Aspects of Hyperelliptic Curves over Large Prime Fields in Software Implementations

Roberto Maria Avanzi
Institute for Experimental Mathematics (IEM)
Ellernstrasse 29, D-45326 Essen, Germany
mocenigo@exp-math.uni-essen.de *

December 13, 2003

Abstract

This paper presents an implementation of genus 2 and 3 hyperelliptic curves over prime fields, with a comparison with elliptic curves. To achieve a fair comparison, we developed an ad-hoc arithmetic library, designed to remove most of the overheads that penalise implementations of curve-based cryptography over prime fields. These overheads get worse for smaller fields, and thus for large genera. We also use techniques such as "lazy" and "incomplete" modular reduction, originally developed for performing arithmetic in field extensions, to reduce the number of modular reductions occurring in the formulae for the group operations.

The result is that the performance of hyperelliptic curves of genus 2 over prime fields is much closer to the performance of elliptic curves than previously thought. For groups of 192 and 256 bits the difference is about 18% and 15% respectively.

Introduction

Background

In 1988 Koblitz [26] proposed to use the Jacobian varieties of hyperelliptic curves (HEC) as an alternative to elliptic curves (EC) for constructing cryptographic systems (short: cryptosystems) based on the *discrete logarithm problem* (DLP): *Given a cyclic group G of order n, a generator D of G, and $E \in G$, determine an integer k such that $E = kD = D + D + \cdots + D$ (k summands).*

For the same security, elliptic curve cryptosystems (ECC) require a much shorter key than RSA or systems based on the DLP in finite fields. A 160-bit ECC key is considered to offer security

*The work described in this paper has been supported by the Commission of the European Communities through the IST Programme under Contract IST-2001-32613 (see http://www.arehcc.com).

equivalent to that of a 1024 bit RSA key [37]. This is due to the following facts: (i) The best known algorithms for solving the DLP in EC have complexity exponential in the logarithm of the group order [51, 48] – more precisely this complexity is $O(\sqrt{n})$; (ii) the best methods for solving the DLP in finite fields [10, 19, 1] or for factoring integers [35, 36] are sub-exponential. Since solving the DLP on HEC of genus smaller than 4 has complexity $O(\sqrt{n})$, these curves offer the same security level as EC for the same key size. Curves of high genus are insecure [2, 6], and curves of smaller genus, but greater than 3, are also weaker than EC [13, 62].

A vast amount of research has been devoted to cryptographic applications of EC, but the cryptographic potential of HEC has not been investigated as thoroughly. In 1999, Smart [59] concluded that HEC seemed not practical, because of the greater difficulty of finding suitable curves and their poor performance with respect to EC.

In the subsequent years the landscape changed significantly.

Firstly, it is now possible to efficiently construct genus 2 and 3 HEC whose divisor class group has almost prime order of cryptographic relevance. For curves over prime fields, a genus 2 analogue of Schoof's point counting algorithm can be used: The first version [14] was too slow, but the improvements of [44] and further work by Gaudry and Schost made it possible to count points on large enough curves [15].

In [16], the method is described and cryptographically suitable examples are given. Another technique is the complex multiplication method: The genus 2 case is handled by Mestre in [42], improvements and a partial extension to genus 3 can be found in [65].

For small characteristic, Satoh [55] proposed a fast point counting algorithm for elliptic curves, later extended to higher genus and improved by many, including Satoh, Skjernaa and Taguchi [56], Vercauteren [64, 63], Gaudry, Harley and Fouquet [12], Mestre [43], Kedlaya [25], Lauder and Wan [34], and Gerkmann [17].

Secondly, the performance of the HEC group operations has been considerably improved. The first explicit formulae for genus 2 [21, 45, 61] have been followed by the extensive work of Lange [30, 31, 32, 33]. For genus 3, there are formulae by Pelzl [49] (see also [50]), improving on [29].

HEC are attractive to designers of embedded hardware since they require smaller fields than EC to attain the same security level. The order of the Jacobian of a HEC of genus g over a field with q elements is $\approx q^g$. This means that a 160-bit group is given by an EC with $q \approx 2^{160}$, by an HEC of genus 2 with $q \approx 2^{80}$, and genus 3 with $q \approx 2^{53}$.

Recently, there has been also research on securing implementations of HEC-based cryptosystems on embedded devices against differential power analysis [3].

