Rethinking Low Genus Hyperelliptic Jacobian Arithmetic over Binary Fields: Interplay of Field Arithmetic and Explicit Formulæ

R. Avanzi[1], N. Thériault[2], and Z. Wang[2]

[1] Faculty of Mathematics and Horst Görtz Institute for IT-Security
Ruhr University Bochum, Germany
Roberto.Avanzi@ruhr-uni-bochum.de
[2] University of Waterloo, Department of Combinatorics and Optimization, Canada
ntheriau@math.uwaterloo.ca and z3wang@uwaterloo.ca

Abstract. This paper is an extensive study of the issues of efficient software implementation of low genus hyperelliptic Jacobians over binary fields.
We first give a detailed description of the methods by which one obtains explicit formulæ from Cantor's algorithm. We then present improvements on the best known explicit formulæ for curves of genus three and four. Special routines for multiplying vectors of field elements by a fixed quantity (which are much faster than performing the multiplications separately) are also deployed, and the explicit formulæ for all genera are redesigned or re-implemented accordingly.
To allow a fair comparison of the curves of different genera, we use a highly optimized software library for arithmetic in binary fields. Our goals in its development were to minimize the overheads and performance penalties associated to granularity problems, which have a larger impact as the genus of the curves increases. The current state of the art in attacks against the discrete logarithm problem is taken into account for the choice of the field and group sizes and performance tests are done on a personal computer.
Our results can be shortly summarized as follows: Curves of genus three provide performance similar, or better, to that of curves of genus two, and these two types of curves perform consistently around 50% faster than elliptic curves; Curves of genus four attain a performance level comparable to, and more often than not, better than, elliptic curves. A large choice of curves is therefore available for the deployment of curve based cryptography, with curves of genus three and four providing their own advantages as larger cofactors can be allowed for the group order.

Keywords. Elliptic and hyperelliptic curves, cryptography, efficient implementation, binary field arithmetic, explicit formulæ

1 Introduction

The *discrete logarithm problem* (DLP) is a quite popular primitive in the design of cryptosystems. It can be formulated as follows: Given a group G generated by an element g, and a second element $h \in G$, find some $t \in \mathbb{Z}$ with $t \cdot g = h$ (this integer is called the discrete logarithm of h with respect to g). The computation of scalar multiples of elements

of a group (i.e. given an integer t and an element g, compute $t \cdot g$) is the fundamental operation of a DLP-based cryptosystem.

Systems based on the DLP in the Jacobian of curves over finite fields were considered as early as 1985, when Miller [44] and Koblitz [27] independently proposed to use elliptic curves (EC). Shortly thereafter Koblitz [29] suggested the Jacobians of hyperelliptic curves (HEC) of higher genus (EC are HEC of genus one).

After a slow start, the use of EC in cryptography has now gained momentum and there are several books covering the subject (for example [6, 23, 7, 1]). HEC have been enjoying increasing attention in recent years. They have long been considered as not competitive with EC because of construction and performance issues, but the situation has changed in the last few years. It is now possible to efficiently construct HEC whose Jacobian has cryptographically good order (for an overview see [36]), and the performance has been considerably improved. For curves of genus two over binary fields, the fastest explicit formulæ are given in [34], and active research has also been done for other genera. A recent summary can be found in [9, 10], but of course research has not stopped: Recent improvements for genus three can be found in [13], and *further results for genus three and four are the main subject of the present paper.*

No subexponential algorithm is known for solving the DLP on elliptic and hyperelliptic curves of genus at most four (cfr. [3–5] for an overview of the techniques involved). For curves of genus one and two, the index calculus method is not faster than Pollard's methods, therefore the security level of these curves is assessed by the number of operations required to solve the DLP with Pollard's Rho algorithm.

For curves defined over a fixed field and increasing genus the complexity of computinging the discrete logarithm becomes subexponential in the group order [12] by using *index calculus* methods (see also [4]), but for "small", fixed genus the complexity of the methods remains exponential. Starting with genus three, one has to take into account a loss of security by a constant factor, and therefore one has to increase the field size because the index calculus techniques start to become faster than Pollard's methods.

On the other hand, the best algorithms for solving the factorization problem and the DLP in finite fields are subexponential. Therefore, to achieve a security increase equivalent to *doubling* the RSA key size, one needs to *add only a few bits* to an EC group. For example, according to [35] the security of 1323 bit RSA (or of a 137 bits subgroup of a 1024 bit finite field) is attained by an EC over a 157 bit field and with a group of prime order (of 157 bits). For that same level of security, the field would have 79 bits for a HEC of genus two, 59 bits for genus three and 53 bits for genus four (for all three, the curve must have a prime subgroup of order at least 157 bits); In comparison, the security of 2048 bits RSA is (roughly) achieved by 200-bits curve groups, and that of 3072 bits RSA by 240-bits curve groups [11]. NIST [49] suggests to use 224 and 256 bit groups in place of the values 200 and 240. There are obvious bandwidth and performance advantages in using curve based systems, in particular when the security requirements increase: Curves of genus up to three are now heavily investigated alternatives. *One of the purposes of this paper will be to show that systems based on curves of genus four may also prove interesting.*

In this paper we reconsider the issue of efficient implementation of low genus hyperelliptic Jacobians. Our focus is on curves over fields of characteristic two. We briefly recall the state of the art of implementation of HEC arithmetic in low genus: This will motivate our investigations.

Explicit formulæ for curves of genus two have been studied extensively [24, 40, 31, 51, 34, 33], both in the odd and even characteristic cases. An overview of the different results can be found in [10]. In particular, the fastest known algorithm for curves over binary fields makes use of the doubling formula by Lange and Stevens [34]: In that paper, the authors use Shoup's NTL software library [57] for the arithmetic in the base field, and obtain that the best case for curves of genus two can perform about 10% better than elliptic curves.

There are also a number of papers on explicit formulæ for curves of genus three [47, 31, 21, 13] in both odd and even characteristic, with a very complete description of all cases in [21]. A detailed comparisons between the performances of curves of genus one, two, and three in odd characteristic can be found in [2] and for characteristic two some comparisons are found in [59].

The literature is much more restricted when it comes to curves of genus four, with only the formulæ of Pelzl, Wollinger and Paar for characteristic two [51], and no detailed comparison with the performance of other genera that includes a security analysis and a careful choice of fields.

This paper reconsiders the issue of implementing low genus hyperelliptic Jacobian arithmetic over finite fields of even characteristic. A difference with respect to existing literature is that we simultaneously address field arithmetic enhancements, the derivation of the explicit formulæ and the impact of recent attacks, and consider the interplay of these factors. This produces surprising implementation results, which can be listed in the following two groupings:

1. A thorough approach to finite field implementation can be used to deliver performance essentially independent of the granularity of the computing architecture employed. In other words, the timings of the multiplication in the field is a close approximation of the quadratic bit complexity of (small) multiplication. Special finite field functions can be used to speed up explicit formulæ by up to 20% and the impact of such functions increases with the genus.
2. The genus two formulæ by Lange and Stevens in even characteristic can be used to beat EC performance by as much as 50% but curves of genus three deliver similar performance while using even smaller fields. Moreover, curves of genus four perform in fact quite well, often matching or improving EC performance.

Section 2 gives a detailed overview the general technique used to derive explicit formulæ. In Section 3 we describe our approach to implementation of the underlying field arithmetic: The technique of *sequential multiplications* is introduced. Our improvements to the explicit formulæ are presented in Section 4. Security considerations are the subject of Section 5, where key and field size equivalence issues are addressed. A description of our experiments and the corresponding results are given in Section 6. Finally, in Section 7 we draw conclusions.

2 From Cantor's algorithm to explicit formulæ

An excellent, low brow, introduction to hyperelliptic curves is [43], including elementary proofs of many facts used implicitly below. A more geometric presentation of the theoretical background is given in [1].

Let $\mathbb{F}_q$ be a finite field of characteristic two. Let us consider a hyperelliptic curve of genus g explicitly given by an equation of the form

$$C \ : \ y^2 + h(x)y = f(x)$$

over the field $\mathbb{F}_q$, with $\deg(f) = 2g+1$ and $deg(h) \leq g$.

Let ∞ be the point at infinity on the curve. In general, the points on a hyperelliptic curve do *not* form a group (the notable exception being represented by the hyperelliptic curves of genus one, i.e. the elliptic curves). Instead, the *divisor class group* of C is used: We briefly recall its main properties. The divisor class group is isomorphic to, and sometimes identified with, the algebraic variety called the *Jacobian* of C, which we do not define nor study here.

A *divisor* D is a formal sum of points on the curve, considered with multiplicities, or, in other words, any element of the free Abelian group $\mathbb{Z}[C(\overline{\mathbb{F}_q})]$. Its *degree* is the sum of those multiplicities, and its *support* the set of points with nonzero multiplicity. We are interested in the divisors of degree zero given by sums of the form

$$\sum_{i=1}^{k} m_i P_i - m\infty \quad : \quad P_i \in C \setminus \{\infty\} \tag{1}$$

where $m = \sum_{i=1}^{k} m_i$. The *degree* of the associated effective divisor $\sum_{i=1}^{k} m_i P_i$ is the integer m. The points P_i form the *finite support* of D. The *principal divisors* are the divisors of functions, i. e. those whose points are the poles and zeros of a rational function on the curve, the multiplicity of each point being the order of the zero or minus the order of the pole at that point. The *divisor class group* is the quotient group of the degree zero divisors modulo the principal divisors. In each divisor class there exists a unique element of the form (1) with (effective) degree $m \leq g$. Such an element is called a *reduced divisor*.

The group elements are these reduced divisors and they can be represented as pairs of polynomials $[u(x), v(x)]$ satisfying:

1. $\deg(u) \leq g$ (i.e. the roots of $u(x)$ are the x–coordinates of the points belonging to the finite support of the divisor);
2. $\deg(v) < \deg(u)$;
3. $v(x_{P_i}) = y_{P_i}$ for all $1 \leq i \leq m$ (i.e. the polynomial $v(t)$ interpolates these points);
4. $u(x)$ divides $v(x)^2 + v(x)h(x) - f(x)$.

This representation is usually attributed to Mumford [46]. If the first degree condition is not satisfied, the divisor is called *semi-reduced*.

For computation purposes, the group operation is based on Cantor's algorithm [8], that operates directly with elements in Mumford's representation. Cantor's original version worked only in odd characteristic and was extended for all fields by Koblitz [29]. Algorithm 1 gives the algorithm restricted to curves of characteristic two.

Algorithm 1. Group operation for hyperelliptic Jacobians in characteristic two

INPUT: Reduced divisors $D_1 = [u_1(x), v_1(x)]$ and $D_2 = [u_2(x), v_2(x)]$
OUTPUT: Reduced divisor $D_3 = [u_3(x), v_3(x)]$, $D_3 = D_1 + D_2$

1. **Composition:** $[u_C, v_C] = D_1 + D_2$ (semi-reduced)
2. $d_1 \leftarrow \gcd(u_1, u_2)$
   where $d_1 = s_1 u_1 + s_2 u_2$ [Extended Euclidean Algorithm]
3. $d(x) \leftarrow \gcd(d_1, v_1 + v_2 + h_2)$
   where $d = t_1 d_1 + t_2(v_1 + v_2 + h)$ [Extended Euclidean Algorithm]
4. $r_1 \leftarrow s_1 t_1$, $r_2 \leftarrow s_2 t_1$ and $r_3 \leftarrow t_2$
5. $u_C \leftarrow u_1 u_2 / d^2$
6. $v_C \leftarrow v_2 + \frac{u_2}{d} r_2 (v_1 + v_2) + r_3 \frac{v_2^2 + h v_2 + f}{d}$
7. **Reduction:** $D_3 = [u_3, v_3]$ (reduced)
8. $\tilde{u}_0 \leftarrow u_C$, $\tilde{v}_0 \leftarrow v_C$
9. **for** $i = 0$ **while** $\deg(\tilde{u}_i) > g$ **do**
10. $\tilde{u}_{i+1} \leftarrow Monic\left(\frac{\tilde{v}_i^2 + h\tilde{v}_i + f}{\tilde{u}_i}\right)$
11. $\tilde{v}_{i+1} \leftarrow \tilde{v}_i + h \bmod \tilde{u}_{i+1}$
12. $i \leftarrow i + 1$
13. $u_3 \leftarrow \tilde{u}_i$, $v_3 \leftarrow \tilde{v}_i$

Note that at step 6 we are simply computing $v_C(x)$ to be congruent to $v_1(x)$ modulo $u_1(x)/d(x)$ and congruent to $v_2(x)$ modulo $u_2(x)/d(x)$.

The idea behind explicit formulæ is to replace the polynomial-based form of Cantor's algorithm by a coefficient-based approach. These formulæ are case-specific, i.e. they depend on whether the divisors are distinct (*addition*) or equal (*doubling*), on the degrees of the polynomials involved, etc. (For a detailed case consideration in genus two see three see for example [33]; for genus three see [21].) This approach has a number of advantages which result in a significant speed-up in the computations:

- Conditional statements can be reduced to a minimum. Polynomial arithmetic is inherently dependent on conditional loops (mainly on the degree of the polynomial), which cannot be avoided in a general setting. Although checking a conditional statement (for example "is $k < \deg(u)$?") is not very expensive on its own, the cumulative impact over the whole algorithm should not be ignored.
- Coefficients which have no impact on the final result are no longer computed. This is quite evident in step 10 where we do not compute the coefficients of x of degree less than $\deg(\tilde{u}_i)$ in $\tilde{v}_i^2 + h\tilde{v}_i + f$, since we know that the division $\frac{\tilde{v}_i^2 + h\tilde{v}_i + f}{\tilde{u}_i}$ is exact and thus has no fractional part.
- In Cantor's algorithm, some of the partial computations may be done twice, with only the variable names being different. These duplications are avoided in the explicit formulæ – by keeping those values in memory.

– Parts of the algorithm can be replaced by more efficient techniques that cannot be used in a general setting. Sections 4.1 and 4.2 are good examples of these situations.

We also take advantage of the following observations:

– For almost all reduced divisors $D = [u(x), v(x)]$, $u(x)$ has degree g.
– For almost all pairs of polynomials u and v such that u divides $v^2 + hv + f$, v mod u has degree $\deg(u) - 1$.
– Almost all randomly chosen polynomials are relatively coprime.

(These are standard assumptions which are made by nearly every author in the development of explicit formulæ, beginning with Harley [24]). In all three of these cases, "almost-all" can be interpreted as "all but a proportion of size $O(g/q)$". This means that if we concentrate on developing additions formulæ which apply to the most general case (i.e. assuming that all polynomials have maximal degree and non-related polynomials are coprime) then only a negligible proportion of all group operations requires a different implementation. From the point of view of efficiency, we can handle all other cases with the general Cantor algorithm without having a noticeable impact on the computation of $[e]D$ (via a double-and-add approach), so only the general case is discussed here.

To reduce computational cost (mostly for the doublings), we restrict ourselves to curves of the form

$$y^2 + y = x^7 + f_5x^5 + f_3x^3 + f_1x + f_0 \tag{2}$$

for genus three and of the form

$$y^2 + y = x^9 + f_7x^7 + f_5x^5 + f_3x^3 + f_1x + f_0 \tag{3}$$

for genus four (the security of curves of these special forms is discussed in Section 5). As already mentioned, we consider only the most common case of the addition and doubling formulæ, i.e. when the degrees are maximal, and (for the addition formula) when u_1 and u_2 are coprime.

The form of the most common case for the addition and doubling formulæ are very similar for genus three and four (except for the degrees of the polynomials, obviously) since we need to go through the reduction loop twice to obtain a reduced divisor. The only major difference is that $\frac{\tilde{v}_1^2 + h\tilde{v}_1 + f}{\tilde{u}_1}$ is already monic in the genus three formulæ (since the leading coefficient in the denominator comes from $f(x)$) but must be made monic for the genus four formulæ.

Since the formulæ are in terms of the coefficients instead of polynomials, we will denote p_i the coefficient of x^i in $p(x)$. As there could easily be confusions with the polynomials $u_1(x)$, $u_2(x)$ and $u_3(x)$, as well as $v_1(x)$, $v_2(x)$ and $v_3(x)$, we will denote their coefficients differently:

– $u_1(x) = a_0 + a_1x + a_2x^2 + x^3$ (genus $g = 3$) or
  $u_1(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + x^4$ (genus $g = 4$);
– $u_2(x) = b_0 + b_1x + b_2x^2 + x^3$ ($g = 3$) or $u_2(x) = b_0 + b_1x + b_2x^2 + b_3x^3 + x^4$ ($g = 4$);
– $v_1(x) = c_0 + c_1x + c_2x^2$ ($g = 3$) or $v_1(x) = c_0 + c_1x + c_2x^2 + c_3x^3$ ($g = 4$);
– $v_2(x) = d_0 + d_1x + d_2x^2$ ($g = 3$) or $v_2(x) = d_0 + d_1x + d_2x^2 + d_3x^3$ ($g = 4$);

– $u_3(x) = e_0 + e_1x + e_2x^2 + x^3$ ($g = 3$) or $u_3(x) = e_0 + e_1x + e_2x^2 + e_3x^3 + x^4$ ($g = 4$);
– $v_3(x) = \varepsilon_0 + \varepsilon_1x + \varepsilon_2x^2$ ($g = 3$) or $v_3(x) = \varepsilon_0 + \varepsilon_1x + \varepsilon_2x^2 + \varepsilon_3x^3$ ($g = 4$).

We also replace $\tilde{u}_1(x)$ and $\tilde{v}_1(x)$ from steps 10 and 11 by $u_T(x)$ and $v_T(x)$.

2.1 Addition formula

For the addition, we want to compute $D_1 + D_2$ where $D_1 = [u_1(x), v_1(x)]$ and $D_2 = [u_2(x), v_2(x)]$, with $\deg(u_1) = \deg(u_2) = g$, $\deg(v_1) = \deg(v_2) = g-1$ and $\gcd(u_1, u_2) = 1$.

At step 2, we have $d_1 = \gcd(u_1, u_2) = 1$ with $s_1 \equiv u_1^{-1} \bmod u_2$ and $s_2 \equiv u_2^{-1} \bmod u_1$. For step 3, we simply have $d = \gcd(1, v_1 + v_2 + h) = 1$ with $t_1 = 1$ and $t_2 = 0$, so we get $r_1 = u_1^{-1} \bmod u_2$, $r_2 = u_2^{-1} \bmod u_1$ and $r_3 = 0$ at step 4. We then have

$$u_C = u_1u_2$$

at step 5 and

$$v_C = v_1 + u_1(u_1^{-1} \bmod u_2)(v_1 + v_2) \bmod u_C$$

at step 6.

The next idea consists of writing $v_C(x)$ in terms of multiples of $u_1(x)$, i.e. as $v_1(x) + s(x)u_1(x)$ where $s = (u_1^{-1} \bmod u_2)(v_1 + v_2) \bmod u_2$. Since we must do two reduction steps to obtain a reduced divisor, we can substitute the values of u_C and v_C (and $h(x) = 1$) in the first reduction step to simplify the equations:

$$\begin{aligned} u_T &= Monic\left(\frac{(v_1 + su_1)^2 + (v_1 + su_1) + f}{u_1u_2}\right) \\ &= Monic\left(\frac{(v_1^2 + s^2u_1^2 + v_1 + su_1 + f}{u_1u_2}\right) \\ &= Monic\left(\frac{v_1^2 + hv_1 + f}{u_1u_2} + \frac{s^2u_1}{u_2} + \frac{s}{u_2}\right) \end{aligned} \quad (4)$$

and

$$v_T = v_1 + su_1 + 1 \bmod u_T \quad .$$

By construction of u_C and v_C, the division in step 10 is exact, so we can look at the quotient (denoted $[\cdot]$) and ignore the fractional parts of each of the terms in Equation 4.

