Finding Roots in $\mathbb{F}_{p^n}$ with the Successive Resultants Algorithm

Christophe Petit *

UCL Crypto Group

To appear in the LMS Journal of Computation and Mathematics, as a special issue for ANTS (Algorithmic Number Theory Symposium) conference.

Abstract. The problem of solving polynomial equations over finite fields has many applications in cryptography and coding theory. In this paper, we consider polynomial equations over a "large" finite field with a "small" characteristic. We introduce a new algorithm for solving this type of equations, called the *Successive Resultants Algorithm* (SRA) in the sequel. SRA is radically different from previous algorithms for this problem, yet it is conceptually simple. A straightforward implementation using Magma was able to beat the built-in function *Roots* for some parameters. These preliminary results encourage a more detailed study of SRA and its applications. Moreover, we point out that an extension of SRA to the multivariate case would have an important impact on the practical security of the elliptic curve discrete logarithm problem in small characteristic.

1 Introduction

Let p be a "small" prime number and let d and n be two natural numbers. Let $\mathbb{F}_{p^n}$ be the finite field with p^n elements, and let f be a polynomial of degree d over $\mathbb{F}_{p^n}$. The *root-finding problem* is the problem of computing one, several or all elements $x \in \mathbb{F}_{p^n}$ such that

$$f(x) = 0.$$

This problem has a lot of applications, in particular for the more general problem of factoring f and its applications [19], but also in cryptography and in coding theory.

Many algorithms have been proposed to solve this problem. Most of them first reduce f to a square-free and split polynomial, and then progressively factor this polynomial through successive attempts [1,13,17,4].

In this paper, we introduce the *Successive Resultant Algorithm* (SRA), a new deterministic algorithm to solve this problem. Our approach is conceptually simple, yet radically different from previous ones. We show that SRA has an asymptotic complexity comparable to Berlekamp's well-known trace algorithm for large degree polynomials ($d^2 > n$ or $d > n$ depending on the type of polynomial arithmetic) and in all cases if certain field constants used in the algorithm are precomputed. We also provide a straightforward implementation using Magma [21] and we emphasize some parameter

* Supported by an F.R.S.-FNRS postdoctoral research fellowship at Université catholique de Louvain, Louvain-la-Neuve.

sets for which this implementation has beaten Magma's corresponding built-in function *Roots*.

We finally discuss open problems and a potential extension of our work. In particular, we believe that our ideas form an important step towards a much more efficient resolution of polynomial systems arising from a Weil descent in the multivariate case [11]. We stress that a multivariate version of SRA would have a very strong impact on the practical security of the Elliptic Curve Discrete Logarithm Problem in the small characteristic case.

1.1 Outline

This paper is organized as follows. In Section 2, we review the basics of finite field arithmetic and previous root finding algorithms in $\mathbb{F}_{p^n}$. In Sections 3 and 4, we provide both a basic version of our algorithm and an optimized version for fast arithmetic. We also analyze the complexity of our algorithms in these sections. In Section 5, we provide experimental timings obtained with a Magma implementation of our algorithm. We finally conclude the paper and present interesting open problems in Section 6.

2 Preliminaries

2.1 Finite Field and Polynomial Ring Arithmetic

Let p be a "small" prime number, let n be a positive integer, let $\mathbb{F}_{p^n}$ be the finite field with p^n elements and let f be a univariate polynomial of degree d over $\mathbb{F}_{p^n}$. We also define s as the number of solutions of f over $\mathbb{F}_{p^n}$.

We will suppose that p is small enough for us to treat it as a constant in our estimations. Unless explicitely mentioned, we take an operation over $\mathbb{F}_p$ as a basic step in all our complexity evaluations. We use both the "big O" and "big O tilde" notations in our estimations. Remember that f is $\tilde{O}(g)$ if and only if f is $O(g \log^c(g))$ for some constant c. Solving a linear system of size m over $\mathbb{F}_p$ has a cost $O(m^\omega)$, where ω is the linear algebra constant. The best algorithms today achieve ω as small as 2.3727 for generic systems [22].

We denote by $\mathbf{a}(n)$ and $\mathbf{m}(n)$ the cost of an addition and a multiplication over $\mathbb{F}_{p^n}$, and by $\mathbf{A}(d)$ and $\mathbf{M}(d)$ the cost of an addition and a multiplication of two polynomials of degree d over $\mathbb{F}_{p^n}$. We also denote by $\mathbf{G}(d)$ the cost of computing the greatest common divisor of two polynomials of degree d over $\mathbb{F}_{p^n}$. We will consider both "classical" and "fast" polynomial arithmetics in this paper.

Classical arithmetic is a reasonable choice today for small and medium parameter sizes for which the overhead of fast arithmetic algorithms is significant. Using this type of arithmetic, field additions and polynomial additions are respectively executed in $O(n)$ and $O(dn)$. Polynomial multiplications are performed in a straightforward way with a quadratic cost with respect to the degree. As a result, we have $\mathbf{m}(n) = O(n^2)$ and $\mathbf{M}(d) = O(d^2n^2)$.

Using fast arithmetic, polynomial multiplications are performed in quasi-linear time with FFT-based methods [15]. As a result, we have $\mathbf{m}(n) = \tilde{O}(n)$ and $\mathbf{M}(d) = \tilde{O}(dn)$.

Multiplications modulo a polynomial of degree d can be performed at essentially the same cost. Field additions and polynomial additions are executed in $O(n)$ and $O(dn)$ as before. Fast arithmetic is available today in the computer algebra system Magma [21].

The greatest common divisor (gcd) of two polynomials of degree d can be computed in $O(d^2)$ field operations using the Euclidean algorithm or $\tilde{O}(d)$ field operations using a more involved Schönhage-type algorithm [14,16]. In our estimations, we will assume for simplicity that the Euclidean algorithm is always used together with classical arithmetic and that fast gcd algorithms are always used together with fast arithmetic. Table 1 summarizes the various costs respectively for "classical" and "fast" arithmetics with this convention.

Table 1: Costs of finite field and polynomial arithmetic

	$\mathbf{a}(n)$	$\mathbf{m}(n)$	$\mathbf{A}(d)$	$\mathbf{M}(d)$	$\mathbf{G}(d)$
Classical	$O(n)$	$O(n^2)$	$O(dn)$	$O(d^2n^2)$	$O(d^2n^2)$
Fast	$O(n)$	$\tilde{O}(n)$	$O(dn)$	$\tilde{O}(dn)$	$\tilde{O}(dn)$

2.2 Finding Roots in $\mathbb{F}_{p^n}$

Let f be a univariate polynomial over $\mathbb{F}_{p^n}$ with degree d having exactly s distinct roots. The problems of computing one, several or all roots of f have many applications in cryptography and coding theory. Several algorithms have been proposed for this problem, with complexities depending on the arithmetic type and on the parameters d and n.

In most root-finding algorithms, the polynomial f is assumed to be split and squarefree (all its irreducible factors are linear and distinct), hence $s = d$. Given an arbitrary polynomial f, its squarefree split part is easily recovered through the gcd computation $\gcd(x^{p^n} - x, f(x))$, after successively computing the polynomials $x^{p^i} \bmod f(x)$ for $i = 0, \ldots, n-1$ with a square-and-multiply algorithm. These computations require $O(d^2n)$ operations over $\mathbb{F}_p^n$ or $O(d^2n^3)$ operations over $\mathbb{F}_p$ using standard arithmetic, and only $\tilde{O}(dn^2)$ over $\mathbb{F}_p$ using fast arithmetic.

