ARM and SRA represent the affine subspaces with minimal polynomials, BTA represents them by means of intersections of affine polynomials of the form $\mathrm{Tr}(\alpha X) - r$.

D. Average running time

On average, ARM and BTA stop at height h, before reaching the bottom of the product tree. SRA, on the other hand, stops at height h, before reaching the root of the same tree. Her,e we give estimates for the average height h in each algorithm.

Average case running time of ARM. As already observed by Menezes, Van Oorschot and Vanstone [11], the statistical behavior of the height H_d of the tree built by ARM on split polynomials with d distinct roots chosen at random is related to the height of a data structure called *digital trie* in the Bernoulli model. This observation allows them to upper bound the expected value $\mathrm{E}(H_d)$ by $2\log_q d + O(1)$ [1]. Using the results of Szpankowski [22] it is straightforward to show that

$$\mathrm{E}(H_d) \leq \min(2\log_q d + 1 + \frac{1 - \ln 2}{\ln q} + O(d^{-1}), n).$$

and assuming further that $d < q^{n/2}$ we have

$$\Pr(H_d < x) \approx \exp(-\exp(-\ln q \cdot (x - 2 \cdot \log_q d)))$$

and

$$\mathrm{Var}\, H_d \approx \frac{\pi^2}{6 \cdot (\ln q)^2}.$$

[1] There is typo in the manuscript: $2\log_q n + O(1)$ should read $2\log_q d + O(1)$. The evaluation of the complexity of the Affine method, ARM, and BTA should be corrected accordingly.

Those two results express the fact that the number of iterations of the algorithm does not diverge much from its average most of the time.

Average case running time of SRA. In SRA, for a fixed polynomial of degree d, the number of iterations h_d is the smallest value for which $\deg f^{(h)} = q^{n-h}$. Note that $\deg f^{(i)} \leq d$ for any i, thus necessarily $h \geq n - \log_q d$.

Thus, h_d is the smallest value for which the projections through L_i of all the roots of f cover V_h^*. Therefore, $n - h_d$ is the length of the shortest path in the associated digital trie. B.Pittel [23] proved that the random variable H'_d representing the shortest path of the digital trie with d records generated according to the Bernoulli model satisfies $H'_d / \log_q d \underset{d \to +\infty}{\longrightarrow} 1$ a.s. It follows that the random variable H_d representing the number of iterations is $H_d \approx n - \log_q d$.

Average case running time of BTA. This analysis is due to Menezes, Van Oorschot, and Vanstone [11] and is also related to the statistical properties of digital tries. It can be shown that the average complexity is $O((n\mathsf{M}(d) \log q + q\mathsf{M}(d) \log d) \log d)$.

IV. Variant and Extensions

A. Improving the worst-case complexity

ARM may perform poorly in the worst case. When the number of iterations h is maximal, the term corresponding to the computation of the $\ell_{i,\rho}$'s in step 7 becomes $O(n^2 d)$, which may become the bottleneck of the algorithm. Note that the same computation is also performed in SRA.

If a normal basis with fast arithmetic is available for $\mathbb{F}_{q^n}/\mathbb{F}_q$ (e.g., using the generic construction of [24]), the $\ell_{i,\rho}$'s may be computed more efficiently.

At every iteration, for each node $f_{i+1,\rho}$ that needs to be branched, the values $\ell_{i,\rho}$ corresponding to its q children may be computed from $\ell_{i+1,\rho} = L_{i+1}(\rho)$ by solving the equation

$$\ell_{i,\rho}^q - \beta_i^{q-1} \ell_{i,\rho} = \ell_{i+1,\rho}\,,$$

which follows from the definition of the L_i's. Rewriting the above equation as

$$(\beta_i^{-1}\ell_{i,\rho})^q - (\beta_i^{-1}\ell_{i,\rho}) = \beta_i^{-q}\ell_{i+1,\rho}\,,$$

it follows that the set of $\ell_{i,\rho}$'s corresponding to the q children of the node $f_{i+1,\rho}$ is given by

