A Comparison and a Combination of SST and AGM Algorithms for Counting Points of Elliptic Curves in Characteristic 2

Pierrick Gaudry

Laboratoire d'Informatique (CNRS/UMR 7650), École polytechnique,
91128 Palaiseau Cedex, France
gaudry@lix.polytechnique.fr

Abstract. Since the first use of a p-adic method for counting points of elliptic curves, by Satoh in 1999, several variants of his algorithm have been proposed. In the current state, the AGM algorithm, proposed by Mestre is thought to be the fastest in practice, and the algorithm by Satoh–Skjernaa–Taguchi has the best asymptotic complexity but requires precomputations. We present an amelioration of the SST algorithm, borrowing ideas from the AGM. We make a precise comparison between this modified SST algorithm and the AGM, thus demonstrating that the former is faster by a significant factor, even for small cryptographic sizes.

1 Introduction

In the design of an elliptic public key cryptosystem, parameter initialization is a difficult task; it is required to count points of curves until one is found with almost prime group order. In the early ages of elliptic curve cryptography, the only way to achieve this was to use curves with special properties like having complex multiplication by a small discriminant or being supersingular, though taking random curves might be seen as the most secure. The first polynomial time algorithm for point-counting was designed in 1985 by Schoof [11], but was not fast enough to deal with cryptographical sizes. A decade of theoretical and practical improvements by Atkin, Couveignes, Dewaghe, Elkies, Lercier, Morain, Müller lead to a situation where point-counting was efficiently feasible for cryptographical sizes (see the survey [1] and the references therein). However, the cost of parameter initialization remained high (in runtime and in complexity of programming) compared to other systems like RSA or XTR. Situation changed in 1999 when Satoh [8] proposed a new algorithm for counting points of elliptic curves over finite field of small characteristic. His method is based on the computation of the canonical lift of the curve in a p-adic local ring. The theoretical complexity is asymptotic better than all the variants of Schoof's algorithm. Further work by Fouquet–Gaudry–Harley [2, 3], Skjernaa [12], Vercauteren et al. [13], Satoh–Skjernaa–Taguchi [9, 10], Hae Young Kim et al. [7] made this algorithm practical, in particular in characteristic 2, which is the most important in practice. Another closely related method, based on the algebraic-geometric mean (AGM) was found by Mestre, and an implementation by Harley [5] proved it to be very efficient. For a curve over $\mathbb{F}_{2^n}$, the complexity of all these method is in $O(n^{3+\varepsilon})$, except for the Satoh–Skjernaa–Taguchi (SST) method which achieves a complexity in $O(n^{2.5+\varepsilon})$ but requires precomputations.

In characteristic 2, with the current state of the art, AGM is thought to be the fastest for cryptographical sizes, due to a very small constant, and is also the best algorithm for computing records. The first drawback of the SST algorithm is the precomputation stage which is not feasible for records, but this is not at all a problem for cryptographical sizes, where this is easily doable and the storage of the precomputed data is manageable. Another problem in the SST method is that the defining polynomial for the local ring is dense, thus increasing the cost of a multiplication by a factor of 3 compared to the sparse structure used in AGM.

Our contribution is two-sided: firstly we mix ideas of SST and AGM to get what we call the modified SST algorithm (MSST) which is faster by a constant factor compared to the original SST algorithm. Our improvement modifies only the lifting phase; the norm computation which is common to both algorithm is not modified. Then we make a precise comparison between MSST and AGM algorithms, based on an evaluation of the number of operations required at each precision. This turns out to be in favor of MSST, even for small cryptographical sizes, as confirmed by some experiments we did on a Pentium III: for curves over $\mathbb{F}_{2^{163}}$ we get a speed-up by a factor of 4.15 and for curves over $\mathbb{F}_{2^{239}}$ the factor is 4.90.

The paper is organized as follows: in Section 2 we recall some basics and fix notations. In Section 3 and 4, we give a brief description of the original SST and AGM algorithms. Section 5 is devoted to the mix of SST and AGM algorithms. Section 6 contains a theoretical comparison between MSST and AGM methods, and Section 7 contains the numerical experiments.

2 General Setting and Notations

Let $\mathbb{F}_q$ be a finite field of characteristic 2, and let n be such that $q = 2^n$. Let E be a non-supersingular elliptic curve over $\mathbb{F}_q$. For the purpose of point-counting, without lost of generality, and perhaps considering the quadratic twist of E, we can assume that E has an equation of the form $y^2 + xy = x^3 + a_6$. Then its j-invariant is given by $j = a_6^{-1}$. Denote by N the group order of E. The *trace* of E is defined by $\mathrm{Tr}(E) = q + 1 - N$ and Hasse's theorem states that $|\mathrm{Tr}(E)| \leq 2\sqrt{q}$.

All the p-adic point-counting methods proceed in the same way: lift some data from $\mathbb{F}_q$ to a p-adic local ring with enough precision, and deduce a p-adic approximation of the $\mathrm{Tr}(E)$ which might be enough to conclude due to Hasse's bound.

In our case, the 2-adic local ring we shall consider is the ring of integers of the degree n unramified extension of $\mathbb{Q}_2$. In the following we denote this ring by $\mathbb{Z}_q$ (noted $W(\mathbb{F}_q)$ in some papers, or simply R in [10]). Note that $\mathbb{Z}_q$ has nothing to do with $\mathbb{Z}/q\mathbb{Z}$, just like the ring of 2-adic integers $\mathbb{Z}_2$ is not $\mathbb{Z}/2\mathbb{Z}$. The ring $\mathbb{Z}_q$ is equipped with a cyclic $\mathbb{Z}_2$-automorphism of order n which reduces to the 2-nd power Frobenius automorphism of $\mathbb{F}_q$.