– $\left[\frac{v_1^2 + hv_1 + f}{u_1u_2}\right]$ is linear of the form $x + \ell$ where ℓ is $a_{g-1} + b_{g-1}$ (the sum of the coefficients of x^{g-1} in u_1 and u_2).
– $\left[\frac{s}{u_2}\right] = 0$ since $\deg(s) < \deg(u_2)$.
– The bulk of the computation is in $\left[\frac{s^2u_1}{u_2}\right]$. Even though su_1 is required to compute v_T, it is more efficient to compute s^2u_1 by first squaring s and then multiplying by u_1 (in s^2, odd powers of x have coefficient 0).

We now observe that the leading term in $\left[\frac{s^2u_1}{u_2}\right]$ (and therefore of $\frac{v_C^2+hv_C+f}{u_C}$ as well) is the square of the leading term of s. An easy way of making u_T monic is then to first make s monic and use this new polynomial (we call it $\tilde{s}$) in the computation of u_T. In terms of the number of operations, it is less expensive to compute $\tilde{s}$ and $\left[\frac{\tilde{s}^2u_1}{u_2}\right]$ and multiply $\left[\frac{v_1^2+hv_1+f}{u_1u_2}\right]$ by the corresponding factor than to compute the whole polynomial from s before making it monic.

On its own, this change has a minimal impact, but it becomes very successful when combined with another idea: Replacing $u_1^{-1} \bmod u_2$ (in the computation of s) by an *almost inverse* (the product of the inverse by a constant r). Many methods to compute $u_1^{-1} \bmod u_2$ do in fact compute an almost inverse first and then multiply it by r^{-1} to find the real inverse. n the extended Euclidean algorithm for example, one can delay all the field inversions and work with multiples of the intermediate polynomials by a common factor, thus computing an almost inverse (the final common factor is our constant r). Then, the inverse is obtained via a final division by r (as Nagao [47] and [37] do), which we however do not perform. This idea is quite easy to implement. From the $inv(x)$, the almost inverse of $u_1(x)$ modulo $u_2(x)$, we can define $s'(x) = inv(x)(v_1(x) + v_2(x)) \bmod u_2(x)$ and use it instead of $s(x)$.

The main advantage is that computing $\tilde{s}$ from s or s' requires the same amount of work, so it is much more efficient to skip the computation of s. Once $s'(x)$ and r are known, is it relatively easy to compute $\tilde{s}$, s_{g-1} and ${s_{g-1}}^{-2}$, so u_T and v_T can be computed as

$$\begin{aligned} u_T &= \left[\frac{\tilde{s}^2u_1}{u_2}\right] + {s_{g-1}}^{-2}(x + a_{g-1} + b_{g-1}) \\ v_T &= v_1 + s_{g-1}\tilde{s}u_1 + 1 \bmod u_T \quad . \end{aligned}$$

2.2 Doubling formula

The reason for choosing $h(x) = 1$ becomes apparent in the doubling formula. Here we want to compute $D_1 + D_1$ with $D_1 = [u_1(x), v_1(x)]$, $\deg(u_1) = g$ and $\deg(v_1) = g - 1$.

At step 2, we have $d_1 = \gcd(u_1, u_1) = u_1$ (we can set $s_1 = 1$ and $s_2 = 0$, but this is of no consequence). In step 3, we find $d = \gcd(u_1, 2u_1 + h) = \gcd(u_1, 1) = 1$ with $t_1 = 0$ and $t_2 = 1$, so $r_1 = r_2 = 0$ and $r_3 = 1$ in step 4. This choice of r_1, r_2 and r_3 is very favorable since we get $u_C = u_1^2$ (at step 5) and $v_C = v_1^2 + f \bmod u_C$ (at step 6). The computation of $u_C(x)$ and $v_C(x)$ can then be done in only $2g$ field squarings (g to compute u_1^2 and g to compute v_1^2).

The composition step can therefore be computed much faster than for generic curves (with $\deg(h) = g$). Since u_C and v_C can be computed at very little cost, there is no need to combine this step with the first reduction. The two reduction steps are then done as in Cantor's algorithm, but with a number of coefficients known to be zero (which further reduces the cost).

Algorithm 2. Group addition, most common case

INPUT: Reduced divisors $D_1 = [u_1(x), v_1(x)]$ and $D_2 = [u_2(x), v_2(x)]$
OUTPUT: Reduced divisor $D_3 = [u_3(x), v_3(x)]$, $D_3 = D_1 + D_2$

1. Almost inverse, $inv(x) = r \cdot u_1(x)^{-1} \bmod u_2(x)$, via Cramer's rule
2. $r = inv(x) \cdot u_1(x) \bmod u_2(x)$
3. $s'(x) = r \cdot s(x)$, where $s(x) = u_1(x)^{-1} \cdot (v_2(x) + v_1(x)) \bmod u_2(x)$
4. computation of inverses and $\tilde{s}(x)$ ($s'(x)$ made monic)
5. $u_T(x) = \left[\frac{\tilde{s}(x)^2 u_1(x)}{u_2(x)}\right] + {s_{g-1}}^{-2}\left(x + a_{g-1} + b_{g-1}\right)$
6. $z(x) = \tilde{s}(x) u_1(x)$
7. $v_T(x) = s_{g-1} z(x) + v_1(x) + 1 \bmod u_T(x)$
8. $u_3(x) = Monic\left(\frac{f(x) + v_T(x) + v_T(x)^2}{u_T(x)}\right)$
9. $v_3(x) = v_T(x) + 1 \bmod u_3(x)$

Algorithm 3. Group doubling, most common case

INPUT: Reduced divisors $D_1 = [u_1(x), v_1(x)]$ and $D_2 = [u_2(x), v_2(x)]$
OUTPUT: Reduced divisor $D_3 = [u_3(x), v_3(x)]$, $D_3 = D_1 + D_2$

1. $u_C(x) = u_1(x)^2$
2. $v_C(x) = v_1(x)^2 + f(x) \bmod u_C(x)$
3. computation of inverses
4. $u_T(x) = Monic\left(\frac{f(x) + v_C(x) + v_C(x)^2}{u_C(x)}\right)$
5. $v_T(x) = v_C(x) + 1 \bmod u_T(x)$
6. $u_3(x) = Monic\left(\frac{f(x) + v_T(x) + v_T(x)^2}{u_T(x)}\right)$
7. $v_3(x) = v_T(x) + 1 \bmod u_3(x)$

3 Field Arithmetic

Field arithmetic efficiency is crucial to the speed of the implementation of curve arithmetic. Its realization is often the Achilles' heel of HEC implementations. In [2] it is shown that HEC in odd characteristic are heavily penalized in most comparisons to EC, because of many types of overheads in the field arithmetic whose impact increases as field sizes get smaller. This is also the case in characteristic 2, but the nature of the worst overheads and the techniques used to address them differ.

1. *Using loops to process operands produces expensive branch mispredictions, whose cost is heavier for shorter loops.*
   Smaller fields are thus more penalized. This issue is addressed by full loop unrolling for all input sizes, for example in the implementation of the schoolbook multiplication used to implement the underlying integer arithmetic. This is a very common implementation practice. Loop unrolling is also useful in even characteristic, but the way it is used is different: For details see Subsection 3.1.
2. *Inlining can often be used to reduce function call overheads.*
   For prime field of small sizes, inlining multiplications makes a big difference. However, the binary field multiplication code is much larger than the code for a multiprecision integer multiplication of the same size. Therefore inlining would result in code size explosion causing big performance drops. In the even characteristic case, all multiplications, squarings and inversions are done using function calls, and only additions and comparisons are inlined.
3. *The cost of a modular reduction relative to the multiprecision multiplication increases as operands size decreases.*
   Therefore in odd characteristic HEC implementations more time is spent doing modular reductions than in EC implementations. This issue was addressed in [2] by delaying modular reductions for sums of products of two operands.
   In even characteristic, the reduction modulo the irreducible polynomial defining the field extension is much cheaper, and the advantages of delaying modular reductions is debatable: The additional memory traffic involved can lead to reduced performance. After doing some atomic operation counts and some testing, we opted not to implement it.
4. *Architecture* granularity *also induces irregular performance penalties.*
   A 32-bits CPU usually processes 32-bits operands, and we say that the architecture has a granularity of 32 bits. Similarly, Granularity issues may affect curves of higher genus more than elliptic curves: An elliptic curve over a 223-bits field uses 7 words operands, but a curve of genus two offering similar security needs a field of approximately 112 bits, i.e. 4 words operands. The number of field multiplications for a group operation increases roughly quadratically with the genus, but in this scenario the cost of a field multiplication in the smaller field is about 33% of the cost of a multiplication in the larger field. As a result the HEC of genus two is penalized by a factor of 1.32by granularity alone.
   Not much can be done to defeat granularity problems in the large prime field case. A partial solution [2] applies when operand sizes are between $32n-31$ and $32n-16$ bits. It consists of using half-word operands for the most significant bits, reducing

memory accesses, and speeding up modular reduction. The savings are however limited and are more noticeable for $n = 2$ (i.e. in the 33 to 48 bits range) than for larger values of n. The technique can also be used in even characteristic, however, the problem of granularity in even characteristic can be addressed more thoroughly as we show in Subsection 3.1.

5. *In the explicit formulæ, sometimes several different field elements are multiplied by the same field element.*
   This situation is similar to the multiplication of vector by a scalar, and it is possible to speed up these multiplications appreciably. The technique for doing this is described in Subsection 3.2. Similar optimization techniques do not seem to be possible in the prime field case.

The next two Subsections look at the implementation of field multiplication (Subsection 3.1) and at the technique of sequential multiplications (Subsection 3.2). Squaring are described in Subsection 3.3, modular inversion in Subsection 3.4, and modular reduction in Subsection 3.5. We conclude the section with the performance results for field arithmetic (Subsection 3.6).

A field $\mathbb{F}_{2^n}$ is represented using a polynomial basis as the quotient ring $\mathbb{F}_2[t]/(p(t))$, where $p(t)$ is an irreducible polynomial of degree n. Elements of the field are represented by binary polynomials of degree less than n. To perform multiplication (resp. squaring) in $\mathbb{F}_{2^n}$, the polynomial(s) representing the input(s) are first multiplied together (resp. squared), and the result is then reduced modulo $p(t)$.

3.1 Field multiplication and architecture granularity

The starting point for our implementation of field multiplication is the algorithm by López and Dahab [39], which is based on the comb exponentiation method by Lim and Lee [38]. ([23, Subsection 2.3] also describes this method.) It is given as Algorithm 4.

Note that if $u = 0$ in Step 6, then Steps 7–8 may be skipped. However, inserting an "**if** $u \neq 0$" statement before Step 7 slows down the algorithm in practice, because the implied branch cannot successfully be predicted by the CPU, and frequent pipeline flushes cannot be avoided.

There are a few obvious optimizations of Algorithm 4. The first one applies if the operands are at most $s\gamma - w + 1$ bits long. In this case the polynomials $P_j(t)$ fit in s words, and the loop beginning at Step 7 requires k to go from 0 to $s-1$ only.

The second optimization applies if operands are between $s\gamma - w + 2$ and $s\gamma$ bits long. We proceed as follows: First, zero the $w-1$ most significant bits of A for the computations in Steps 2 and 3, thus obtaining the polynomials $P_j(t)$ that fit in s words; Then, perform the computation as in the case of shorter operands, thus in Step 7 the upper bound for k is $s-1$; Finally, add the multiples of $B(t)$ corresponding to the most $w-1$ most significant bits of A to R before returning the result. This leads to a much faster implementation since a lot of memory writes in Step 2 and memory reads in Step 8 are traded for a minimal amount of memory operations later. This approach is commonly used for generic implementations of binary polynomial multiplication.

More optimizations can be applied if the field size is known in advance. Firstly, as the lengths of all the loops are known in advance it is possible to do partial or full

Algorithm 4. Field multiplication [LD00]

INPUT: $A = (A_{s-1}, \ldots, A_0)$, $B = (B_{s-1}, \ldots, B_0)$, and a comb size w
OUTPUT: $R = (R_{2s-1}, \ldots, R_0) = A \times B$

1. **for** $j = 0$ **to** $2^w - 1$ **do**
2. $\quad P_j(t) \leftarrow (j_{w-1}t^{w-1} + \cdots + j_1 t + j_0)A(t)$ where $j = (j_{w-1} \ldots j_2\, j_1\, j_0)_2$.
   [Here the polynomial $P_j(t)$ is at most $s+1$ words long]
3. **for** $i = 0, \ldots, 2s-1$ **do** $R_i \leftarrow 0$
4. **for** $j = s-1$ **downto** 0 **do**
5. $\quad$ **for** $i = 0$ **to** $s-1$ **do**
6. $\quad\quad u \leftarrow (B_i >> jw) \bmod t^w$ [mask out w bits at time]
7. $\quad\quad$ **for** $k = 0$ **to** s **do**
8. $\quad\quad\quad R_{k+i} \leftarrow R_{k+i} \oplus P_u[k]$
9. $\quad$ If $j \neq 0$ **then** $R \leftarrow t^w R$ [bit shift]
10. **return** R

loop unrolling. Secondly, some operands (containing the most significant bits of R) are known to be zero in the first runs of the loops, hence parts of the computation can be skipped. To do this the main loop in j will be split in two (or more) parts whose ranges cover the full range of the original loop, but some operations will be omitted in the first parts.

We considered different comb sizes and different loop splitting and unrolling techniques for 25 different input sizes, ranging from 41 to 269 bits. Of these sizes, only 22 were actually used with the group operations for our tests.

Special treatment is reserved to polynomials whose size is just a few bits more than a multiple k of the architecture granularity. Two such cases are 67 and 97 bits polynomials where the values of k are 64 and 96, resp. We perform scalar multiplication of the polynomials consisting of the lower k bits first, and then handle the remaining most significant bits one by one. This gives significantly faster code (by 10 to 15%) than optimizing and shortening the 96 and 128 bits multiplication by the previous approach.

Multiplication of polynomials whose degree is high enough is also done using the well known trick of Karatubsa [26] that reduces the multiplication of two s-bits polynomials to three multiplications of $s/2$-bits polynomials. We also observe that on the PowerPC, Karatsuba multiplication already performs slightly better than comb-based multiplication with operands that are 6 words long (the 3 half-size multiplications are then performed by the comb method).

The performance of multiplication routines can be seen (among other things) in Table 1 and Figure 1. This will be discussed in detail in Subsection 3.6, but we can already observe that the cost of multiplication grows quite smoothly with the field size, and in

fact approaches the curve of the quadratic theoretical bit complexity of multiplication much better than a simpler approach that works at the word level.

3.2 Sequential multiplications

In the explicit formulæ for the curves of genus two to four, we can find several sets of multiplications with one of terms in common. At this point it is natural to decide to preserve the precomputations (Steps 1 and 2 of Algorithm 4) associated to one field element and to re-use them in the next multiplications. However, this would require a lot of extra variables in the implementation of the explicit formulæ and memory management techniques. We therefore opted for a simpler approach: We wrote routines that perform the precomputations once and then repeat the Steps 3 to 10 of Algorithm 4 for each multiplication. We call this type of operation *sequential multiplication* (scalar multiplication would be more correct, but in our context this terminology is already used...).

It turns out that the optimal comb size for a single multiplication (on a given architecture) may not be optimal for 2, 3, or more multiplications, so we adapt the comb size to the number of multiplications that are performed together. For example, for the 73 bits field on the PowerPC, a comb of size 3 is optimal for the single multiplication and for pairs of multiplications (sequences of 2 multiplications), but for 3 to 5 multiplications the optimal comb of size is 4. For 89 bits fields, the optimal comb size for single multiplications is still 3, but it is 4 already for the double multiplication.

If a comb method is used for 6-word fields on our target architecture, then 4 is the optimal size for single multiplications and 5 is the optimal size for groups of at least 3 multiplications. As we mentioned in Subsection 3.1, Karatsuba performs slightly better for the single multiplications and the same method is also used in the sequential multiplications, where a sequential multiplication of s-word operands is turned into three sequential multiplications of $s/2$-word operands.

In order to keep function call overheads low and to avoid the use of virtual parameter lists in C, the sequential multiplication procedures for at least 3 multiplications require the input and output vectors to consist of elements which are adjacent in memory. This can also speed up memory accesses, since successive parameters are stored in consecutive memory locations, better exploiting the structure of modern caches.

For historical reasons, we need to mention that using static precomputations has already been done for multiplications by a constant parameter (coming from the curve or the field) – this is for example suggested in the context of square root extraction in [14]. We found no references on adapting the comb size to the number of multiplications by the same value.

3.3 Polynomial squaring

Squaring is a linear operation in even characteristic: If $p(t) = \sum_{i=0}^{n} e_i t^i$ where $e_i \in \mathbb{F}_2$, then $\big(p(t)\big)^2 = \sum_{i=0}^{n} e_i t^{2i}$. In other words, the result is obtained by inserting a zero bit between each two adjacent bits of the polynomial to be squared. To efficiently implement this process, a 512-byte table is precomputed for converting 8-bits polynomials into their expanded 16-bits counterparts [56].

3.4 Modular inversion

There are several different algorithms for computing the inverse of a polynomial $a(t)$ modulo another polynomial $p(t)$, where both polynomials are defined over the field $\mathbb{F}_2$.

In [22] three different methods are described (see also [23, Subsection 2.3]):

- The Extended Euclidean Algorithm (EEA), where the partial quotients are approximated by powers of t, hence no polynomial division is required, but only a comparison of degrees.
- The Almost Inverse Algorithm (AIA), which is a variant of the binary extended GCD algorithm where the final result is given as a polynomial $b(t)$ together with an integer k such that $b(t)a(t) \equiv t^k \pmod{p(t)}$. The final result must then be adjusted.
- The Modified Almost Inverse Algorithm (MAIA), which is a variant of the binary extended GCD algorithm which returns the correct inverse as a result.

We refer to [22] for details.

Depending on the nature of the different parameters, it can be assumed that each algorithm can be faster than the other two. However our experiments agree with the results in [22] and we find that EEA performs consistently better than the other two methods.

In our implementations with inputs of up to 8 words we always keep all words (limbs) of all multiprecision operands in separate integer variables explicitly, not in indexed arrays. This allows the compiler to reserve a register for each of these integer variables if enough registers are provided by the architecture (such as on RISC processors). Furthermore, it does not penalize register-starved CISC architectures: the contents of many registers containing individual words are spilled on the stack, but this data would be stored in memory anyway if we used arrays.

Another advantage of the EEA is that we have good control on the bit lengths of the intermediate variables. We can therefore split the main loop in several copies depending on the sizes of the operands, with n sections of code for inputs of n words. These sections are written to avoid operations between registers which are known to be zero (for example, with the most significant words of some variables). At the same time we can reduce the local usage of registers, allowing an increase in the size of the inputs before the compiler has to produce code that spills some information to memory. GCC 4.0 has decent register coloring algorithms and analysis of the code showed very good reuse of registers across different blocks of code. See Subsection 3.6 for the impact of this approach on performance on RISC architectures.