Results

Because of the security considerations above, we made a thorough, fair and unbiased comparison of the relative performance merits of *generic* EC and HEC of small genus 2 or 3 over *prime fields*. We consider the different coordinate systems available for those curves. We are *not* interested in comparing against *very special* classes of curves or in the use of prime moduli of special form (such as $p = 2^{192} - 2^{64} - 1$, for example).

There have been several software implementations of HEC on personal computers and workstations. Most of those are in even characteristic [54, 53, 29, 49, 50, 66], some are over prime fields [28, 53, 30], and a few over optimal extension fields (OEF) [45, 40]. It is now known that in *even characteristic*, HEC can offer performance comparable to EC. Until now there have been no concrete results showing the same for prime fields.

Traditional implementations such as [28, 30] are based on general purpose software libraries like gmp [20], NTL [58], or similarly designed packages. They all introduce fixed overheads for every procedure call and loop, which are usually negligible for very large operands, but become the dominant part of the computations for small operands such as those occurring in curve cryptography. The smaller the field becomes, the higher the time wasted in the overheads will be, and HEC implementations usually suffer from a much bigger performance hit than EC. Furthermore, gmp has no native support for fast modular reduction techniques such as Montgomery's [46].

In our modular arithmetic library nuMONGO [4] we made every effort to avoid such overheads. A description is given in Subsection 2.1. The current version is designed mainly for 32-bit CPUs and is fairly portable. Our implementation has been tested on a PC with a 1 Ghz AMD Athlon Model 4 processor. We get a boost of a factor 2 to 5 over gmp for operations in fields of cryptographic relevance (see Table 2 and also Tables 7 and 8). The larger speed-up is achieved in the smaller fields, such as those used for HEC. We also exploit two techniques, called *lazy* and *incomplete modular reduction* (see [5]), to reduce the number of modular reductions occurring in the formulae for the group operations.

> *We thus show that the performance of genus 2 HEC over prime fields is much closer to the performance of EC than previously thought. For groups of 192, resp. 256 bits the difference is approximately 18%, resp. 15%. The gap with genus 3 curves has been reduced too. For very large groups the performances of genus 2 and genus 3 curves get close. More precise results are stated in Section 3.*

While the only significant constraint in workstations and commodity PCs may be processing power, the results of our work should also be applicable to other more constrained environments, such as Palm platforms, which are also based on general-purpose processors.

The structure of the paper is the following. In the next section we review the arithmetic on EC and HEC and all coordinate systems currently available for generic curves. The implementation is documented in Section 2. We conclude with experimental results in Section 3, including timings for EC and HEC both with gmp and with our library.

Acknowledgements. The author wishes to express his gratitude to Gerhard Frey, who introduced him to the world of cryptographic applications of arithmetic geometry. Tanja Lange left many marks on this paper, directly because of proofreading, and indirectly because of fruitful discussions on the subject matter. She also implemented the elliptic curve group operations used in the comparisons based on nuMONGO "for fun" (sic) following the example of our hyperelliptic curve implementation. The author acknowledges the great feedback of Christophe Doche, Sylvain Duquesne, Kim Nguyen, Christoph Paar, Jan Pelzl, Nicolas Thériault and Thomas Wollinger.

1 Arithmetic

We use the following abbreviations: w is the bit length of the characteristic of the prime field. M, S and I denote a multiplication, a squaring and an inversion respectively. m denotes a multiplication of two w bit integers with a $2w$ bit result. R denotes a modular (or Montgomery) reduction of a $2w$ bit integer with a w bit result.

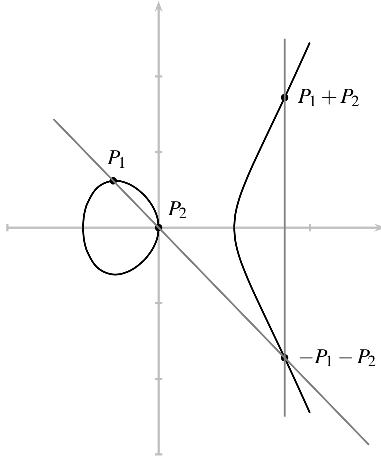
1.1 Elliptic Curves

In this subsection we follow [9]. An elliptic curve E defined over a field F of characteristic zero or greater than 3 can be given by an equation in Weierstrass form