We give a constructive way to see the ring $\mathbb{Z}_q$. Let $\overline{f}(t)$ be the irreducible polynomial of degree n over $\mathbb{F}_2$ chosen to define $\mathbb{F}_q = \mathbb{F}_2[t]/(\overline{f}(t))$. Consider any monic polynomial $f(t)$ with coefficients in $\mathbb{Z}_2$ which reduces to $\overline{f}(t)$ modulo 2. Then $f(t)$ is irreducible and $\mathbb{Z}_q$ can be defined by $\mathbb{Z}_q = \mathbb{Z}_2[t]/(f(t))$. Different choices for the polynomial $f(t)$ lead to isomorphic rings. From an algorithmic point of view two strategies can be used: choosing a sparse $f(t)$ with small coefficients speeds up the basic arithmetic, because the reduction modulo $f(t)$ is almost for free; this is the representation used in the AGM algorithm. On the other hand, lifting $f(t)$ in a careful way can give a representation in which the Frobenius substitution is efficiently computable, the price to pay is a dense polynomial $f(t)$, hence a non-negligible reduction modulo $f(t)$; this is the representation used in the SST algorithm.

2.1 Notations in $\mathbb{Z}_q$

The ring $\mathbb{Z}_q$ comes with the natural "reduction modulo 2" homomorphism onto $\mathbb{F}_q$. For $x \in \mathbb{Z}_q$, the notation $x \bmod 2$ means the element of $\mathbb{F}_q$ image of x by this homomorphism. The ring $\mathbb{Z}_q$ also comes with a *valuation*. Let x and y be elements of $\mathbb{Z}_q$; the valuation of $(x-y)$ is high when x and y are close to each other. More precisely, the valuation of $(x-y)$ is k if $(x-y)$ is in $2^k\mathbb{Z}_q$; then we write $x \equiv y \mod 2^k\mathbb{Z}_q$, or simply $x \equiv y \mod 2^k$.

In an algorithm, when we say "compute $x := \ldots \mod 2^k$", this means that the expression for computing x involves quantities which are known to precisions such that the result is known to precision at least k. The variable x is then assigned an element which is congruent to the result modulo 2^k.

We use the same notation σ for *all* kinds of 2-nd power Frobenius action: ring/field automorphisms of $\mathbb{F}_q$ and $\mathbb{Z}_q$, and also their coordinate-wise extensions to isogenies from an elliptic curve to its conjugate. There should be no confusion, because all the domain of these maps are distinct and all the reduction-diagrams involving two of these σ commute.

2.2 Canonical Lift of an Elliptic Curve

It is easy to find many curves over $\mathbb{Z}_q$ whose equations reduce to the equation of E modulo 2. However, there is a canonical way to do it and keep information on the group order.

Theorem 1 (Lubin–Serre–Tate). *Let E be a non supersingular elliptic curve over $\mathbb{F}_q$. Then, up to isomorphism, there exists a unique curve $\mathcal{E}$ defined over $\mathbb{Z}_q$, such that:*

1. *The equation of $\mathcal{E}$ reduces to the equation of E modulo 2;*
2. $\mathrm{End}(\mathcal{E}) \cong \mathrm{End}(E)$.

A consequence of this theorem is the following commutative diagram

$$\begin{array}{ccc} \mathcal{E} & \xrightarrow{\sigma} & \mathcal{E}^\sigma \\ \downarrow & & \downarrow \\ E & \xrightarrow{\sigma} & E^\sigma \end{array}$$

where σ on the top arrow is the 2-nd power Frobenius isogeny from $\mathcal{E}$ to $\mathcal{E}^\sigma$. This isogeny being of degree 2, the modular equation of degree 2 relates the j-invariants of the canonically lifted curves:

$$\Phi_2(j(\mathcal{E}), j(\mathcal{E}^\sigma)) = 0,$$

where $\Phi_2(X, Y)$ is the symmetric bivariate polynomial

$$\begin{aligned} X^3 + Y^3 - X^2Y^2 + 1488(XY^2 + X^2Y) - 162000(X^2 + Y^2) \\ + 40773375XY + 8748000000(X + Y) - 157464000000000. \end{aligned}$$

Notation: Let c be an element of $\mathbb{Z}_q$ and let $\overline{c}$ be its reduction modulo 2. If $\overline{c}$ is not the j-invariant of a supersingular curve, then we denote by $c^\uparrow$ the j-invariant of the canonical lift of an elliptic curve whose invariant is $\overline{c}$.

3 Satoh–Skjernaa–Taguchi Algorithm

We do not describe all the details of the SST algorithm: we concentrate on the main two steps: firstly the j-invariant of the canonical lift of E is computed to some precision, then the norm of a quantity is computed, yielding the trace. We refer to the original paper [10] for a description of the missing steps.

3.1 Canonical Lifting of the j-Invariant

We are given $j(E)$ in $\mathbb{F}_q \setminus \mathbb{F}_4$, and we want to compute $j(\mathcal{E})$. The value of $j(\mathcal{E})$ is determined one bit after the other: assume that we know J such that $J \equiv j(\mathcal{E}) \mod 2^k$, then writing $j(\mathcal{E}) = J + 2^k e$, and plugging it formally into the equation $\Phi_2(j(\mathcal{E}), j(\mathcal{E}^\sigma)) = 0$, one gets an equation yielding e modulo 2. Hence we have gained one bit on the approximation of $j(\mathcal{E})$.