A limited number of registers disadvantages inversion, but a slow memory bus is not a big problem on RISC architectures: since all the inputs can be stored in internal registers, most of the computations take place without memory accesses. In comparison, a slow memory bus penalizes the multiplication much more than register paucity. These facts are reflected in the low inversion to multiplication (I/M) ratio (often between 4 and 6) on the Powerpc G4 (slow memory interface), with an increase of this ratio to about 10 on Powerpc G5 (many registers, but also a fast bus, which advantages the multiplication) or even 12 on a Pentium 3 or Athlon k6 CPU (very few registers, slow bus). On the Pentium 4 because of a slower shift operation the inversion tends to be less efficient (the I/M ratio can in some cases exceed 20).

3.5 Modular reduction

For modular reduction we implemented two sets of routines.

The generic routine reduces arbitrary polynomials over $\mathbb{F}_2$ modulo arbitrary polynomials over $\mathbb{F}_2$. This code is in fact rather efficient, and a reduction by means of this routine can often take less than 20% of the time of a multiplication. The approach is similar to the one taken in SUN's contributions to OpenSSL or in NTL's code base, and exploits a compact representation of the reduction polynomial. The reduction polynomial is given as a decreasing sequence of integers $(n_0, n_1, n_2, ..., n_{k-1})$ and it is equal to $\sum_{i=0}^{k-1} t^{n_i}$. Whenever possible we use a trinomial ($k = 3$), otherwise we use a pentanomial ($k = 5$). Only in very few cases an eptanomial is used when its reduction is more efficient than that of the most efficient pentanomial.

The second set of routines uses fixed reduction polynomials, and is therefore specific for each polynomial degree. The code is very compact and streamlined. In this situation we sometimes prefer reduction eptanomials to pentanomials when the reduction code is shorter due to the form of the polynomial. For example, for degree 59 we have two good irreducible polynomials: $f_1(t) = t^{59} + (t+1)(t^5 + t^3 + 1)$ and $f_2(t) = t^{59} + t^7 + t^4 + t^2 + 1$.

The C code takes a polynomial of degree up to 116 ($= 2 \cdot 58$) stored in variables `r3` (most significant word), `r2`, `r1` and `r0` (least significant word), and reduces it modulo $f_1(t)$, leaving the result in `r1` and `r0`

```
#define bf_mod_59_6_5_4_3_1_0(r3,r2,r1,r0) do {                              \
    r3  = ((r3) <<  5) ^ ((r3) <<  6); r1 ^= (r3) ^ ((r3) << 3) ^ ((r3) << 5); \
    r3  = ((r2) <<  5) ^ ((r2) <<  6); r0 ^= (r3) ^ ((r3) << 3) ^ ((r3) << 5); \
    r3  = ((r2) >> 22) ^ ((r2) >> 21); r1 ^= (r3) ^ ((r3) >> 2) ^ ((r3) >> 5); \
    r2  = (r1) >> 27; r2 ^= (r2) << 1;                                       \
    r1 &= 0x07ffffff; r0 ^= (r2) ^ ((r2) << 3) ^ ((r2) << 5);                \
    } while (0)
```

and the C code to reduce the same input modulo $f_2(t)$ is

```
#define bf_mod_59_7_4_2_0(r3,r2,r1,r0) do {                          \
    r1 ^= ((r3) <<  5) ^ ((r3) <<  7) ^ ((r3) <<  9) ^ ((r3) << 12); \
    r2 ^=                ((r3) >> 25) ^ ((r3) >> 23) ^ ((r3) >> 20); \
    r0 ^= ((r2) <<  5) ^ ((r2) <<  7) ^ ((r2) <<  9) ^ ((r2) << 12); \
    r1 ^= ((r2) >> 27) ^ ((r2) >> 25) ^ ((r2) >> 23) ^ ((r2) >> 20); \
    r2  = (r1) >> 27;  r1 &= 0x07ffffff;                             \
    r0 ^= (r2) ^ ((r2) << 2) ^ ((r2) << 4) ^ ((r2) << 7);            \
    } while (0)
```

Comparing the two codes, we find that the first one is slightly more efficient. A similar choice occurs at degree 107 (the irreducible polynomial is $t^{107} + (t^6 + t^2 + 1)(t + 1)$), and the idea of factoring the "lower degree part" of the reduction polynomial is also used for degree 109 (the polynomial is $t^{109} + (t^6 + 1)(t + 1)$).

These considerations have been applied to degrees 43, 53, 59, 67, 71, 73, 79, 83, 89, 101, 107, 109, 127, 137, 139, 157, 179, 199, 211, 239, 251, and 269 (which are used in the tests) as well as the intermediate values 97, 131, and 149.

By doing this, we can keep the time for the modular reduction between 3 and 5% of the time required for a multiplication in the case the reduction polynomial is a trinomial, and between 6 and 10% in the other cases. Reduction modulo a trinomial is about as

twice fast as polynomial squaring (Subsection 3.3), because in the latter there are more memory accesses.

For degrees 47, 71, 79, 89, 97, 127, 137, 159, 199, and 239 we used a trinomial. For degrees 43, 53, 67, 73, 83, 101, 109, 131, 139, 149, 179, 211, 251, and 269 we used a pentanomial. For degrees 59, 107, and 219 we opted for eptanomials, when they were reducible and had a sedimentary part of lower degree than the optimal pentanomial (see remarks above).

As operand sizes increase, the relative cost of modular reduction decreases considerably and becomes in practice negligible.

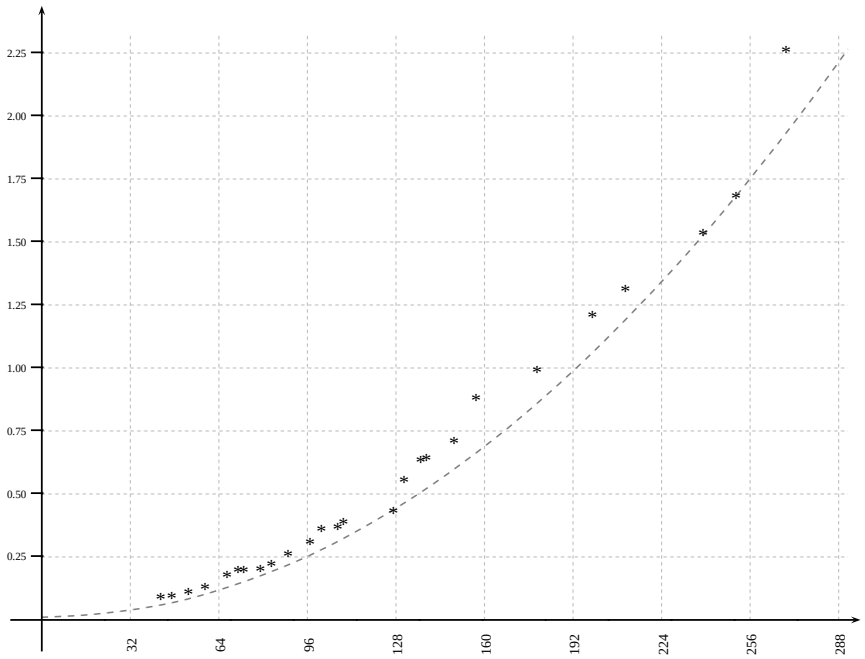
3.6 Field arithmetic and its performance

Table 1, contains the timings of the fundamental operations in the fields discussed in the previous section. These operations are: Single multiplication (Mul), squaring (Sqr), multiplication of 2 to 5 different field elements by a fixed one (columns from Mul2 to Mul5) and inversion (Inv). The timings for multiplications (both single and sequentials) and squaring include the modular reduction. We first give the absolute times in microseconds, and then the relative times compared to a single multiplication (with reduction). Figure 1 displays in a graphical way the results of the single multiplications for the different field sizes.

Table 1. *Timings of field operations in µsec (1.5 GHz Powerpc G4) and ratios*

	Absolute timings							Timings relative to multiplication					
Bits	Mul	Sqr	Mul2	Mul3	Mul4	Mul5	Inv	Sqr	Mul2	Mul3	Mul4	Mul5	Inv
47	.087	.014	.142	.192	.243	.296	.483	.164	1.640	2.225	2.795	3.421	5.511
53	.102	.019	.169	.238	.305	.372	.517	.182	1.654	2.330	2.982	3.641	5.063
59	.120	.020	.206	.271	.332	.396	.536	.170	1.713	2.253	2.766	3.295	4.460
67	.169	.025	.299	.374	.470	.563	.832	.148	1.774	2.220	2.791	3.340	4.935
71	.188	.019	.317	.457	.555	.654	.858	.102	1.689	2.430	2.954	3.483	4.569
73	.191	.024	.326	.470	.575	.676	.868	.125	1.702	2.453	3.004	3.530	4.536
79	.193	.019	.330	.466	.567	.670	.905	.099	1.712	2.416	2.937	3.476	4.693
83	.213	.050	.387	.518	.640	.760	.922	.236	1.818	2.430	3.003	3.569	4.329
89	.254	.025	.420	.538	.667	.796	.962	.100	1.653	2.119	2.627	3.136	3.790
97	.311	.025	.455	.612	.761	.913	1.589	.081	1.462	1.967	2.445	2.933	5.105
101	.351	.057	.535	.732	.916	1.099	1.634	.164	1.526	2.088	2.613	3.135	4.661
107	.361	.059	.547	.782	.983	1.181	1.669	.162	1.513	2.162	2.720	3.267	4.618
109	.381	.029	.553	.814	1.021	1.231	1.702	.075	1.454	2.139	2.683	3.233	4.470
127	.415	.053	.674	.957	1.201	1.447	1.832	.128	1.625	2.306	2.894	3.486	4.415
131	.460	.067	.763	1.046	1.329	1.605	3.664	.147	1.659	2.275	2.890	3.489	7.968
137	.625	.062	1.084	1.485	1.907	2.325	3.733	.100	1.734	2.375	3.050	3.718	5.969
139	.635	.067	1.113	1.506	1.931	2.360	3.761	.105	1.753	2.371	3.042	3.718	5.924
149	.677	.090	1.191	1.680	2.170	2.656	3.908	.133	1.760	2.482	3.206	3.924	5.773
157	.992	.146	1.921	2.631	3.368	4.067	3.991	.147	1.937	2.654	3.397	4.101	4.025
179	1.182	.197	2.192	2.937	3.720	4.508	5.547	.166	1.855	2.484	3.147	3.814	4.693
199	1.344	.184	2.406	3.229	4.088	4.944	12.166	.137	1.790	2.403	3.042	3.679	9.053
211	1.506	.250	2.677	3.707	4.746	5.748	12.624	.166	1.777	2.462	3.152	3.817	8.383
239	1.528	.187	2.630	3.637	4.604	5.638	14.828	.122	1.722	2.381	3.014	3.691	9.706
251	1.675	.288	2.978	4.157	5.297	6.498	15.157	.172	1.778	2.482	3.163	3.879	9.050
269	2.254	.384	4.124	5.672	7.284	8.774	20.413	.213	1.831	2.519	3.234	3.896	9.158

Fig. 1. Field multiplication performance for fields of different sizes (timings in μsec). The stars represent the multiplication timings, whereas the arc of parabola represent a routine performing $c_1 \cdot n^2 + c_0$ bit operations and interpolating the best performance values. The vertical lines are spaced 32 bits from each other. Several fields which have less-than-optimal performance are defined by pentanomials and not by trinomials.



The timings were obtained on a 1.5 Ghz PowerPC G4 (Motorola 7447) CPU, a 32-bits RISC architecture with 32 general purpose integer registers. This means that we can put several intermediate operands in the registers. Note that we do not use specific compiler directives, but we simply declare all limbs of all intermediate operands as single word variables, and operate on them with the usual C language logic and shift operations. The code in fact compiles correctly on any 32-bits architecture supported by the gnu compiler collection (we used gcc 4.0.1 under Mac OS X 10.4.3).

The register abundance easily explains not only the very good performance (multiplications are often 30% faster than on a 2.8 Ghz Pentium 4 processor), but also the low timings for the inversion. If the operands of the EEA all fit in the registers (which was the case for elements of up to 6 words), then the whole inversion is performed exclusively in the registers, without accessing external memory except to load the inputs and to store the final result. The "bump" in inversion performance occurs when the registers are no longer sufficient, and the compiler must store some partial data in main memory (as confirmed by analysis of the disassembly of the compiled code).

Note that the usage of trinomials and pentanomials or eptanomials is reflected in the timings. The ratio Sqr/Mul is higher if a pentanomial is used because the reduction is slower than it would have been if a trinomial of the same degree had existed. The reduction has a small relative impact on the field multiplication, but it makes a bigger difference for the field squaring (since polynomial squaring is very inexpensive).

4 Improvements in the formulæ

In this section, we describe the algorithmic methods used to improve the formulæ of Guyot, Kaveh and Patankar (for genus three, [21]) and of Pelzl, Wollinger and Paar (for genus four, [52, 59]). We follow three main approaches:

1. *Reduce the number of inversions.* As inversions are usually much more costly than other operations, it is often a good idea to combine as many of them together, even if this means increasing the number of multiplications. This was common for genus two and three, but it can be pushed further for genus four as is described in Section 4.1.
2. *Reduce the number of multiplications.* It goes without saying that if the number of multiplications in the explicit formulæ can be reduced, the overall performance will improve. Most explicit formulædo this by introducing Karatsuba multiplication to compute the product of polynomials. We obtain further improvements by selecting faster algorithms (Section 4.2), combining multiplications using Karatsuba-like tricks (Section 4.3) and keeping products in memory if they are used again later in the formula.
3. *Combine multiplications with a repeated operand.* The goal here is to make use of the sequential multiplications. Although this does not affect the total operation count, this approach reduced the effective cost of some (but not all) steps by more than 30%.

Note that approaches 2 and 3 are not always compatible. In most cases, reducing the number of multiplications using Karatsuba-like tricks will hinder the use of sequential

multiplications. For example, the computation of $inv(x) \cdot (v_2(x) + v_1(x))$ in the genus four addition formula takes 16 multiplications using classical methods. With sequential multiplications, we still have the same number of multiplications, but the effective cost is close to that of 11.4 normal multiplications (using estimates for sequential multiplications extrapolated from Table 1 – see subsection 4.5 for the exact equivalences adopted). With two layers of Karatsuba multiplications, we can do this in 9 multiplications, but then no operand is used in more than one product so we cannot use sequential multiplications to get any further savings.

In this example it is clear that a Karatsuba-like approach is a better choice. In many cases however, the choice is not always so clear as the savings obtained by giving precedence to Karatsuba-like tricks or to sequential multiplications may be very close and will vary depending on the processor and the field size. To avoid writing a different formula for each field size, the formulæ described in this paper assume the average case. For some field sizes, the formula may not be completely optimal, although the saving that could be obtained by using a field-specific formula would be marginal at best.

4.1 Inversions

As for curves of genus three, the common case of group addition for curves of genus four requires two reduction steps. However, $\frac{f+v_t+v_T^2}{u_T}$ in the second reduction step (Step 8 of Algorithm 2) for genus four is not automatically monic since the leading term in $f + v_t + v_T^2$ is ${v_{T,5}}^2 x^{10}$. For these curves, we must therefore compute the inverses of s_3 (to make u_T monic), r (to get s_3, for the computation of v_T), s_3' (to make s' monic) and $v_{T,5}$ (to make u_3 monic).

The inverses of s_3, r and s_3' can be combined in the same way it is done for curves of genus three, but this still leaves us with two inversion to be performed. At the time the inverses of s_3' and s_3 are obtained, only $inv(x)$, r and $s'(x)$ have been computed. To minimize the number of inversions, one has to express $v_{T,5}$ in terms of r and the coefficients $s'(x)$ and $u_2(x)$.

To improve readability, we use the coefficients of the polynomials $u_T(x)$, $z(x) = \tilde{s}(x)u_1(x)$, $s(x) = \frac{1}{r}s'(x)$ and $\tilde{s}(x) = \frac{1}{s_3'}s'(x)$, even though those polynomials are not computed before the inverses. Since $v_T(x) = s_3 z(x) + v_1(x) + 1 \bmod u_T(x)$, we have

$$v_{T,5} = s_3\left(z_5 + u_{T,4} + u_{T,5}(z_6 + u_{T,5})\right) \quad .$$

Furthermore, replacing $\tilde{s}(x)^2 u_1(x)$ by $\tilde{s}(x)z(x)$ in the equation for $u_T(x)$, we have

$$u_T(x) = \left[\frac{\tilde{s}(x)z(x)}{u_2(x)}\right] + s_3^{-2}(x + a_3 + b_3) \quad ,$$

so $u_{T,5} = z_6 + \tilde{s}_2 + b_3$ and $u_{T,4} = z_5 + \tilde{s}_2 z_6 + \tilde{s}_1 + b_2 + u_{T,5}b_3$. Substituting back into the equation of $v_{T,5}$, we get

$$\begin{aligned} v_{T,5} &= s_3\left(\tilde{s}_2 z_6 + \tilde{s}_1 + b_2 + u_{T,5}b_3 + u_{T,5}(z_6 + u_{T,5})\right) \\ &= s_3\left(\tilde{s}_2 z_6 + \tilde{s}_1 + b_2 + u_{T,5}\tilde{s}_2\right) \\ &= s_3\left(\tilde{s}_2 z_6 + \tilde{s}_1 + b_2 + (z_6 + \tilde{s}_2 + b_3)\tilde{s}_2\right) \\ &= s_3\left(\tilde{s}_1 + b_2 + \tilde{s}_2^2 + b_3\tilde{s}_2\right) \quad . \end{aligned}$$

Using the definitions of $s(x)$ and $\tilde{s}(x)$, we get

$$v_{T,5} = \frac{s'_3}{r}\left(\frac{s'_1}{s'_3} + b_2 + \left(\frac{s'_2}{s'_3}\right)^2 + b_3\frac{s'_2}{s'_3}\right) = \frac{1}{rs'_3}\left({s'_2}^2 + s'_3(s'_1 + b_3 s'_2 + b_2 s'_3)\right) \ .$$

Since $1/rs'_3$ is not yet known, we replace the computation of $v_{T,5}$ by the computation of

$$rs'_3 v_{T,5} = {s'_2}^2 + s'_3(s'_1 + b_3 s'_2 + b_2 s'_3) \ .$$

To obtain the inverses of s_3, r, s'_3 and $v_{T,5}$, we first compute rs'_3 and $rs'_3 v_{T,5}$, and let

$$t = \frac{1}{(rs'_3)\cdot(rs'_3 v_{T,5})} \ .$$

Then, the inverses of $v_{T,5}$ and rs'_3 are obtained as

$$t\cdot(rs'_3)^2 = \frac{1}{v_{T,5}} \quad \text{and} \quad t\cdot(rs'_3 v_{T,5}) = \frac{1}{rs'_3} \ .$$

The inverses of s_3, r and s'_3 are then obtained as before. In this manner, it is possible to combine the final inversion with the previous ones, at the cost of an extra five multiplications and two squarings. Note that the computation of $v_{T,5}$ (which is needed for the final step) can then be done in one multiplication instead of two and that we no longer need to compute z_5 (which saves one more multiplication). This approach reduces computation time if an inversion costs more than three multiplications and two squarings.