$$E \;:\; Y^2Z = X^3 + a_4XZ^2 + a_6Z^3 \tag{1}$$

with $4a_4^2 + 27a_6^2 \neq 0$. The last condition is equivalent to requiring that the polynomial $x^3 + a_4x + a_6$ has no multiple roots. The set of points of an elliptic curve over (any extension of) the field F forms a group. The triples $(X : Y : Z)$ satisfying (1) represent the points in projective space. We can normalize the points by dividing through the Z-coordinate $(X : Y : Z) \mapsto (x, y) := (X/Z, Y/Z)$, and by introducing the special symbol O for the point $(0 : 1 : 0)$. The group law on these affine points is depicted in Figure 1. This rule is easily transformed into an explicit formula which manipulates the coordinates. To double a point P, we consider the tangent line to the curve at P instead of the secant.

Figure 1: Addition on EC



Affine coordinates ($\mathcal{A}$). Let $P_1 = (x_1, y_1)$, $P_2 = (x_2, y_2)$ and $P_3 = (x_3, y_3)$. Then

$$x_3 = \lambda^2 - x_1 - x_2,\; y_3 = \lambda(x_1 - x_3) - y_1$$

where $\lambda = (y_1 - y_2)/(x_1 - x_2)$. For doubling set $\lambda = (3x_1^2 + a_4)/(2y_1)$. Thus an addition and a doubling require respectively $\mathtt{I} + 2\mathtt{M} + \mathtt{S}$ and $\mathtt{I} + 2\mathtt{M} + 2\mathtt{S}$.

Projective coordinates ($\mathcal{P}$). The coordinates are $(X : Y : Z)$ and the equation is (1). Let $P_1 = (X_1 : Y_1 : Z_1)$, $P_2 = (X_2 : Y_2 : Z_2)$ and $P_3 = (X_3 : Y_3 : Z_3)$. The addition is

$$X_3 = vA,\; Y_3 = u(v^2X_1Z_2 - A) - v^3Y_1Z_2,\; Z_3 = v^3Z_1Z_2 \quad ,$$

where $u = Y_2Z_1 - Y_1Z_2$, $v = X_2Z_1 - X_1Z_2$, and $A = u^2Z_1Z_2 - v^3 - 2v^2X_1Z_2$. The doubling is

$$X_3 = 2hs,\; Y_3 = w(4B - h) - 8Y_1^2s^2,\; Z_3 = 8s^3 \quad ,$$

where $w = a_4Z_1^2 + 3X_1^2$, $s = Y_1Z_1$, $B = X_1Y_1s$ and, $h = w^2 - 8B$. No inversions are needed and the operation counts are $12\mathtt{M} + 2\mathtt{S}$ and $7\mathtt{M} + 5\mathtt{S}$ respectively.

Jacobian and Chudnovsky Jacobian coordinates ($\mathcal{J}$ and $\mathcal{J}^c$). In *Jacobian coordinates* the equation of E is

$$E: Y^2 = X^3 + a_4XZ^4 + a_6Z^6 \quad ,$$

where $x = X/Z^2$ and $y = Y/Z^3$. Let (X_i, Y_i, Z_i) be the coordinates of the point P_i. The addition $P_3 = P_1 + P_2$ is given by

$$X_3 = -H^3 - 2U_1H^2 + r^2,\ Y_3 = -S_1H^3 + r(U_1H^2 - X_3),\ Z_3 = Z_1Z_2H \quad ,$$

where $U_1 = X_1Z_2^2$, $U_2 = X_2Z_1^2$, $S_1 = Y_1Z_2^3$, $S_2 = Y_2Z_1^3$, $H = U_2 - U_1$ and $r = S_2 - S_1$. The doubling is given by

$$X_3 = T,\ Y_3 = -8Y_1^4 + M(S - T),\ Z_3 = 2Y_1Z_1 \quad ,$$

where $S = 4X_1Y_1^2$, $M = 3X_1^2 + a_4Z_1^4$ and $T = -2S + M^2$. The operation counts are $12\mathtt{M} + 4\mathtt{S}$ and $4\mathtt{M} + 6\mathtt{S}$ respectively.

For a point P_i, the quintuple $(X_i, Y_i, Z_i, Z_i^2, Z_i^3)$ are the *Chudnovsky (Jacobian) coordinates*. The same formulae as for $\mathcal{J}$ are used, and we do not have to compute Z_1^2, Z_2^2, Z_1^3 and Z_2^3, but we must compute Z_3^2 and Z_3^3. The operation counts become $11\mathtt{M} + 3\mathtt{S}$ and $5\mathtt{M} + 6\mathtt{S}$.