$$\{\beta_i \cdot \ell \mid \ell^q - \ell = \beta_i^{-q}\ell_{i+1,\rho}\}\,.$$

In ARM and SRA, we only need to determine one such element. We do this by first computing the right side of the equation in $O(\log q)$ operations in $\mathbb{F}_{q^n}$. If the field extension $\mathbb{F}_{q^n}/\mathbb{F}_q$ is represented by a normal basis, a particular solution of the equation may be derived by solving a bidiagonal system in n variables belonging to $\mathbb{F}_q$, which requires $O(n)$ operations in $\mathbb{F}_q$ or $O(1)$ operations in $\mathbb{F}_{q^n}$. Finally, the solution is multiplied by β_i again at a cost $O(1)$. It follows that for each node $f_{i+1,\rho}$, the computation of one child $\ell_{i,\rho}$ costs $O(\log q)$ operations in $\mathbb{F}_{q^n}$. As the number of leaves at every iteration is upper bounded by d, the overall cost for computing all the $\ell_{i,\rho}$'s in step 7 is $O(d \log q \log_q d)$ on average and $O(nd \log q)$ in the worst case, which is no more a bottleneck.

B. Variant with partial precomputation

The pre-computation step of ARM and SRA costs $O(n^2 \log q)$ to determine the α_i's involved in the decomposition of the field equation.

Performing on average $2\log_q d$ iterations, most of the time, ARM only needs the last $2\log_q d$ values α_i to split f completely. In what follows, h is the *a priori* bound on the expected number of iterations of ARM, typically $c \cdot \log_q d$ with $c \geq 2$.

As seen in Eq. (4), for any linearized polynomial L_i there exists a unique linearized polynomial L_i^* such that $(L_i^* \circ L_i)(X) = (L_i \circ L_i^*)(X) = X^{q^n} - X$. Therefore, we start by computing $L_{n-h}^*(X)$ using a basis $\{v_{n-h+1}, \ldots, v_n\}$ of a vector space of dimension h of $\mathbb{F}_{q^n}/\mathbb{F}_q$. We deduce that

$$X^{q^n} - X = (X^q - \alpha_n) \circ \cdots \circ (X^q - \alpha_{n-h+1}X) \circ L_{n-h}(X)\,,$$

where $L_{n-h}(X)$ is unique and not yet computed. This requires $O(h^2 \log q)$ operations in $\mathbb{F}_{q^n}$. For i starting from n and going down to $n-h$, we compute recursively the $L_i(X)$'s which satisfy

$$X^{q^n} - X = (X^q - \alpha_n) \circ \cdots \circ (X^q - \alpha_{i-1}X) \circ L_i(X)\,.$$

or equivalently $L_{i+1}(X) = L_i(X)^q - \alpha_{i+1}L_i(X)$. The $L_i(X)$'s being linearized polynomials, each of these equations reduces to a bidiagonal system, which may be solved at the cost $O(n \log q)$ or $O(hn \log q)$ for the h polynomials L_i. Note that step 3 of algorithm 3 which consists in computing $L_i \bmod f$ need to be slightly modified. The values $X^{q^j} \bmod f$ are first computed in $O(\mathsf{M}(d) n \log q)$ operations, then $L_i \bmod f$ can be computed in $O(dn)$ operations. This does not change the complexity of the step. Step 7 also needs to be modified because we don't have the matrix Γ defined in Eq. (3). Instead of computing in step 7 the values $\ell_{i-1,\rho}$ for all the nodes of the tree, we compute the values $\ell_{i-1,\rho} = L_{i-1}(\rho)$ by simple evaluation of the polynomial L_{i-1} for any any root approximation $\rho = \sum_j r_j v_j$ candidate appearing in step 11. Each value $\ell_{i-1,\rho}$ requires $O(n \log q)$ operations, and there are at most $d \cdot h$ such value to compute. The overall complexity for the computation of these values is $O(hnd \log q)$, which is not a bottleneck in the average case.

Depending on where the roots of f are located, the a priori bound h may, in fact, not be sufficient to completely factor f. In this case, we launch the algorithm on the remaining factors of f at level h with a newly generated vector basis with h' elements. This value should be chosen as a multiple of $\log_q \max(\deg f_{h,\rho})$ where $f_{h,\rho}$ are the remaining factors of f at level h.

Finally, note that it is obvious that SRA may take advantage of the same procedure, but in this case, the trick consists simply of stopping computing the α_i's when the last resultant is computed.

C. Precomputation when n is composite

The variants of ARM and SRA are presented in Section IV-A do not need the full matrix Γ as input. Rather, only the diagonal elements β_i are required to perform the root refinement steps. In this case, we may speed up the precomputation of Lemma 1 by directly computing those elements. It is unknown whether a sub-quadratic algorithm to compute the eigenvalues of Γ exists in general. This section provides an improvement for the case where the degree n is highly composite.