For the doubling formula, the approach must be modified slightly since in this case $u_C(x)$ and $v_C(x)$ are computed directly. This time, we need the inverses of $v_{C,7}$ and $v_{T,5}$ and we have

$$v_{C,7}v_{T,5} = {v_{C,6}}^2 + v_{C,5}v_{C,7} + u_{C,6}({v_{C,7}}^2) \ .$$

We then compute

$$t = \frac{1}{v_{C,7}(v_{C,7}v_{T,5})}$$

and the inverses of $v_{T,5}$ and $v_{C,7}$ are found as

$$t\cdot({v_{C,7}}^2) = \frac{1}{v_{T,5}} \quad \text{and} \quad t\cdot(v_{C,7}v_{T,5}) = \frac{1}{v_{C,7}} \ .$$

The second inverse is then replaced by five multiplications and two squarings, of which one (${v_{C,6}}^2$) is used again at a later step. Since two of these multiplications can be combined, the exchange is advantageous if one inversion cost more than (roughly) 4.7 multiplications and one squaring. This trade-off is slightly to our disadvantage on the PowerPC G4 (by about 0.2 multiplications on average for the fields used for genus four testing), but we still opted to implement it as it always produces significant savings in other processors.

4.2 Almost inverse

One of the most costly step of explicit formulæ is the computation of the almost inverse. Most of the explicit formulæ published so far [24, 47, 40, 31, 51, 50, 52, 21, 34, 33] find the almost inverse via the computation of a resultant, however this approach is not optimal. For every genus bigger than two, the almost inverse can be computed with fewer field multiplications using Cramer's rule (for genus two, the cost are the same if both methods are implemented carefully). This approach has the added bonus of taking full advantage of sequential multiplications, making it even more efficient.

Cramer's rule computes a solution $\mathbf{v}$ to the system $M\mathbf{v} = \mathbf{w}$ where M is an $n \times n$ matrix. The solution is

$$\mathbf{v} = \frac{1}{|M|} \begin{pmatrix} |Sub_0(M,\mathbf{w})| \\ |Sub_1(M,\mathbf{w})| \\ \vdots \\ |Sub_{n-1}(M,\mathbf{w})| \end{pmatrix}$$

where $Sub_i(M,\mathbf{w})$ is matrix M with the i^{th} column replaced by $\mathbf{w}$.

To apply this method to computing the inverse of polynomials, we need a map between polynomials (of degree smaller then n) and vectors. To the coefficient of x^i in the polynomial $p(x)$, we associate the i^{th} coordinate of the vector $\mathbf{p}$ (and vice versa).

To compute the inverse of $a(x)$ modulo $b(x)$, we use the matrix M where the i^{th} row corresponds to $x^i a(x) \bmod b(x)$. We then solve for the vector $\mathbf{w} = \mathbf{1} = (1,0,0,\ldots,0)^t$ (the constant polynomial 1). Since the matrices have many coefficients in common, most of the computations can be combined using a bottom-to-top approach (see Appendix A for details).

As we do not require the inverse, but rather an almost inverse (the product of the inverse by a constant multiple r), we compute all the determinants used in Cramer's rule, but we avoid the division by $|M|$. We obtain:

$$\mathbf{inv} = \begin{pmatrix} |Sub_0(M,\mathbf{1})| \\ |Sub_1(M,\mathbf{1})| \\ \vdots \\ |Sub_{n-1}(M,\mathbf{1})| \end{pmatrix} \qquad \text{and} \qquad r = |M| = \sum_{i=0}^{n-1} M_{0,i}|Sub_i(M,\mathbf{1})| \ .$$

Note that in the formulæ, the computation of r is done with some of the computations for $s'(x)$ in order to combine some of the multiplications into pairs (and use sequential multiplications).

4.3 Polynomial divisions

Although Karatsuba-like multiplications are most commonly applied to polynomial multiplication, they can also be used when dealing with polynomial divisions (both for the quotient and the remainder). We consider three main cases: The reduction of $inv(x)(v_2(x)+v_1(x))$ modulo $u_2(x)$ (computation of $s'(x)$), the division on $(\tilde{s}(x)^2)\cdot u_1(x)$ by $u_2(x)$ (computation of $u_T(x)$), and the reduction of $v_T(x)+h(x)$ modulo $u_3(x)$ (computation of $v_3(x)$).

For the computation of $s'(x)$, let us assume that the product $inv(x) \cdot (v_2(x) + v_1(x))$ is already computed (even though the product is intermingled with the reduction modulo $u_2(x)$ in practice) and let us call it

$$p(x) = p_6x^6 + p_5x^5 + p_4x^4 + p_3x^3 + p_2x^2 + p_1x + p_0 \ .$$

We want to reduce $p(x)$ modulo $u_2(x)$, i.e. we want to write $p(x) = \lambda(x)u_2(x) + s'(x)$ with $\lambda(x) = \lambda_2x^2 + \lambda_1x + \lambda_0$. Note that although we do not really want $\lambda(x)$, its computation is necessary to the efficient computation of $s'(x)$. The computations take the form:

$$\begin{aligned}
\lambda_2 &= p_6 \\
\lambda_1 &= p_5 + b_3\lambda_2 \\
\lambda_0 &= p_4 + b_3\lambda_1 + b_2\lambda_2 \\
s'_3 &= p_3 + b_3\lambda_0 + b_2\lambda_1 + b_1\lambda_2 \\
s'_2 &= p_2 + b_2\lambda_0 + b_1\lambda_1 + b_0\lambda_2 \\
s'_1 &= p_1 + b_1\lambda_0 + b_0\lambda_1 \\
s'_0 &= p_0 + b_0\lambda_0 \ .
\end{aligned}$$

One important thing to notice is that although the coefficients of $s'(x)$ cannot be computed before λ_0, there is no problem (other than memory requirements) with computing some of the products containing λ_1 or λ_2 before computing λ_0. The maximum number of multiplications that can be saved using a Karatsuba-like approach is three. One can combine the pairs of multiplications in the sums $b_3\lambda_1 + b_2\lambda_2$, $b_3\lambda_0 + b_1\lambda_2$ and $b_2\lambda_0 + b_1\lambda_1$ (where λ_0 is obtained before the last two pairs are handled). The result is 6 products with no terms in common and 3 multiplications by b_0 (handled sequentially), for an effective cost of around 8.3 multiplications. Note that in this case, using 3 sequential multiplications (by λ_2, λ_1 and λ_0) on the original set of operations would have had an effective cost of around 8.45 multiplications. (Again, see subsection 4.5 for the equivalences.)

For the computation of $u_T(x)$, most of the work involves computing the quotient of $(\tilde{s}(x)^2 \cdot u_1(x))$ divided by $u_2(x)$. Since $u_2(x)$ has degree 4, the coefficients of x^i for $i < 4$ in $\tilde{s}(x)^2 \cdot u_1(x)$ do not have to be computed as they have no impact on the result. This time, we assume that we have already computed

$$p(x) = \tilde{s}(x)^2 \cdot u_1(x) = x^{10} + p_9x^9 + p_8x^8 + p_7x^7 + p_6x^6 + p_5x^5 + p_4x^4 + \ldots$$

(with $p_9 = a_3$) and we see that the coefficients of $u_T(x)$ are:

$$\begin{aligned}
u_{T,5} &= a_3 + b_3 \\
u_{T,4} &= p_8 + b_2 + b_3u_{T,5} \\
u_{T,3} &= p_7 + b_1 + b_3u_{T,4} + b_2u_{T,5} \\
u_{T,2} &= p_6 + b_0 + b_3u_{T,3} + b_2u_{T,4} + b_1u_{T,5} \\
u_{T,1} &= p_5 + b_3u_{T,2} + b_2u_{T,3} + b_1u_{T,4} + b_0u_{T,5} \\
u_{T,0} &= p_4 + b_3u_{T,1} + b_2u_{T,2} + b_1u_{T,3} + b_0u_{T,4} \ .
\end{aligned}$$

However, the products of b_3, b_2, b_1 and b_0 by $u_{T,5} = (a_3 + b_3)$ are already known from the computation of the almost inverse (using Cramer's rule), so we only need to compute products of the form

$$\begin{aligned} u_{T,3} &= sum_3 + b_3 u_{T,4} \\ u_{T,2} &= sum_2 + b_3 u_{T,3} + b_2 u_{T,4} \\ u_{T,1} &= sum_1 + b_3 u_{T,2} + b_2 u_{T,3} + b_1 u_{T,4} \\ u_{T,0} &= sum_0 + b_3 u_{T,1} + b_2 u_{T,2} + b_1 u_{T,3} + b_0 u_{T,4} \ . \end{aligned}$$

After a few checks, one might be tempted to conclude that the most that can be saved using Karatsuba-like tricks in this situation is one multiplication, but in fact it is not the case. Although only three combinations ($b_3 u_{T,3} + b_2 u_{T,4}$, $b_3 u_{T,2} + b_2 u_{T,3}$ and $b_2 u_{T,3} + b_1 u_{T,4}$) could be used without requiring extra products (and no two of these can be used successfully at the same time), it is possible to reduce the operation count by adding one new product. If one also computes $b_1 u_{T,2}$, then it becomes possible to add two more combinations ($b_3 u_{T,2} + b_1 u_{T,4}$ and $b_2 u_{T,2} + b_1 u_{T,3}$) to $b_3 u_{T,3} + b_2 u_{T,4}$, in effect reducing the number of multiplication by two instead of one. Once again, it is slightly more efficient to use sequential multiplications after reducing the number of multiplications as much as possible than using them on the original equations.

Let us now consider the final step of the the genus four formulæ (that step is essentially identical for both the addition and the doubling) where we compute $v_3(x) = v_T(x) + 1 \bmod u_3(x)$. We have

$$\begin{aligned} t &= v_{T,4} + e_3 v_{T,5} \\ \varepsilon_3 &= v_{T,3} + e_2 v_{T,5} + e_3 t \\ \varepsilon_2 &= v_{T,2} + e_1 v_{T,5} + e_2 t \\ \varepsilon_1 &= v_{T,1} + e_0 v_{T,5} + e_1 t \\ \varepsilon_0 &= v_{T,0} + 1 + e_0 t \end{aligned}$$

which, at first glance, requires 8 multiplications. Although the use of Karatsuba multiplications (combining $e_2 v_{T,5} + e_3 t$ and $e_0 v_{T,5} + e_1 t$) reduces the number of multiplications by 2 whereas the use of sequential multiplications reduces the cost to around 5.7 multiplications, the Karatsuba approach is in fact better. The idea here is to reorder the 6 products into three pairs:

- $e_3 v_{T,5}$ and $e_1 v_{T,5}$ (done first, so t can be computed)
- $e_2 t$ and $e_0 t$
- $(e_3 + e_2)(v_{T,5} + t)$ and $(e_1 + e_0)(v_{T,5} + t)$ to obtain the two sets of combined products).

By computing these pairs as sequential multiplications, we can get the effective cost down to around 5.1 multiplications.

For the genus four addition, there are three remaining cases that we do not discuss where a Karatsuba-like approach can be considered:

- The computation of $z(x) = \tilde{s}(x) \cdot u_1(x)$. The pattern of multiplications encountered are the same as for the reduction of $inv(x)(v_2(x) + v_1(x))$ modulo $u_2(x)$, but viewed upside down since z_5 is not required (see Section 4.1).

– The computation of $(\tilde{s}(x)^2) \cdot u_1(x)$ (before the division by $u_2(x)$). Since only the coefficients of powers of x greater than 4 are needed and $\tilde{s}(x)^2$ only contains even powers of x, it is not possible to obtain any saving from this approach.
– The computation of $u_3(x)$. Here we have only one possible way of reducing multiplications, by combining the products in $e_3 u_{T,4} + e_2 u_{T,5}$.

4.4 Variables

A very serious issue with the genus four formulæ (and, to a lesser extent, the genus three formulæ as well), is the number of variables involved. An implementation of the addition operation for genus four that does not worry about memory requirements would use 234 variables (240 to take into account the "sequential" form of our implementation of sequential multiplications). This number of variables is obviously too large for constrained environments, and even with two or three words per field elements (as is the case for the security levels considered in this paper), this would mean a storage in the order of one kilobyte for the variables alone. At this point, the memory allocation might affect the performance of the computations also for high-end processors: even if this memory is allocated statically, its use may still take up a non-negligible part of the level 1 cache of the processor.

The natural solution is to use the same variables multiple times. We decided to minimize the number of variables as much as possible without losing any (significant) efficiency in exchange. The result is a more compact and portable code which can be used even in constrained environments. To minimize memory allocations as much as possible, we chose to define an array of field elements, whose address is passed as part of the function calls for the addition and doubling and where all the intermediate results of the group operations are stored.

To improve readability, the formulæ in the appendix are given in terms of distinct variables and they are accompanied by an allocation schedule for the variable array. In a few cases, a variable is copied into a different location in the array to obtain an adjacent sequence that can be used for the sequential multiplications (this is due to our choice for the function call), or a long sum is broken into two shorter sums to free some of the variables. The final values are kept with the other variables until the last minute (at which point they are copied into the space allocated to a divisor) so the function's output can replace one of its inputs if desired.

The resulting formulæ require 14 variables for the genus three addition and doubling, 19 for the genus four doubling and 23 for the addition. In all cases, these numbers are minimal with the present formulæ as there are "bottlenecks" where all the variables are either in use for the current operation or contain values that were computed earlier and will be used later.

4.5 Operation counts

The addition and doubling formulæ for genus three, with the tables of variable allocation, are in Appendix B, and those for genus four are in Appendix C. Tables 2 and 3 compare the operation counts of our formulæ with previous works (note that for the

genus four addition, changing from $h = x$ to $h = 1$ only affects the number of field additions, the difference in operation counts comes from our improvements). We give to sets of numbers for our formulæ: One where sequential multiplications are handled as normal multiplication (the "classical" coost) and one where they are done sequentially as in Subsection 3.2 (the "effective" cost). To obtain the effective cost, we use an average value from Table 1 (over the fields of 47 to 101 bits), with pairs of multiplications costing 1.7 single multiplication, sequences of 3 costing 2.3, sequences of 4 costing 2.85, and sequences of 5 costing 3.4. For Cantor's and Nagao's algorithms, we use the odd characteristic numbers as an indicator of the cost in characteristic two (the numbers that would be obtained by adapting these algorithms to even characteristic should differ only slightly).

Table 2. *Field operation counts for genus three group operations*

Reference	Characteristic	Curve properties	Addition	Doubling
Cantor [8]	odd	$h = 0$	4 I + 200 M/S	4 I + 207 M/S
Nagao [47]	odd	$h = 0$	2 I + 144 M/S	2 I + 153 M/S
Pelzl et al. [50]	two	$h = 1$, $f_6 = 0$	1 I + 65 M + 6 S	1 I + 14 M + 11 S
Guyot et al. [21]	two	$h = 1$, $f_6 = 0$	1 I + 58 M + 6 S	1 I + 11 M + 11 S
this work, classical	two	$h = 1$, $f_6 = 0$	1 I + 57 M + 6 S	1 I + 11 M + 11 S
this work, effective	two	$h = 1$, $f_6 = 0$	1 I + 47.7 M + 6 S	1 I + 9.3 M + 11 S

Table 3. *Field operation counts for genus four group operations*

Reference	Characteristic	Curve properties	Addition	Doubling
Cantor [8]	odd	$h = 0$	6 I + 386 M/S	6 I + 359 M/S
Nagao [47]	odd	$h = 0$	2 I + 289 M/S	2 I + 268 M/S
Pelzl et al. [52]	two	$h = x$, $f_8 = 0$	2 I + 148 M + 6 S	2 I + 75 M + 14 S
this work, classical	two	$h = 1$, $f_8 = 0$, $f_7 \neq 0$	1 I + 119 M + 10 S	1 I + 28 M + 16 S
this work, effective	two	$h = 1$, $f_8 = 0$, $f_7 \neq 0$	1 I + 98.15 M + 10 S	1 I + 23.7 M + 16 S

5 Security

In this section, we describe how the different security levels are selected, as well as the security of the form of curve used.

For curves of genus one and two, the fastest known attack is Pollard's Rho algorithm which take $O(\sqrt{\text{group order}})$ group operations. Since the group order of a curve of genus g over a field of q elements is $q^g + O(gq^{g-1/2})$, this means $O(\sqrt{q_1})$ group

operations for elliptic curves over the field $\mathbb{F}_{q_1}$ and $O(q_2)$ group operations for curves of genus two over the field $\mathbb{F}_{q_2}$.

For curves of genus three and four, the fastest known attack is the index calculus algorithm. Using the double large prime variations of Gaudry, Thomé, Thériault and Diem [20] and Nagao [48], and ignoring logarithmic terms, this attack requires $O(q^{2-2/g})$ group operations for a genus g over of field of q elements. For a curve of genus three over $\mathbb{F}_{q_3}$, this means $O({q_3}^{4/3})$ group operations, and for a curve of genus four over $\mathbb{F}_{q_4}$, this means $O({q_4}^{3/2})$ group operations.

To obtain a precise comparison of the security levels, we would need to take into account any logarithmic term present in the index calculus running time, as well as the underlying constants in both algorithms. To simplify the analysis, we assume no logarithmic term and identical constants. This assumption should in fact disadvantage slightly the curves of genus three and four: The constants are most likely of similar size, while proven results on the double large prime index calculus contain an extra logarithmic factor, so we are underestimating the cost for genus three and four.

For the discrete log to require the same amount on each curve, we need

$$\frac{1}{2}\log(q_1) \approx \log(q_2) \approx \frac{4}{3}\log(q_3) \approx \frac{3}{2}\log(q_4) \ ,$$

where q_g is the order of the field of definition for the curve of genus g. To compare with an EC over a field of n bits, we need a field of $n/2$ bits for genus two, $3n/8$ bits for genus three and $n/3$ bits for genus four.

Since Pollard Rho can be adapted to take advantage of the existence of subgroups or knowledge of the key size, curves of genus one and two are assumed to groups of order twice a prime (the form of the curves forces the group order to be even) with keys of n bits.

For genus three and four, the situation is different since the index calculus algorithm works on the algebraic group as a whole, so it cannot take advantage of the existence of subgroups or any information on the key (including the bit size). On the other hand, Pollard Rho could still be used if the subgroups were small enough or if the keys were short enough, so to have an equivalent security the curves must have a prime-ordered subgroup of at least n bits. Similarly, the keys used must also be at least n bits long, but keys of more than n bits do not give any added security since they are attacked using index calculus.

The last remark is very important from an efficiency point of view, since it means that the same key (scalar) can be used for all four genera instead of having to increase the key length for genus three and four. The (sub)group sizes are also of interest, since curves of genus three could allow a cofactor of up to $n/8$ bits and curves of genus four could have a cofactor of $n/4$ bits (the group orders have $9n/8$ and $4n/3$ bits respectively), which could make the search for a "good" curve much easier.

A final concern in the choice of the field is the Weil descent attack, which is a risk for some field extensions (see [19] for EC, [58] for HEC). Although these attacks may not always be a risk for every curve over a given field extension, recent developments [25, 42] show that for some extension degrees a large proportion of curves are at risk. Gaudry [18] also showed that small factors in the extension degree can expose all

curves defined over that field to a Weil descent-like attack. However, no known variation exists for prime extensions, so we avoid the issue of "how likely is a Weil descent to work on a given field?", by choosing all fields of the form $\mathbb{F}_{2^p}$, where p is a prime.

The only security aspect that remains to be discussed is the form of the defining equation of the curves. For genus one and two, curves of the form $y^2+y=f(x)$ are supersingular and are exposed to the MOV [41] or Frey-Rück [16] attack and their hyperelliptic variant [17] (all of which can be subsumed under the treatment of Tate-Lichtenbaum pairings) and, hence, they should be avoided for designing DL systems. Hence we selected curves of the form $y^2+xy=f(x)$ for security and efficiency. As we choose curves of the form $y^2+y=f(x)$ for genus three and four, it is natural to ask whether they are supersingular or not. Using results of Scholten and Zhu [55], we know that none of the curves of genus three over binary fields are supersingular, while the only supersingular curves of genus four over binary fields are of the (simplified) form $y^2+y=x^9+f_5x^5+f_3x^3+f_1x+f_0$. We can then safely use the special form for curves of genus three, and the only condition required for genus four is to insure that $f_7\neq 0$, which is easily verified.