The simplest algorithms for the root finding problem are variants of exhaustive search. A better approach was proposed by Berlekamp et al. in [3]. This algorithm first constructs a polynomial L such that $L(x) = \sum_{i=0}^{d-1} L_i x^{p^i}$ and f divides L. The computation of L only requires computing $x^{p^i} \bmod f(x)$ for $i = 0, \ldots, d-1$ and then solving a $d \times d$ linear system over $\mathbb{F}_{p^n}$. Since L is a linear application over $\mathbb{F}_p$, the algorithm of [3] then solves L with linear algebra over $\mathbb{F}_p$ and tests each solution for f. The algorithm is still not very efficient in general since L may have up to p^d solutions in the worst case, and all these solutions are tested to identify the roots of f.

The best known algorithm for computing all the roots is probably Berlekamp's trace algorithm (BTA) that was originally presented in his celebrated paper on the factorization of polynomials [1]. This algorithm tries to factor a split squarefree polynomial f

as

$$f(x) = \prod_{r \in \mathbb{F}_p} \gcd\left(f(x), \mathrm{Tr}(\alpha^i x) - r\right)$$

for some $\alpha \in \mathbb{F}_{p^n}$ with algebraic degree n and for various $i \in \{0, \ldots, n-1\}$. Each gcd computation costs $O(d^2n^3)$ or $\tilde{O}(dn^2)$ operations over $\mathbb{F}_p$, using respectively classical or fast arithmetics. It is known that at least one value of i leads to a non-trivial factorization [1] and that testing $O(\log n)$ of them is required on average [10]. Once f is split into at least two distinct factors, the process is then recursively applied to all these factors. Since the recursive step has a cost larger than any linear function in d, we can in fact recover all the linear factors of f using $O(d^2n^3)$ or $\tilde{O}(dn^2)$ operations over $\mathbb{F}_p$, depending on the arithmetic type [18, Th. 14.11].

Other splitting strategies are also possible. When p is odd, Rabin's root-finding algorithm [13] computes $\gcd\left(f(x), (x+\delta)^{(p^n-1)/2} - 1\right)$ for a random $\delta \in \mathbb{F}_{p^n}$. The total complexity of this approach is similar to BTA.

Compared to BTA, the Affine Method of van Oorschot and Vanstone [17] first computes a polynomial L as in [3]. The trace function used in BTA is generalized to other polynomials $B(x)$ that are also linear over $\mathbb{F}_p$. The gcd between f and B is then computed in two steps as $\gcd(f(x), \gcd(L(x), B(x))$. The affine method is more efficient than BTA when $d < n$ and standard arithmetic is used, since their respective costs are then equivalent to $O(d^2n)$ and $O(dn^2)$ multiplications over $\mathbb{F}_{p^n}$ [17]. However with fast arithmetic, the computation $B(x) \bmod L(x)$ alone already requires $\tilde{O}(dn^2)$ following the method of [17], so the Affine Method is at best as fast as BTA.

The modular Frobenius exponentiation $x \to x^{p^i} \bmod f(x)$ is a key ingredient of all the methods described above. Von zur Gathen and Shoup [20] suggested to use repeated modular compositions and multipoint evaluation instead of the straightforward square and multiply algorithm to perform these exponentiations. This idea led to the asymptotically fastest polynomial factorization algorithms today. Kaltofen and Shoup [8] proposed an algorithm running in a time $\tilde{O}(d^{1.815}n^2)$, though not completely practical since it relies on fast matrix multiplication. By introducing new, asymptotically faster algorithms for the modular composition problem, Kedlaya and Umans [9] derived a randomized algorithm to factor f entirely in time $\tilde{O}(d^{3/2}n + dn^2)$.

Our new algorithm has an asymptotic complexity $O(n^4 + d^2n^3)$ with standard arithmetic and $\tilde{O}(n^3 + dn^2)$ with fast arithmetic, where the n^4 and n^3 terms are spent on computing certain field constants. This asymptotic complexity is similar to BTA for large degree polynomials or if the field constants are precomputed. Our experiments suggest that the new algorithm may compete with the ones currently used in practice for some parameters.

3 The Successive Resultants Algorithm

We now describe our new algorithm for solving polynomial equations over finite fields of small characteristic.

3.1 A polynomial system

Let $\{v_1, \ldots, v_n\}$ be an arbitrary basis of $\mathbb{F}_{p^n}$ over $\mathbb{F}_p$. From this basis, we recursively define $n+1$ functions $L_0, L_1, \ldots, L_n$ from $\mathbb{F}_{p^n}$ to $\mathbb{F}_{p^n}$ such that

$$\begin{cases} L_0(z) = z \\ L_1(z) = \prod_{i \in \mathbb{F}_p} L_0(z - iv_1) \\ L_2(z) = \prod_{i \in \mathbb{F}_p} L_1(z - iv_2) \\ \ldots \\ L_n(z) = \prod_{i \in \mathbb{F}_p} L_{n-1}(z - iv_n). \end{cases}$$

The functions L_j are examples of *linearized polynomials* as defined in [2, Ch. 11]. They satisfy the following properties.

Lemma 1. *a) Each polynomial L_i is split and its roots are all elements of the vector space generated by $\{v_1, \ldots, v_i\}$. In particular, we have $L_n(z) = z^{p^n} - z$.*
b) We have $L_i(z) = L_{i-1}(z)^p - a_i L_{i-1}(z)$ where $a_i := (L_{i-1}(v_i))^{p-1}$.
c) If we identify $\mathbb{F}_{p^n}$ with the vector space $(\mathbb{F}_p)^n$, then each L_i is a p to 1 linear map of $L_{i-1}(z)$ and a p^i to 1 linear map of z.

Proof. Part a) is clear by construction. We first prove Part b) for L_1. We have $z^p - z = \prod_{i \in \mathbb{F}_p}(z - i)$ by identification of the roots on both sides. Substituting x by z/v_1, we deduce $z^p - v_1^{p-1} z = \prod_{i \in \mathbb{F}_p}(z - iv_1) = L_1(z)$. From this equality, it is clear that L_1 is a linear map over $\mathbb{F}_p$, and in particular $L_1(z - iv_2) = L_1(z) - iL_1(v_2)$ for all $i \in \mathbb{F}_p$. Substituting z by $L_j(z)$ and v_1 by $L_j(v_{j+1})$, part b) follows by induction. For part c), notice that the kernel of the linear map $z \to \prod_{i \in \mathbb{F}_p}(z - i)$ has size p.

We now consider the following polynomial system:

$$\begin{cases} f(x_1) = 0 \\ x_j^p - a_j x_j = x_{j+1} \quad j = 1, \ldots, n-1 \\ x_n^p - a_n x_n = 0 \end{cases} \tag{1}$$

where the $a_i \in \mathbb{F}_{p^n}$ are defined as in Lemma 1. Any solution of this system provides us with a root of f by the first equation, and the n last equations together imply that this root belongs to $\mathbb{F}_{p^n}$.

Lemma 2. *Let $(x_1, x_2, \ldots, x_n)$ be a solution of System (1). Then $x_1 \in \mathbb{F}_{p^n}$ is a solution of f. Conversely, given a solution $x_1 \in \mathbb{F}_{p^n}$ of f, we can reconstruct a solution of System (1) by setting $x_2 = x_1^p - a_1 x_1$, etc.*

Proof. By Lemma 1, the equations of System (1) imply $x_i = L_{i-1}(x_1)$, and in particular $x_n^p - a_n x_n = x_1^{p^n} - x_1$ so $x_1 \in \mathbb{F}_{p^n}$.

3.2 Solving System (1) with Resultants

In order to solve System (1), we notice that it has a *quasi-diagonal* structure: the first equation only depends on x_1, each equation $x_j^p - a_j x_j = x_{j+1}$ only depends on x_j and x_{j+1}, and the last equation only depends on x_n. Our new algorithm will exploit this structure to solve System (1), hence the polynomial f.

