Fast algorithms for computing the eigenvalue in the Schoof-Elkies-Atkin algorithm

Pierrick Gaudry, François Morain

Fast algorithms for computing the eigenvalue in the Schoof-Elkies-Atkin algorithm

P. Gaudry
LORIA, Vandoeuvre-Les-Nancy, France
LIX, École polytechnique, Palaiseau, France
gaudry@lix.polytechnique.fr

F. Morain
LIX, École polytechnique, Palaiseau, France
morain@lix.polytechnique.fr

ABSTRACT

The Schoof-Elkies-Atkin algorithm is the only known method for counting the number of points of an elliptic curve defined over a finite field of large characteristic. Several practical and asymptotical improvements for the phase called eigenvalue computation are proposed.

1. INTRODUCTION

The aim of the Schoof-Elkies-Atkin (SEA) algorithm is to compute the cardinality of elliptic curves defined over finite fields. While the case of small characteristic is more efficiently covered by p-adic methods (see [15, 11]), SEA is still the only efficient known method used in the case of large characteristic.

The SEA algorithm for a curve defined over a finite field $\mathbb{F}_q$ of large characteristic p performs a lot of polynomial operations of various natures. It computes modulo polynomials of degree ranging from 2 to about $\log q$. For these, all the standard machinery of asymptotically fast operations is used (see in particular [10] and [17]).

A lot of algorithms have been designed to make SEA efficient (see [1, 7, 14, 16]). The aim of this article is to describe a fast version of the search for the eigenvalue in the cases of Elkies primes as well as Schoof's basic algorithm. We explain the impact of these improvements on the recent records obtained for large q's.

The eigenvalue phase has been studied by Maurer and Müller in [13]. Their approach is based on a baby step giant step procedure. It splits in three computationnally important parts: first the computation of the initial data (X^q modulo the eigenfactor of the division polynomial); second the computation of the two lists of elements that are to be matched; third the finding of the match between the two lists. This third step is usually trivial for baby step giant step approaches, but in the present situation, the elements in the lists are computed and stored in a non-unique representation in order to avoid costly inversions modulo a large degree polynomial. We present theoretical and practical improvements for these three phases of the eigenvalue computation.

In Section 2, we recall briefly the SEA algorithm. In Section 3 we focus more precisely on the eigenvalue computation and distinguish the three main steps that are developped in Sections 4, 5 and 6, that contain the new material. In Section 7 we provide some practical experiments showing that some asymptotically fast algorithms are not to be used for the current range of applicability. We also give some data from a new record obtained using the techniques described in the following work.

We use the standard notation $\mathsf{M}(n)$ to designate the time needed to compute the product of two degree n polynomials over our base field.

2. THE SEA ALGORITHM

Throughout the article, let E denote an elliptic curve defined over $\mathbb{F}_q$ by an equation of the form $Y^2 = X^3 + AX + B$. We refer for instance to [2, 12] for the following facts.

There is a group law on an elliptic curve, that is known as the tangent-and-chord method. Over any field, the formulae are rational. Repeated use of this law leads to the introduction of *division polynomials*. Define the bivariate division polynomials in $\mathbb{F}_q[X, Y]$ associated to E via:

$$\Psi_{-1} = -1,\ \Psi_0 = 0,\ \Psi_1 = 1,\ \Psi_2 = 2Y,$$

$$\Psi_3 = 3X^4 + 6AX^2 + 12BX - A^2,$$

$$\Psi_4 = 4Y(X^6 + 5AX^4 + 20BX^3 - 5A^2X^2 - 4ABX - 8B^2 - A^3),$$

and for $n \geq 1$,

$$\Psi_{2n} = \Psi_n(\Psi_{n+2}\Psi_{n-1}^2 - \Psi_{n-2}\Psi_{n+1}^2)/(2Y),$$

$$\Psi_{2n+1} = \Psi_{n+2}\Psi_n^3 - \Psi_{n+1}^3\Psi_{n-1}.$$

A classical modification of the division polynomials leads to a univariate version which is more convenient to use. Indeed, if n is odd, after replacing each occurence of Y^2 by $X^3 + AX + B$, the polynomial Ψ_n becomes a polynomial in X only, that we call f_n. And if n is even, Ψ_n becomes of the form $2Y$ times a polynomial in X alone that defines f_n also in that case. The recurrence formulas for Ψ_n can be rewritten

in terms of f_n and we get

$$f_{2m} = f_m(f_{m+2}f_{m-1}^2 - f_{m-2}f_{m+1}^2), \quad (1)$$

$$f_{2m+1} = \begin{cases} f_{m+2}f_m^3 - R^2 f_{m+1}^3 f_{m-1} & \text{if } m \text{ is odd,} \\ R^2 f_{m+2}f_m^3 - f_{m+1}^3 f_{m-1} & \text{if } m \text{ is even,} \end{cases} \quad (2)$$

where $R = 4(X^3 + AX + B)$.