6 Timings and comparisons

In Table 4 we show the timings of our implementation of EC and of HEC jacobians of genus up to four. Since our goal is to make a comparison between the performance of curves of different genera offering the same security level, we attempted to find quadruplets of degrees of field extensions (p_1,p_2,p_3,p_4) where $p_2\approx p_1/2$, $p_3\approx 3\,p_1/8$, and $p_4\approx p_1/3$ (see Section 5), and used randomly chosen curves of genus i over $\mathbb{F}_{p_i}$ for $i=1,2,3$, and 4. We admitted tolerances of at most 2% (in bits) of security level between the "most" and the "least" secure curves in each quadruplet. Due to the highly irregular distribution of primes, we could not find neat matches for all security levels, but we could find 9 good sets, from low-cost security (roughly 140 bits), to high security (270 bits). In four of these sets some curves are missing, but we included them to offer a broad range of cases.

For each curve and security level we report the timings for doubling of a point (DBL) and addition of two different points on the curve (ADD), then scalar multiplication timings using the non-adjacent form (NAF) and a signed windowing method based on the NAF_w. We also compare the timings of our implementations using sequential multiplications (cfr. Section 3.2) and without using sequential multiplications (i.e. each sequential multiplication is replaced by several multiplication).

We did not devise and implement projective coordinates for curves of genus greater than one. We either looked at the formulæ currently available in literature, or estimated the number of multiplications that such formulæ would require, and verified that in our situation (with relatively low inversion to multiplication ratios) projective coordinates would not give a performance gain. For EC the situation is slightly different. We show timings for both operations in affine coordinates and mixed affine/López-Dahab coordinates (also using a mixed coordinate approach to reduce some of the precomputation costs of the NAF_w scalar multiplication).

In Figure 2 we show the best timings for each curve type and security level, whereas in Figure 3 the timings of curves of genus at least two are normalized with respect to EC performance.

It is immediate that the performance of curves of genus four is comparable to that of EC, and is often better. Curves of genus two and three have similar performance, with genus three winning in some cases despite the use of the Lange-Stevens doubling (the genus two addition formula also profits from the use of sequential multiplication) and both perform better than curves of genus one and four.

This result is obtained by using sequential multiplications. The gain (as seen by Table 4) is often around 15% *for curves of genus three with regard to an implementation that does not use sequential multiplications. For genus four, this gain approaches* 20%*. Because of the nature of the arithmetic, there are no gains for EC in affine coordinates and a very small gain using López-Dahab coordinates (less than* 1%*.) For curves of genus two the improvement is around* 2%*.*

It may be interesting to compare our results to those obtained by Pelzl, Wollinger, and Paar in [59] and [52]. We collected some of their timings in Table 5, and grouped them according to our security level criteria to ease comparison with ours. The processor used in these papers is a 1.8 Ghz Pentium 4. In our tests, such a machine is usually slightly slower than our 1.5 Ghz Powerpc G4 on multiplication, and significantly slower in the inversion (with an inversion to multiplication ratio between 9 and 12). In fact, for the field $\mathbb{F}_{2^{47}}$ they measure 3.752 μsec for an inversion and 0.407 μsec for a multiplication, giving a ratio of 9.33, whereas for the same field we have 0.483 μsec for an inversion, 0.087 μsec for a multiplication, and a ratio of 5.511 (cf. Figure 1).

On their architecture, the fastest coordinate system for EC is therefore one of the projective systems, and the disparity between point addition and doubling in their timings seems to confirm that they used such a coordinate system (although they do not give information about this). For higher genera they use affine coordinates, just as we do. They also seem to use a windowing scalar multiplication method. On the security side, they do not hesitate to use non-prime extension fields, generating curves which are possibly weaker because of Weil descent attacks (see Section 5). We note that their arithmetic is faster on $\mathbb{F}_{2^{63}}$ than on $\mathbb{F}_{2^{59}}$, suggesting that fast field arithmetic techniques were used for fields of non-prime extension, for instance to speed-up inversions, thus skewing the comparison of performance at given security levels in an unfair way towards potentially weaker fields. They do not employ optimizations depending on field size (except when the extension degree is composite), nor serial multiplication techniques, and do not restrict keysizes.

A comparison of the results must then be viewed with some care, but our results show a clear improvement in the performance. This is already significant for genus 2 and 3 where in [59, 52] the performance roughly matches that of elliptic curves, while we show significant gains. Their results for genus 4 would suggest that the performance of such curves makes them unsuitable for practical applications, with a increase in costs by a factor of close to 4, but our results show similar performance to EC. A comparison of the two architectures and of the EC results clearly shows that our EC implementation was given at least as much care as theirs.

The disparity in the timings for smaller fields also reflects that we have been able to efficiently remove the overheads in the software implementation, thus reducing the penalities suffered by curves of higher genus.

To put things in a broader perspective, Table 6 gives timings for other implementations over fields of even characteristic. Computer architectures evolve very quickly and, as we already mentioned, not all operations become faster at the same rate (the discussion about inversion speed on the PowerPC G4 and Pentium 4 architectures is just one of many examples). However, we can try to compare results to show the evolution of implementation performance. Roughly speaking, the computer used in [30] is between 20 and 25 times slower than our reference architecture (1.5 GHz Powerpc G4), the computer in [53] about 3 to 4 times slower, the one used in [54] 2 to 3 times slower, and the machine employed in [32] is at least 30% slower. It should be noted that [53] and [54] use 64-bits processors, so the fields of 41 and 59 bits require only single-register arithmetic, which may explain the sharp increase in timings when the field size increases to 89 and 113 bits for the curves of genus 3.

The timings show an improvement by a factor close to 60 in scalar multiplication performance on a curve of genus three over a binary field of 59 bits from [54] to our implementation, which amounts to a factor between 15 and 20 once the processor speeds or the choices of scalar (including the use of a NAF) are taken into account. This increase in performance is due to the use of highly optimised explicit formulæ in place of Cantor's algorithm. A comparison with [59] (see Table 5) suggests that our improved formulæ and field arithmetic implementations are responsible for a speed-up by a factor in the order of 5 to 8.

7 Conclusions

In this paper we reconsidered the issue of implementing low genus hyperelliptic Jacobian arithmetic over fields of even characteristic. Instead of independently addressing field arithmetic, the derivation of explicit formulæ, and the impact of recent security research, we studied the interplay of these issues.

Our work starts in Section 2 with a thorough reconsideration of the methodology used to derive explicit formulæ from Cantor's algorithm. In fact, the paper at hand contains the first comprehensive discussion of the techniques involved, at least for the case of affine coordinates. In particular we collect and discuss ideas previously scattered in a lot of different papers, including, for example [47], [24], and [32].

In Section 3 we moved to the issues surrounding efficient implementation of finite fields of characteristic two and of prime extension degree. Even though some of the implementation techniques were already known, we pushed them as much as possible in order to reduce the performance penalties associated to architecture granularity. The results, while offering very good performance, show also a behaviour very close to the theoretical bit-complexity for a wide range of fields.

When explicit formulæ are designed, it is customary to speed-up polynomial operations by means of Karatsuba-like tricks. We found that using sequential multiplication routines (also described in Section 3) in combination with these Karatsuba-like tricks

Table 4. *Scalar multiplication timings in µsec (1.5 GHz Powerpc G4)*

Sec. lvl.	Curve	Using sequential mults				No sequential mults			
		DBL	ADD	NAF	NAF_w	DBL	ADD	NAF	NAF_w
140	ec(139) A	6.20	6.21	1151	1029	6.20	6.21	1151	1029
	ec(139) A/LD	4.83	7.59	1033	893	4.83	7.78	1048	904
	g2(71)	2.39	5.28	577	492	2.39	5.51	588	499
	g3(53)	1.93	5.89	542	446	2.06	6.59	592	486
	g4(47)	3.07	10.35	906	739	3.66	12.98	1110	900
160	ec(157) A	6.80	6.81	1421	1267	6.80	6.81	1421	1267
	ec(157) A/LD	6.09	10.20	1517	1284	6.09	10.60	1539	1295
	g2(79)	2.39	5.17	645	546	2.39	5.41	656	554
	g3(59)	2.14	6.79	692	562	2.37	7.65	772	626
	g4(53)	3.67	12.18	1214	980	4.28	14.81	1446	1162
180	ec(179) A	8.37	8.37	1994	1771	8.37	8.37	1994	1771
	ec(179) A/LD	7.33	12.27	2082	1756	7.33	12.47	2094	1762
	g2(89)	2.78	6.42	880	734	2.78	6.67	895	743
	g3(67)	3.00	9.97	1132	904	3.40	11.93	1320	1047
	g4(59)	4.18	13.72	1567	1253	4.84	17.02	1883	1494
200	ec(199) A	14.34	14.38	3807	3370	14.34	14.38	3807	3370
	ec(199) A/LD	7.62	12.69	2410	2066	7.62	13.20	2443	2083
	g2(101)	4.14	8.94	1417	1183	4.14	9.41	1448	1202
	g3(no curve)	–	–	–	–	–	–	–	–
	g4(67)	5.89	20.59	2538	1999	6.78	24.87	2998	2348
210	ec(211) A	15.45	15.47	4344	3838	15.45	15.47	4344	3838
	ec(211) A/LD	8.36	14.13	2827	2407	8.36	14.67	2865	2426
	g2(107)	4.32	9.16	1557	1299	4.32	9.65	1592	1320
	g3(79)	3.34	11.10	1484	1172	3.65	13.02	1686	1319
	g4(71)	6.22	22.34	2884	2254	7.07	26.32	3342	2601
220	ec(no curve)	–	–	–	–	–	–	–	–
	g2(109)	4.41	9.56	1673	1389	4.41	10.05	1708	1410
	g3(83)	3.74	12.54	1743	1371	4.11	14.40	1960	1533
	g4(73)	6.50	23.02	3119	2436	7.36	26.93	3594	2796
240	ec(239) A	17.97	17.93	5709	5025	17.97	17.93	5709	5025
	ec(239) A/LD	7.87	13.87	3058	2600	7.87	14.16	3081	2612
	g2(no curve)	–	–	–	–	–	–	–	–
	g3(89)	3.90	13.45	2003	1561	4.55	16.71	2429	1878
	g4(79)	6.46	23.09	3398	2636	7.40	27.49	3976	3068
250	ec(251) A	18.90	18.90	6317	5555	18.90	18.90	6317	5555
	ec(251) A/LD	9.38	15.78	3758	3193	9.38	16.37	3808	3218
	g2(127)	4.67	11.36	2115	1721	4.67	11.81	2152	1743
	g3(no curve)	–	–	–	–	–	–	–	–
	g4(83)	7.37	25.82	3993	3098	8.26	30.08	4571	3529
270	ec(269) A	24.28	24.28	8697	7635	24.28	24.28	8697	7635
	ec(269) A/LD	11.19	21.31	5021	4249	11.19	21.89	5079	4276
	g2(137)	7.53	17.14	3576	2925	7.53	17.51	3610	2945
	g3(101)	5.44	18.28	3116	2421	6.37	23.28	3805	2921
	g4(89)	7.76	27.64	4583	3532	9.27	34.62	5618	4302

Fig. 2. Scalar multiplication performance for curves of different genera at various security levels (timings in msec)

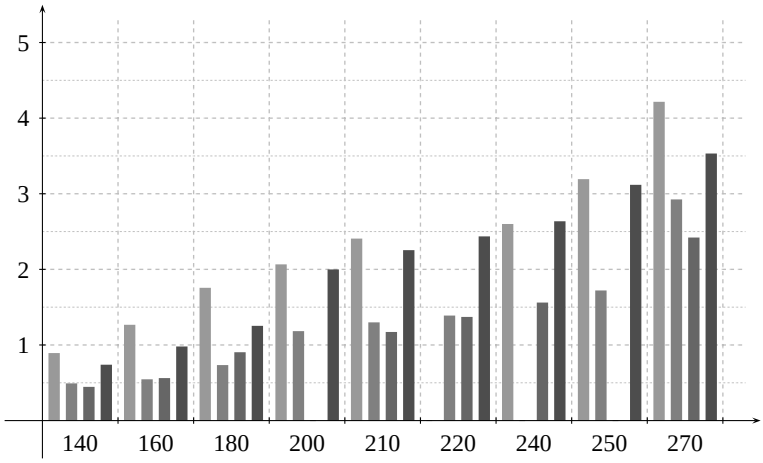


Fig. 3. Scalar multiplication performance for curves of genera two, three and four at various security levels, relative to EC performance (normalized to 1.0)

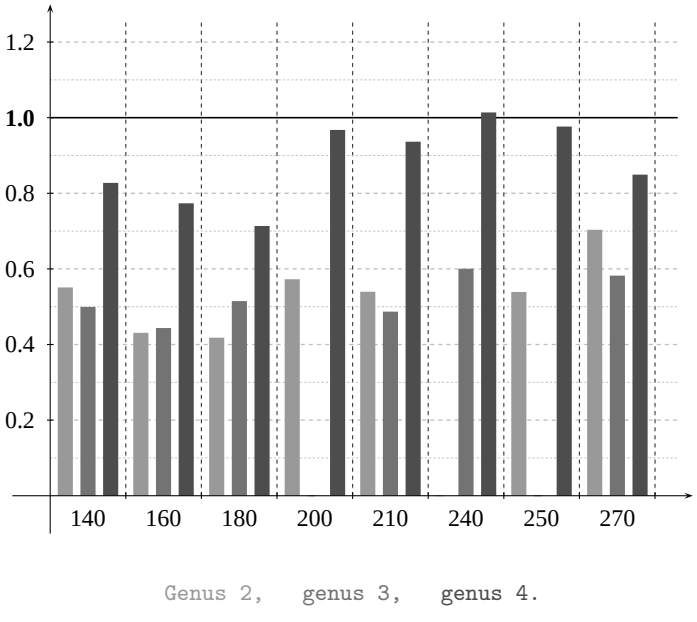


Table 5. *Timings of group operation and scalar multiplication according to Pelzl et al. [59], except for the starred results, which are from [52] (all tests on the same 1.8 GHz Pentium 4 workstation)*

Sec. Lvl.	Curve	ADD μsec.	DBL μsec.	Scal. Mult. msec.
162	ec (163)	18.3	9.4	2.60
	g2 (81)	18.7	11.7	2.73
	g3 (54)	24.8	8.9	2.56
	g4 (47)	53.8	30.7	8.59
	g4 (47)*	50.3	29.3	8.05
180	ec (179)	16.9	9.8	2.80
	g2 (91)	21.1	13.7	3.50
	g3 (63)	25.3	9.2	3.10
	g4 (59)	65.2	37.9	10.36
190	ec (191)	15.4	8.7	2.78
	g2 (95)	19.0	12.6	3.41
	g4 (63)	63.7	33.1	9.50
	g4 (63)*	51.6	29.8	8.43

Table 6. *Performance of other hyperelliptic curve software implementations over binary fields.*

Paper	Architecture	Curve	Timing (msec.)
[30]	Pentium 100 Mhz	g2 (64)	730
		g3 (42)	1200
		g4 (31)	1100
		g5 (25)	1800
[53]	Alpha 21164A 467 Mhz	g3 (59)	83.3
		g4 (41)	96.6
		g3 (89)	2570
		g3 (113)	3790
[54]	Alpha21164A 600 Mhz	g3 (59)	40
		g4 (41)	43
[32]	Pentium 4 1.5 Ghz	g2 (83)	18.875
		g2 (97)	27.188

brings substantial performance gains, especially as the genus increases. This is one of the key observations in Section 4, and these are reflected in the real world results reported in Section 6. For genera three, resp. four, the gains are often close to 15%, resp. 20%. Smaller gains can be obtained even when using López-Dahab coordinates for elliptic curves as well as in the affine genus two formulæ: this has been taken into account in our comparisons. Our genus four formulæ are much better than the best ones previously published (for example, doublings costs decrease from 2I+75M+14S in [52] to 1I+28M+16S in this paper) and are further improved by the use of sequential multiplications

Moreover, we restrict the computations to specific subgroups or subsets of the considered algebraic groups to speed up scalar multiplication provided there is no reduction in security (Section 5) by doing so. This is due to the fact that index calculus methods attack the DLP in the whole algebraic group and do not consider subgroups or key sizes - whereas square root methods (such as Pollard's methods) can be restricted to search in subgroups or among keys of a certain size.

Some interesting results and observations stemming from the benchmarks (Section 6) are:

- Curves of genus three provide similar performance to curves of genus two, and perform even better in some circumstances.
  If sequential multiplications can successfully be implemented in hardware, these curves could be very interesting for hardware implementors since smaller multiplying units are required for genus three than for genus two.
- Cryptographically secure curves of genus three and four may be easier to find since we can allow larger cofactors than in the genus one and two cases.
- Genus four is still interesting, when used wisely, as its performance is comparable to that of elliptic curves.

References

1. R. Avanzi, H. Cohen, C. Doche, G. Frey, T. Lange, K. Nguyen, and F. Vercauteren. *The Handbook of Elliptic and Hyperelliptic Curve Cryptography*. CRC Press, 2005.
2. R. Avanzi. Aspects of hyperelliptic curves over large prime fields in software implementations. In *Cryptographic Hardware and Embedded Systems – CHES 2004*, LNCS 3156, 148–162, Springer-Verlag, 2004.
3. R. Avanzi. Generic Algorithms for Computing Discrete Logarithms. In [1], chapter 19, 477–494.
4. R. Avanzi and N. Thériault. Index Calculus. In [1], chapter 20, 495–509.
5. R. Avanzi and N. Thériault. Index Calculus for Hyperelliptic Curves. In [1], chapter 21, 511–527.
6. I. F. Blake, G. Seroussi, and N. P. Smart. *Elliptic curves in cryptography*. *London Mathematical Society Lecture Note Series*, vol. 265. Cambridge University Press, Cambridge, 1999.
7. I. F. Blake, G. Seroussi, and N. P. Smart. *Advances in Elliptic Curve Cryptography*. *London Mathematical Society Lecture Note Series*, vol. 317. Cambridge University Press, Cambridge, 2005.
8. D. G. Cantor. Computing in the Jacobian of hyperelliptic curves. *Mathematics of Computation*, 48 (177), 95–101, 1987.