In the first step of the algorithm, we successively compute $f^{(1)} = f, f^{(2)}, \ldots, f^{(n)}$ such that $f^{(j)}$ has the same degree as f and only depends on the variable x_j. Let f_i be the coefficients of f, such that $f(x) = \sum_{i=0}^{d} f_i x^i$. We compute $f^{(2)}$ as

$$f^{(2)}(x_2) = \mathrm{Res}_{x_1}(f^{(1)}(x_1), x_2 - (x_1^p - a_1 x_1))$$

$$= \begin{vmatrix}
1 & 0 & \ldots & 0 & -a_1 & -x_2 & 0 & 0 & & & \ldots & & & 0 \\
 & 1 & 0 & \ldots & 0 & -a_1 & -x_2 & 0 & 0 & & & \ldots & & 0 \\
 & & & & & & \ldots & & & & & & & \\
 & & & & & & & & \ldots & & & & & \\
 & & & & & & & & 1 & 0 & \ldots & 0 & -a_1 & -x_2 & 0 \\
 & & & & & & & & & 1 & 0 & \ldots & 0 & -a_1 & -x_2 \\
f_d & f_{d-1} & f_{d-2} & \ldots & f_{p+1} & f_p & f_{p-1} & \ldots & f_2 & f_1 & f_0 & 0 & \ldots & 0 & 0 \\
 & f_d & f_{d-1} & f_{d-2} & \ldots & f_{p+1} & f_p & f_{p-1} & \ldots & f_2 & f_1 & f_0 & 0 & \ldots & 0 \\
 & & & & & & & & \ldots & & & & & & \\
 & & & f_d & f_{d-1} & f_{d-2} & \ldots & f_{p+1} & f_p & f_{p-1} & \ldots & f_2 & f_1 & f_0 & 0 \\
 & & & & f_d & f_{d-1} & f_{d-2} & \ldots & f_{p+1} & f_p & f_{p-1} & \ldots & f_2 & f_1 & f_0
\end{vmatrix}, \qquad (2)$$

which is clearly a polynomial in x_2 only. Its degree is exactly d since the variable x_2 appears exactly d times in the above determinant, in different rows and columns. We then successively compute

$$f^{(j+1)}(x_{j+1}) = \mathrm{Res}_{x_j}\left(f^{(j)}(x_j), x_{j+1} - (x_j^p - a_j x_j)\right)$$

for $j = 2, \ldots, n-1$, which all have degree d for the same reasons. A simple algorithm to compute these resultants is provided in Section 3.4 below.

In the second step of our algorithm, we successively recover values for $x_n, x_{n-1}, \ldots$, and finally x_1. We first compute

$$g^{(n)}(x_n) := \gcd\left(f^{(n)}(x_n), x_n^p - a_n x_n\right).$$

By construction, $g^{(n)}$ is a polynomial of degree at most p, dividing $x_n^p - a_n x_n$. If this polynomial is a non zero constant, then f has no solution over $\mathbb{F}_{p^n}$. Otherwise, it follows from Lemma 1 c) that $g^{(n)}$ is split. Its roots $\hat{x}_n$ correspond to the values of the variable x_n in the solutions of System (1). For each of these $\hat{x}_n$ values, we then compute

$$g^{(n-1)}(x_{n-1}) := \gcd\left(f^{(n-1)}(x_{n-1}), \hat{x}_n - (x_{n-1}^p - a_{n-1} x_{n-1})\right). \qquad (3)$$

By construction, $g^{(n-1)}$ is a polynomial of degree at most p, dividing $\hat{x}_n - (x_{n-1}^p - a_{n-1} x_{n-1})$. If this polynomial is a constant, then there is no solution. Otherwise, it follows from Lemma 1 c) that $g^{(n-1)}$ is split. We compute the factorization of $g^{(n-1)}$ using any classical root-finding algorithm, or any dedicated root-finding algorithm for

the linear polynomial $\hat{x}_n - (x_{n-1}^p - a_{n-1}x_{n-1})$ followed by a small exhaustive search. The roots of $g^{(n-1)}$ correspond to the values of the variables x_{n-1} in the solutions of System (1). Proceeding recursively, we finally obtain all x_1 values that satisfy the equation $f^{(1)}(x_1) = 0$. The whole algorithm is deterministic if no probablistic algorithm is used for the resultant computation and small degree root-finding routines.

3.3 Example

We provide a small example of SRA execution when $p = 2$, $n = 5$ and $d = 6$. Let α be a root of $t^5 + t^2 + 1$ over $\mathbb{F}_{2^5}$. Let $v_i := \alpha^{i-1}$, $i = 1, \ldots, 5$. The precomputation step of SRA leads to $a_1 = 1$, $a_2 = \alpha^{19}$, $a_3 = \alpha^6$, $a_4 = \alpha^4$ and $a_5 = \alpha^2$.

Let now $f(x) := x^5 + \alpha^{20}x^4 + \alpha^{27}x^3 + \alpha^4x^2 + \alpha^{14}x + \alpha^9$. In the first step of SRA, we successively compute

$$\begin{aligned}
f^{(1)}(x_1) &= x_1^5 + \alpha^{20}x_1^4 + \alpha^{27}x_1^3 + \alpha^4x_1^2 + \alpha^{14}x_1 + \alpha^9,\\
f^{(2)}(x_2) &= x_2^5 + \alpha^{28}x_2^4 + \alpha^{23}x_2^3 + \alpha^4x_2^2 + \alpha^{12}x_2 + \alpha^{19},\\
f^{(3)}(x_3) &= x_3^5 + \alpha x_3^4 + \alpha^{23}x_3^3 + \alpha^{23}x_3^2 + x_3,\\
f^{(4)}(x_4) &= x_4^5 + \alpha^4x_4^4 + \alpha^7x_4^3 + \alpha^{11}x_4^2,\\
f^{(5)}(x_5) &= x_5^5 + \alpha x_5^3.
\end{aligned}$$

In the second step of SRA, we then compute

$$\begin{aligned}
g^{(5)}(x_5) &= \gcd(f^{(5)}(x_5), x_5^2 + a_5x_5) = x_5,\\
g^{(4)}(x_4) &= \gcd(f^{(5)}(x_5), x_4^2 + a_4x_4) = x_4^2 + \alpha^4x_4 = x_4(x_4 + \alpha^4).
\end{aligned}$$

The root $\hat{x}_4 = \alpha^4$ leads to

$$\begin{aligned}
g^{(3)}(x_3) &= \gcd(f^{(3)}(x_3), \hat{x}_4 + x_3^2 + a_3x_3) = x_3 + \alpha^3,\\
g^{(2)}(x_2) &= \gcd(f^{(2)}(x_2), \hat{x}_3 + x_2^2 + a_2x_2) = x_2 + \alpha,\\
g^{(1)}(x_1) &= \gcd(f^{(1)}(x_1), \hat{x}_2 + x_1^2 + a_1x_1) = x_1 + \alpha^3.
\end{aligned}$$

The root $\hat{x}_4 = 0$ leads to

$$g^{(3)}(x_3) = \gcd(f^{(3)}(x_3), \hat{x}_4 + x_3^2 + a_3x_3) = x_3^2 + \alpha^6x_3 = x_3(x_3 + \alpha^6).$$

The root $\hat{x}_3 = 0$ leads to

$$\begin{aligned}
g^{(2)}(x_2) &= \gcd(f^{(2)}(x_2), \hat{x}_3 + x_2^2 + a_2x_2) = x_2 + \alpha^{19},\\
g^{(1)}(x_1) &= \gcd(f^{(1)}(x_1), \hat{x}_2 + x_1^2 + a_1x_1) = x_1 + \alpha^{18}.
\end{aligned}$$

The root $\hat{x}_3 = \alpha^6$ leads to

$$\begin{aligned}
g^{(2)}(x_2) &= \gcd(f^{(2)}(x_2), \hat{x}_3 + x_2^2 + a_2x_2) = x_2 + \alpha^{30},\\
g^{(1)}(x_1) &= \gcd(f^{(1)}(x_1), \hat{x}_2 + x_1^2 + a_1x_1) = x_1 + \alpha^{19}.
\end{aligned}$$

The solution set of f is therefore $\{\alpha^3, \alpha^{18}, \alpha^{19}\}$. For this example, the computation of this set required 5 resultants, 10 gcds and the factorizations of 2 degree 2 (linear over $\mathbb{F}_2$) polynomials.