The degree of f_n is $(n^2-1)/2$ if n is odd and $(n^2-4)/2$ if n is even. These polynomials describe the multiplication by n of a point on E in a very explicit way:

PROPOSITION 2.1. *For any $P = (X, Y)$ in E and for any $n \geq 1$, we have*

$$[n]P = \left(X - \frac{\Psi_{n-1}\Psi_{n+1}}{\Psi_n^2}, \frac{\Psi_{n+2}\Psi_{n-1}^2 - \Psi_{n-2}\Psi_{n+1}^2}{4Y\Psi_n^3} \right).$$

Furthermore, if P is not a 2-torsion point, then $[n]P = 0$ if and only if $f_n(X) = 0$.

Over an algebraic closure of $\mathbb{F}_q$, let φ be the Frobenius endomorphism of E that sends (X, Y) to (X^q, Y^q). It verifies an equation of the form $\varphi^2 - t\varphi + q = 0$ and the number of $\mathbb{F}_q$-rational points of E is $\#E = q + 1 - t$. Hasse's bound ensures that the absolute value of the trace t is bounded by $2\sqrt{q}$. Schoof's algorithm proceeds by computing t modulo small primes ℓ using the action of φ on the set of ℓ-torsion points $E[\ell]$, until enough modular information is known to reconstruct the trace and the cardinality of E by the Chinese Remainder Theorem.

For every prime ℓ one needs to find the value of t mod ℓ such that the equation

$$\varphi^2(P) - [t]\varphi(P) + [q]P = 0 \quad (3)$$

holds for all P in $E[\ell]$. Using $f_\ell(X)$, this boils down to testing:

$$(X^{q^2}, Y^{q^2}) + [q](X, Y) = [t](X^q, Y^q) \quad (4)$$

where all operations are to be thought of as modulo $f_\ell(X)$, and modulo the equation of E.

In the so-called Elkies case, the characteristic polynomial of φ has two linear factors modulo ℓ; therefore the restriction of φ to $E[\ell]$ has two rational eigenspaces, one of these (call it V) being characterised by some polynomial of degree $(\ell-1)/2$, that we note $g_\ell(X)$ and which is a divisor of $f_\ell(X)$. The action of the Frobenius on $(X, Y) \in V$ is simply $\varphi(X, Y) = (X^q, Y^q)$, with the first term being reduced modulo $g_\ell(X)$ and the second modulo $g_\ell(X)$ and $Y^2 - (X^3 + AX + B)$. We shall be interested in computing the eigenvalue of φ, namely the integer k, $0 < k < \ell$ s.t. $\varphi(X, Y) = [k](X, Y)$ or:

$$(X^q, Y^q) = [k](X, Y). \quad (5)$$

Indeed, the trace modulo ℓ is then deduced from the formula $t \equiv k + q/k \mod \ell$.

Detecting the primes ℓ for which we are in the Elkies case amounts essentially to finding roots of the modular equation of degree ℓ. Then building the corresponding factor of the division polynomial is a non-trivial task for which several techniques exist. For those steps, we refer to [16, 14].

On a heuristic basis, for a general curve, we expect to be in the Elkies case for about half of the primes ℓ. Combining this with Hasse's bound and the prime number theorem, we obtain that, asymptotically, the largest ℓ we have to consider is about $\log q$ (here, log is the natural logarithm).

3. COMPUTING THE EIGENVALUE

We turn to the main question we are interested in, namely the computation of the eigenvalue in the SEA algorithm when ℓ is an Elkies prime: we are given a factor g_ℓ of the division polynomial, of degree $(\ell-1)/2$, and we want to find an integer k such that equation (5) is verified in the algebra $\mathcal{A} = \mathbb{F}_q[X, Y]/(Y^2 - (X^3 + AX + B), g_\ell(X))$.

Although this case is the most important one in practice, we put it in a slightly more general context, so that we also include the resolution of equation (4) and the isogeny cycle approach that finds t mod ℓ^r (see Remark 6.1 below).

Let us assume we are given P_1 and P_2 two points of E defined over an algebra $\mathcal{A}$ and we want to find a value k modulo ℓ such that $P_2 = [k]P_1$. This covers the resolution of equations (4) and (5), but the initialisation of the points is different. Also in the case of (4), k can be zero whereas in (5) this is not possible. Since $k = 0$ is easily detected anyway, we assume from now that k is in $[1, \ell - 1]$.

3.1 Baby Step Giant Step Approaches

Müller and Maurer [13] have designed several algorithms for solving the eigenvalue problem that are faster than plain enumeration of all k's that costs $O(\ell)$ operations. We describe several other ways of implementing these and analyze their cost in detail. The principle is to use some baby step giant step approaches.

The first approach is the classical one. Write k in base u as $k = cu + d$ where $0 \leq d < u$ and $0 \leq c < \ell/u$ (we could do slightly better by testing $\pm k$ at the same time). Then we rewrite our equation as:

$$P_2 - [c]([u]P_1) = [d]P_1.$$

We precompute all $[d]P_1$'s for a cost of u elliptic operations over $\mathcal{A}$ and have c vary, taking at most ℓ/u values. This algorithm is minimized for $u = \sqrt{\ell}$, yielding an algorithm that requires $O(\sqrt{\ell})$ operations in the elliptic curve to compute the points to match.