9. C. Doche and T. Lange. Arithmetic of Elliptic Curves. In [1], chapter 13, 267–302.
10. S. Duquesne and T. Lange. Arithmetic of Hyperelliptic Curves. In [1], chapter 14, 303–353.
11. ECRYPT. Ecrypt yearly report on algorithms and keysizes. Technical report, ECRYPT, 2005.
12. A. Enge. Computing discrete logarithms in high-genus hyperelliptic Jacobians in provably subexponential time. *Mathematics of Computation*, 71 (238), 729–742, 2002.
13. X. Fan, T. Wollinger and Y. Wang. Efficient Doubling on Genus 3 Curves over Binary Fields. IACR ePrint 2005/228.
Available from `http://eprint.iacr.org/2005/228`
14. K. Fong, D, Hankerson, J, López, A, Menezes:. Field Inversion and Point Halving Revisited. IEEE Trans. Computers 53(8), 1047–1059, 2004.
15. G. Frey, M. Müller, and H. Rück. The Tate pairing and the discrete logarithm applied to elliptic curve cryptosystems. IEEE Transactions on Information Theory, 45 (5), 1717–1719, 1999.
16. G. Frey and H. Rück. A remark concerning $m-$divisibility and the discrete logarithm problem in the divisor class group of curves. *Mathematics of Computation*, 62 (206), 865–874, 1994.
17. S. D. Galbraith. Supersingular Curves in Cryptography. In: *Advances in Cryptology – Asiacrypt 2001*. LNCS 2248, 49–513, Springer-Verlag, 2001.
18. P. Gaudry. Index calculus for abelian varieties and the elliptic curve discrete logarithm problem IACR ePrint 2004/073.
Available from `http://eprint.iacr.org/2004/073`
19. P. Gaudry, F. Hess and N. P. Smart. Constructive and destructive facets of Weil descent on elliptic curves. *J. Cryptology*, **15**, no. 1, 19-46, 2002.
20. P. Gaudry, E. Thomé, N. Thériault and C. Diem. A double large prime variation for small genus hyperelliptic index calculus. IACR ePrint 2004/153.
Available from `http://eprint.iacr.org/2004/153`
21. C. Guyot, K. Kaveh and V. M. Patankar. Explicit algorithm for the arithmetic on the hyperelliptic Jacobians of genus 3. *Journal of the Ramanujan Mathematical Society*, **19**, no. 2, 75–115, 2004.
22. D. Hankerson, J. López-Hernandez, and A. Menezes. Software Implementation of Elliptic Curve Cryprography over Binary Fields. In: *Cryptographic Hardware and Embedded Systems – CHES 2000*. LNCS 1965, 1–24, Springer-Verlag, 2001.
23. D. Hankerson, A. J. Menezes, and S. A. Vanstone. *Guide to elliptic curve cryptography*. Springer-Verlag, 2003.
24. R. Harley. Fast arithmetic on genus two curves.
Available at: http://cristal.inria.fr/~harley/hyper/, 2000
25. F. Hess. The GHS Attack Revisited. In: *Advances in Cryptology – Eurocrypt 2003*. LNCS 2656, 374–387, Springer-Verlag, 2003.
26. A. Karatsuba and Y. Ofman. Multiplication of Multidigit Numbers on Automata. *Soviet Physics - Doklady*, **7**, 595–596, 1963.
27. N. Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48 (177), 203–209, 1987.
28. N. Koblitz. *Algebraic aspects of cryptography.* Springer-Verlag, 1998.
29. N. Koblitz. Hyperelliptic cryptosystems. *J. Cryptology*, 1, 139–150, 1989.
30. U. Krieger. `signature.c`, Masters thesis. Fachbereich Mathematik und Informatik, Universität Duisburg–Essen, Essen, Germany, February 1997.
31. J. Kuroki, M. Gonda, K. Matsuo, J. Chao and S. Tsujii. Fast genus three hyperelliptic curve cryptosystems. The 2002 Symposium on Cryptography and Information Security, Japan – SCIS 2002, 29 Jan. – 1 Feb. 2002.

32. T. Lange. Efficient Arithmetic on Genus 2 Hyperelliptic Curves over Finite Fields via Explicit Formulae. Cryptology ePrint Archive, Report 2002/121, 2002, http://eprint.iacr.org/.
33. T. Lange. Formulae for arithmetic on genus 2 hyperelliptic curves. Appl. Algebra Engrg. Comm. Comput. **15**, no. 5, 295–328, 2005.
34. T. Lange and M. Stevens. Efficient doubling for genus two curves over binary fields. In: *Selected Areas in Cryptography – SAC 2004*. LNCS 3357, 170–181, Springer-Verlag, 2005.
35. A. K. Lenstra, E. R. Verheul. Selecting Cryptographic Key Sizes. *J. Cryptology*, **14** (4), 255–293, 2001.
36. R. Lercier, D. Lubicz and F. Vercauteren. Point Counting on Elliptic and Hyperelliptic Curves. In [1], chapter 17, 407–453.
37. C.-H. Lim and H.-S. Hwang. Fast Implementation of Elliptic Curve Arithmetic in GF(p^n) In Public Key Cryptography, Proceedings of PKC 2000, 405–421. Springer-Verlag 2000.
38. C. Lim and P. Lee. More flexible exponentiation with precomputation. In: *Advances in Cryptology - Crypto '94*. LNCS 839, 95–107, Springer-Verlag 1994.
39. J. López and R. Dahab. High-speed software multiplication in $\mathbb{F}_{2^m}$. Progress in Cryptology - INDOCRYPT 2000. LNCS 1977, 203–212, Springer-Verlag, 2000.
40. K. Matsuo, J. Chao and S. Tsujii. Fast genus two hyperelliptic curve cryptosystems. In: ISEC2001-31, IEICE, 2001.
41. A. Menezes, T. Okamoto and S. Vanstone. Reducing elliptic curve logarithms in a finite field. IEEE Transactions on Information Theory, vol. IT-39 (5), 1639–1646, 1993.
42. A. Menezes and E. Teske. Cryptographic Implications of Hess' Generalized GHS Attack. IACR ePrint 2004/235.
Available from `http://eprint.iacr.org/2004/235`
43. A. Menezes, Y.-H. Wu and R. Zuccherato. An Elementary Introduction to Hyperelliptic Curves. In [28]. Pages 155–178.
44. V. S. Miller. Use of elliptic curves in cryptography. In: *Advances in Cryptology – Crypto 1985*. LNCS 218, 417–426. Springer-Verlag, 1986.
45. Y. Miyamoto, H. Doi, K. Matsuo, J. Chao, and S. Tsuji. A Fast Addition Algorithm of Genus Two Hyperelliptic Curve. In: SCIS 2002, IEICE Japan, 2002, 497–502. In Japanese.
46. D. Mumford. *Tata Lectures on Theta II*. Birkhäuser, 1984.
47. K. Nagao. Improving group law algorithms for Jacobians of hyperelliptic curves. In: *Algorithmic Number Theory – ANTS-IV*. LNCS 1838, 439–448, Springer-Verlag, 2000.
48. K. Nagao. Improvement of Thériault Algorithm of Index Calculus for Jacobian of Hyperelliptic Curves of Small Genus. IACR ePrint 2004/161.
Available from `http://eprint.iacr.org/2004/161`
49. NIST. Key Management Guideline - Workshop Document. Draft, October 2001.
Available from
`http://csrc.nist.gov/encryption/kms/key-management-guideline-(workshop).pdf`
50. J. Pelzl, T. Wollinger, J. Guajardo and C. Paar. Hyperelliptic curve cryptosystems: closing the perfomance gap to elliptic curves (Update). IACR ePrint 2003/026.
Available from `http://eprint.iacr.org/2003/026`
51. J. Pelzl, T. Wollinger and C. Paar. High Performance Arithmetic for Hyperelliptic Curve Cryptosystems of Genus Two. IACR ePrint 2003/212.
Available from `http://eprint.iacr.org/2003/212`
52. J. Pelzl, T. Wollinger and C. Paar. Low cost Security: Explicit Formulae for Genus 4 Hyperelliptic Curves. In: *Selected Areas in Cryptography – SAC 2003*. LNCS 3006, 1–16, Springer-Verlag, 2004.
53. Y. Sakai and K. Sakurai. Design of Hyperelliptic Cryptosystems in small Characteristic and a Software Implementation over $\mathbb{F}_{2^n}$. In Advances in Cryptology - ASIACRYPT 98. LNCS 1514, 80–94. Springer Verlag, 1998.0

54. Y. Sakai and K. Sakurai. On the Practical Performance of Hyperelliptic Curve Cryptosystems in Software Implementation. In IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences, vol. E83-A NO.4, April 2000, 692–703.
55. J. Scholten and H. J. Zhu. Hyperelliptic Curves in Characteristic 2. Inter. Math. Research Notices, 17, 905-917, 2002.
56. R. Schroeppel, H. Orman, S. O'Malley and O. Spatscheck. Fast key exchange with elliptic curve systems. In: *Advances in Cryptology - Crypto '95*. LNCS 963, 43–56, Springer-Verlag, 1995.
57. V. Shoup. NTL: A Library for doing number theory.
Available from: `http://shoup.net/ntl/`
58. N. Thériault. Weil Descent for Artin-Schreier Curves.
Available from: `http://www.cacr.math.uwaterloo.ca/~ntheriault`
59. T. Wollinger, J. Pelzl, and C. Paar. Cantor versus Harley: Optimization and Analysis of Explicit Formulae for Hyperelliptic Curve Cryptosystems. to appear in IEEE Transactions on Computers.

A Almost inverse using Cramer's rule

In this section, we describe how to compute the almost inverse in the genus four group addition. The situation for genus three is a simple restriction of the genus four case.

To compute the almost inverse of $u_1(x)$ modulo $u_2(x)$, we apply Cramer's rule to the matrix M given by:

$$M = \begin{pmatrix} M_{0,0} & M_{0,1} & M_{0,2} & M_{0,3} \\ M_{1,0} & M_{1,1} & M_{1,2} & M_{1,3} \\ M_{2,0} & M_{2,1} & M_{2,2} & M_{2,3} \\ M_{3,0} & M_{3,1} & M_{3,2} & M_{3,3} \end{pmatrix}$$

where the i^{th} column corresponds to $x^i u_1(x) \bmod u_2(x)$. To compute M, we compute the columns inductively, starting from the left ($u_1(x) \bmod u_2(x)$), multiplying by x and reducing modulo $u_2(x)$ at each step:

$$M = \begin{pmatrix} a_0+b_0 & M_{0,3}\cdot b_0 & M_{1,3}\cdot b_0 & M_{2,3}\cdot b_0 \\ a_1+b_1 & M_{0,0}+M_{0,3}\cdot b_1 & M_{1,0}+M_{1,3}\cdot b_1 & M_{2,0}+M_{2,3}\cdot b_1 \\ a_2+b_2 & M_{0,1}+M_{0,3}\cdot b_2 & M_{1,1}+M_{1,3}\cdot b_2 & M_{2,1}+M_{2,3}\cdot b_2 \\ a_3+b_3 & M_{0,2}+M_{0,3}\cdot b_3 & M_{1,2}+M_{1,3}\cdot b_3 & M_{2,2}+M_{2,3}\cdot b_3 \end{pmatrix}$$

Note that the products in the computation of the second column appear again at the later step of the group addition (see Subsection 4.3).

In Subsection 4.2, we showed that obtaining $r = det(M)$ is straightforward once inv_0, inv_1, inv_2 and inv_3 are computed, and these can be written as:

$$inv_0 = det\begin{pmatrix} M_{1,1} & M_{1,2} & M_{1,3} \\ M_{2,1} & M_{2,2} & M_{2,3} \\ M_{3,1} & M_{3,2} & M_{3,3} \end{pmatrix}, \quad inv_1 = det\begin{pmatrix} M_{1,0} & M_{1,2} & M_{1,3} \\ M_{2,0} & M_{2,2} & M_{2,3} \\ M_{3,0} & M_{3,2} & M_{3,3} \end{pmatrix},$$

$$inv_2 = det\begin{pmatrix} M_{1,0} & M_{1,1} & M_{1,3} \\ M_{2,0} & M_{2,1} & M_{2,3} \\ M_{3,0} & M_{3,1} & M_{3,3} \end{pmatrix}, \quad inv_3 = det\begin{pmatrix} M_{1,0} & M_{1,1} & M_{1,2} \\ M_{2,0} & M_{2,1} & M_{2,2} \\ M_{3,0} & M_{3,1} & M_{3,2} \end{pmatrix}.$$

We compute these four determinants using the schoolbook inductive approach, starting from the top row of each matrix. Since all four matrices are in fact submatrices of M, each 2×2 determinant of square submatrices coming from the two lower rows of M appear in two of these computations, but they only need to be computed once each. We write the 2×2 determinants as:

$$\begin{aligned}
\alpha_{0,1} &= det \begin{pmatrix} M_{2,0} & M_{2,1} \\ M_{3,0} & M_{3,1} \end{pmatrix} \quad , \quad \alpha_{0,2} = det \begin{pmatrix} M_{2,0} & M_{2,2} \\ M_{3,0} & M_{3,2} \end{pmatrix} \quad , \\
\alpha_{0,3} &= det \begin{pmatrix} M_{2,0} & M_{2,3} \\ M_{3,0} & M_{3,3} \end{pmatrix} \quad , \quad \alpha_{1,2} = det \begin{pmatrix} M_{2,1} & M_{2,2} \\ M_{3,1} & M_{3,2} \end{pmatrix} \quad , \\
\alpha_{1,3} &= det \begin{pmatrix} M_{2,1} & M_{2,3} \\ M_{3,1} & M_{3,3} \end{pmatrix} \quad , \quad \alpha_{2,3} = det \begin{pmatrix} M_{2,2} & M_{2,3} \\ M_{3,2} & M_{3,3} \end{pmatrix} \quad .
\end{aligned}$$

The algorithm can now be written simply in terms of multiplications: We first compute

$$\alpha_{i,j} = M_{2,i} \cdot M_{3,j} + M_{2,j} \cdot M_{3,i}$$

for $0 \leq i < j \leq 3$, then

$$inv_i = \sum_{j \neq i} \alpha_{k_1,k_2} \cdot M_{1,j}$$

for $0 \leq i \leq 3$ (where $\{k_1, k_2\}$ is the ordered set given by $\{0,1,2,3\} \setminus \{i, j\}$), and finaly

$$r = inv_0 \cdot M_{0,0} + inv_1 \cdot M_{0,1} + inv_2 \cdot M_{0,2} + inv_3 \cdot M_{0,3} \quad .$$

These computations require a total of 40 multiplications: 12 to compute M, 12 for the $\alpha_{i,j}$'s, 12 for the inv_i's and 4 for r. The regular structure of these multiplications makes them very convenient to do in combination with sequential multiplications, giving 11 blocks of products (3 to compute M and 4 each for the $\alpha_{i,j}$'s and the inv_i's), leaving only the products in the computation of r as single multiplications. These 4 remaining products are then combined into pairs of multiplications with some of the products required for the computations of $s'(x)$.

B Genus three formulæ

Addition formula $D_3 = D_1 \oplus D_2$

Inputs:
$D_1 = [u_1(x), v_1(x)]$, $u_1(x) = a_0 + a_1x + a_2x^2 + x^3$ and $v_1(x) = c_0 + c_1x + c_2x^2$
$D_2 = [u_2(x), v_2(x)]$, $u_2(x) = b_0 + b_1x + b_2x^2 + x^3$ and $v_2(x) = d_0 + d_1x + d_2x^2$
$C : y^2 + h(x)y = f(x)$ with $h(x) = h_0 \in \mathbb{F}_q$ and $f(x) = f_0 + f_1x + f_2x^2 + f_3x^3 + f_4x^4 + f_5x^5 + x^7$

Outputs:
$D_3 = [u_3(x), v_3(x)]$, $u_3(x) = e_0 + e_1x + e_2x^2 + x^3$ and $v_3(x) = \varepsilon_0 + \varepsilon_1x + \varepsilon_2x^2$

Step	Operations	Cost	Effective
1	Almost inverse, $inv(x) = r \cdot u_1(x)^{-1} \bmod u_2(x)$, via Cramer's rule, $inv(x) = inv_0 + inv_1x + inv_2x^2 + inv_3x^3$ $M_{0,0} = b_0 + a_0$; $M_{1,0} = b_1 + a_1$; $M_{2,0} = b_2 + a_2$; $(M_{0,1}, T_0, T_1) = M_{2,0} \cdot (b_0, b_1, b_2)$; $M_{1,1} = T_0 + M_{0,0}$; $M_{2,1} = T_1 + M_{1,0}$; $(M_{0,2}, t_2, t_3) = M_{2,1} \cdot (b_0, b_1, b_2)$; $M_{1,2} = t_2 + M_{0,1}$; $M_{2,2} = t_3 + M_{1,1}$; $(t_4, t_5) = M_{1,0} \cdot (M_{2,2}, M_{2,1})$; $(t_6, t_7) = M_{1,1} \cdot (M_{2,2}, M_{2,0})$; $(t_8, t_9) = M_{1,2} \cdot (M_{2,0}, M_{2,1})$; $inv_0 = t_6 + t_9$; $inv_1 = t_4 + t_8$; $inv_2 = t_5 + t_7$; If r is 0 use Cantor's algorithm	12 M	9.7 M
2	$r = inv(x) \cdot u_1(x) \bmod u_2(x)$ $q_0 = d_0 + c_0$; $q_1 = d_1 + c_1$; $q_2 = d_2 + c_2$; $(t_{10}, T_{11}) = inv_0 \cdot (M_{0,0}, q_0)$; $(t_{12}, \lambda_1) = inv_2 \cdot (M_{0,2}, q_2)$; $t_{13} = t_{12} + t_{10}$; $(t_{14}, T_{15}) = inv_1 \cdot (M_{0,1}, q_1)$; $r = t_{13} + t_{14}$;	6 M	5.1 M
3	$s'(x) = r \cdot s(x)$, where $s(x) = u_1(x)^{-1} \cdot (v_2(x) + v_1(x)) \bmod u_2(x)$, $s'(x) = s'_0 + s'_1x + s'_2x^2$ $t_{16} = inv_0 + inv_1$; $t_{17} = inv_0 + inv_2$; $t_{18} = inv_1 + inv_2$; $t_{19} = q_1 + q_2$; $t_{20} = q_1 + q_0$; $t_{21} = q_0 + q_2$; $t_{22} = t_{17} \cdot t_{21}$; $t_{23} = t_{18} \cdot t_{19}$; $(t_{24}, t_{25}) = \lambda_1 \cdot (b_1, b_2)$; $\lambda_0 = t_{25} + T_{15} + \lambda_1 + t_{23}$; $(t_{26}, t_{27}) = \lambda_0 \cdot (b_0, b_2)$; $s'_2 = t_{22} + T_{11} + \lambda_1 + T_{15} + t_{24} + t_{27}$; $s'_0 = t_{26} + T_{11}$; $t_{28} = t_{20} \cdot t_{16}$; $t_{29} = \lambda_0 + \lambda_1$; $t_{30} = b_0 + b_1$; $t_{31} = t_{29} \cdot t_{30}$; $s'_1 = t_{31} + s'_0 + T_{15} + t_{24} + t_{28}$;	8 M	7.4 M
4	computation of inverses and $\tilde{s}(x)$ ($s'(x)$ made monic), $\tilde{s}(x) = \tilde{s}_0 + \tilde{s}_1x + x^2$ $t_{32} = r \cdot s'_2$ If t_{32} is 0 use Cantor's algorithm $t_{33} = 1/t_{32}$; $t_{34} = (s'_2)^2$; $(t_{35}, s_2) = t_{33} \cdot (r, t_{34})$; $(T_{36}, \tilde{s}_0, \tilde{s}_1) = t_{35} \cdot (r, s'_0, s'_1)$;	1 I + 6 M + 1 S	1 I + 5 M + 1 S
5	$u_T(x) = \left\lfloor \frac{\tilde{s}(x)^2 u_1(x)}{u_2(x)} \right\rfloor + s_2{}^{-2}(x + a_2 + b_2)$, $u_T(x) = u_{T,0} + u_{T,1}x + u_{T,2}x^2 + u_{T,3}x^3 + x^4$ $t_{37} = (\tilde{s}_0)^2$; $t_{38} = (\tilde{s}_1)^2$; $(t_{39}, t_{40}) = t_{38} \cdot (a_1, a_2)$; $u_{T,3} = a_2 + b_2$; $u_{T,2} = t_{38} + a_1 + b_1 + T_1$; $(t_{41}, t_{42}) = u_{T,2} \cdot (b_1, b_2)$; $l_1 = t_{42} + a_0 + b_0 + T_0 + t_{40}$; $t_{43} = l_1 \cdot b_2$; $l_0 = t_{43} + t_{37} + M_{0,1} + t_{39} + t_{41}$; $t_{44} = (T_{36})^2$; $t_{45} = t_{44} \cdot u_{T,3}$; $u_{T,1} = t_{44} + l_1$; $u_{T,0} = t_{45} + l_0$;	6 M + 3 S	5.4 M + 3 S
6	$z(x) = \tilde{s}(x)u_1(x)$, $z(x) = z_0 + z_1x + z_2x^2 + z_3x^3 + z_4x^4 + x^5$ $(t_{46}, t_{47}) = \tilde{s}_1 \cdot (a_1, a_2)$; $(z_0, t_{48}) = \tilde{s}_0 \cdot (a_0, a_2)$; $t_{49} = \tilde{s}_0 + \tilde{s}_1$; $t_{50} = a_0 + a_1$; $t_{47} + z_3 = a_1 + \tilde{s}_0$; $t_{51} = t_{49} \cdot t_{50}$; $z_2 = a_0 + t_{46} + t_{48}$; $z_1 = t_{51} + z_0 + t_{46}$;	5 M	4.4 M
7	$v_T(x) = s_3z(x) + v_1(x) + 1 \bmod u_T(x)$, $v_T(x) = v_{T,0} + v_{T,1}x + v_{T,2}x^2 + v_{T,3}x^3$ $t_{52} = \tilde{s}_1 + u_{T,3} + a_2$; $(t_{53}, t_{54}, t_{55}, t_{56}) = t_{52} \cdot (u_{T,0}, u_{T,1}, u_{T,2}, u_{T,3})$; $t_{57} = t_{53} + z_0$; $t_{58} = t_{54} + u_{T,0} + z_1$; $t_{59} = t_{55} + u_{T,1} + z_2$; $t_{60} = t_{56} + u_{T,2} + z_3$; $(t_{61}, t_{62}, t_{63}, v_{T,3}) = s_2 \cdot (t_{57}, t_{58}, t_{59}, t_{60})$; $T_{64} = t_{61} + c_0$; $v_{T,1} = t_{62} + c_1$; $v_{T,2} = t_{63} + c_2$;	8 M	5.7 M
8	$u_3(x) = Monic\left(\frac{f(x) + v_T(x) + v_T(x)^2}{u_T(x)}\right)$, $u_3(x) = e_0 + e_1x + e_2x^2 + x^3$ $t_{65} = (v_{T,3})^2$; $t_{66} = (v_{T,2})^2$; $e_2 = t_{65} + u_{T,3}$; $(t_{67}, t_{68}) = e_2 \cdot (u_{T,2}, u_{T,3})$; $e_1 = t_{68} + f_5 + u_{T,2}$; $t_{69} = u_{T,3} \cdot e_1$; $e_0 = t_{66} + f_4 + u_{T,1} + t_{67} + t_{69}$;	3 M + 2 S	2.7 M + 2 S
9	$v_3(x) = v_T(x) + 1 \bmod u_3(x)$, $v_3(x) = \varepsilon_0 + \varepsilon_1x + \varepsilon_2x^2$ $(t_{70}, t_{71}, t_{72}) = v_{T,3} \cdot (e_0, e_1, e_2)$; $\varepsilon_2 = v_{T,2} + t_{72}$; $\varepsilon_1 = v_{T,1} + t_{71}$; $\varepsilon_0 = T_{64} + t_{70}$;	3 M	2.3 M
	Total	1 I + 57 M + 6 S	1 I + 47.7 M + 6 S