3.4 Computing the Resultants

Resultants are the basic operations in the first step of SRA algorithm. Under simple row manipulations, we have

$$
\begin{aligned}
f^{(2)}(x_2) &= \mathrm{Res}_{x_1}(f^{(1)}(x_1), x_2 - (x_1^p - a_1 x_1)) \\
&= \begin{vmatrix}
1 & 0 & \dots & 0 & -a_1 & -x_2 & 0 & 0 & \dots & & & 0 \\
 & 1 & 0 & \dots & 0 & -a_1 & -x_2 & 0 & 0 & \dots & & 0 \\
 & & & & & \dots & & & & & & \\
 & & & & & & & \dots & & & & \\
 & & & & & & & 1 & 0 & \dots & 0 & -a_1 & -x_2 & 0 \\
 & & & & & & & & 1 & 0 & \dots & 0 & -a_1 & -x_2 \\
 & & & & & & & & & F_{p-1,p-1} & \dots & F_{p-1,2} & F_{p-1,1} & F_{p-1,0} \\
 & & & & & & & & & F_{p-2,p-1} & \dots & F_{p-2,2} & F_{p-2,1} & F_{p-2,0} \\
 & & & & & & & & & & & \dots & & \\
 & & & & & & & & & F_{1,p-1} & \dots & F_{1,2} & F_{1,1} & F_{1,0} \\
 & & & & & & & & & F_{0,p-1} & \dots & F_{0,2} & F_{0,1} & F_{0,0}
\end{vmatrix} \\
&= \begin{vmatrix}
F_{p-1,p-1} & \dots & F_{p-1,2} & F_{p-1,1} & F_{p-1,0} \\
F_{p-2,p-1} & \dots & F_{p-2,2} & F_{p-2,1} & F_{p-2,0} \\
 & & \dots & & \\
F_{1,p-1} & \dots & F_{1,2} & F_{1,1} & F_{1,0} \\
F_{0,p-1} & \dots & F_{0,2} & F_{0,1} & F_{0,0}
\end{vmatrix}
\end{aligned}
$$

where $F_{j,i}$ satisfy

$$\sum_{i=0}^{p-1} F_{j,i}(x_2) x_1^i = x_1^j f(x_1) \bmod (x_2 - (x_1^p - a_1 x_1)) .$$

In particular, $p \deg F_{j,i} + i \leq d + j$. The resultant can therefore be computed as follows:

1. Reduce the last row of (2) by the first d ones to obtain the coefficients $F_{0,i}$. This amounts to computing

$$h(x_1, x_2) := \sum_{i=0}^{p-1} F_{0,i}(x_2) x_1^i = f(x_1) \bmod (x_1^p - a_1 x_1 - x_2). \tag{4}$$

2. Shift these coefficients on the left and further reduce by the first d rows to obtain all coefficients $F_{i,j}$. This amounts to successively computing

$$
\begin{aligned}
\sum_{i=0}^{p-1} F_{j,i}(x_2) x_1^i &= x_1^j h(x_1, x_2) \bmod (x_1^p - a_1 x_1 - x_2) \\
&= x_1 \left(x_1^{j-1} h(x_1, x_2) \right) \bmod (x_1^p - a_1 x_1 - x_2) \\
&= \sum_{i=1}^{p-1} F_{j-1,i-1}(x_2) x_1^i + F_{j-1,p-1}(x_2) \left(a_1 x_1 + x_2 \right)
\end{aligned}
$$

3. Compute the last determinant with a few multiplications of polynomials with degrees smaller than d.

3.5 Complexity Analysis

The complexity of SRA can be analyzed as follows. First, we note that all the values a_i of Lemma 1 can be (pre)computed at a total cost of $O(n^2)$ operations over $\mathbb{F}_{p^n}$, that is $O(n^4)$ operations over $\mathbb{F}_p$ using classical arithmetic or $\tilde{O}(n^3)$ operations over $\mathbb{F}_p$ using fast arithmetic.

Next we evaluate the cost of the resultant algorithm of Section 3.4. The last row can be computed with $O(d)$ elementary row reduction steps, each one involving $O(p \cdot \frac{d}{p}) = O(d)$ multiplications over $\mathbb{F}_{p^n}$. All the other polynomials $F_{i,j}$ can then be computed using $O(d)$ operations over $\mathbb{F}_{p^n}$. Finally, the last determinant requires $O(p^3)$ multiplications of polynomials of degrees at most d over $\mathbb{F}_{p^n}$. Computing one resultant therefore costs $O(d^2n^2)$ operations over $\mathbb{F}_p$ using standard arithmetic and $\tilde{O}(d^2n)$ operations over $\mathbb{F}_p$ using fast arithmetic. Completing the first step of SRA costs n resultants, that is $O(d^2n^3)$ operations over $\mathbb{F}_p$ using standard arithmetic and $\tilde{O}(d^2n^2)$ operations over $\mathbb{F}_p$ using fast arithmetic.

In the second step of SRA, we compute several gcds between a degree d and a degree p polynomial over $\mathbb{F}_{p^n}$. This requires $O(d)$ operations over $\mathbb{F}_{p^n}$ for each gcd. We also need to factor all the polynomials $g^{(j)}$ that have degree larger than 1. Since each polynomial $g^{(j)}$ has degree at most p, each factorization costs $O(n^3)$ operations with classical arithmetic or $\tilde{O}(n^2)$ operations with fast arithmetic, using a classical equal-degree factorization algorithm like Berlekamp trace algorithm [1] or Cantor-Zassenhaus algorithm [4]. Note that these algorithms are only applied here on polynomials with degree smaller than $p = O(1)$. Alternatively, we can also factor the polynomials $\hat{x}_j - (x_{j-1}^p - a_{j-1}x_{j-1})$ using linear algebra over $\mathbb{F}_p$, and test each solution in $g^{(j-1)}(x_{j-1})$.

The number of times these two operations will be repeated in SRA second step depends on the number of solutions for each of $x_1, x_2, \ldots, x_n$. By the properties of resultants, any solution for x_i leads to at least one solution for all x_j, $j \leq i$. If f has exactly $s \leq d$ roots, then these roots are solutions for x_1, but several of these solutions may "merge" into common solutions for x_2, x_3, etc., and x_n can of course take at most p values. In any case, at most ns polynomials $g^{(j)}$ will be computed, and at most $s/2$ of them will need to be factored. The second step of our algorithm can therefore be completed with $O(dn^3s + n^3s) = O(dn^3s)$ operations over $\mathbb{F}_p$ using classical arithmetic and $\tilde{O}(dn^2s + n^2s) = \tilde{O}(dn^2s)$ operations over $\mathbb{F}_p$ using fast arithmetic. Note that the complexity of the second step is identical if f has more than s roots but we are only interested in computing s of them.

The total complexity of SRA is therefore $O(d^2n^3 + n^4)$ using classical arithmetic and $\tilde{O}(d^2n^2 + n^3)$ using fast arithmetic. When only classical arithmetic is available, this complexity is similar to BTA if $d^2 > n$ or if the field constants a_i in Lemma 1 are precomputed.

4 Fast SRA

When fast polynomial arithmetic is available, the basic SRA algorithm presented above does not compete with BTA. To compete again with BTA in this context, we introduce

two new algorithms to perform SRA first and second steps. The first algorithm simply uses the linearity of the Frobenius. The second one uses multipoint evaluation of polynomials, hence it crucially relies on fast polynomial arithmetic.