Since the group law on an elliptic curve is expensive, it is better to use a multiplicative decomposition of the value k we are looking for: the integer k is to be thought of modulo ℓ and cannot be zero modulo ℓ. Therefore we look for an identity of the form

$$[i]P_2 = [j]P_1$$

in E over $\mathcal{A}$. Taking two values I and J such that $IJ > \ell$, it is always possible to find $1 \leq i \leq I$ and $1 \leq \pm j \leq J$ such that $k = j/i$ (see for instance [10, §5.10]). Taking $I = J = \lceil \sqrt{\ell} \rceil$, we obtain again a complexity of $O(\sqrt{\ell})$ operations in the curve to compute the points to match. The difference with

the additive decomposition of k is that we now have pure multiples of a point on each side. This is easier to handle and allows to perform the computations with only the point abscissae.

REMARK 3.1. *Maurer and Müller have proposed another method. The idea is to use the fact that doubling a point is easy and we look for $k = j/2^i \bmod \ell$ for $1 \leq i, j \leq I, J$. This makes sense in the case where one computes in a non-projective manner, allowing divisions; otherwise doubling is not really faster than the use of division polynomials as presented in Section 5. For large values of ℓ, inversions must be avoided as much as possible, and the doubling strategy is no longer a competitive alternative. Furthermore, the values for I and J are expected to be $O(\sqrt{\ell})$ for most primes ℓ, but in the case where 2 has a small order modulo ℓ, this is not true.*

3.2 Elementary Steps of the Algorithm

The algorithm for finding k decomposes in the following elementary steps that will be studied in turn:

1. Compute initial data. This means basically computing P_1 and P_2, but depending on cases, only their abscissae might be required.
2. Compute the abscissa of $[i]P_1$ and the abscissa of $[i]P_2$ for $1 \leq i \leq \lceil\sqrt{\ell}\rceil$ in projective form.
3. Find a match between the two lists of rational fractions in $\mathcal{A}$.

3.2.1 Computation of initial data.

In the case of equation (4), one needs to compute X^q, Y^q, X^{q^2} and Y^{q^2}. We will give in Section 4 a fast way to deduce X^q from Y^q and X^{q^2} from Y^{q^2}. This means that only two (expensive) binary powerings are to be done.

In the case of equation (5), one needs X^q and Y^q. However if furthermore $\ell \equiv 3 \bmod 4$, a trick by Dewaghe [6] allows to fully determine the eigenvalue if it is known up to sign. It means that one can work with abscissae only and compute only X^q. Hence only one expensive binary powering is required. In the other case, where $\ell \equiv 1 \bmod 4$, Müller and Maurer were computing both X^q and Y^q in order to get the full signed eigenvalue. By our algorithm in Section 4, one can improve this and compute first Y^q by binary powering, and then deduce X^q.

3.2.2 Construction of the lists.

Once the abscissae of P_1 and P_2 are known, one computes the abscissae of multiples of them. We insist on the fact that even if the ordinates are known, since we are working in projective form and want several consecutive multiples, it is better not to use them at this stage. The match will give only the solution up to sign, and then using the ordinate or Dewaghe's trick the sign is computed. In Section 5 we show an efficient way of computing these multiples, thus improving by a constant factor the strategy of Müller and Maurer.

3.2.3 The matching problem.

Since we deal with projective coordinates to avoid inversions, the matching in the two lists is not immediate. Several approaches are possible, starting from redoing all the inversions, to completely avoiding them. We recall some of them and propose a new one in Section 6.

4. RECOVERING X^q FROM Y^q

Let $h(X)$ be a polynomial of degree n over $\mathbb{F}_q$. In the eigenvalue computation, h is g_ℓ. It is clear that computing $X^q \bmod h(X)$ and $Y^q = Y(X^3+AX+B)^{(q-1)/2} \bmod h(X)$ costs $O((\log q)\mathsf{M}(n))$ operations in $\mathbb{F}_q$. It turns out that we can in fact compute $X^q \bmod h(X)$ from the value of Y^q with less operations than by a direct binary powering computation.

4.1 Two Algorithms

We can write $F(X) = Y^{2q} \equiv (X^3+AX+B)^q \bmod h(X) \equiv (X^q)^3 + AX^q + B$. We note that $W = X^q$ satisfies $W^3 + AW + B = F(X) \bmod h$, but also that W is a root of h. Hence, we should recover X^q as the (hopefully unique) root of

$$\gcd(W^3 + AW + B - F(T), h(W)) \tag{6}$$

computed in $\mathcal{B}[W]$, where $\mathcal{B} = \mathbb{F}_q[T]/(h(T))$. This gcd is easy to compute, since all operations are performed over $\mathcal{B}$. Moreover, its cost is dominated by the first reduction, namely $h(W) \bmod (W^3 + AW + B - F(T))$, and is readily seen to be $O(n\mathsf{M}(n))$ operations in $\mathbb{F}_q$, which means that this approach will be faster than simply computing X^q when $n \ll \log q$. Some special code for computing this reduction has to be written, benefiting from the very special form of the cubic polynomial.