Let $n = ms$ and set $Q = q^s$, we can decompose the field equation $X^{q^{ms}} - X$ in two steps: first, as the equation of the extension $\mathbb{F}_{q^{ms}}/\mathbb{F}_{q^s}$, then by plugging the decomposition of $X^{q^s} - X$ in. Let then $\{\upsilon_1, \ldots, \upsilon_m\}$ be a basis of $\mathbb{F}_{q^{ms}}/\mathbb{F}_{q^s}$, using Lemma 1 we can compute a decomposition

$$X^{Q^m} - X = (X^Q - \beta_m^{Q-1}X) \circ \cdots \circ (X^Q - \beta_1^{Q-1}X) \quad (21)$$

using $O(m^2 \log Q) = O(m^2 s \log q)$ operations.

Observe that for any β_i, we can rewrite the members on the RHS above as

$$X^Q - \beta_i^{Q-1}X = (\beta_i^Q X) \circ (X^Q - X) \circ (X/\beta_i). \quad (22)$$

Now, we can compute recursively a decomposition for $X^Q - X$. Let

$$X^{q^s} - X = (X^q - \delta_s^{q-1}X) \circ \cdots \circ (X^q - \delta_1^{q-1}X) \quad (23)$$

be such a decomposition; we substitute this expression inside Eq. (22). To this extent, remark that

$$(X^q - \delta^{q-1}X) \circ (X/\beta^{q^j}) = (X/\beta^{q^{j+1}}) \circ (X^q - (\delta\beta^{q^j})^{q-1}X).$$

By iterating s times this equality in Eq. (22), we obtain a decomposition

$$X^Q - \beta_i^{Q-1}X = (X^q - (\delta_s \beta_i^{q^{s-1}})^{q-1}X) \circ \cdots \circ (X^q - (\delta_1\beta_i)^{q-1}X).$$

Plugging this equation inside Eq. (21), gives a complete decomposition for $X^{q^{ms}} - X$, at a cost of $O(ms \log q)$ additional operations

Denote by $G(s)$ the cost of computing a decomposition for $X^{q^s} - X$, by the discussion above we deduce that

$$G(ms) = O(m^2 s \log q) + G(s).$$

We can apply this technique recursively to $\mathbb{F}_{q^s}$, hence if $n = n_1 \cdots n_r$, we can compute a decomposition for $X^{q^n} - X$ using $O(n_{\min} n \log q + n_{\max}^2 \log q)$ operations, where $n_{\min}$ is the smallest of the factors of n, and $n_{\max}$ is the largest.

Projecting to smaller fields. We can further refine this decomposition so that the polynomial $L_{(m-1)s}$ is a projection to the field $\mathbb{F}_{q^s}$. Indeed, if we take a basis $\{\upsilon_1, \ldots, \upsilon_m\}$ of $\mathbb{F}_{q^{ms}}/\mathbb{F}_{q^s}$ such that $\upsilon_1, \ldots, \upsilon_{m-1}$ all have trace 0, we obtain a decomposition

$$X^{Q^m} - X = (X^Q - X) \circ \mathrm{Tr}_{\mathbb{F}_{Q^m}/\mathbb{F}_Q}(X).$$

Thus $L_{(m-1)s}(X) = \mathrm{Tr}_{\mathbb{F}_{Q^m}/\mathbb{F}_Q}(X)$ is a projection onto $\mathbb{F}_Q$. This construction can now be iterated on $X^Q - X$, thus providing a decomposition of $X^{q^n} - X$ given by projections to subfields that get progressively smaller.

This construction is especially interesting for SRA because it forces the polynomials $f^{(i)}$ to have coefficients in smaller subfields. Although it does not provide an asymptotic improvement, it is interesting in practice when the field arithmetic of $\mathbb{F}_{q^n}$ is designed to take advantage of working in subfields, e.g., when using towers of finite fields [25], [26].

Fig. 1. Timings in seconds for $\mathbb{F}_{2^{10}}$ (left) and $\mathbb{F}_{5^4}$ (right). Abscissa is polynomial degree.

V. Experimental results

To compare the behavior of the various algorithms, we implemented ARM, SRA and BTA in the computer algebra system Sage [27], and compared them with the default root-finding algorithms implemented in Pari/GP [28] and NTL [29]. Due to the limitations of these systems, we could only test the algorithms as described in Section III: Indeed, we did not have at our disposal any implementation of normal basis arithmetic to implement the variant described in Section IV-A.