For a more precise setting, we take a Taylor expansion:

$$\begin{aligned} 0 &= \Phi_2(j(\mathcal{E}), j(\mathcal{E})^\sigma) = \Phi_2(J + 2^k e, J^\sigma + 2^k e^\sigma) \\ &= \Phi_2(J, J^\sigma) + 2^k e \partial_X \Phi_2(J, J^\sigma) + 2^k e^\sigma \partial_Y \Phi_2(J, J^\sigma) + 2^{2k-1}(\textit{element of } \mathbb{Z}_q). \end{aligned}$$

In this equation, $\Phi_2(J, J^\sigma)$ is zero modulo 2^k, we can therefore divide everything by 2^k. Furthermore the Kronecker relation implies that $\partial_X \Phi_2(J, J^\sigma)$ is zero modulo 2 and that $\partial_Y \Phi_2(J, J^\sigma)$ is different from zero modulo 2. Finally we have $e^\sigma \equiv e^2 \mod 2$ and we get

$$e^2 \equiv \frac{\Phi_2(J, J^\sigma)}{2^k \partial_Y \Phi_2(J, J^\sigma)} \mod 2.$$

To turn this into an algorithm, we need to apply σ to elements of $\mathbb{Z}_q$. For this, the polynomial defining $\mathbb{Z}_q$ over $\mathbb{Z}_2$ is chosen such that its roots are $(q-1)$-th roots of unity. This polynomial is precomputed once for all, for each base field. Hence Frobenius substitution is just reordering the coefficients and reducing modulo the defining polynomial (see [9] for details).

Then we get the following first lifting algorithm: (inverse of Frobenius is used, thus saving the computation of the square root of e in the previous formula)

Algorithm 1.

Input: j and a desired precision k.
Output: $j^{\uparrow}$ to precision k.

1. $d := 1/\partial_Y \Phi_2(\sigma^{-1}(j), j)$;
2. $y := j$;
3. For i from 1 to $k-1$ do
4. $\quad x := \sigma^{-1}(y) \mod 2^{i+1}$;
5. $\quad y := y - d\,\Phi_2(x, y) \mod 2^{i+1}$;
6. Return y;

In a point counting context, we need $k \approx \frac{n}{2}$. Running the i-th loop requires 1 Frobenius substitution and $O(1)$ multiplications of elements of $\mathbb{Z}_q$ at precision 2^{i+1}. Therefore the cost of Algorithm 1 is in $O(n^{1+2\mu})$ bit-operations, which is the cost of other lifting methods. Here μ is a real such that multiplication of k bit objects can be done in time $O(k^{\mu})$.

When looking at what is going on in Step 5, we see that $\Phi_2(x, y)$ is very close to zero and only the small non-zero piece of information is used to update y. It looks sub-optimal to recompute all the time $\Phi_2(x, y)$ from scratch: the values of x and y at step $i+1$ are close to the ones at step i, therefore $\Phi_2(x, y)$ at step $i+1$ can be deduced from its value at step i and some adjustment involving partial derivatives. By precomputing the partial derivatives modulo 2^W, one can update $\Phi_2(x, y)$ during W iterations, then one needs to recompute one time $\Phi_2(x, y)$ from scratch before doing again W iterations with only cheap updates. These ideas yield the SST lifting algorithm, a sketch of which is reproduced in Algorithm 2.

In [10] it is shown that Algorithm 2 runs in time $O(n^{2\mu+1/(\mu+1)})$, when one chooses $W = n^{\mu/(\mu+1)}$. Therefore, it is always better than Algorithm 1, because $\mu > 1$. If an FFT-based multiplication algorithm is used, $\mu = 1 + \varepsilon$, and we get a complexity of $O(n^{2.5+\varepsilon})$.

From the lifted j, a quantity can be derived which is a rational fraction in j, such that the norm of this quantity gives the trace of E.

Algorithm 2. SST canonical lifting

Input: j and a desired precision k; a parameter W.
Output: $j^{\uparrow}$ to precision k.

1. $y := j^{\uparrow} \mod 2^W$, computed via Algorithm 1;
2. $D_X := \partial_X \Phi_2(\sigma^{-1}(y), y) \mod 2^W$;
   $D_Y := \partial_Y \Phi_2(\sigma^{-1}(y), y) \mod 2^W$;
3. For m from 1 to $\lfloor \frac{k-1}{W} \rfloor$ do
4. Lift arbitrarily y modulo $2^{(m+1)W}$;
5. $V := \Phi_2(\sigma^{-1}(y), y) \mod 2^{(m+1)W}$;
6. For i from 0 to $W-1$ do
7. Compute $y := j^{\uparrow} \mod 2^{mW+i+1}$;
8. Update $V := \Phi_2(\sigma^{-1}(y), y) \mod 2^{mW+i+1}$;
   // Steps (7) and (8) use only operations modulo 2^W;
9. Return y;

3.2 Fast Norm Computation

In [10] a fast norm computation is described, that is well-suited to the case of point-counting in characteristic 2. It is based on the following equation:

$$\mathrm{Norm}(x) = \exp(\mathrm{Tr}(\log(x))),$$

which holds whenever log and exp converge. Computing a trace is far easier than computing a norm, and the subsequent exponential is very cheap. Therefore the main cost is a log evaluation, which is performed by fast evaluation of power series. We refer to the original paper for details, and will not discuss this part anymore, since the norm computation is a step which has to be done for any variant of the algorithm, and it is not the place where one is better than the other.

4 AGM Algorithm

We recall here the principles of the AGM algorithm. We give no proof of the results, they can be derived in the similar manner as for the other point-counting algorithms and are out of the scope of this paper.