Modified Jacobian coordinates ($\mathcal{J}^m$). This set of coordinates was introduced by Cohen et al. [9]. It is based on $\mathcal{J}$ but the internal representation of a point P is the quadruple (X, Y, Z, a_4Z^4). The formulae are almost the same as for $\mathcal{J}$, the main difference being the introduction of $U = 8{Y_1}^4$ so that $Y_3 = M(S - T) - U$ and $a_4Z_3^4 = 2U(a_4Z_1^4)$. An addition takes $13\mathtt{M} + 6\mathtt{S}$ and a doubling $4\mathtt{M} + 4\mathtt{S}$. Since $\mathtt{I}$ takes on average between 9 and $40\mathtt{M}$ and $\mathtt{S}$ is about $0.5\mathtt{M}$ to $0.8\mathtt{M}$, this system provides the fastest doubling in practice.

Mixed Coordinate Systems. Different coordinate systems can be used together to perform scalar multiplications. It is always advantageous to keep the base point and all precomputed points in $\mathcal{A}$, since additions by those points will be less expensive. For the doublings one should choose the coordinate system which has the fastest doubling. More refined strategies are possible: see [9], where detailed operation counts for group operations with mixed coordinates are given. The same approach can be followed for genus 2 HEC (see Subsubsection 2.3.1 and Subsection 2.4).

1.2 Hyperelliptic Jacobians

An excellent, low brow, introduction to hyperelliptic curves is given in [41], including proofs of the facts used below. Our notation is slightly different, but conform to that of [30, 31, 32, 33, 50].

1.2.1 Equation and Divisor Representation

A hyperelliptic curve $\mathcal{C}$ of genus g over a finite field $\mathbb{F}_q$ of odd characteristic is defined by a *Weierstrass equation*

$$\mathcal{C}\ :\ y^2 = f(x) \quad , \tag{2}$$

where f is a monic square-free polynomial of degree $2g + 1$ in x. Let ∞ be the point at infinity on the curve. In general, the points on a hyperelliptic curve do *not* form a group. Instead, the

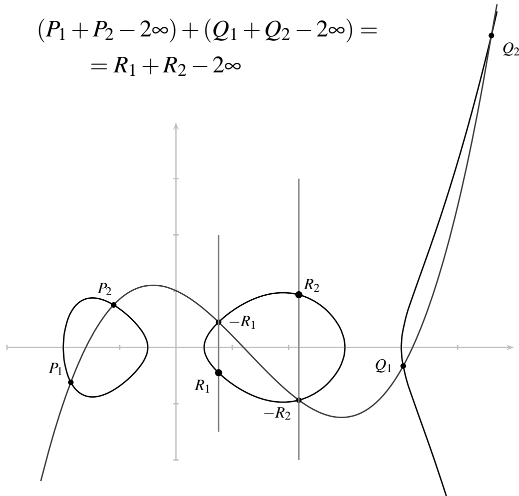
divisor class group of $\mathcal{C}$ is used: We briefly recall its main properties. The divisor class group is isomorphic to the variety called the *Jacobian* of $\mathcal{C}$, which we do not define nor study here.

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 $\mathbb{Z}[\mathcal{C}(\overline{\mathbb{F}_q})]$. 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

$$\sum_{i=1}^{m} P_i - m\infty \quad : \quad P_i \in \mathcal{C} \setminus \{\infty\} \ . \tag{3}$$

The *degree* of the associated effective divisor 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 (3) with (effective) degree $m \leq g$.

Figure 2: Addition on genus 2 HEC



To add two classes c_1, c_2 we formally add the representing divisors D_1, D_2. In general, this sum has degree $> g$, in which case it must be *reduced* to the equivalent divisor of effective degree $\leq g$. To do this, we consider a plane curve passing through the finite points of D_1 and D_2, counted with their multiplicities – i. e. we determine a function f on the curve vanishing on the points on $D_1 + D_2$, with multiplicities taken into account, the only pole being at ∞. Put D_3 the divisor whose (finite) support consists of the other points of intersection of the curve (with multiplicities). The class c_3 of D_3 satisfies $c_1 + c_2 + c_3 = 0$. It suffices to replace each point in D_3 with its opposite (i.e. invert the y–coordinates) to get a divisor equivalent to $D_1 + D_2$, but with smaller degree. We might have to repeat this step more than once to get the reduced element. This addition is depicted in Figure 2 for a genus 2 curve "geometrically".