4.1 Improved Resultant Algorithm

The first step of the basic SRA algorithm consists in computing n resultants using the algorithm of Section 3.4. The most expensive part of this algorithm is the computation of the polynomial

$$h(x_1, x_2) := f(x_1) \bmod (x_1^p - a_1 x_1 - x_2)$$

in a straightforward way. We now present an alternative algorithm taking advantage of the linearity of the Frobenius.

Let $k := \lfloor \log_p d \rfloor$ and let $h_k(x_1, x_2) := f(x_1)$. The alternative algorithm first computes $a_1^{p^i}$ for $i = 0, \ldots, k-1$ in time $\tilde{O}(n \log_p d)$. It then successively computes

$$h_i\left(x_1, x_2^{p^i}\right) := h_{i+1}\left(x_1, x_2^{p^{i+1}}\right) \bmod \left(x_1^{p^{i+1}} - a_1^{p^i} x_1^{p^i} - x_2^{p^i}\right) \tag{5}$$

for $i = k-1, \ldots, 0$. We observe that

$$h_i\left(x_1, x_2^{p^i}\right) = f(x_1) \bmod (x_1^p - a_1 x_1 - x_2)^{p^i}$$

so in particular $h(x_1, x_2) = h_0(x_1, x_2)$.

The polynomial $h_i\left(x_1, x_2^{p^i}\right)$ has degree at most p^{i+1} in x_1 and at most p^{k-i} in $x_2^{p^i}$. Each reduction step (5) involves reducing at most $(p-1)p^{i+1}$ terms $c_j\left(x_2^{p^i}\right) \cdot x_1^j$ where c_j are polynomials of degree p^{k-i}, so it takes $\tilde{O}(dn)$. There are $\log_p d$ steps so the total cost to compute $h(x_1, x_2)$ is also $\tilde{O}(dn)$. Using fast polynomial multiplication to compute the determinant as in Section 3.4, each resultant in the first step of SRA can be computed using only $\tilde{O}(dn)$ operations over $\mathbb{F}_p$.

As a consequence, the first step of SRA can be performed in time $\tilde{O}(dn^2)$ operations using fast arithmetic.

4.2 Simultaneous Evaluation of g_j for all $\hat{x}_{j+1}$

The second step of SRA requires the evaluation of

$$g^{(j-1)}(x_{j-1}) := \gcd\left(f^{(j-1)}(x_{j-1}), \hat{x}_j - (x_{j-1}^p - a_{j-1} x_{j-1})\right)$$

for all solutions $\hat{x}_j$, and for $j = n-1, \ldots, 2$. We notice that unless

$$f^{(j-1)}(x_{j-1}) = 0 \bmod \left(\hat{x}_j - (x_{j-1}^p - a_{j-1} x_{j-1})\right),$$

we have

$$g^{(j-1)}(x_{j-1}) = \gcd\left(f^{(j-1)}(x_{j-1}) \bmod \left(\hat{x}_j - (x_{j-1}^p - a_{j-1} x_{j-1})\right), \hat{x}_j - (x_{j-1}^p - a_{j-1} x_{j-1})\right).$$

Moreover, all the polynomials

$$f^{(j-1)}(x_{j-1}) \bmod \left(x_j - (x_{j-1}^p - a_{j-1}x_{j-1})\right) = \sum_{i=0}^{p-1} F_{0,i}^{(j-1)}(x_j) x_{j-1}^i$$

are computed in the first step of SRA.

Using a multipoint evaluation algorithm, each polynomial $F_{0,i}^{(j-1)}(x_j)$ (that has degree smaller than d) can be evaluated at the almost $s \leq d$ solutions $\hat{x}_j$ in time $\tilde{O}(dn)$. Once these values have been computed, each final gcd is performed on two polynomials of degrees smaller than $p = O(1)$, hence it only requires $O(1)$ operations over $\mathbb{F}_{p^n}$. The total cost for computing the polynomial $g^{j-1}(x_{j-1})$ for all solutions $\hat{x}_j$ is therefore $\tilde{O}(dn)$ instead of $\tilde{O}(dns)$, and the cost of the second step of SRA decreases from $\tilde{O}(dn^2 s)$ to $\tilde{O}(dn^2)$ using this algorithm.

4.3 Complexity of Fast SRA

Using the algorithms of Sections 4.1 and 4.2, the cost of SRA with fast arithmetic can be reduced to $\tilde{O}(dn^2 + n^3)$, where the n^3 term comes from the (pre)-computation of the a_i in Lemma 1. Possibly up to logarithmic factors, this complexity is similar to BTA if $d > n$ or if the field constants a_i are precomputed.

5 Proof-of-Concept Implementation Results

As a proof of concept, we implemented both the basic and fast versions of the Successive Resultant Algorithm in Magma [21]. We chose Magma for its simplicity of use and because it provides many of the subroutines that we need in our algorithm. We point out that Magma claims to have efficient fast algorithmic routines. The code of the basic version (given in Appendix A) is only a few lines. To implement multipoint evaluation in Fast SRA, we followed the description of [5]. We stress that we did not put any effort in optimizing neither the basic nor the fast SRA implementations. On the contrary, when a generic Magma function was available for a specific task, we always used this function, even if the particular inputs used in our algorithms could open the way to more efficient implementations. In particular, we did not implement straightforward simplifications when $p = 2$.

5.1 Experiments

We tested our implementation against Magma *Roots* function for $p \in \{2, 3, 5, 7, 11, 13, 17\}$, for $n = 2^{e_n}$ and $d = 2^{e_d}$ with $e_d, e_n \in \{2, \ldots, 12\}$, and for three types of polynomials:

- Random polynomials: polynomials of degree d over $\mathbb{F}_{p^n}$ with randomly chosen coefficients.
- Random polynomials with a single root: to generate these polynomials, we chose random polynomials as above and we used Magma functions to test whether it had a single root or not.

– Split polynomials: polynomials $f(x) := \prod_{i=1}^{d}(x - x_i)$ for x_i randomly chosen in $\mathbb{F}_{p^n}$.

For every polynomial, we recorded the time needed by Magma *Roots* function as well as the precomputing time and the time for the first and second steps of both Basic SRA and Fast SRA. All timings recorded were real time (seconds). We repeated every experiment ten times and we averaged computation times over these ten experiments. All experiments were performed on an Intel Xeon CPU X5500 processor running at 2.67 GHz, with 24 GB RAM.

5.2 Selected Results

With the exception of very small d values, the precomputation part of SRA had always a small or negligible cost compared to the first and second steps of the algorithm. For random polynomials and polynomials with only one root, we observed that the first part of our algorithm was by far the most time-consuming one. For split polynomials, the first and second parts tended to be more balanced.

Figures 1 and 2 show log-log graphs of the timings obtained for $p = 2$ and split polynomials, respectively as a function of n for various d and as a function of d for various n. We observed that the *Roots* function generally performed significantly better, and the two variants of SRA generally had similar timings. The timing evolutions with n is similar for the three algorithms, but both versions of SRA seem less efficient than *Roots* as d increases.

For larger p values, the gap between *Roots* and SRA performances is considerably reduced or completely removed, suggesting that even a slightly optimized version of *BasicSRA* could become competitive with respect to *Roots*. Table 2 reports some parameters and timing results for which either *BasicSRA* or *FastSRA* was the most efficient algorithm to compute roots. All these parameters involve split polynomials.

We believe that the relatively poor performances of both SRA implementations with respect to *Roots* for $p = 2$ are due to a default of optimizations to this case in our implementations with respect to Magma's *Roots* function. The advantage of *FastSRA* over *BasicSRA* will probably become more obvious for larger parameter sizes.