We can design a faster method as follows. Let $P(W) = W^3 + AW + B$, $H = P - F(T)$, so that $P = F \bmod H$. Write $h(W)$ in base P as:

$$h(W) = \sum_i h_i(W)P^i$$

with $\deg(h_i) \leq 2$. According to [10, Theorem 9.15], this costs $O(\mathsf{M}(n)\log n)$. Then:

$$\begin{aligned} h(W) \bmod H &\equiv \sum_i h_i(W)F^i \\ &= \sum_i (h_{i,0} + h_{i,1}W + h_{i,2}W^2)F^i \\ &= (\sum_i h_{i,0}F^i) + W(\sum_i h_{i,1}F^i) + W^2(\sum_i h_{i,2}F^i) \end{aligned}$$

for a cost of three modular compositions (modulo $h(X)$), or $O(n^{1/2}\mathsf{M}(n)+n^{(\omega+1)/2})$, where ω is the complexity exponent of matrix multiplication.

In the special case where h is a divisor of f_ℓ, one can replace (6) by:

$$\gcd(W^3 + AW + B - F(T), f_\ell(W)), \tag{7}$$

still to be computed in $\mathcal{B}[W]$. Even in the case where $f_\ell(W)$ has a degree which is higher than the degree of h, this can be worthwile, since f_ℓ can be evaluated quickly using (1) and

(2): at each step in the recursive process, the polynomials are reduced modulo $W^3 + AW + B - F(T)$, so that computing $f_\ell(W)$ modulo $W^3 + AW + B - F(T)$ requires requires $O(\log \ell)$ operations in $\mathcal{B}$, that is $O((\log \ell)M(n))$.

4.2 An Example

Consider the curve $E : Y^2 = X^3 + 2X + 3$ over $\mathbb{F}_q = \mathbb{F}_{1009}$. For $\ell = 13$, one eigenfactor is given by

$$g_{13}(T) = T^6 + 641T^5 + 755T^4 + 993T^3 + 468T^2 + 183T + 503,$$

$$\begin{aligned} F(T) &\equiv (T^3 + 2T + 3)^q \\ &= 974T^5 + 964T^4 + 475T^3 + 902T^2 + 945T + 832, \end{aligned}$$

$$\begin{aligned} &\gcd(W^3 + 2W + 3 - F(T), g_{13}(W)) \\ &\quad = W - (614T^5 + 667T^4 + 441T^3 + 130T^2 + 283T + 190) \end{aligned}$$

the root of which is easily checked to be indeed T^q mod $g_{13}(T)$.

4.3 Failures

The gcd could be trivial in rare cases. These are easy to discover and in that event, switching to the traditional computation of X^q is easy. In our numerous experiments this never happened, except for curves that we especially designed for the only purpose of making this method fail.

4.4 Application to Schoof's Basic Algorithm

As an application, we could speed up Schoof's original design. The optimal way to check (3) is as follows:

1. Compute $Y^q \equiv YZ(X) \equiv Y(X^3 + AX + B)^{(q-1)/2}$ mod $f_\ell(X)$;

2. Deduce X^q from Y^q as just shown;

3. Compute (X^{q^2}, Y^{q^2}) as $X^{q^2} = X^q \circ X^q$, $Y^{q^2} = (YZ(X))^q = YZ(X)(Z(X) \circ X^q)$ by two modular compositions with the same X^q;

4. Compute $(X^{q^2}, Y^{q^2}) + [q](X, Y)$, and then look for t.

Asymptotically, we would replace one modular composition in 3. by the same trick again for computing X^{q^2} from Y^{q^2}.

4.5 Application to our Original Problem

In the case of the eigenvalue computation for Elkies primes, we work modulo g_ℓ of degree $(\ell - 1)/2$. The complexity of computing X^q by binary powering is $O((\log q)\mathsf{M}(\ell))$. Deducing it from Y^q by a naive GCD computation has a complexity of $O(\ell\mathsf{M}(\ell))$. Since most of the time is spent on primes ℓ of size about $\log q$, this does not give any asymptotic improvement. Computing the GCD using modular compositions has a complexity of $O(\sqrt{\ell}\mathsf{M}(\ell) + \ell^{(\omega+1)/2})$ which is asymptotically faster by log factors for $\omega = 3$, and even better if one takes $\omega < 3$. And the last method of computing the GCD using f_ℓ has a complexity of $O((\log \ell)\mathsf{M}(\ell))$, which is asymptotically the best of the three methods.

5. COMPUTATIONS WITH DIVISION POLYNOMIALS

5.1 Incremental computation of multiples of a point

In order to test equation 5, one has to compute the consecutive multiples of a torsion point. This is not exactly the same question as the classical scalar multiplication problem where variants of the binary powering algorithm have been studied at length since this is the basic operation in cryptographic protocols.

We are in a context (operations modulo h) where inversions are very expensive compared to multiplications so the projective variants have to be used and the classical formulae will yield a cost of a dozen of multiplications per element to compute. In the following, we will see furthermore that in all cases we can work with (projective) abscissae only. We refer for instance to [4] for some details on how to compute efficiently n times a point in this context.

To compute all the multiples of a point, we use division polynomials, and improve slightly the constant in the complexity given in [13]. Assume that we have computed the values of f_i for $1 \leq i \leq n + 2$, evaluated at the abscissa of a point $P = (x, y)$. Then by the formulae of Proposition 2.1, we can deduce the abscissae of $[i]P$ for $1 \leq i \leq n$ in an affine or projective form.