As expected, the dominant costs for ARM were the computation of the polynomials in Step 3 and the GCD calculations in Step 11. The dominant costs for SRA were the resultant computations in Step 3 and the evaluations in Step 12. Unsurprisingly, ARM and BTA showed remarkably close performances.

We tested our algorithms on an Intel Core i5-3427U CPU at 1.80GHz. The source code and the test cases can be downloaded from http://github.com/defeo/root_finding/. We selected various base fields $\mathbb{F}_{q^n}$ of characteristic 2 and 5. We ran all algorithms on randomly selected split (non-necessarily square-free) polynomials of increasing degree. The best performers were Pari/GP and ARM; the worst was NTL. The gap between the best and the worst algorithms was never higher than a factor of ≈ 5. Figure 1 shows the running times for two finite fields of size roughly 10 bits.

We used the versions of Pari/GP (v2.7.1) and NTL (v6.1.0) shipped in the latest Sage distribution (v6.4.1). It should be noted that this is not the latest NTL version, however, we are not aware of any performance improvements in the latest version (v8.0) concerning root finding. It is also worth mentioning that the implementation of polynomials over $\mathbb{F}_{q^n}$ in Sage 6.4 are backed by the NTL library (more precisely, by the multi-precision type `ZZ_pEX`), thus the performance of our implementation is essentially to be compared with NTL's one.

To the best of our knowledge, both Pari/GP and NTL implement the variant of Cantor-Zassenhaus [7] described in [20]. The difference in performances is likely the best explained by the fact that the NTL type `ZZ_pEX` is not the best fit for small characteristics, while Pari/GP optimizes the field element representation in this case. The fact that our own implementations performed between the two, and sometimes

even better, shows that these algorithms are practical. However, the observed differences in performance being so small, they are more likely attributed to differences in the specific implementation rather than to the algorithms themselves.

We conclude that the algorithms reviewed in this paper are of practical interest for finding polynomial roots in small characteristic finite fields. In some cases, this is in contrast with what was previously thought, e.g., for ARM. Thus we think that these algorithms may find useful applications in particular finite field problems.

VI. Conclusion

In this paper, we revisit several algorithms for root-finding over finite fields with small characteristics. We provide variants of ARM and SRA with improved average time complexity and show that these two algorithms are, in a sense, dual. We also detail the relationship between ARM and the well-known BTA. Finally, we implement all these algorithms in Sage, showing that they perform equally well in practice up to a small constant factor.

Aknowledgements. Luca De Feo would like to thank the Pari/GP team for kindly hosting him in Bordeaux during the preparation of the paper and for giving him useful insights into the internals of Pari/GP. He also thanks Jean-Pierre Flori for his invaluable help figuring out the gory details of the Sage/NTL interface.

References

[1] R. McEliece, "A Public-key Cryptosystem Based on the Aalgebraic Cdoing Theory," *Technical Report DSN 42-44, JPL, Pasadena*, 1978.

[2] J. von zur Gathen, "Who Was Who in Polynomial Factorization," in *Symbolic and Algebraic Computation, International Symposium, ISSAC 2006, Genoa, Italy, July 9-12, 2006, Proceedings*, B. M. Trager, Ed. ACM, 2006, p. 2. [Online]. Available: http://doi.acm.org/10.1145/1145768.1145770

[3] E. R. Berlekamp, "Factoring Polynomials over Finite Fields," *Bell System Tech. J.*, vol. 46, pp. 1853–1859, 1967.

[4] ——, "Factoring Polynomials over Large Finite Fields," *Math. Comp.*, vol. 24, pp. 713–735, 1970.

[5] R. T. Moenck, "On the Efficiency of Algorithms for Polynomial factoring," vol. 31, no. 137, pp. 235–250, Jan. 1977.

[6] M. O. Rabin, "Probabilistic Algorithms in Finite Fields," *SIAM J. Comput*, vol. 9, pp. 273–280, 1979.

[7] D. G. Cantor and H. Zassenhaus, "A New Algorithm for Factoring Polynomials over Finite Fields," *Mathematics of Computation*, pp. 587–592, 1981.

[8] E. Berlekamp, *Algebraic Coding Theory*. Aegan Park Press, 1984.