6 Conclusion and Open Problems

In this paper, we presented the Successive Resultant Algorithm (SRA), a new algorithm for finding roots in extension fields $\mathbb{F}_{p^n}$ with a small characteristic. The preliminary analysis conducted here suggests that SRA has an asymptotic complexity similar to Berlekamp's well-known trace algorithm for "large" polynomials ($d^2 \geq n$ with classical arithmetic, $d \geq n$ with fast arithmetic) in general, and for any parameters if certain field constants used in SRA are precomputed. Preliminary performance results obtained with a straightforward Magma implementation suggest that SRA could also become competitive with currently used algorithms in practice.

We leave a more thorough comparison analysis of our algorithm with previous work, including logarithmic factors and dependency in p, to further work. We also leave as an

Table 2: Timings for selected parameters. The average total time needed by the full basic and fast SRA is provided as a quotient with respect to the average time needed by Roots; the other timings are given in seconds. bSRA=Basic SRA, fSRA=Fast SRA, p=precomputation, f=first step, s=second step, t= total.

p	n	d	type	Roots	bSRAt	fSRAt	bSRAp	bSRAf	bSRAs	fSRAp	fSRAf	fSRAs
5	32	128	Split	1.17	0.91	1.33	0.01	0.64	0.42	0.01	0.63	0.93
5	64	64	Split	1.85	0.83	1.22	0.02	0.83	0.68	0.02	0.83	1.40
5	128	32	Split	3.39	1.00	1.33	0.11	1.71	1.56	0.11	1.62	2.76
5	256	32	Split	25.08	0.92	1.10	0.67	11.45	11.03	0.68	10.57	16.37
5	128	64	Split	10.99	0.83	1.02	0.11	4.79	4.23	0.11	4.43	6.67
5	64	128	Split	5.93	0.79	1.05	0.02	2.65	1.98	0.02	2.73	3.45
5	32	256	Split	3.78	0.95	1.17	0.01	2.25	1.32	0.00	2.18	2.22
5	64	256	Split	19.60	0.82	0.99	0.02	9.66	6.46	0.02	9.93	9.41
5	128	128	Split	35.58	0.81	0.94	0.11	15.76	13.00	0.11	15.23	18.17
5	256	64	Split	80.28	0.80	0.90	0.69	32.82	30.59	0.69	28.56	42.79
5	256	128	Split	257.45	0.78	0.82	0.67	106.21	93.30	0.68	94.05	116.64
5	128	256	Split	112.35	0.88	0.93	0.11	55.75	43.00	0.11	55.04	49.66
5	64	512	Split	62.00	0.94	1.01	0.02	36.09	22.31	0.02	37.52	25.21
7	8	256	Split	0.56	0.97	1.14	0.00	0.36	0.18	0.00	0.28	0.36
7	128	128	Split	105.48	0.94	0.97	0.30	54.84	44.03	0.30	47.60	54.82
11	8	256	Split	0.67	0.97	1.35	0.00	0.42	0.23	0.00	0.34	0.56
11	8	512	Split	2.22	0.99	1.08	0.00	1.50	0.69	0.00	1.14	1.26
13	4	512	Split	0.13	0.95	2.18	0.00	0.06	0.06	0.00	0.08	0.21
13	4	1024	Split	0.42	0.91	1.56	0.00	0.23	0.16	0.00	0.29	0.37
13	32	256	Split	11.17	0.91	1.22	0.01	5.90	4.25	0.01	5.70	7.87
13	4	2048	Split	1.39	0.96	1.53	0.00	0.91	0.43	0.00	1.42	0.71
13	32	512	Split	36.04	0.90	1.09	0.01	19.37	13.01	0.01	18.80	20.55
13	64	256	Split	65.39	0.97	1.14	0.06	36.27	27.12	0.06	33.56	41.24
13	128	256	Split	473.03	0.95	1.06	0.36	254.74	193.41	0.36	224.49	275.74
13	64	512	Split	208.16	0.97	1.07	0.06	116.97	84.03	0.06	109.83	113.69
13	128	512	Split	1470.15	0.94	1.00	0.36	791.85	587.40	0.36	712.66	759.20
17	32	256	Split	14.91	0.88	1.28	0.01	7.72	5.45	0.01	7.88	11.26
17	32	512	Split	47.08	0.81	1.11	0.01	22.72	15.26	0.01	23.38	28.74
17	64	256	Split	88.08	0.87	1.20	0.06	45.13	31.81	0.06	47.40	58.11
17	128	256	Split	585.80	0.99	1.22	0.32	339.87	238.68	0.32	326.88	385.81
17	64	512	Split	277.01	0.77	1.06	0.06	125.70	88.81	0.05	135.35	156.97
17	32	1024	Split	148.99	0.83	1.05	0.01	75.58	47.98	0.01	78.30	78.74
17	64	1024	Split	868.82	0.77	1.00	0.06	393.51	275.06	0.06	438.24	433.83
17	128	512	Split	1838.89	0.88	1.08	0.33	943.57	682.31	0.32	923.52	1057.06
5	256	256	Split	805.93	0.86	0.85	0.68	377.99	314.11	0.68	347.00	335.68
5	256	512	Split	2856.05	1.01	0.95	0.81	1587.23	1284.40	0.80	1615.68	1105.31
7	8	512	Split	1.86	1.02	0.97	0.00	1.36	0.54	0.00	0.98	0.82
7	8	1024	Split	5.99	1.17	0.93	0.00	5.32	1.71	0.00	3.66	1.92
7	128	256	Split	329.42	1.00	0.97	0.31	186.53	142.08	0.30	165.90	153.01

open problem to propose an optimized implementation of SRA together with parameters of practical interest for which SRA would consistently perform better than previous algorithms. On the algorithmic side, we believe that efficiency improvements can be achieved in SRA through a careful choice of the basis used in Lemma 1.

Our algorithm is radically different from previous ones. While traditional root-finding algorithms have used various strategies to separate the root set, SRA first "merges" the roots together using successive resultants with the polynomials $x_{j+1} - (x_j^p - a_j x_j)$, and it then progressively separates them using gcds and root-finding algorithms on polynomials of small degrees only. It would be interesting to explore alternative merging strategies, in other words to take successive resultants with polynomials $x_{j+1} - \tilde{L}_j(x_j)$ where the functions $\tilde{L}_j$ would be other non injective functions. An alternative multipoint evaluation method could then be used instead of the dedicated Frobenius approach of Section 4.1 to preserve the resultant computation complexity with fast arithmetic.[1]

To conclude this paper, we would like to mention a very interesting and important open problem. This problem is the extension of our work to solve *multivariate* polynomials $f(x_1, \ldots, x_m) = 0$ under *linear constraints* $x_i \in V_i$, where $V_i \subset \mathbb{F}_{p^n}$ are vector spaces of dimension $n' \approx n/m$ over $\mathbb{F}_p$. This problem is of great interest in cryptography, to the factorization problem in $SL(2, \mathbb{F}_{2^n})$ and to various discrete logarithm problems in small characteristic [6,7,11]. Following the same reasoning as in Section 3.1, we can write a polynomial system

$$\begin{cases} f(x_{1,1}, \ldots, x_{m,1}) = 0 & \\ x_{ij}^p - a_{ij} x_{ij} = x_{i,j+1} & i = 1, \ldots, m; j = 1, \ldots, n' - 1 \\ x_{i,n'}^p - a_{i,n'} x_{i,n'} = 0 & i = 1, \ldots, m \end{cases} \tag{6}$$

which includes the linear constraints and has a "block diagonal" structure. This system can clearly be solved by construction of new polynomials $f^{(i_1,\ldots,i_m)}$ where the variables are successively replaced with resultants as well. However, we have not been able to design an algorithm that does not increase the degree of the new polynomials, and we could therefore not provide any good complexity bound. Nevertheless, we believe that this approach is very promising. Besides proving Petit and Quisquater's conjecture to some extent [11], it may also lead to huge practical improvements on the cryptanalysis of ECDLP in characteristic 2 if the time and memory required to solve a multivariate polynomial with linear constraints were significantly decreased.