Mumford [47] introduced a representation of the elements of the divisor class group as polynomial pairs, for which Cantor [7] provided an explicit arithmetic algorithm. Let $D = \sum_{i=1}^{m} P_i - m\infty$ be a divisor with $m \leq g$. The ideal class associated to D is represented by a unique pair of polynomials $U(x), V(x) \in \mathbb{F}_q[x]$ with $g \geq \deg U > \deg V$, U monic and such that: $U(x) = \prod_{i=1}^{m}(x - x_{P_i})$ (i.e. the roots of $U(x)$ are the x–coordinates of the points belonging to the divisor); $V(x_{P_i}) = y_{P_i}$ for all $1 \leq i \leq m$ (i.e. the polynomial $V(t)$ interpolates those points); and $U(x)$ divides $V(x)^2 - f(x)$. We say that the pair $[U(x), V(x)]$ represents the *reduced divisor D*.

1.2.2 Composition Algorithms

We consider the group operations on curves of arbitrary genus here. For explanations and proofs we refer the interested reader to [7]. The operation is split into two algorithms.

The first algorithm *composes* two divisors, and the result is in *semi-reduced* form: the output is a divisor $D = [U,V]$ where the condition $g \geq \deg_t U$ is not necessarily satisfied, but $\deg_t U > \deg_t V$. This corresponds to the addition two divisors in the geometric way.

Algorithm 1: ***Divisor Composition***

`Input:` $D_1 = [U_1, V_1], D_2 = [U_2, V_2]$,
`Output:` $D = [u, v]$ semi-reduced with $D \equiv D_1 + D_2$.

1. `compute` d_1, e_1, e_2 : $d_1 = \gcd(U_1, U_2) = e_1 U_1 + e_2 U_2$;
2. `compute` d, c_1, c_2 : $d = \gcd(d_1, V_1 + V_2) = c_1 d_1 + c_2 (V_1 + V_2)$;
3. $s_1 \leftarrow c_1 e_1,\ s_2 \leftarrow c_1 e_2,\ s_3 \leftarrow c_2$; `[i.e.` $d = s_1 U_1 + s_2 U_2 + s_3 (V_1 + V_2)$`]`
4. $u \leftarrow (U_1 U_2)/(d^2)$;
$v \leftarrow (s_1 U_1 V_2 + s_2 U_2 V_1 + s_3 (V_1 V_2 + f))/d \bmod U$.

The second algorithm computes the Mumford representation of the unique representant of effective degree at most g in the divisor class of the input.

Algorithm 2: ***Divisor Reduction***

`Input:` $D = [U,V]$ semi-reduced.
`Output:` $D' = [U', V']$ reduced with $D \equiv D'$.

1. $U' \leftarrow (f - V^2)/U,\ V' \leftarrow -V \bmod U'$;
2. `if` $\deg U' > g$ `then put` $U \leftarrow U', V \leftarrow V'$; `goto` **1**;
3. `make` U' `monic.`

1.2.3 Coordinate Systems

1.2.3.1 The general case

Let a reduced divisor D of a genus g curve be represented by two polynomials $U(x), V(x) \in \mathbb{F}_q[x]$ with $g \geq \deg U > \deg V$, and U monic. In the most common case, write $U(x) = x^g + \sum_{i=0}^{g-1} U_i x^i$ and $V(x) = \sum_{i=0}^{g-1} V_i x^i$. The *affine coordinates* of D are the $2g$-tuple $[U_{g-1}, \ldots, U_1, U_0, V_{g-1}, \ldots, V_1, V_0]$. For genus 3 this is the only coordinate system available.

1.2.3.2 Genus 2

For genus 2 there are two more coordinate systems besides affine ($\mathcal{A}$).

Miyamoto, Doi, Matsuo, Chao, and Tsuji [45] introduced projective coordinates ($\mathcal{P}$): a quintuple $[U_1,U_0,V_1,V_0,Z]$ corresponds to the divisor class represented by $[t^2+U_1/Zt+U_0/Z, V_1/Zt+V_0/Z]$. Lange improved the operation counts of [45] and extended the work to even characteristic. To convert to the system $\mathcal{A}$ we need $\mathtt{I}+4\mathtt{M}$.