4.1 The Arithmetic-Geometric Mean (AGM) Sequence

Let a_6 be an element of $\mathbb{F}_q^*$ and let E be the curve of equation $y^2 + xy = x^3 + a_6$ with j-invariant $j(E) = a_6{}^{-1}$. We denote also by a_6 an arbitrary element of $\mathbb{Z}_q$ that reduces to a_6 modulo 2. We then have recursive formulae which give a well-defined sequence (A_i, B_i) of elements of $\mathbb{Z}_q$:

$$A_0 = 1 + 8a_6, \quad B_0 = 1,$$

$$A_{i+1} = \frac{A_i + B_i}{2}, \quad B_{i+1} = \sqrt{A_i B_i},$$

where the square root is chosen to be congruent to 1 modulo 4. Indeed, by induction, we show that if $A_i \equiv B_i \equiv 1 \mod 4$ and $\frac{A_i}{B_i} = 1 + 8\alpha$ for an invertible α, then the squareroot is possible to be taken and if the one which is chosen is congruent to 1 modulo 4, the same properties hold at step $i + 1$.

This sequence (A_i, B_i) is called the *AGM sequence* and we can associate to it the sequence of elliptic curves E_i of equations $y^2 = x(x - A_i^2)(x - B_i^2)$. We denote by j_i the j-invariant of the curve E_i.

4.2 Link with Canonical Lifting

It is well known that AGM is linked to isogenies of degree 2 between elliptic curves. This in turn gives a link with the canonical lifting as follows:

Theorem 2. *Let j_i be the j-invariant of the curve E_i attached to the AGM sequence. Then the sequence j_i verifies:*

$$j_0 \equiv a_6^{-2} \mod 2,$$

$$j_{i+1} \equiv j_i^2 \mod 2,$$

$$j_i \equiv j_i^{\uparrow} \mod 2^{i+2}.$$

The first assertion shows that E_0 is isomorphic to the conjugate of a lift of the initial curve E modulo 2. The second relation states that all the curves E_i also reduce modulo 2 to conjugates of the curve E (up to isomorphism). The third one is the heart of the AGM algorithm: it means that when progressing along the AGM sequences, we get closer and closer to the canonical lift.

This yields immediately a straightforward algorithm for computing the canonical lift. Starting with the initial values of (A_0, B_0), we apply the recursive formula to compute successive values of (A_i, B_i). After k steps, we can compute the j-invariant of the associated curve which is close to the canonical lifting of a conjugate of E up to precision about 2^k.

The link with the trace is given by the following result:

Theorem 3. *Let $i > 0$ and let c_i be* $\mathrm{Norm}_{\mathbb{Z}_q/\mathbb{Z}_p}\left(\frac{A_{i+1}}{A_i}\right)$. *Then*

$$c_i + \frac{q}{c_i} \equiv \mathrm{Tr}(E) \mod 2^{i+4}.$$

A point-counting algorithm follows easily: one computes the AGM sequence with enough steps, then a norm computation gives the trace of the initial curve up to some precision which is equal to the number of steps plus a constant. A practical complication arises: on a computer one cannot really deal with elements of $\mathbb{Z}_q$, but with truncated ones. At first sight, it seems that we need a high starting precision for A_0 and B_0, because we get less and less significant digits on A_i and B_i when at the same time the j_i gets closer to the canonical lift. This problem can be overturned by adding arbitrary noise to A_i and B_i just before doing an operation which "loses" precision like a square root or a division by 2.

After having cleaned the details we get the following algorithm:

Algorithm 3. AGM point-counting

Input: a_6 in $\mathbb{F}_{2^n}^*$.
Output: Trace of the curve $y^2 + xy = x^3 + a_6$.

1. $a := 1 + 8a_6 \mod 16$; $b := 1 \mod 16$; $k := 4$;
2. Repeat until $k = \lceil \frac{n}{2} \rceil + 3$:
3. Lift arbitrarily a and b modulo 2^{k+2};
4. $(a,\ b) := \left(\frac{a+b}{2} \mod 2^{k+1},\ \sqrt{ab} \mod 2^{k+1} \right)$;
5. $k := k + 1$;
6. $a' := \frac{a+b}{2}$;
7. Return $\mathrm{Norm}(\frac{a'}{a}) \mod 2^{\lceil \frac{n}{2} \rceil + 2}$ as a signed integer in $[-2\sqrt{2^n}, 2\sqrt{2^n}]$.

4.3 Runtime Analysis

The AGM algorithm requires $O(n)$ operations between elements of $\mathbb{Z}_q$ with maximal precision in $O(n)$, and then a norm computation. As said before, the norm computation will not be discussed here, because it is the same in every algorithm. The cost of the lifting process is in $O(n^{1+2\mu})$, which is the same as Algorithm 1. Hence Algorithm 2 by Satoh–Skjernaa–Taguchi is asymptotically faster than the AGM for the lifting phase (but requires precomputation). However the AGM algorithm has a very low constant, due to the small number of operations at each step and the fact that a sparse defining polynomial for $\mathbb{Z}_q$ can be used. The figures given in [10] suggest that for cryptographical sizes, AGM remains faster.

5 AGM-Aided SST Algorithm

The AGM algorithm as stated before does not appear to be mixable with the SST idea. Therefore, before doing so we need to rewrite it in a univariate way to reveal the hidden modular equation which can then be used instead of Φ_2 in the SST algorithm. This is also this version of the AGM algorithm that we shall use for the comparison and the implementation in the next sections. We do not expect any speed difference between the univariate and the bivariate AGM.

5.1 Univariate AGM Algorithm

Taking again the AGM sequence as a starting point, we define a new sequence

$$\lambda_i = \frac{A_i}{B_i}.$$

The corresponding curves have equation $y^2 = x(x-1)(x-\lambda_i^2)$. An easy computation shows that λ_{i+1} can be computed directly from λ_i by

$$\lambda_{i+1} = \frac{1+\lambda_i}{2\sqrt{\lambda_i}}.$$

Another important fact is that $\lambda_{i+1} \equiv \lambda_i^\sigma \mod 2^{i+4}$, which corresponds to the fact that we are jumping from a curve to an approximation of its conjugate.