After some initialisations to deal with small indices, assume that we have computed all the values of f_i up to $i < k$. Then f_k can be computed with the help of (1) or (2) depending on the congruence of k modulo 4. In order not to recompute several times the same values, we introduce the polynomials $U_i = f_{i-1}f_{i+1}$ and $V_i = f_i^2$, so that the formulae simplify to

$$f_{2m} = V_{m-1}U_{m+1} - V_{m+1}U_{m-1}, \tag{8}$$

$$f_{2m+1} = V_m U_{m+1} - R^2 V_{m+1} U_m, \tag{9}$$

if m is odd and

$$f_{2m+1} = R^2 V_m U_{m+1} - V_{m+1} U_m \tag{10}$$

otherwise, where $R = 4(x^3 + Ax + b)$ is the square of the ordinate of the point P.

Therefore, in order to compute all the f_i, U_i, V_i up to $i \leq n+2$, we need $3.5n + O(1)$ products and $n + O(1)$ squares in the ring where the abscissa x of the point P we want to multiply lives. The abscissae of the multiples of P can be written in the form $N_i(x)/D_i(x)$, where N_i and D_i are polynomials that are computed at a cost of two more multiplications using the formulae

$$\frac{N_i}{D_i}(x) = \begin{cases} x - \dfrac{U_i}{RV_i} & \text{if } i \text{ is even,} \\ x - \dfrac{RU_i}{V_i} & \text{if } i \text{ is odd.} \end{cases} \tag{11}$$

PROPOSITION 5.1. *For any $P = (x, y)$ in E, the abscissae of the points $[i]P$ for $1 \leq i \leq n$ can be computed in a projective form at a cost of $3.5n + O(1)$ products and $n + O(1)$ squares in the ring that contains x.*

5.2 Application to SEA

In Schoof's original algorithm, Proposition 5.1 is used with points (X^q, Y^q) as well as $(X^{q^2}, Y^{q^2}) + [q](X, Y)$ in the algebra $\mathbb{F}_q[X,Y]/(f_\ell(X), Y^2 - (X^3 + AX + B))$. In the eigenvalue finding phase of the SEA algorithm, the points concerned are (X, Y) and (X^q, Y^q) in the algebra $\mathbb{F}_q[X,Y]/(g_\ell(X), Y^2 - (X^3 + AX + B))$. In this latter case, we note that when the abscissa is precisely the element X that is used to build the extension of $\mathbb{F}_q$, as long as n is less than about the square root of the degree of g_ℓ, the polynomials f_i, U_i, V_i have a degree small enough so that no reduction modulo g_ℓ occurs, so that the computations are much faster than for the other abscissa. Furthermore, in that case, since X, R and R^2 are polynomials of small constant degrees, multiplication by them takes a negligible time. Therefore, computing the abscissae of the first n multiples of (X, Y) takes $3n + O(1)$ multiplications and $n + O(1)$ squares, most of them being between polynomials of small degrees and without any reduction modulo g_ℓ.

REMARK 5.1. *The difference (by a constant factor) in the complexities of computing the multiples of (X, Y) and (X^q, Y^q) means that in practice one should adjust the baby step giant step procedure in order to minimize the overall cost. However, tuning this can be done only for a particular implementation since the difference between the complexities is related to the cost of the multiplication of polynomials of all degrees from 1 to $(\ell - 1)/2$.*

6. TESTING FOR RATIONAL EQUALITY

The problem we want to adress is the following: we are given I 4-tuples of polynomials a_i, b_i, c_i, d_i for $1 \le i \le I \le n$ of degree less than n and a polynomial h of degree n. Assuming that b_i and d_i are invertible modulo h for all i, we want to find (if they exist) two indices i and j such that

$$\frac{a_i(X)}{b_i(X)} \equiv \frac{c_j(X)}{d_j(X)} \bmod h(X).$$

Computing all the possible crossproducts leads to a complexity which is quadratic in I, which annihilates the benefit of the baby step giant step approach.

6.1 Algorithm 1

The simplest idea for performing the task is to compute inverses. After computing the inverses of all the b_i and d_i modulo h, it amounts to $2I$ multiplications modulo h. Hence $2I$ inverses and $2I$ multiplications modulo h are enough to find the matching indices.

However, computing an inverse modulo h has an asymptotical cost of $O(\mathsf{M}(n)\log n)$ which is larger than for a multiplication. Quite often in our context, this theoretical prediction is also true in practice. Hence one wants to save inversions.

For that, the classical trick by Montgomery can be used: computing the $2I$ inverses can be done using only one inversion and $6I-6$ multiplications. Hence the complexity of the test is reduced to $8I - 6$ multiplications and 1 inversion.

We propose a variant that requires no inversions. The idea is to test $a_i/b_i \equiv c_j/d_j \bmod h$ by

$$\frac{a_i \prod_{k\neq i} b_k}{\prod_k b_k} \equiv \frac{c_j \prod_{k\neq j} d_k}{\prod_k d_k} \bmod h$$

or $\tilde{a}_i = \tilde{c}_j$ where

$$\tilde{a}_i = a_i \Big(\prod_{k\neq i} b_k\Big)\Big(\prod_k d_k\Big) \bmod h,$$

$$\tilde{c}_j = c_j \Big(\prod_{k\neq j} d_k\Big)\Big(\prod_k b_k\Big) \bmod h.$$

Let us explain how we can compute the $\tilde{a}_i$'s and $\tilde{c}_j$'s in a quick way. Denoting $a[k..l]$ the product $a_k a_{k+1} \cdots a_l$, we first evaluate the following quantities that we put in a diagram in which each line will give one of the a_i.