Lange's *new* coordinates [32] ($\mathcal{N}$) are a weighted system, in the spirit of elliptic Jacobian coordinates: The sextuple $[U_1,U_0,V_1,V_0,Z_1,Z_2]$ corresponds to the divisor class $[t^2+U_1/Z_1^2t+U_0/Z_1^2, V_1/Z_1^3Z_2t+V_0/Z_1^3Z_2]$. The system $\mathcal{N}$ provides the fastest doubling, and is therefore important in scalar multiplications, where the doublings are the most common operation.

The operation counts for the different operations are given in Table 1, courtesy of Lange. We do not list the group operations here. Full details are given in [33].

Table 1: Addition and Doubling in Different Systems, genus 2

Doubling		**Addition**			
operation	costs	operation	costs	operation	costs
$2\mathcal{N}=\mathcal{P}$	7S, 38M	$\mathcal{N}+\mathcal{N}=\mathcal{P}$	7S, 51M	$\mathcal{A}+\mathcal{N}=\mathcal{P}$	5S, 41M
$2\mathcal{P}=\mathcal{P}$	6S, 38M	$\mathcal{N}+\mathcal{P}=\mathcal{P}$	4S, 51M	$\mathcal{A}+\mathcal{P}=\mathcal{P}$	3S, 40M
$2\mathcal{N}=\mathcal{N}$	7S, 34M	$\mathcal{N}+\mathcal{N}=\mathcal{N}$	7S, 47M	$\mathcal{A}+\mathcal{N}=\mathcal{N}$	5S, 36M
$2\mathcal{P}=\mathcal{N}$	6S, 34M	$\mathcal{N}+\mathcal{P}=\mathcal{N}$	5S, 45M	$\mathcal{A}+\mathcal{P}=\mathcal{N}$	3S, 35M
$2\mathcal{A}=\mathcal{P}$	5S, 25M	$\mathcal{P}+\mathcal{P}=\mathcal{P}$	5S, 45M	$\mathcal{A}+\mathcal{A}=\mathcal{N}$	3S, 25M
$2\mathcal{A}=\mathcal{N}$	5S, 21M	$\mathcal{P}+\mathcal{P}=\mathcal{N}$	5S, 41M		
$2\mathcal{A}=\mathcal{A}$	1I, 5S, 22M	$\mathcal{A}+\mathcal{A}=\mathcal{A}$	1I, 3S, 22M		

2 Implementation

2.1 Prime Field Library

We already mentioned in the introduction that standard software libraries for performing arithmetic computations introduce several types of overheads. One is the fixed function call overhead. Other ones come from the fact they process operands of variable length in loops: They are usually negligible for very large operands, since the conditional branches in the loops in the code are most of the time taken for free. In fact, the branch prediction unit of the CPU will almost always guess the right branch, and the only misprediction shall occur when exiting the loop, i.e. the only time the branch is not taken. For operands of size relevant for curve cryptography the CPU will spend more time performing jumps and paying big penalties because of branch mispredictions than doing arithmetic. Thus, The smaller the field becomes, the higher will be the time wasted in the overheads. Because of the larger number of field operations in smaller fields, HEC suffer from a much larger performance loss than EC.

Our software library nuMONGO was designed to allow efficient reference implementations of EC and HEC over prime fields. It actually implements arithmetic operations in rings $\mathbb{Z}/N\mathbb{Z}$ with N odd, where the elements are stored in Montgomery's representation [46], and the reduction algorithm is Montgomery's REDC. It is written in the programming language C++ in order to take advantage of inline functions, operator and function overloading, but it contains no classes. All data structures are as simple and spartan as possible. The routines aim at the atomicity of assembler operations, and in this sense this is a *RISC-like* software library: The least possible number of routines are implemented which still allow to perform all desired field operations, and are heavily optimized.

All operations are built from elementary operations working on single words. These elementary operations are available as C macros and as assembler macros for Intel and AMD x86 processors (support for more CPUs is planned). They assume a CPU able to work on 32-bit operands and not a 32-bit CPU – the library in fact worked also on an Alpha. As mentioned in the introduction, we wanted to reduce as much as possible the overhead associated with each function call. Therefore, inlining was used extensively to build all multi-precision routines from the single word operations mentioned above. There are almost no conditional branches, so the CPU will seldom make very expensive branch mispredictions.