[9] A. Menezes, P. C. van Oorschot, and S. A. Vanstone, "Some Computational Aspects of Root Finding in GF(q^m)." in *ISSAC*, ser. Lecture Notes in Computer Science, P. M. Gianni, Ed., vol. 358. Springer, 1988, pp. 259–270. [Online]. Available: http://dblp.uni-trier.de/db/conf/issac/issac88.html#MenezesOV88

[10] P. C. van Oorschot and S. A. Vanstone, "A Geometric Approach to Root Finding in GF(q^m)," *IEEE Transactions on Information Theory*, vol. 35, no. 2, pp. 444–453, 1989. [Online]. Available: http://dx.doi.org/10.1109/18.32139

[11] A. J. Menezes, P. C. Van Oorschot, and S. Vanstone, "Subgroup Refinement Algorithms for Roots Finding in GF(q)*," *SIAM J. Comput.*, vol. 21, no. 2, pp. 228–239, apr 1992.

[12] V. Shoup, "A Fast Deterministic Algorithm for Factoring Polynomials over Finite Fields of Small Characteristic," in *Proceedings of the 1991 International Symposium on Symbolic and Algebraic Computation, ISSAC '91, Bonn, Germany, July 15-17, 1991*, S. M. Watt, Ed. ACM, 1991, pp. 14–21. [Online]. Available: http://doi.acm.org/10.1145/120694.120697

[13] H. Niederreiter, "Factoring Polynomials over Finite Fields Using Differential Equations and Normal Bases," *Mathematics of Computation*, vol. 62, no. 206, pp. 819–830, 1994.

[14] P. Fleischmann and P. Roelse, "Comparative Implementations of Berlekamp's and Niederreiter's Polynomial Factorization Algorithms," in *FINITE FIELDS AND APPLICATIONS*. Cambridge University Press, 1996, pp. 73–84.

[15] C. Petit, "Finding Roots in GF(p^n) with the Successive Resultant Algorithm," *LMS Journal of Computation and Mathematics*, 8 2014, (to appear).

[16] P. C. van Oorschot and S. A. Vanstone, "On Splitting Sets in Block Designs and Finding Roots of Polynomials," *Discrete Mathematics*, vol. 84, no. 1, pp. 71–85, 1990. [Online]. Available: http://dx.doi.org/10.1016/0012-365X(90)90274-L

[17] J. V. Z. Gathen and J. Gerhard, *Modern Computer Algebra*, 2nd ed. New York, NY, USA: Cambridge University Press, 2003.

[18] A. Schönhage, "Schnelle Berechnung von Kettenbruchentwicklungen," *Acta Inf.*, vol. 1, pp. 139–144, 1971. [Online]. Available: http://dx.doi.org/10.1007/BF00289520

[19] K. Thull and C. K. Yap, "A Unified Approach to HGCD Algorithms for Polynomials and Integers," 1990.

[20] J. von zur Gathen and V. Shoup, "Computing Frobenius Maps and Factoring Polynomials," *Computational Complexity*, vol. 2, pp. 187–224, 1992. [Online]. Available: http://dx.doi.org/10.1007/BF01272074

[21] K. S. Kedlaya and C. Umans, "Fast Polynomial Factorization and Modular Composition," *SIAM J. Comput.*, vol. 40, no. 6, pp. 1767–1802, Dec. 2011. [Online]. Available: http://dx.doi.org/10.1137/08073408X

[22] W. Szpankowski, "On the Analysis of the Average Height of a Digital Trie: Another Approach," 1986.

[23] B. Pittel, "Paths in a Random Digital Tree: Limiting Distributions," *Advances in Applied Probability*, vol. 18, pp. 139–155, 1986.

[24] J.-M. Couveignes and R. Lercier, "Elliptic periods for finite fields," *Finite Fields and Their Applications*, vol. 15, no. 1, pp. 1–22, Feb. 2009. [Online]. Available: http://dx.doi.org/10.1016/j.ffa.2008.07.004

[25] L. De Feo and É. Schost, "Fast arithmetics in Artin-Schreier towers over finite fields," *J. Symb. Comput.*, vol. 47, no. 7, pp. 771–792, 2012. [Online]. Available: http://dx.doi.org/10.1016/j.jsc.2011.12.008

[26] L. De Feo, J. Doliskani, and É. Schost, "Fast algorithms for ℓ-adic towers over finite fields," in *ISSAC'13*. ACM, 2013, pp. 165–172.

[27] W. A. Stein and Others, *Sage Mathematics Software (Version 6.4)*, The Sage Development Team, 2014.

[28] *PARI/GP, version* `2.7.1`, The PARI Group, Bordeaux, 2014.

[29] V. Shoup, "NTL: A Library for Doing Number Theory, version `6.1.0`," http://www.shoup.net/ntl, 2014.