Variable schedule for the genus 3 addition formula

0	1	2	3	4	5	6	7	8	9	10	11	12	13
$M_{0,0}$	$M_{1,0}$	$M_{2,0}$	$M_{1,1}$	$M_{2,1}$	$M_{0,2}$	t_2	t_3	t_6	t_4	t_5	$M_{0,1}$	T_0	T_1
t_{12}	t_7	q_1	t_8	λ_1	t_{14}	$M_{1,2}$	$M_{2,2}$	inv_0	inv_1	inv_2	t_{49}	t_{43}	t_{42}
t_{13}	q_0	t_{20}	q_2	t_{32}	t_{16}	t_{10}	t_9	t_{17}	t_{18}	t_{19}	t_{54}	l_0	l_1
r	t_{21}	t_{31}	t_{22}	t_{34}	t_{29}	T_{15}	T_{11}	t_{27}	t_{25}	t_{24}	t_{58}	t_{50}	t_{48}
t_{39}	t_{23}	s'_1	s'_2	t_{37}	t_{33}	T_{36}	t_{28}	t_{30}	λ_0	t_{41}	t_{66}	t_{55}	t_{56}
t_{45}	t_{26}	t_{38}	t_{35}	t_{47}	z_0	t_{51}	$\tilde{s}_0$	$\tilde{s}_1$	s_2	t_{46}	e_0	t_{59}	t_{60}
$u_{T,0}$	s'_0	$u_{T,2}$	$u_{T,3}$	z_3	t_{61}	z_1	z_2	t_{52}	t_{69}	t_{53}		t_{68}	t_{65}
	t_{40}	t_{71}	t_{72}		T_{64}	t_{62}	t_{63}	$v_{T,3}$		t_{57}		e_1	e_2
	t_{44}				ε_0	$v_{T,1}$	$v_{T,2}$			t_{67}			
	$u_{T,1}$					ε_1	ε_2						
	t_{70}												

Doubling formula $D_3 = [2]D_1$

Inputs:
$D_1 = [u_1(x), v_1(x)]$, $u_1(x) = a_0 + a_1x + a_2x^2 + x^3$ and $v_1(x) = c_0 + c_1x + c_2x^2$
$C : y^2 + h(x)y = f(x)$ with $h(x) = h_0 \in \mathbb{F}_q$ and $f(x) = f_0 + f_1x + f_2x^2 + f_3x^3 + f_4x^4 + f_5x^5 + x^7$

Outputs:
$D_3 = [u_3(x), v_3(x)]$, $u_3(x) = e_0 + e_1x + e_2x^2 + x^3$ and $v_3(x) = \varepsilon_0 + \varepsilon_1x + \varepsilon_2x^2$

Step	Operations	Cost	Effective
1	$u_C(x) = u_1(x)^2$, $u_C(x) = u_{C,0} + u_{C,2}x^2 + u_{C,4}x^4 + x^6$ $u_{C,0} = (a_0)^2$; $u_{C,2} = (a_1)^2$; $u_{C,4} = (a_2)^2$;	3 S	3 S
2	$v_C(x) = v_1(x)^2 + f(x) \bmod u_C(x)$, $v_C(x) = v_{C,0} + v_{C,1}x + v_{C,2}x^2 + v_{C,3}x^3 + v_{C,4}x^4 + v_{C,5}x^5$ $t_0 = (c_0)^2$; $t_1 = (c_1)^2$; $t_2 = (c_2)^2$; $v_{C,0} = f_0 + t_0$; $v_{C,1} = f_1 + u_{C,0}$; $v_{C,2} = f_2 + t_1$; $v_{C,3} = f_3 + u_{C,2}$; $v_{C,4} = f_4 + t_2$; $v_{C,5} = f_5 + u_{C,4}$; If $v_{C,5}$ is 0 use Cantor's algorithm	3 S	3 S
3	computation of inverse $T_3 = 1/v_{C,5}$;	1 I	1 I
4	$u_T(x) = Monic\left(\frac{f(x)+v_C(x)+v_C(x)^2}{u_C(x)}\right)$, $u_T(x) = u_{T,0} + u_{T,1}x + u_{T,2}x^2 + x^4$ $u_{T,1} = (T_3)^2$; $t_4 = (v_{C,4})^2$; $t_5 = (v_{C,3})^2$; $(t_6, t_7) = u_{T,1} \cdot (t_4, t_5)$; $u_{T,2} = t_6 + u_{C,4}$; $t_8 = u_{C,2} + t_7$; $(t_9, T_{10}, T_{11}) = u_{T,2} \cdot (u_{C,4}, v_{C,5}, v_{C,4})$; $u_{T,0} = t_8 + t_9$;	5 M + 3 S	4 M + 3 S
5	$v_T(x) = v_C(x) + 1 \bmod u_T(x)$, $v_T(x) = v_{T,0} + v_{T,1}x + v_{T,2}x^2 + v_{T,3}x^3$ $t_{12} = v_{C,4} \cdot u_{T,0}$; $t_{13} = v_{C,4} + v_{C,5}$; $t_{14} = u_{T,0} + u_{T,1}$; $t_{15} = t_{13} \cdot t_{14}$; $T_{16} = v_{C,0} + t_{12}$; $v_{T,1} = v_{C,1} + t_{15} + t_{12} + T_3$; $v_{T,2} = v_{C,2} + T_{11} + T_3$; $v_{T,3} = v_{C,3} + T_{10}$;	2 M	2 M
6	$u_3(x) = Monic\left(\frac{f(x)+v_T(x)+v_T(x)^2}{u_T(x)}\right)$, $u_3(x) = e_0 + e_1x + e_2x^2 + x^3$ $e_2 = (v_{T,3})^2$; $t_{17} = (v_{T,2})^2$; $t_{18} = e_2 \cdot u_{T,2}$; $e_1 = u_{T,2} + f_5$; $e_0 = u_{T,1} + t_{17} + t_{18} + f_4$;	1 M + 2 S	1 M + 2 S
7	$v_3(x) = v_T(x) + 1 \bmod u_3(x)$, $v_3(x) = \varepsilon_0 + \varepsilon_1x + \varepsilon_2x^2$ $(t_{19}, t_{20}, t_{21}) = v_{T,3} \cdot (e_0, e_1, e_2)$; $\varepsilon_0 = t_{19} + T_{16}$; $\varepsilon_1 = t_{20} + v_{T,1}$; $\varepsilon_2 = t_{21} + v_{T,2}$;	3 M	2.3 M
	Total	1 I + 11 M + 11 S	1 I + 9.3 M + 11 S

Variable schedule for the genus 3 doubling formula

0	1	2	3	4	5	6	7	8	9	10	11	12	13
$u_{C,4}$	$v_{C,5}$	$v_{C,4}$	$v_{C,3}$	$v_{C,2}$	$v_{C,1}$	$v_{C,0}$	$u_{T,1}$	t_6	t_0	t_1	t_2	$u_{C,0}$	$u_{C,2}$
$u_{T,0}$	t_{15}	t_{13}	$v_{T,3}$	$v_{T,2}$	$v_{T,1}$	T_{16}	e_0	$u_{T,2}$	t_5	t_4	t_7	T_3	t_8
t_{14}	t_{18}							e_1	t_9	T_{10}	T_{11}	t_{20}	t_{12}
t_{17}									e_2		t_{19}	ε_1	t_{21}
											ε_0		ε_2

C Genus four formulæ

Addition formula $D_3 = D_1 \oplus D_2$

Inputs:
$D_1 = \langle u_1(x), v_1(x)\rangle$, $u_1(x) = x^2 + a_{1,1}x + a_{1,0}$ and $v_1(x) = c_{1,1}x + c_{1,0}$; $D_2 = \langle u_2(x), v_2(x)\rangle$, $u_2(x) = x^2 + b_{2,1}x + b_{2,0}$ and $v_2(x) = d_{1,1}x + d_{1,0}$;
$C: y^2 + h(x)y = f(x)$ with $h(x) = h_2x^2 + h_1x + h_0 \in \mathbb{F}_q$ and $f(x) = f_5x^5 + f_4x^4 + f_3x^3 + f_2x^2 + f_1x + f_0$

Outputs:
$D_3 = \langle u_3(x), v_3(x)\rangle$, $u_3(x) = x^2 + e_{3,1}x + e_{3,0}$ and $v_3(x) = e_{2,1}x + e_{2,0}$

Step	Operations	Cost	Effective
1	Almost inverse, $inv(x) = r \cdot u_1(x)^{-1} \bmod u_2(x)$, via Cramer's rule, $inv(x) = inv_0 + inv_1x + inv_2x^2 + inv_3x^3$ $M_{0,0} = a_0 + b_0$; $M_{1,0} = a_1 + b_1$; $M_{2,0} = a_2 + b_2$; $M_{3,0} = a_2 + b_2$; $M_{3,0} = M_{3,0} \cdot (b_0 \cdot b_1, b_2, b_3)$; $M_{1,1} = t_0 + M_{1,0}$; $M_{2,1} = t_1 + M_{1,0}$; $M_{3,1} = M_{2,0} + t_2$; $\langle M_{0,2}, t_3, t_4, t_5\rangle = M_{3,1} \cdot (b_0, b_1, b_2, b_3)$; $M_{1,2} = M_{0,1} + t_3$; $M_{2,2} = M_{1,1} + t_4$; $M_{3,2} = M_{2,1} + t_5$; $\langle M_{0,3}, t_6, t_7, t_8\rangle = M_{3,2} \cdot (b_0, b_1, b_2, b_3)$; $M_{1,3} = M_{0,2} + t_6$; $M_{2,3} = M_{1,2} + t_7$; $M_{3,3} = t_8 + M_{2,2}$; $\langle t_9, t_{10}, t_{11}\rangle = M_{2,2} \cdot (M_{3,3}, M_{3,0}, M_{3,1})$; $copy(M_{3,1})$; $\langle t_{12}, t_{13}, t_{14}\rangle = M_{2,3} \cdot (M_{3,0}, M_{3,1}, M_{3,2})$; $copy(M_{3,0})$; $\langle t_{15}, t_{16}, t_{17}\rangle = M_{2,0} \cdot (M_{3,1}, M_{3,2}, M_{3,3})$; $c_{0,2} = t_{16} + t_{10}$; $c_{0,3} = t_{12} + t_{17}$; $c_{2,3} = t_{14} + t_9$; $\langle t_{18}, t_{19}, t_{20}\rangle = M_{2,1} \cdot (M_{3,2}, M_{3,3}, M_{3,0})$; $c_{0,1} = t_{20} + t_{15}$; $c_{1,2} = t_{18} + t_{11}$; $c_{1,3} = t_{13} + t_{19}$; $T_{21} = M_{2,1} + M_{1,0}$; $T_{22} = M_{3,1} + M_{2,0}$; $copy(c_{0,3})$; $copy(c_{1,3})$; $copy(c_{0,2})$; $\langle t_{23}, t_{24}, t_{25}\rangle = M_{1,2} \cdot (c_{0,3}, c_{0,1}, c_{1,3})$; $copy(c_{0,1})$; $\langle t_{26}, t_{27}, t_{28}\rangle = M_{1,0} \cdot (c_{1,3}, c_{2,3}, c_{1,2})$; $\langle t_{29}, t_{30}, t_{31}\rangle = M_{1,3} \cdot (c_{1,2}, c_{0,2}, c_{0,1})$; $copy(c_{0,2})$; $\langle t_{32}, t_{33}, t_{34}\rangle = M_{1,1} \cdot (c_{0,3}, c_{0,2}, c_{2,3})$; $inv_0 = t_{29} + t_{25} + t_{34}$; $inv_1 = t_{30} + t_{23} + t_{27}$; $inv_2 = t_{31} + t_{32} + t_{26}$; $inv_3 = t_{28} + t_{33} + t_{24}$; $T_{35} = M_{1,1} + M_{0,0}$;	36 M	26.95 M
2	$r = inv(x) \cdot u_1(x) \bmod u_2(x)$ $q_0 = c_0 + d_0 \cdot q_2 = c_2 + d_2$; $\langle t_{36}, T_{37}\rangle = inv_0 \cdot (M_{0,0}, q_0)$; $\langle t_{38}, T_{39}\rangle = inv_2 \cdot (M_{0,2}, q_2)$; $q_1 = c_1 + d_1$; $q_3 = c_3 + d_3$; $\langle t_{40}, T_{41}\rangle = inv_1 \cdot (M_{0,1}, q_1)$; $\langle t_{42}, t_{43}\rangle = inv_3 \cdot (M_{0,3}, q_3)$; $r = t_{36} + t_{40} + t_{38} + t_{42}$; If r is 0 use Cantor's algorithm	8 M	6.8 M
3	$s'(x) = r \cdot s(x)$, where $s(x) = u_1(x)^{-1} \cdot (v_2(x) + v_1(x)) \bmod u_2(x)$, $s'(x) = s'_0 + s'_1x + s'_2x^2 + s'_3x^3$ $t_{43} = q_0 + q_1$; $t_{44} = q_2 + q_3$; $t_{45} = q_0 + q_2$; $t_{46} = q_1 + q_3$; $t_{47} = inv_0 + inv_1$; $t_{48} = inv_2 + inv_3$; $t_{49} = inv_2 + inv_0$; $t_{50} = inv_3 + inv_1$; $t_{51} = t_{43} \cdot t_{47}$; $t_{52} = t_{44} \cdot t_{48}$; $t_{53} = t_{45} \cdot t_{49}$; $t_{54} = t_{46} \cdot t_{50}$; $t_{55} = t_{43} + t_{44}$; $t_{56} = t_{48} + t_{47}$; $t_{57} = t_{55} \cdot t_{56}$; $t_{58} = \lambda_2 \cdot b_2$; $\lambda_1 = t_{52} + T_{39} + \lambda_2 + t_{58}$; $t_{59} = \lambda_1 \cdot b_2$; $t_{60} = \lambda_1 + \lambda_2$; $t_{61} = b_2 + b_3$; $t_{62} = t_{60} \cdot t_{61}$; $\lambda_0 = t_{54} + T_{41} + \lambda_2 + T_{39} + t_{62} + t_{58} + t_{59}$; $t_{63} = \lambda_0 \cdot b_1$; $t_{64} = \lambda_0 + \lambda_1$; $t_{65} = \lambda_0 + b_2$; $t_{66} = b_1 + b_2$; $t_{67} = b_1 + b_3$; $t_{68} = t_{58} + t_{54} + t_{57} + t_{52}$; $t_{69} = t_{64} \cdot t_{66}$; $t_{70} = t_{65} \cdot t_{67}$; $s'_3 = t_{68} + t_{51} + t_{53} + T_{37} + T_{41} + T_{39} + \lambda_2 + t_{70} + t_{59} + t_{63}$; $\langle t_{71}, t_{72}, t_{73}\rangle = h_0 \cdot (\lambda_0, \lambda_1, \lambda_2)$; $s'_2 = t_{59} + t_{53} + T_{37} + T_{39} + T_{41} + t_{69} + t_{63} + t_{73}$; $s'_1 = t_{51} + T_{37} + T_{41} + t_{63} + t_{72}$; $s'_0 = T_{37} + t_{71}$;	14 M	13.3 M
4	computation of inverses and $\tilde{s}(x)$ ($\tilde{s}'(x)$ made monic), $\tilde{s}(x) = \tilde{s}_0 + \tilde{s}_1x + \tilde{s}_2x^2 + x^3$ $t_{74} = b_3 \cdot s'_2$; $\langle t_{75}, t_{76}\rangle = s'_3 \cdot (b_2, t)$; $t_{77} = (s'_3)^2$; $t_{78} = t_{74} + t_{75} + s'_1$; $t_{79} = t_{78} \cdot s'_3$; $t_{80} = (s'_2)^2$; $t_{81} = (t_{76})^2 \cdot t_{82} = t_{79} + t_{80}$; $t_{83} = t_{76} \cdot t_{82}$; If t_{83} is 0 use Cantor's algorithm $t_{84} = 1/t_{83}$; $\langle t_{85}, t_{87}\rangle = t_{84} \cdot (t_{82}, t_{81})$; $\langle s_3, v_{T,5}, t_{96}\rangle = t_{85} \cdot (r, s'_0, s'_1, s'_2)$; $T_{89} = (t_{87})^2$; $t_{90} = (t_{88})^2$;	11 + 14 M + 5 S	11 + 11.55 M + 5 S
5	$u_T(x) = \dfrac{\tilde{s}(x)^2 u_1(x)/u_2(x) + s_3^{-2}(x + a_3 + b_3)}{}$, $u_T(x) = u_{T,0} + u_{T,1}x + u_{T,2}x^2 + u_{T,3}x^3 + u_{T,4}x^4 + u_{T,5}x^5 + x^6$ $t_{91} = (s_0)^2$; $t_{92} = (\tilde{s}_1)^2$; $t_{93} = (\tilde{s}_2)^2$; $\langle t_{94}, t_{95}, t_{96}, t_{97}\rangle = t_{93} \cdot (a_0, a_1, a_2, a_3)$; $\langle t_{98}, t_{99}\rangle = t_{92} \cdot (a_2, a_3)$; $u_{T,5} = a_3 + b_3$; $u_{T,4} = T_{22} + a_2 + b_2 + t_{93}$; $\langle t_{100}, t_{101}\rangle = u_{T,4} \cdot (b_3, b_0)$; $u_{T,3} = T_{21} + a_1 + b_1 + t_{97} + t_{100}$; $t_{102} = t_{94} + t_{101}$; $t_{103} = u_{T,3} + u_{T,4}$; $t_{104} = b_2 + b_3$; $t_{105} = t_{103} \cdot t_{104}$; $t_{106} = u_{T,3} \cdot b_2$; $u_{T,2} = T_{35} + a_0 + b_0 + t_{96} + t_{92} + t_{105} + t_{106} + t_{100}$; $t_{107} = u_{T,2} + u_{T,4}$; $t_{108} = b_1 + b_3$; $t_{109} = u_{T,2} \cdot b_1$; $t_{110} = t_{107} \cdot t_{108}$; $l_1 = M_{0,1} + s_5 + t_{99} + t_{106} + t_{110} + t_{100} + t_{109}$; $t_{111} = u_{T,2} + u_{T,3}$; $t_{112} = b_1 + b_2$; $t_{113} = l_1 \cdot b_3$; $t_{114} = t_{111} \cdot t_{112}$; $l_0 = t_{102} + t_{98} + t_{91} + t_{113} + t_{114} + t_{106} + t_{109}$; $u_{T,1} = l_1 + T_{90}$; $u_{T,0} = l_0 + t_{115}$;	15 M + 3 S	13.25 M + 3 S
6	$z(x) = \tilde{s}(x)u_1(x)$, $\tilde{z}(x) = z_0 + z_1x + z_2x^2 + z_3x^3 + z_4x^4 + z_5x^5 + z_6x^6 + x^7$ $\langle t_{116}, z_0\rangle = \tilde{s}_0 \cdot (a_3, a_0)$; $\langle t_{117}, t_{118}\rangle = \tilde{s}_1 \cdot (a_3, a_1)$; $t_{119} = a_2 \cdot \tilde{s}_2$; $t_{120} = a_0 + a_1$; $t_{121} = a_0 + a_2$; $t_{122} = a_1 + a_2$; $t_{123} = \tilde{s}_0 + \tilde{s}_1$; $t_{124} = \tilde{s}_0 + \tilde{s}_2$; $t_{125} = \tilde{s}_1 + \tilde{s}_2$; $t_{126} = t_{120} \cdot t_{123}$; $t_{127} = t_{121} \cdot t_{124}$; $t_{128} = t_{122} \cdot t_{125}$; $z_4 = a_1 + \tilde{s}_0 + t_{117} + t_{119} + t_{126}$; $z_5 = a_0 + a_2 + t_{116} + t_{128} + t_{119} + t_{118}$; $z_6 = t_{118} + t_{127} + t_{119} + z_0$; $z_1 = t_{126} + t_{118} + z_0$; $z_2 = a_3 + \tilde{s}_2$;	8 M	7.4 M
7	$v_T(x) = s_3z(x) + v_1(x) + 1 \bmod u_T(x)$, $v_T(x) = v_{T,0} + v_{T,1}x + v_{T,2}x^2 + v_{T,3}x^3 + v_{T,4}x^4 + v_{T,5}x^5$ $t_{129} = u_{T,5} + c_1$; $\langle t_{130}, t_{131}, t_{132}, t_{133}, t_{134}\rangle = t_{129} \cdot (u_{T,0}, u_{T,1}, u_{T,2}, u_{T,3}, u_{T,4})$; $t_{135} = c_0 + t_{130}$; $t_{136} = z_1 + t_{131} + u_{T,0}$; $t_{137} = z_2 + t_{132} + u_{T,1}$; $t_{138} = z_3 + t_{133} + u_{T,2}$; $t_{139} = z_4 + t_{134} + u_{T,3}$; $\langle v_{T,4}, t_{140}, t_{141}, t_{142}, t_{143}\rangle = s_3 \cdot (t_{139}, t_{138}, t_{137}, t_{136}, t_{135})$; $T_{144} = t_{143} + c_0$; $v_{T,1} = t_{142} + c_1$; $v_{T,2} = t_{141} + c_2$; $v_{T,3} = t_{140} + c_3$;	10 M	6.8 M
8	$u_3(x) = Monic\left(\left(f(x) + v_T(x) + v_T(x)^2\right) / u_T(x)\right)$, $u_3(x) = e_0 + e_1x + e_2x^2 + e_3x^3 + x^4$ $t_{145} = (v_{T,3})^2$; $t_{146} = (v_{T,4})^2$; $t_{147} = f_6 + t_{145} \cdot copy(f_7)$; $e_3 = T_{89} + u_{T,5}$; $\langle t_{148}, t_{149}, t_{150}\rangle = T_{89} \cdot (t_{146}, t_{147}, f_7)$; $\langle t_{151}, t_{152}\rangle = e_3 \cdot (u_{T,5}, u_{T,3})$; $e_2 = t_{148} + t_{151} + u_{T,4}$; $t_{153} = e_2 + e_3 \cdot t_{154} = u_{T,4} + u_{T,5}$; $t_{155} = e_2 \cdot u_{T,4}$; $t_{156} = t_{153} \cdot t_{154}$; $e_1 = t_{150} + t_{156} + t_{155} + t_{151} + u_{T,3}$; $t_{157} = e_1 \cdot u_{T,5}$; $e_0 = t_{152} + t_{149} + t_{155} + t_{157} + u_{T,2}$;	8 M + 2 S	7 M + 2 S
9	$v_3(x) = v_T(x) + 1 \bmod u_3(x)$, $v_3(x) = e_0 + e_1x + e_2x^2 + e_3x^3$ $\langle t_{158}, t_{159}\rangle = v_{T,5} \cdot (e_1, e_3)$; $t_{160} = v_{T,4} + t_{159}$; $t_{161} = t_{160} + v_{T,5}$; $t_{162} = e_0 + e_1$; $t_{163} = e_2 + e_3$; $\langle t_{164}, t_{165}\rangle = t_{160} \cdot (e_0, e_2)$; $\langle t_{166}, t_{167}\rangle = t_{161} \cdot (t_{162}, t_{163})$; $e_0 = t_{164} + T_{144}$; $e_1 = t_{166} + t_{158} + t_{164} + v_{T,1}$; $e_2 = t_{158} + t_{165} + v_{T,2}$; $e_3 = t_{167} + t_{159} + v_{T,3}$;	6 M	5.1 M
	Total	11 + 119 M + 10 S	11 + 98.15 M + 10 S