The corresponding univariate AGM algorithm is as follows:

Algorithm 4. Univariate AGM

Input: a_6 in $\mathbb{F}_{2^n}^*$.
Output: Trace of the curve $y^2 + xy = x^3 + a_6$.

1. $\lambda := 1 + 8a_6 \mod 16$; $\quad k := 4$;
2. Repeat until k is $\lceil \frac{n}{3} \rceil + 3$:
3. Lift arbitrarily λ modulo 2^{k+2};
4. $\lambda := \frac{1+\lambda}{2\sqrt{\lambda}} \mod 2^{k+1}$;
5. $k := k + 1$;
6. Return $\mathrm{Norm}(\frac{2\lambda}{1+\lambda}) \mod 2^{\lceil \frac{n}{2} \rceil + 2}$ as a signed integer in $[-2\sqrt{2^n}, 2\sqrt{2^n}]$.

Implementation of the Square Root. The main step of this algorithm is Step (4) in which we have to compute the inverse of the square root of λ and to multiply the result by $\frac{1+\lambda}{2}$. The inverse of the square root is computed via a Newton iteration that can be done without inversion. First we note that λ is always of the form $\lambda \equiv 1 + 8\alpha \mod 16$. Then $\frac{1}{\sqrt{\lambda}} \equiv 1 - 4\alpha \mod 8$, and this will be the initialization of the lift. Then the iteration is

$$x_{n+1} := x_n + \frac{x_n}{2}(1 - \lambda x_n^2).$$

If for some n, x_n is congruent to $\frac{1}{\sqrt{\lambda}}$ modulo 2^k, then one can show that x_{n+1} is equal to $\frac{1}{\sqrt{\lambda}}$ modulo 2^{2k-1} (see for instance Lemma 2.7 in [2]).

5.2 Modified Modular Equation

In the previous algorithm, the λ_i which is computed at the last step of the loop is (the conjugate of) a solution of the following equations:

$$Z \equiv 1 + 8a_6 \mod 16,$$

$$(Z^\sigma)^2(1+Z)^2 - 4Z \equiv 0 \mod 2^i.$$

But this is precisely this kind of system that SST algorithm is meant to solve. It remains to remove the leading non-significant bits in λ_i and to prove the same result on partial derivatives that made Algorithm 1 work.

Let $E(X,Y)$ be the AGM modular equation:

$$E(X,Y) = Y^2(1+X)^2 - 4X = 0.$$

We make a change of variables $X \leftarrow 1+8X$, $Y \leftarrow 1+8Y$, and the modular equation becomes:

$$\tilde{E}(X,Y) = (X+2Y+8XY)^2 + Y + 4XY = 0,$$

which has to be solved, subject to the conditions that X is known and non-zero modulo 2 and $Y = X^\sigma$.

The partial derivatives of $\tilde{E}$ evaluated at (X,Y) give

$$\begin{aligned}\partial_X \tilde{E}(X,Y) &= 2(X+2Y+8XY)(1+8Y) + 4Y \\ \partial_Y \tilde{E}(X,Y) &= (1+4X)(1+4(X+2Y+8XY))\end{aligned}$$

This proves that $\partial_X \tilde{E}(X,Y)$ is congruent to 0 modulo 2, whereas $\partial_Y \tilde{E}(X,Y)$ is 1 modulo 2, thus yielding the required asymmetry for the SST algorithm to converge. Note also that the partial derivative with respect to Y is 1 modulo 2, so that it is no longer necessary to compute d in Step (1) of Algorithm 1.

5.3 Modified Satoh–Skjernaa-Taguchi (MSST) Algorithm

According to the previous section, it is possible to use SST algorithm to compute the lifted invariant of the curve (or more precisely some kind of Legendre's invariant of the canonical lift of the curve). It remains to compute the data whose norm will give the result. This is actually much simpler than in the original SST algorithm: transposing the results of the AGM method, we can see that if λ is a solution of $\tilde{E}(X,X^\sigma)$ and $\lambda \equiv a_6 \mod 2$ then the following holds:

$$\mathrm{Tr}(E) \equiv \mathrm{Norm}\left(\frac{1}{1+4\lambda}\right) \mod 2^n.$$

We obtain Algorithm 5.

The advantage of the MSST algorithm is 2-sided: firstly the modular equation is smaller thus reducing by a constant factor the number of operations, secondly the intermediate step between the lift and the norm does not exist any more, thus simplifying the code and giving a slight speed-up.

6 Theoretical Comparison

The MSST algorithm is always faster than the plain SST algorithm because it involves strictly less operations. It remains to compare it to the AGM algorithm.

Algorithm 5. MSST point-counting

Input: a_6 in $\mathbb{F}_{2^n}^{\times}$.
Output: Trace of the curve $y^2 + xy = x^3 + a_6$.

1. $y := a_6$; // arbitrary lift to 2^W.
2. For i from 1 to $W-1$ do
3. $\quad x := \sigma^{-1}(y) \mod 2^{i+1}$;
4. $\quad y := y - \tilde{E}(x,y) \mod 2^{i+1}$;
5. $x := \sigma^{-1}(y) \mod 2^W$;
6. $D_X := \partial_X \tilde{E}(x,y) \mod 2^W$; $\quad D_Y := \partial_Y \tilde{E}(x,y) \mod 2^W$;
7. For m from 1 to $\lceil \frac{n}{2W} \rceil$ do
8. $\quad$ Lift arbitrarily y modulo $2^{(m+1)W}$;
9. $\quad x := \sigma^{-1}(y) \mod 2^{(m+1)W}$;
10. $\quad V := \tilde{E}(x,y) \mod 2^{(m+1)W}$;
11. $\quad$ For i from 0 to $W-1$ do // break if $i + mW \geq \lceil \frac{n}{2} \rceil$
12. $\qquad \Delta_Y := -2^{-mW} V \mod 2^W$;
13. $\qquad \Delta_X := \sigma^{-1}(\Delta_Y) \mod 2^W$;
14. $\qquad y := y + 2^{mW} \Delta_Y \mod 2^{(m+1)W}$;
15. $\qquad V := V + 2^{mW}(D_X \Delta_X + D_Y \Delta_Y) \mod 2^{(m+1)W}$;
16. Return $\mathrm{Norm}(\frac{1}{1+4y}) \mod 2^{\lceil \frac{n}{2} \rceil + 2}$ as a signed integer in $[-2\sqrt{2^n}, 2\sqrt{2^n}]$.