	$d[1..I]b[2..I]$	$a[1]$
$b[1]$	$d[1..I]b[3..I]$	$a[2]$
$b[1..2]$	$d[1..I]b[4..I]$	$a[3]$
...	...	...
...	$d[1..I]b[I-1..I]$	$a[I-2]$
$b[1..I-2]$	$d[1..I]b[I]$	$a[I-1]$
$b[1..I-1]$		$a[I]$
$b[1..I]$		

A similar picture is drawn for c_k and d_k. The first step is to compute the elements in the first columns. This costs $2I$ multiplications. The last element of the second picture is the complete product of the d_k. We will then compute the elements of the second column of the first picture using this product. And similarly, we compute the second column of the second picture using the complete product of the b_k that is taken from the first picture. This costs again $2I$ multiplications. Once the data in the picture are known, it suffices to multiply the elements in each line to get all the $\tilde{a}_k$ and $\tilde{c}_k$, at a cost of $4I$ multiplications. The total time is therefore $8I$ multiplications modulo h and *no* inversions.

6.2 Algorithm 2

This algorithm is due to Shoup. We recall it here and analyze its complexity.

Let $\vec{w}$ be a random vector of $\mathbb{F}_q^n$. For a polynomial $a(X)$ in $\mathbb{F}_q[X]$ of degree less than n, denote by $\vec{a}$ the coefficient vector of $a(X) = a_0 + a_1 X + \cdots + a_{n-1}X^{n-1}$, or more precisely $\vec{a} = [a_0, a_1, \ldots, a_{n-1}]$. Define the linear map associated to $\vec{w}$:

$$\begin{array}{rccc} L_w & : & \mathbb{F}_q[X]_{<n} & \to & \mathbb{F}_q \\ & & a(X) & \mapsto & \vec{a}\cdot\vec{w} \end{array}$$

where $\cdot$ denotes the scalar product of two vectors. Define the $n \times n$ matrix of the multiplication by $a(X)$ in $\mathbb{F}_q[X]/(h(X))$:

$$M_a = \left(\vec{a} | a\vec{X} | \cdots | a\vec{X^{n-1}}\right),$$

where the coefficient vector of $a(X)X^i \bmod h(X)$ forms the i-th column of M_a.

PROPOSITION 6.1. *For any $d(X)$ in $\mathbb{F}_q[X]/(h(X))$, one has*

$$L_w(a(X)d(X)) = ({}^tM_a\vec{w})\cdot\vec{d}.$$

If $a_i(X)d_j(X) - c_j(X)b_i(X) \neq 0$, then with high probability, $L_w(a_i(X)d_j(X) - c_j(X)b_i(X)) \neq 0$. Let us fix i in $[1, I]$, so that we have to evaluate $L_w(a_i(X)d_j(X))$ and $L_w(b_i(X)c_j(X))$ for j in $[1, I]$.

Using Proposition 6.1, if we have precomputed ${}^tM_{a_i}\vec{w}$ and ${}^tM_{b_i}\vec{w}$, then the check boils down to scalar multiplications. The matrix M_{a_i} is a matrix of multiplication and therefore the product of this matrix times a vector has a computational cost of $O(\mathsf{M}(n))$. By Tellegen's principle [18] the transpose of this matrix can also be multiplied by a vector within the same complexity. Hence the computation of ${}^tM_{a_i}\vec{w}$ and ${}^tM_{b_i}\vec{w}$ costs $O(\mathsf{M}(n))$ for each i. Then the dot products can be organized in matrices so that in the end the overall cost is $O(I\mathsf{M}(n) + nI^{\omega-1})$.

For the application to the SEA algorithm, we have $I \approx \sqrt{n}$, so that we have replaced the inversion by an operation which is asympotically much more costly. On the other hand the constants are much smaller.

6.3 Incremental Computations

Algorithms 1, 2 have been analyzed for worst-cases. In an average case analysis, assuming that the matching pair is uniformly distributed, we can improve the constant in front of the $I\mathsf{M}(n)$ part of the complexity by reordering the computations in Algorithms 1. The idea is to use the available data as soon as possible to test for a match. We skip the details and only mention that for Algorithms 1 one can get an average complexity of $6I$ modular multiplications, whereas the worst case requires $8I$ modular multiplications. In practice, we use these incremental versions.

6.4 Theoretical Comparison

In the case of the eigenvalue computation, we work modulo g_ℓ which is of degree $(\ell-1)/2$, and I is $\lceil\sqrt{\ell}\rceil$. The complexity of Algorithm 1 is then $O(\sqrt{\ell}\mathsf{M}(\ell))$ and the complexity of Algorithm 2 is $O(\sqrt{\ell}\mathsf{M}(\ell) + \ell^{(\omega+1)/2})$.

We assume that asymptotically fast algorithms are used for polynomial multiplication so that $\mathsf{M}(n)$ is $O(n \log n \log\log n)$ and we assume that naive algorithms are used for matrix multiplication so that $\omega = 3$. This corresponds to the implementation in NTL. Then Algorithms 1 has complexity $\tilde{O}(\ell^{1.5})$ and Algorithm 2 has complexity $\tilde{O}(\ell^2)$. Hence Algorithm 2 should be slower than Algorithm 1.