Variable schedule for the genus four addition formula (copies indicated by *)

0	M_{00}	q_1	t_{47}	t_{57}	t_{69}	t_{74}	t_{79}	s_3	t_{155}	t_{161}									
1	M_{02}	q_3	t_{48}	t_{56}	t_{63}	t_{75}	t_{80}	t_{84}	$v_{T,5}$										
2	t_3	M_{03}	t_{52}	t_{70}	t_{76}	t_{86}	T_{89}	t_{156}	t_{163}										
3	t_4	t_6	t_{12}	α_{03}	t_{36}	t_{64}	t_{71}	t_{88}	t_{91}	t_{119}	t_{130}	$v_{T,4}$	t_{160}						
4	t_5	t_7	t_{13}	α_{13}	α^*_{02}	t_{40}	t_{43}	t_{55}	t_{60}	t_{65}	t_{72}	t_{78}	t_{83}	t_{85}	$\tilde{s}_0$	t_{131}	t_{140}	$v_{T,3}$	
5	t_8	M_{33}	t_{14}	α_{23}	t_{38}	t_{44}	t_{61}	t_{66}	t_{73}	t_{81}	$\tilde{s}_1$	t_{126}	t_{132}	t_{141}	$v_{T,2}$				
6	M_{30}	t_{15}	α^*_{12}	t_{32}	t_{42}	t_{45}	t_{62}	t_{67}	$\tilde{s}_2$	t_{133}	t_{142}	$v_{T,1}$							
7	M_{31}	t_{16}	α_{02}	t_{33}	t_{46}	λ_0	T_{90}	t_{116}	t_{134}	t_{143}	T_{144}								
8	M_{32}	t_{17}	t_{18}	α_{12}	α^*_{01}	t_{34}	q_0	λ_1	t_{92}	t_{107}	t_{117}	t_{129}	t_{145}	t_{152}	t_{158}				
9	t_9	t_{19}	α^*_{03}	t_{26}	λ_2	t_{87}	t_{93}	t_{101}	t_{103}	t_{106}	t_{118}	t_{154}	t_{157}	t_{159}					
10	t_{10}	t_{20}	α_{01}	t_{27}	T_{37}	$u_{T,5}$	t_{164}												
11	t_{11}	α^*_{13}	t_{28}	T_{41}	t_{104}	t_{108}	t_{111}	l_0	$u_{T,0}$	t_{151}	t_{165}								
12	M_{01}	l_1	$u_{T,1}$	t_{153}	t_{162}														
13	t_0	M_{11}	T_{35}	$u_{T,2}$															
14	t_1	M_{21}	T_{21}	$u_{T,3}$															
15	t_2	M^*_{31}	T_{22}	$u_{T,4}$	ε_0														
16	M^*_{32}	t_{23}	T_{39}	t_{77}	t_{100}	t_{120}	t_{127}	t_{148}	t_{166}	ε_1									
17	M_{22}	M^*_{33}	t_{24}	q_2	t_{54}	t_{82}	t_{94}	t_{102}	t_{123}	t_{128}	z_6	t_{149}	ε_2						
18	M_{23}	M^*_{30}	t_{25}	r	t_{95}	t_{113}	t_{122}	z_4	t_{139}	t_{150}	t_{167}	ε_3							
19	M_{20}	t_{29}	inv_0	t_{53}	s'_0	t_{96}	t_{109}	t_{121}	z_3	t_{138}	t_{146}	e_0							
20	M_{10}	t_{30}	inv_1	t_{51}	s'_1	t_{97}	t_{105}	t_{110}	t_{114}	t_{124}	z_2	t_{137}	t_{147}	e_1					
21	M_{12}	t_{31}	inv_2	t_{49}	t_{59}	s'_2	t_{98}	t_{125}	z_1	t_{136}	f^*_7	e_2							
22	M_{13}	inv_3	t_{50}	t_{58}	t_{68}	s'_3	t_{99}	t_{112}	t_{115}	z_0	t_{135}	e_3							

Doubling formula $D_3 = [2]D_1$

Inputs:		
$D_1 = [u_1(x), v_1(x)]$, $u_1(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + x^4$ and $v_1(x) = c_0 + c_1x + c_2x^2 + c_3x^3$ $C : y^2 + h(x)y = f(x)$ with $h(x) = h_0 \in \mathbb{F}_q$ and $f(x) = f_0 + f_1x + f_2x^2 + f_3x^3 + f_4x^4 + f_5x^5 + f_6x^6 + f_7x^7 + x^9$		
Outputs:		
$D_3 = [u_3(x), v_3(x)]$, $u_3(x) = e_0 + e_1x + e_2x^2 + e_3x^3 + x^4$ and $v_3(x) = \varepsilon_0 + \varepsilon_1x + \varepsilon_2x^2 + \varepsilon_3x^3$		
Step	Operations	Cost	Effective
1	$u_C(x) = u_1(x)^2$, $u_C(x) = u_{C,0} + u_{C,2}x^2 + u_{C,4}x^4 + u_{C,6}x^6 + x^8$ $u_{C,0} = (a_0)^2$; $u_{C,2} = (a_1)^2$; $u_{C,4} = (a_2)^2$; $u_{C,6} = (a_3)^2$;	4 S	4 S
2	$v_C(x) = v_1(x)^2 + f(x) \bmod u_C(x)$, $v_C(x) = v_{C,0} + v_{C,1}x + v_{C,2}x^2 + v_{C,3}x^3 + v_{C,4}x^4 + v_{C,5}x^5 + v_{C,6}x^6 + v_{C,7}x^7$ $t_0 = (c_0)^2$; $t_1 = (c_1)^2$; $t_2 = (c_2)^2$; $t_3 = (c_3)^2$; $v_{C,0} = f_0 + t_0$; $v_{C,1} = f_1 + u_{C,0}$; $v_{C,2} = f_2 + t_1$; $v_{C,3} = f_3 + u_{C,2}$; $v_{C,4} = f_4 + t_2$; $v_{C,5} = f_5 + u_{C,4}$; $v_{C,6} = f_6 + t_3$; $v_{C,7} = f_7 + u_{C,6}$;	4 S	4 S
3	computation of inverses $t_4 = v_{C,7} \cdot v_{C,5}$; $T_5 = (v_{C,6})^2$; $t_6 = (v_{C,7})^2$; $t_7 = t_6 \cdot u_{C,6}$; $t_8 = T_5 + t_4 + t_7$; $t_9 = v_{C,7} \cdot t_8$; If t_9 is 0 use Cantor's algorithm $t_{10} = 1/t_9$; $(t_{11}, t_{12}) = t_{10} \cdot (t_6, t_8)$; $e_3 = (t_{11})^2$; $u_{T,1} = (t_{12})^2$;	1 I + 5 M + 4 S	1 I + 4.7 M + 4 S
4	$u_T(x) = Monic\left(\frac{f(x) + v_C(x) + v_C(x)^2}{u_C(x)}\right)$, $u_T(x) = u_{T,0} + u_{T,1}x + u_{T,2}x^2 + u_{T,4}x^4 + x^6$ $t_{13} = (v_{C,5})^2$; $t_{14} = (v_{C,4})^2$; $(t_{15}, t_{16}, t_{17}) = u_{T,1} \cdot (T_5, t_{13}, t_{14})$; $u_{T,4} = t_{15} + u_{C,6}$; $(t_{18}, t_{19}) = u_{T,4} \cdot (u_{C,6}, u_{C,4})$; $u_{T,2} = t_{18} + t_{16} + u_{C,4}$; $t_{20} = u_{C,6} \cdot u_{T,2}$; $u_{T,0} = t_{19} + t_{17} + u_{C,2} + t_{20}$;	6 M + 2 S	5 M + 2 S
5	$v_T(x) = v_C(x) + 1 \bmod u_T(x)$, $v_T(x) = v_{T,0} + v_{T,1}x + v_{T,2}x^2 + v_{T,3}x^3 + v_{T,4}x^4 + v_{T,5}x^5$ $(t_{21}, t_{22}, t_{23}) = v_{C,6} \cdot (u_{T,0}, u_{T,4}, u_{T,1})$; $(t_{24}, t_{25}, t_{26}) = v_{C,7} \cdot (u_{T,2}, u_{T,0}, u_{T,4})$; $t_{27} = v_{C,6} + v_{C,7}$; $t_{28} = u_{T,1} + u_{T,2}$; $t_{29} = t_{27} \cdot t_{28}$; $T_{30} = v_{C,0} + t_{21}$; $v_{T,1} = v_{C,1} + t_{23} + t_{25}$; $v_{T,2} = v_{C,2} + t_{29} + t_{23} + t_{24}$; $v_{T,3} = v_{C,3} + t_{24}$; $v_{T,4} = v_{C,4} + t_{22}$; $v_{T,5} = v_{C,5} + t_{26}$;	7 M	5.6 M
6	$u_3(x) = Monic\left(\frac{f(x) + v_T(x) + v_T(x)^2}{u_T(x)}\right)$, $u_3(x) = e_0 + e_1x + e_2x^2 + e_3x^3 + x^4$ $t_{31} = (v_{T,3})^2$; $t_{32} = (v_{T,4})^2$; $t_{33} = f_7 + u_{T,4}$; $t_{34} = f_6 + t_{31}$; $(t_{35}, e_1, t_{36}) = e_3 \cdot (t_{34}, t_{33}, t_{32})$; $e_2 = t_{36} + u_{T,4}$; $t_{37} = e_2 \cdot u_{T,4}$; $e_0 = t_{35} + u_{T,2} + t_{37}$;	4 M + 2 S	3.3 M + 2 S
7	$v_3(x) = v_T(x) + 1 \bmod u_3(x)$, $v_3(x) = \varepsilon_0 + \varepsilon_1x + \varepsilon_2x^2 + \varepsilon_3x^3$ $(t_{38}, t_{39}) = v_{T,5} \cdot (e_1, e_3)$; $t_{40} = v_{T,4} + t_{39}$; $t_{41} = t_{40} + v_{T,5}$; $t_{42} = e_0 + e_1$; $t_{43} = e_2 + e_3$; $(t_{44}, t_{45}) = t_{40} \cdot (e_0, e_2)$; $(t_{46}, t_{47}) = t_{41} \cdot (t_{42}, t_{43})$; $\varepsilon_0 = T_{30} + t_{44}$; $\varepsilon_1 = v_{T,1} + t_{46} + t_{38} + t_{44}$; $\varepsilon_2 = v_{T,2} + t_{38} + t_{45}$; $\varepsilon_3 = v_{T,3} + t_{47} + t_{39} + t_{45}$;	6 M	5.1 M
	Total	1 I + 28 M + 16 S	1 I + 23.7 M + 16 S

Variable schedule for the genus 4 doubling formula

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
$u_{C,2}$	$u_{C,4}$	$u_{C,6}$	t_0	t_1	t_2	t_3	$v_{C,0}$	$v_{C,1}$	$v_{C,2}$	$v_{C,3}$	$v_{C,4}$	$v_{C,5}$	$v_{C,6}$	$v_{C,7}$	$u_{C,0}$	t_{12}	t_{17}	e_3
t_{24}	t_{25}	t_{26}	T_5	t_6	t_7	t_8	T_{30}	$v_{T,1}$	$v_{T,2}$	$v_{T,3}$	$v_{T,4}$	$v_{T,5}$	t_{27}	t_{39}	t_4	t_{16}	t_{23}	
t_{34}	t_{33}	t_{32}	t_{18}	t_{13}	t_{10}	$u_{T,1}$	ε_0	ε_1	ε_2	ε_3	t_{40}		t_{38}		t_9	t_{22}	t_{36}	
t_{41}	t_{42}	t_{37}	t_{43}	t_{19}	t_{14}	t_{29}									t_{11}	e_1	e_2	
			t_{44}	$u_{T,0}$	$u_{T,4}$	t_{47}									t_{15}			
				t_{28}	t_{46}										t_{20}			
				t_{31}											t_{21}			
				t_{45}											t_{35}			
															e_0			