6.1 Constraint Environments

In a constraint environment it might be preferable to choose an algorithm for which no precomputation need to be stored and the RAM requirement stays low. In this context, the AGM algorithm (with the norm computation replaced by an extra loop) is by far the best choice. However, one should keep in mind that for a reasonable key-size, the amount of precomputed data to be stored for the SST algorithm is not so high: this is essentially two elements of $\mathbb{Z}_q$ at maximal precision; for instance, for $\mathbb{F}_{2^{163}}$, it is only a few kilo-bytes. This might be too much for a smart card, but this is not a problem on a PDA.

We consider as unlikely that someone really wants to count points in a highly constraint environment. Indeed, this kind of computation is required during the setup of the system parameters and the result does not need at all to be secret. Hence a card can ask to the server to do the computation if a new parameter setting is required.

In the following, we shall therefore concentrate on the case where we have no constraints and a machine word size of 32 bits (still the most common for PC's).

6.2 Assumptions

We recall that we are interested in cryptographically useful sizes. In that case, it has been shown in [10] that in the SST algorithm it is not worthwhile to use a W parameter different from the machine word size. Therefore we shall always consider that $W = 32$ is the optimal parameter for the MSST algorithm.

Another assumption we make is that multiplying integers of size less than the machine word size is not significantly faster than multiplying integers of size exactly the machine word size. At first sight, this assumption looks reasonable since the assembly instructions for multiplying bytes or short integers usually do not require much less cycles than the instruction for long integers. In fact this is a bit misleading because in both AGM and MSST algorithms several of these operations are parallelizable, one could therefore pack several small integers in a word size integer and perform several multiplication at once, or one could also use specific multimedia instruction like the MMX or SSE2 instruction set in the case of the Pentium. Using those could speed-up both the algorithms and we shall consider that we do not penalize one or the other by always using machine-word-size arithmetic.

6.3 Cost Analysis

In this section we compare the lifting parts of the AGM and the MSST algorithm. In MSST, we use a dense defining polynomial for $\mathbb{Z}_q$ in order to speed-up the Frobenius substitution computation. Therefore the reduction modulo the defining polynomial subsequent to a multiplication or a square is costly. On the other hand, in the AGM we have no Frobenius substitution to perform and it is possible to use a sparse defining polynomial, leading to an almost free reduction. Therefore, reductions will be counted in the MSST algorithm whereas we shall neglect them in the AGM. In MSST, with the dense polynomial, Frobenius substitution can be done at a cost of roughly one multiplication and one reduction, as explained in [9]. We do not count additions and other simpler operations.

We use the following notations for the basic operations in $\mathbb{Z}_q$:

P : unreduced product
S : unreduced square
R : reduction modulo defining polynomial

Furthermore, we can add an index i to each of these symbols to indicate the number of W-bit-digits of each operand.

MSST Algorithm The cost to evaluate the modular equation $\tilde{E}$ is $P+S+2R$. Each of its partial derivatives can be evaluated at a cost of $2P+2R$, and if one wants both derivatives, one can share the result of one product and get a cost of $3P+3R$.

Steps (2)-(4) cost $(W-1)(2P_1+S_1+3R_1)$.

Steps (5)-(6) cost $4P_1+4R_1$.

Next we analyze the cost of Steps (8)-(15) for each value of m:

Steps (9)-(10) cost $2P_{m+1}+S_{m+1}+3R_{m+1}$.

Steps (12)-(15) cost $W(3P_1+2R_1)$. Indeed, the multiplications by powers of 2 are for free, and one reduction can be saved at step (15). Actually, as explained in [10], some operations could be done on one bit operands.

If we need to lift to precision k, the total cost of MSST is then

$$(3k-W+2)P_1+(2k+W+1)R_1+(W-1)S_1+\sum_{1\le m\le\lceil\frac{k-W}{W}\rceil}(2P_{m+1}+S_{m+1}+3R_{m+1}).$$

We then apply this formula with $W=32$, for two base fields.

For $\mathbb{F}_{2^{163}}$, we need a precision $k=82$, and we get

$$C_{MSST}(163)=216P_1+31S_1+197R_1+2P_2+S_2+3R_2+2P_3+S_3+3R_3.$$

For $\mathbb{F}_{2^{239}}$, we need a precision $k=120$, and we get

$$\begin{aligned}C_{MSST}(239)=\;&330P_1+31S_1+273R_1+2P_2+S_2+3R_2+2P_3+S_3+3R_3\\&+2P_4+S_4+R_4.\end{aligned}$$

Hence we readily notice that for cryptographical sizes, only few operations require multiprecision arithmetic.

AGM Algorithm As said above, we study the univariate AGM instead of the bivariate one. The key step is then clearly Step (4), and in this step the crucial part is the Newton iteration for computing the inverse of $\sqrt{\lambda}$:

$$x_{n+1}:=x_n+\frac{x_n}{2}(1-\lambda x_n^2).$$

As usual for a Newton iteration, we need to have operations with variable precision. If we want to compute x_{n+1} with precision k from x_n known at precision roughly $k/2$, at first sight it requires one square and two products computed modulo 2^k. However this can be improved: $(1-\lambda x_n^2)$ is zero at precision $k/2$, because x_n is already a good approximation of the result. Hence when we multiply this further by x_n, this is actually a multiplication at precision $k/2$ that has to be performed. One step further is explained by Karp and Markstein in [6]: the "self-correctingness" of this iteration allows to compute λx_n^2 with two multiplications of an operand at precision k and the other at precision $k/2$.