REMARK 6.1. *In the case where for the given ℓ the eigenvalue is found to have a small order in $\mathbb{F}_\ell^*$, it is interesting to continue the effort and try to get information modulo some power of ℓ as described in [5]. In this method, a factor of f_{ℓ^r} is found by pushing torsion points along a path of isogenies for increasing values of r, in a kind of Hensel lifting process. At each step of this lifting, one has to solve an equation similar to the one we have studied and a baby step giant step is to be used. The algorithms described in the present section apply also in this context, but this time we have to work modulo a polynomial of degree n of the order $d\ell^{r-1}$, where d is the order of the eigenvalue. The bound I is still $\sqrt{\ell}$ and we see that Algorithm 1 has the best asymptotical complexity of $\tilde{O}(d\ell^{r-0.5})$.*

7. PRACTICAL COMPARISONS

7.1 Range of Application

Remember that on a heuristic basis the maximum prime ℓ we are going to deal with is around $\log q$ (natural logarithm of q). In the first step of the SEA algorithm that we do not study here, we need to use the modular equation of degree ℓ. Computing these equations is a challenging task and we refer to [8] for a fast method of doing it. With today's technology, we consider that beyond $\ell = 10,000$, this algorithm takes a time which is no longer reasonable. It means that the maximum size of q we could handle today with the SEA algorithm is about 10^{4300} (but this computation would require a huge amount of resources).

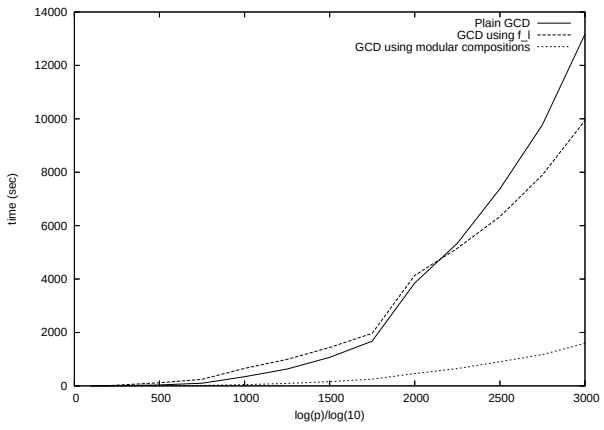
We have implemented the whole SEA algorithm with the different strategies proposed here. We used the C++ programming langage with the NTL library. We compare the different algorithms, keeping in mind the practical limits mentionned in the previous paragraph. All the running times are given on a 2.4GHz Opteron 250 processor. The non-smooth shapes of the curves displaying running times are due to the way the FFT is implemented in NTL, with big gaps at certain values.

7.2 Computing X^q from Y^q

We did experiments for primes q varying from 100 to 3000 decimal digits. For each prime we took the estimated maximal value of $\ell \approx \log q$, and we compared the different methods for computing the GCD of Section 4. The results are shown in Figure 1. Although the method with modular compositions is asymptotically the slowest one, it beats the other methods by a large factor in the current range of applications.

For comparison with the previous algorithm, the time to compute X^q by the binary powering would have been about 1450 seconds for 1000 digits (now 48 seconds), and 16700 seconds for 2000 digits (now 460 seconds).

Figure 1: Methods for computing the GCD when recovering X^q from Y^q.

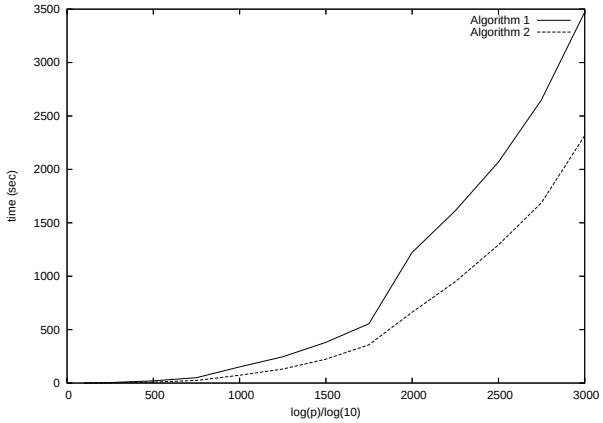


7.3 Testing Rational Equality

We did the same kind of measurements for the algorithms of Section 6. The theoretical complexity of the test of rational

equality using Algorithm 3 looks attractive compared to the complexity of Algorithm 2. However, in practice the constants in the complexity of Algorithm 2 are so small that for all values in the practical range it is faster than Algorithm 3, as demonstrated in Figure 2. Hence our improvements to previous algorithms do not lead to practical speed-ups for the currently feasible sizes.

Figure 2: Testing rational equality.