There are separate addition, subtraction, multiplication and modular inversion routines for all operand sizes, in steps of 32 bits from 32 to 256 bits, as well as for 48–bit fields (80 and 112-bit fields have been implemented too, but gave no speed advantage with respect to the 96 and 128-bit routines). All elements of $\mathbb{Z}/N\mathbb{Z}$ are stored in vectors of the same length of 32-bit words.

With the exception of 32-bit operands, inversion is based on the extended binary GCD, and uses an almost-inverse like algorithm [24, 57] with final multiplication from a table of precomputed, reduced, powers of 2. This is usually the fastest approach up to about 192 bits. For 32-bit operands better performance is attained with an implementation of the extended Euclidean algorithm with separate consideration of small quotients. Inversion was not sped up further for larger input sizes because of the intended usage of the library. In the case of elliptic curves, inversion-free coordinate systems are much faster than affine coordinates, so there is need, basically, only for one inversion at the end of a scalar multiplication. In the case of hyperelliptic curves, fields are quite small (32 to 128 bits in most cases) in which case our inversion routines have optimal performance anyway. Hence, Lehmer's method or the recent improvements by Jebelean [22, 23] or Lercier [38] have not been included.

In Table 2 on the following page we show some timings of basic operations with gmp version 4.1 and nuMONGO. The timings have been measured on a PC with a 1 Ghz AMD Athlon Model 4 processor, as all other timings in this paper, under the Linux operating system (kernel version 2.4.18). Our programs have been compiled with the GNU Compiler Collection (gcc) version 2.95.3. We now describe the meaning of the table entries.

There are three groups of five rows, grouped account to the library used to benchmark the following operations: multiplication of two integers (Mul), modular or Montgomery reduction (Rec or REDC), modular or Montgomery inversion (Inv). Also the ratios of a reduction to a multiplication ($\frac{\texttt{REDC}}{\texttt{Mul}}$ and $\frac{\texttt{Red}}{\texttt{Mul}}$) and of an inversion to the time of a multiplication together with a reduction ($\frac{\texttt{Inv}}{\texttt{Mul+REDC}}$ and $\frac{\texttt{Inv}}{\texttt{Mul+Red}}$) are given: The first ratio tells how many "multiplications" we save each time we save a reduction using the techniques described in the next subsection; the second ratio is the cost of a

Table 2: Timings of basic operations in μsec (1 Ghz AMD Athlon PC) and ratios

Lib / Op / Bits		32	48	64	96	128	160	192	224	256
nuMONGO	Mul	0.0071	0.0223	0.0386	0.054	0.098	0.153	0.23	0.302	0.409
	REDC	0.0267	0.0573	0.0618	0.092	0.153	0.214	0.306	0.382	0.489
	Inv	0.63	1.75	1.95	4.89	8.2	12.3	17.8	25.1	31.8
	$\frac{\text{REDC}}{\text{Mul}}$	3.761	2.569	1.601	1.704	1.561	1.399	1.33	1.265	1.196
	$\frac{\text{Inv}}{\text{Mul+REDC}}$	18.64	21.98	19.42	33.56	32.67	33.51	33.21	36.69	35.77
gmp v. 4.1 (stock)	Mul	0.094	0.155	0.16	0.206	0.238	0.308	0.354	0.44	0.508
	Red	0.234	0.419	0.423	0.65	0.81	0.986	1.154	1.264	1.528
	Inv	2.53	4.74	6.41	9.77	13.3	17.2	21.26	25.84	29.6
	$\frac{\text{Red}}{\text{Mul}}$	2.489	2.703	2.644	3.155	3.403	3.201	3.26	2.873	3.008
	$\frac{\text{Inv}}{\text{Mul+Red}}$	7.713	8.258	10.99	11.41	12.69	13.29	14.1	15.16	14.54
gmp *simplified*	Mul	*0.106*	*0.287*	*0.315*	*0.501*	*0.728*	*1.02*	*1.404*	*1.832*	*2.268*
	Red	*0.247*	*0.411*	*0.46*	*0.682*	*0.934*	*1.266*	*1.742*	*2.196*	*2.664*
	Inv	*3.15*	*5.65*	*7.57*	*12.3*	*17.72*	*23.12*	*29.08*	*36.2*	*42.64*
	$\frac{\text{Red}}{\text{Mul}}$	*2.33*	*1.432*	*1.46*	*1.361*	*1.283*	*1.241*	*1.241*	*1.199*	*1.175*
	$\frac{\text{Inv}}{\text{Mul+Red}}$	*8.923*	*8.129*	*9.768*	*10.39*	*10.66*	*10.11*	*9.243*	*8.987*	*8.646*

field inversion in multiplications. The columns correspond to the bit lengths of the operands.