We note however that in our case a square ideally costs roughly one half of a product and that this "Full times Half" precision product saves one fourth of the operations. Thus the trick of Karp and Markstein does not help in our case.

Estimating the number of operations in a Newton iteration is not that easy, due to the variable precision. To simplify the formulae we shall consider that the precision is exactly doubled at each step (whereas in fact one bit is lost). On the other hand this is easy to write a short program that emulate the algorithm and count the number of operations and the precision required, because there is no branching depending on the input data. Hence for a given base field, it is much simpler to run this emulation to evaluate the cost. We shall compare the results given by both approaches.

Cost of one lift in single precision. The cost of the Newton iteration to get the inverse of the square root at a precision between 2^{k-1} and $2^k < W$ is $(k-1)(2P_1 + S_1)$. After the lift one has to multiply the result by $\frac{1+\lambda}{2}$, which cost another P_1.

Cost of the first W iterations. We split the interval $[0, W]$ into pieces where the cost of the iterations are the same, and add them all. We get the following formula for the cost:

$$\sum_{2 \le k \le \log_2(W)} (k-1)2^{k-1}(2P_1 + S_1) + WP_1,$$

which simplifies to

$$(2 + W(\log_2(W) - 2))\,(2P_1 + S_1) + WP_1.$$

Cost of the W iterations between precision W and $2W$. The last step in a Newton iteration is done at precision of two W-bit digits and all the others are at precision 1. Furthermore, for each lift, the number of iteration at precision 1 is always $(\log_2(W) - 1)(2P_1 + S_1)$. Hence the cost of all the second W operations is

$$W\Big((\log_2(W) - 1)(2P_1 + S_1) + P_1 + 2P_2 + S_2\Big).$$

In this formula, we took into account the fact that in the last iteration at precision 2, one operation has only to be done at precision 1.

Cost of the W iterations between precision $2W$ and $3W$. The last step in a Newton lift is done at a precision of 3 digits, thus reducing to compute the result at a precision between W and $1.5W$. The penultimate step is therefore at a precision of 2 digits, and the remaining steps are done at precision 1. The overall cost is then

$$W\Big((\log_2(W) - 1)(2P_1 + S_1) + P_1 + 2P_2 + S_2 + 2P_3 + S_3\Big).$$

Cost of the W iterations between precision $3W$ and $4W$. The last step in a Newton lift is done at a precision of 4 digits, and the penultimate is done at a precision of 2 digits. The others steps are done at precision 1. Hence the following overall cost:

$$W\Big((\log_2(W) - 1)(2P_1 + S_1) + P_1 + 2P_2 + S_2 + 2P_4 + S_4\Big).$$

Cost of one iteration at a precision of k digits. The Newton iteration starts as always by $(\log_2(W) - 1)$ operations at a precision of 1 digit. Then there are $O(\log_2(k))$ operations at a precision of at most k digits. Including the cost of the multiplication by $\frac{1+\lambda}{2}$, there is at least $2P_k + S_k$.

We apply now our analysis to the same cases than for the MSST algorithm, namely $n = 163$ and $n = 239$, with $W = 32$.

We get

$$C_{AGM,th}(163) = 696P_1 + 306S_1 + 104P_2 + 52S_2 + 40P_3 + 20S_3,$$

and

$$C_{AGM,th}(239) = 1038P_1 + 458S_1 + 180P_2 + 90S_2 + 64P_3 + 32S_3 + 52P_4 + 26S_4.$$

With the emulation program mentioned above we get:

$$C_{AGM,emu}(163) = 694P_1 + 303S_1 + 109P_2 + 57S_2 + 45P_3 + 23S_3,$$

and

$$C_{AGM,emu}(239) = 1033P_1 + 452S_1 + 188P_2 + 98S_2 + 64P_3 + 32S_3 + 57P_4 + 29S_4.$$

These values are close enough to justify the simplifications we made in our analysis.

6.4 Comparison

We first compare the cost of the lifting up to precision $W = 32$, where all the operations that take place are single precision. We have to compare

$$62P_1 + 31S_1 + 93R_1 \quad \text{and} \quad 228P_1 + 98S_1.$$

This clearly depends on the relative costs of R_1 and P_1. It is always possible to do a reduction at the cost of two products, once a small precomputation is done (see [14], page 247). Hence $R_1 \leq 2P_1$. Note that in our implementation (see below) we got a ratio close to 1.5. Also S_1 is usually about 1.5 faster than P_1. With these ratios, the advantage is on MSST side.

Next, we compare the costs for gaining W bits of precision at a higher level. This corresponds to one iteration of Steps (8)-(15) in MSST or W loops of AGM. Let k be the number of digits corresponding to the precision. In MSST, the cost is

$$2P_k + S_k + 3R_k + 96P_1 + 64R_1,$$

whereas in AGM we have

$$64P_k + 32S_k + \cdots + 256P_1 + 128S_1,$$

where the dots contain operations at a precision strictly between 1 and k digits.

Hence MSST is clearly faster than than AGM, and the difference increases with the size of the basefield, due to the higher number of operations at multi-precision in AGM.

7 Practical Experiments

We implemented the MSST and the univariate AGM algorithm in the C programming language, using the GNU MP library [4] for the low-level integer multiplications. Multiplications in $\mathbb{Z}_q$ are done via Karatsuba algorithm. We wrote specific code for the machine word-size precision because in that case many things are simplified and this is critical in both algorithms. We give timings for two field sizes: $n = 163$ and $n = 239$. All the experiments are made on a Pentium III at 700 MHz running Linux. The compiler is `gcc` version 2.96.