REMARK 7.1. *In order to validate our complexity analysis, we have done simulations to find a point where the asymptotically fast algorithms win against the others. We have kept q fixed to 2000 digits, and let ℓ grow, since the dependance in q just appears in the cost of the base field operations and is (in principle) well under control. For ℓ around 10^6, we could see (at last!) the algorithm using f_ℓ for the GCD and the rational equality test by Algorithm 1 becoming faster than the other algorithms.*

7.4 Some Data from a New Record

We computed the cardinality of $E : Y^2 = X^3 + 4589X + 91128$ over the prime field $\mathbb{F}_{10^{2004}+4863}$ (see the announcement [9]). A typical large value of ℓ is 4649. Computing $X^q \bmod \Phi_\ell(X, j(E))$ took 45,192 seconds (without counting the computation time for Φ_ℓ); computing g_ℓ took 500 seconds using the techniques described in [3]; from this, computing $Y^q \bmod g_\ell$ required 16,657 seconds and X^q from Y^q took only 458 seconds using the approach with modular compositions (in which 7 seconds were used to write h in base P, 222 to compute the three modular computations). In the eigenvalue phase, one takes $I = 69$; computing all c_i and d_i costed 310 seconds. One of the eigenvalue turns out to be $k = 43/14 = 4320 \bmod \ell$. The time for computing the 2×14 L functions was 73 seconds, the time for applying them $(13 \times 69 + 43)$ times was 105 seconds, and the time for evaluating a_i, b_i for $i \leq 14$ was 225 seconds.

8. CONCLUSIONS

We have presented several theoretical and practical improvements to the eigenvalue computation phase of the SEA algorithm. These improvements make this phase negligible compared to the main two operations that have to be done for each Elkies prime: computing X^q modulo the modular polynomial $\Phi_\ell(X, j)$ and computing X^q modulo the factor of the division polynomial $g_\ell(X)$. Another part that can be improved to become negligible is the computation of g_ℓ; this is work in progress [3].

The conclusion is that to obtain a fast implementation of the SEA algorithm one only has to concentrate on the optimization of the implementation of the powering of X modulo a polynomial, which is a classical building block in polynomial factoring algorithms.

Acknowledgments. We thank B. Salvy and É. Schost for many discussions on fast polynomial algorithms. A. Enge's help was invaluable at several stages of this work. Besides providing us large modular polynomials he also suggested many improvements to the article.

9. REFERENCES

[1] A. O. L. Atkin. The number of points on an elliptic curve modulo a prime (II). Available on `http://listserv.nodak.edu/archives/nmbrthry.html`, 1992.

[2] I. Blake, G. Seroussi, and N. Smart. *Elliptic curves in cryptography*, volume 265 of *London Math. Soc. Lecture Note Ser.* Cambridge University Press, 1999.

[3] A. Bostan, F. Morain, B. Salvy, and É. Schost. Fast algorithms for computing isogenies between elliptic curves. In preparation, January 2006.

[4] É. Brier and M. Joye. Weierstraß elliptic curves and side-channel attacks. In D. Naccache and Pascal Paillier, editors, *Public Key Cryptography*, volume 2274 of *Lecture Notes in Comput. Sci.*, pages 335–345, 2002.

[5] J.-M. Couveignes and F. Morain. Schoof's algorithm and isogeny cycles. In L. Adleman and M.-D. Huang, editors, *Algorithmic Number Theory*, volume 877 of *Lecture Notes in Comput. Sci.*, pages 43–58. Springer-Verlag, 1994. 1st Algorithmic Number Theory Symposium - Cornell University, May 6-9, 1994.

[6] L. Dewaghe. Remarks on the Schoof-Elkies-Atkin algorithm. *Math. Comp.*, 67(223):1247–1252, July 1998.

[7] N. D. Elkies. Elliptic and modular curves over finite fields and related computational issues. In D. A. Buell and J. T. Teitelbaum, editors, *Computational Perspectives on Number Theory: Proceedings of a Conference in Honor of A. O. L. Atkin*, volume 7 of *AMS/IP Studies in Advanced Mathematics*, pages 21–76. American Mathematical Society, International Press, 1998.

[8] A. Enge. Computing modular polynomials in quasi-linear time. In preparation.

[9] A. Enge, P. Gaudry, and F. Morain. Computing $\#E(GF(10^{2004} + 4863))$. `http://listserv.nodak.edu/archives/nmbrthry.html`, December 2005.

[10] J. von zur Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, 1999.

[11] Kiran S. Kedlaya. Counting points on hyperelliptic curves using Monsky-Washnitzer cohomology. *J. Ramanujan Math. Soc.*, 16(4):323–338, 2001.

[12] S. Lang. *Elliptic curves, diophantine analysis*. Springer, 1978.

[13] M. Maurer and V. Müller. Finding the eigenvalue in Elkies' algorithm. *Experiment. Math.*, 10(2):275–285, 2001.

[14] F. Morain. Calcul du nombre de points sur une courbe elliptique dans un corps fini : aspects algorithmiques. *J. Théor. Nombres Bordeaux*, 7:255–282, 1995.

[15] T. Satoh. The canonical lift of an ordinary elliptic curve over a finite field and its point counting. *J. Ramanujan Math. Soc.*, 15:247–270, 2000.

[16] R. Schoof. Counting points on elliptic curves over finite fields. *J. Théor. Nombres Bordeaux*, 7:219–254, 1995.

[17] V. Shoup. A new polynomial factorization algorithm and its implementation. *J. Symbolic Comput.*, 20:363–397, 1995.

[18] V. Shoup. Efficient computation of minimal polynomials in algebraic extensions of finite fields. In *Proceeding ISSAC*, pages 53–58. ACM, 1999.