Acknowledgements The author would like to thank Tim Hodges, Sylvie Baudine and the program committee of ANTS for carefully reviewing previous versions of this paper. Nicolas Veyrat and Jean-Jacques Quisquater are also thanked for discussions related to this work. Finally, Jens Groth and Alan Lauder are thanked for hosting the author while writing this paper, respectively at University College London and University of Oxford. The research leading to these results has received funding from the Fonds National de

[1] We thank an anonymous reviewer of ANTS for this suggestion.

la Recherche - FNRS and from the European Research Council through the European ISEC action HOME/2010/ISEC/AG/INT-011 B-CCENTRE project.

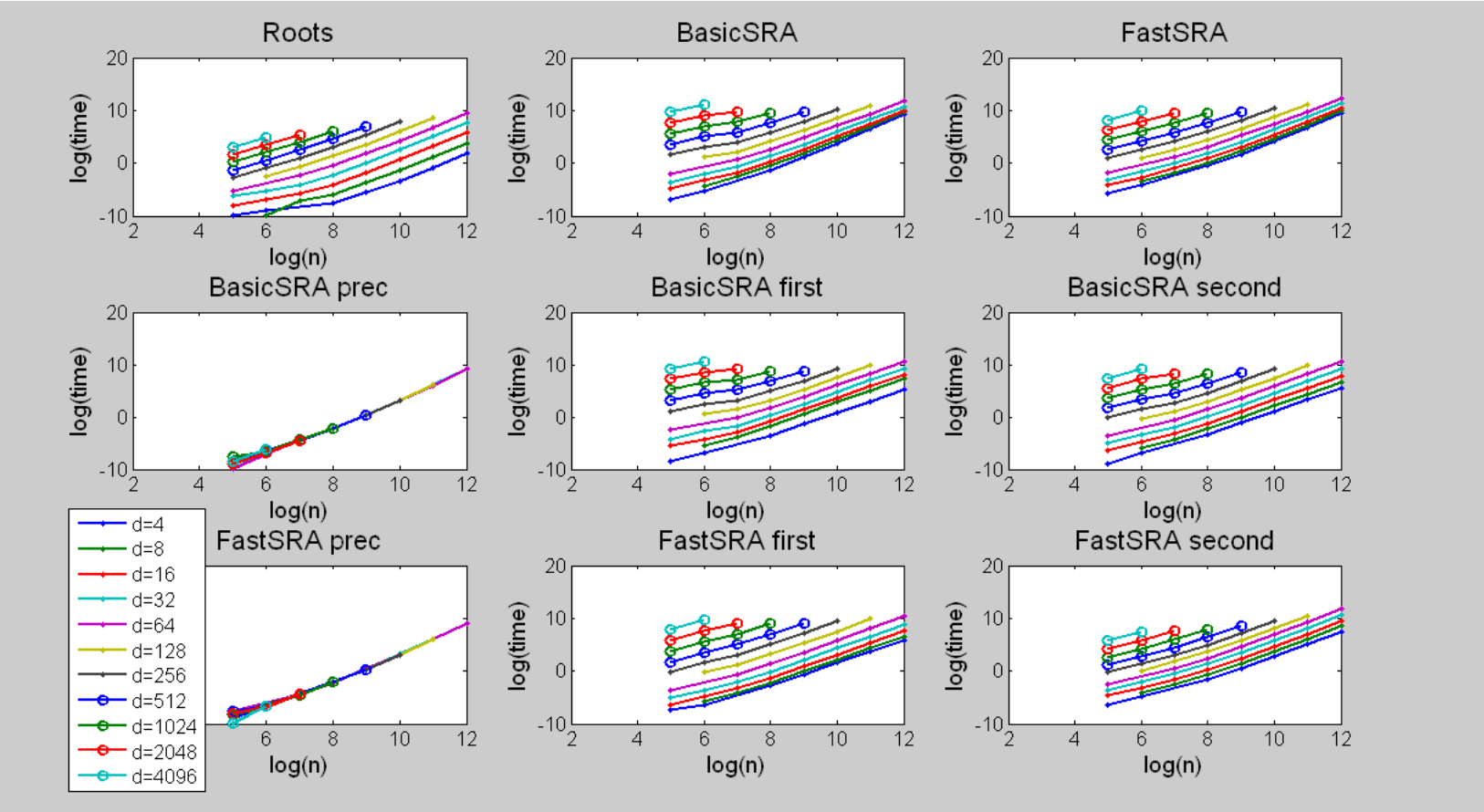

Fig. 1: $\log_2$ of computing times (in seconds) for Magma *Roots* function, basic SRA, fast SRA and their main component parts. The graphs display the curves for several d values.

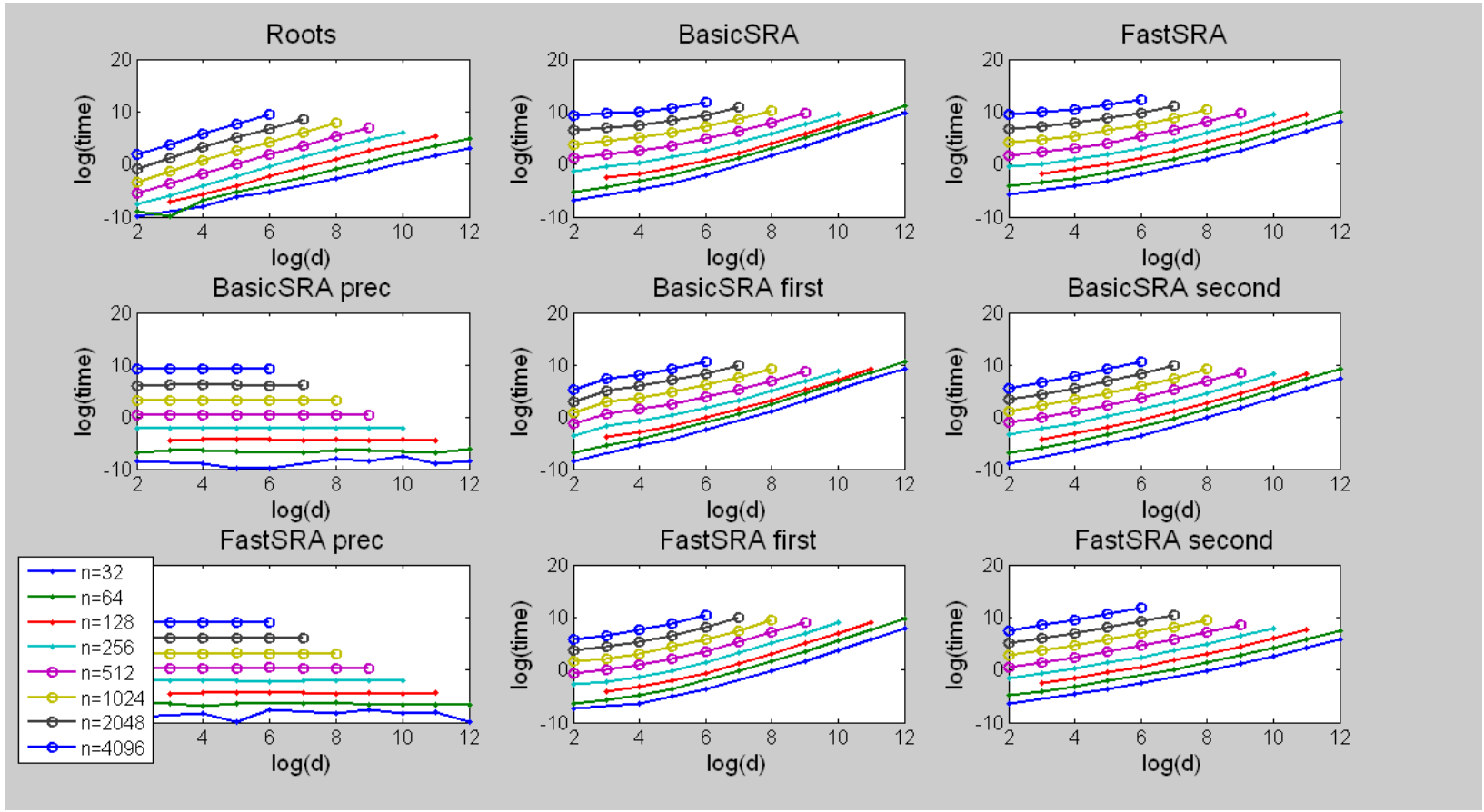

Fig. 2: $\log_2$ of computing times (in seconds) for Magma *Roots* function, basic SRA, fast SRA and their main component parts. The graphs display the curves for several n values.