Field size	Precision	Product	Square	Reduction
163	1 word	0.11 ms	0.07 ms	0.14 ms
	2 words	1.4 ms	0.92 ms	2.3 ms
	3 words	1.8 ms	1.3 ms	3.6 ms
239	1 word	0.21 ms	0.13 ms	0.29 ms
	2 words	2.5 ms	1.7 ms	4.9 ms
	3 words	3.4 ms	2.3 ms	6.8 ms
	4 words	5.4 ms	4.5 ms	10.8 ms

Field size	Lift MSST	Lift AGM	Norm computation	Total MSST	Total AGM
163	0.08 s	0.49 s	0.05 s	0.13 s	0.54 s
239	0.26 s	1.82 s	0.14 s	0.40 s	1.96 s

We see that at low precision, we were able to get a reduction step faster than two times a product. For the two given field sizes, the ratio between the two lifted methods is a factor of 6 to 7 in favor of the MSST algorithm. We implemented the norm computation to get significant runtimes for the complete point-counting computation. However this part is not as well optimized as the first step and the runtime we give might be improved. We only mention that in case of MSST algorithm, after the lift, we switch to the sparse representation before calling the same norm routine as the one used for AGM. This base conversion can be made very quick by precomputing the corresponding matrix. For cryptographical sizes, this matrix fits easily in a few mega-bytes, but for records, this strategy is not feasible. For the complete computation, the overall gain we have by choosing MSST instead of AGM is respectively by a factor of 4.15 and 4.90 for fields of 163 and 239 bits.

For comparison, we recall that the runtimes given in [10] for the original SST algorithm were 0.76s for a field of 163 bits and 2.54s for 239 bits on a 866 MHz Pentium III.

8 Conclusion

We presented a modification of the SST algorithm, using ideas taken from the AGM algorithm to speed-up the lifting phase and remove the second phase; the norm computation is unchanged. We did a precise theoretical and experimental comparison between our method and the AGM. We demonstrate that the number of operations is much smaller for the former. To illustrate this we implemented both methods with the same level of optimization — actually, they use the same time-critical functions. The gain is significant, even for small cryptographical field sizes.

For cryptographical applications, it is required to have an almost prime group order. Therefore it is usually necessary to count $O(\log(n))$ curves before finding one suitable for cryptography. To speed-up this search, in [3] the authors propose to mix the p-adic point-counting method with an early-abort strategy à la Schoof. Indeed for a small prime ℓ, it is possible to decide quickly whether the group order is divisible by ℓ, and if so to switch to another curve without running the p-adic algorithm. The size of the largest ℓ for which we do this depends on the relative costs of the point-counting algorithm and the early-abort. Since [3], the cost of point-counting has been greatly reduced, and the number of ℓ to consider must have diminished accordingly. Therefore it would be nice to also have new ideas in the early-abort stage to obtain another speed-up in the curve construction.

Acknowledgments

We thank Takakazu Satoh, Robert Harley and François Morain for valuable comments and remarks.

References

1. I. Blake, G. Seroussi, and N. Smart. *Elliptic curves in cryptography*, volume 265 of *London Math. Soc. Lecture Note Ser.* Cambridge University Press, 1999.
2. M. Fouquet, P. Gaudry, and R. Harley. An extension of Satoh's algorithm and its implementation. *J. Ramanujan Math. Soc.*, 15:281–318, 2000.
3. M. Fouquet, P. Gaudry, and R. Harley. Finding secure curves with the Satoh-FGH algorithm and an early-abort strategy. In B. Pfitzmann, editor, *Advances in Cryptology – EUROCRYPT 2001*, volume 2045 of *Lecture Notes in Comput. Sci.*, pages 14–29. Springer-Verlag, 2001.
4. T. Granlund. *The GNU Multiple Precision arithmetic library – 4.0.1.* Swox AB, 2002. distributed at `http://swox.com/gmp/`.
5. R. Harley. Counting points with the arithmetic-geometric mean (joint work with J.-F. Mestre and P. Gaudry). Eurocrypt 2001, Rump session.
6. A. Karp and P. Markstein. High precision division and square root. Technical Report 93-93-42, HP Labs, October 1994.
7. H. Kim, J. Park, J. Cheon, J. Park, J. Kim, and S. Hahn. Fast elliptic curve point counting using Gaussian normal basis. In C. Fieker and D. R. Kohel, editors, *ANTS-V*, volume 2369 of *Lecture Notes in Comput. Sci.*, pages 292–307. Springer–Verlag, 2002.
8. 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.

9. T. Satoh. On p-adic point counting algorithms for elliptic curves over finite fields. In C. Fieker and D. R. Kohel, editors, *ANTS-V*, volume 2369 of *Lecture Notes in Comput. Sci.*, pages 43–66. Springer–Verlag, 2002.
10. T. Satoh, B. Skjernaa, and Y. Taguchi. Fast computation of canonical lifts of elliptic curves and its application to point counting. Preprint 2001.
11. R. Schoof. Elliptic curves over finite fields and the computation of square roots mod p. *Math. Comp.*, 44:483–494, 1985.
12. B. Skjernaa. Satoh's algorithm in characteristic 2. To appear in *Math. Comp.*
13. F. Vercauteren, B. Preneel, and J. Vandewalle. A memory efficient version of Satoh's algorithm. In B. Pfitzmann, editor, *Advances in Cryptology – EUROCRYPT 2001*, volume 2045 of *Lecture Notes in Comput. Sci.*, pages 1–13. Springer-Verlag, 2001.
14. J. von zur Gathen and J. Gerhard. *Modern computer algebra*. Cambridge University Press, 1999.