References

1. E. Berlekamp. Factoring polynomials over large finite fields. *Mathematics of computation*, 111:713–735, 1970.
2. E. R. Berlekamp. *Algebraic coding theory*. Aegean Park Press, Laguna Hills, CA, USA, 1984.
3. Elwyn R. Berlekamp, H. Rumsey, and G. Solomon. On the solution of algebraic equations over finite fields. *Information and Control*, 10(6):553–564, June 1967.
4. David G. Cantor and Hans Zassenhaus. A new algorithm for factoring polynomials over finite fields. *Mathematics of Computation*, 36 (154):587592, 1981.
5. Yuanmi Chen and Phong Q. Nguyen. Faster algorithms for approximate common divisors: Breaking fully-homomorphic-encryption challenges over the integers. In Pointcheval and Johansson [12], pages 502–519.
6. Jean-Charles Faugère, Ludovic Perret, Christophe Petit, and Guénaël Renault. New subexponential algorithms for factoring in $SL(2, 2^n)$. Cryptology ePrint Archive, Report 2011/598, 2011. `http://eprint.iacr.org/`.
7. Jean-Charles Faugère, Ludovic Perret, Christophe Petit, and Guénaël Renault. Improving the complexity of index calculus algorithms in elliptic curves over binary fields. In Pointcheval and Johansson [12], pages 27–44.
8. Erich Kaltofen and Victor Shoup. Subquadratic-time factoring of polynomials over finite fields. *Math. Comput.*, 67(223):1179–1197, 1998.
9. Kiran S. Kedlaya and Christopher Umans. Fast polynomial factorization and modular composition. *SIAM J. Comput.*, 40(6):1767–1802, 2011.
10. Alfred Menezes, Paul C. van Oorschot, and Scott A. Vanstone. Subgroup refinement algorithms for root finding in GF(q). *SIAM J. Comput.*, 21(2):228–239, 1992.
11. Christophe Petit and Jean-Jacques Quisquater. On polynomial systems arising from a Weil descent. In Xiaoyun Wang and Kazue Sako, editors, *Asiacrypt*, volume 7658 of *Lecture Notes in Computer Science*, pages 451–466. Springer, 2012.
12. David Pointcheval and Thomas Johansson, editors. *Advances in Cryptology - EUROCRYPT 2012 - 31st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cambridge, UK, April 15-19, 2012. Proceedings*, volume 7237 of *Lecture Notes in Computer Science*. Springer, 2012.
13. Michael O. Rabin. Probabilistic algorithms in finite fields. *SIAM J. Comput.*, 9(2):273–280, 1980.
14. A. A. Schönhage. Schnelle Berechnung von Kettenbruchentwicklungen. *Acta Inf.*, 1:139–144, 1971.
15. Arnold Schönhage. Schnelle Multiplikation von Polynomen über Körpern der Charakteristik 2. *Acta Informatica*, 7 (4):395–398, 1977.
16. K. Thull and C. Yap. A unified approach to HGCD algorithms for olynomials and integers. Manuscript. Available from `http://cs.nyu.edu/cs/faculty/yap/allpapers.html/`, 1990.
17. Paul C. van Oorschot and Scott A. Vanstone. A geometric approach to root finding in GF(q^m). *IEEE Transactions on Information Theory*, 35(2):444–453, 1989.
18. Joachim von zur Gathen and Jürgen Gerhard. *Modern Computer Algebra (2. ed.)*. Cambridge University Press, 2003.
19. Joachim von zur Gathen and Daniel Panario. Factoring polynomials over finite fields: A survey. *J. Symb. Comput.*, 31(1/2):3–17, 2001.
20. Joachim von zur Gathen and Victor Shoup. Computing frobenius maps and factoring polynomials. *Computational Complexity*, 2:187–224, 1992.
21. C. Fieker A. Steel (eds.) W. Bosma, J. J. Cannon. Handbook of Magma functions, edition 2.20. `http://http://magma.maths.usyd.edu.au/magma/`, 2013.
22. Virginia Vassilevska Williams. Multiplying matrices faster than coppersmith-winograd. In Howard J. Karloff and Toniann Pitassi, editors, *STOC*, pages 887–898. ACM, 2012.

A Magma SRA code

```
// given a univariate polynomial over Fq, return its solutions
basicSRA:=function(pol)
```

```
// recover parameters
Rpol:=Parent(pol);
Fq<x>:=BaseRing(Rpol);
p:=Characteristic(Fq);
n:=Degree(Fq);
d:=Degree(pol);


// precomputing part
// ------------------
R:=PolynomialRing(Fq,n,"lex");
AssignNames(~R,["Y" cat IntegerToString(ind): ind in [1..n]]);

// arbitrary choice of basis
basis := [x^ind: ind in [0..n-1]];

// equations relating the Yi variables
Lval:=basis[1]^(p-1);
squareEq:=[];
for ind in [1..n-1] do
        // compute next
        squareEq:=squareEq cat [(R.ind)^p - Lval*R.ind - R.(ind+1)] ;

        // compute Lval = value of last equation evaluated at Y1=basis[ind]
        Lval:=R.1-basis[ind+1];
        for ind2 in [1..ind] do
                Lval:=NormalForm(squareEq[ind2],[Lval]);
        end for;
        Lval:=-MonomialCoefficient(Lval,1)
                                /MonomialCoefficient(Lval,R.(ind+1));
        Lval:=Lval^(p-1);
end for;
// compute last
squareEq:=squareEq cat [(R.n)^p - Lval*R.n] ;


// part depending on pol
// ---------------------
pol:=(hom<Parent(pol) -> R | R.1>)(pol);
polEq:=[pol];

// get an equation in R.n
for var in [1..n-1] do
        pol:=Resultant(pol,squareEq[var],R.var);
```

```
                polEq:=polEq cat [pol];
        end for;

        // compute gcd of pol with last squareEq
        pol:=GCD(pol,squareEq[n]);
        solEq:=[fac[1]: fac in Factorization(pol) | Degree(fac[1]) eq 1] ;

        // successively recover values of Y_n-1, Y_n-2, etc
        for indvar in [n-ind+1: ind in [2..n]] do
                newSolEq:=[];
                for sol in solEq do
                        pol:=GCD(polEq[indvar],NormalForm(squareEq[indvar],[sol]));
                        if Degree(pol) eq 1 then
                                newSolEq:= newSolEq cat [pol];
                        end if;
                        if Degree(pol) gt 1 then
                                newSolEq:= newSolEq cat  [fac[1]: fac in
                                        Factorization(pol) | Degree(fac[1]) eq 1] ;
                        end if;
                end for;
                solEq:=newSolEq;
        end for;

        return solEq;
end function;
```