Note that gmp is very heavily optimized in assembler for the considered architecture, whereas the inner loops of nuMONGO are far less optimized. The timings corresponding to this stock, highly optimized version of gmp are given in the group labeled gmp version 4.1 (stock). It is possible to recompile gmp using only C with some assembler macros, i.e. the same type of approach used for nuMONGO, thus losing performance: This fact has been observed also in [5] in a slightly different contexts. The corresponding timings are given in the group labeled gmp *simplified*.

A few remarks:

(1) nuMONGO can perform better than a far more optimized, but general purpose library.

(2) The disparity with a general purpose library compiled with the same optimization level as nuMONGO is considerable.

(3) For larger operands gmp catches up with nuMONGO, the modular reduction remaining slower because it is not based on Montgomery's algorithm.

(4) nuMONGO has the highest ratio of a field inversion to a field multiplication. This shows how big the overheads in general purpose libraries are for such small inputs. In particular, such ratios are quite close to those in hardware implementation of field arithmetic.

2.2 Lazy and Incomplete reduction

Lazy and incomplete modular reduction are described in [5]. Here, we give a short treatment. Let p be a prime number smaller than 2^w, where w is a fixed integer. We consider the evaluation of expressions of the form $\sum_{i=0}^{d-1} a_i b_i \mod p$ given a_i and b_i with $0 \leq a_i, b_i < p$. Such expressions occur in polynomial multiplication, hence in the explicit formulae for HEC.

To use most modular reduction algorithms or Montgomery's reduction procedure [46] at the *end* of the summation, we have to make sure that all partial sums of $\sum a_i b_i$ are smaller than $p2^w$. Some authors (see for example [39]) suggested to use *small* primes, to guarantee that the condition $\sum a_i b_i < p2^w$ is always satisfied in a given situation. However, doing this would contradict the main design principle of `nuMONGO`, which is to have no restriction on p except that it must fit in the available number of machine words allocated for it.

What we do instead is to ensure that the number obtained by removing the least significant w bits of any intermediate result remains $< p$. We do this by adding the products $a_i b_i$ in succession, and checking if there has been an overflow or if the most significant half of the intermediate sum is $\geq p$: if so we subtract $p2^w$. The last subtraction is in practice performed as follows: since the internal representation of the intermediate result can be seen as $x2^w + y$, with $y \leq 2^w$ but $x \geq p$, we subtract p from x. This requires as many operations as allowing intermediate results in triple precision, but either less memory accesses are needed, or less registers have to be allocated: In practice this leads to a faster approach, and at the end we have to reduce a number bounded by $p2^w$, making the modular reduction easier (or, in Montgomery's case, possible).

Algorithm 3: ***Incomplete reduction***

`Input:` $p < 2^w$, a_i and $b_i < p$ for $i = 0, \ldots, t$,
`Output:` x with $x \equiv \sum_{i=0}^{t} a_i b_i \bmod p2^w$ and $0 \leq x \leq p2^w$
`Notation:` $x = x_{\text{hi}} \cdot 2^w + x_{\text{lo}}$ where $x_{\text{hi}}, x_{\text{lo}} < 2^w$

1. `Initialise` $x \leftarrow a_0 b_0$
`for` $i = 1$ `to` t `do` $\{$
2. $\quad$ `carry` $\cdot 2^{2w} + x \leftarrow x + a_i b_i$ $\quad$ (with $x < 2^{2w}$ and `carry` $\in \{0, 1\}$)
3. $\quad$ `if carry or` $x_{\text{hi}} \geq p$ `then` $x_{\text{hi}} \leftarrow x_{\text{hi}} - p \bmod 2^w$ $\}$
4. `return` x

We implemented some APIs in `nuMONGO` for supporting Lazy (i.e. delayed) and Incomplete (i.e. limited to the most significant half of the considered operands) modular reduction. They can also be used in the implementation of the arithmetic of extension fields.

A straightforward implementation of complex expressions, such as the formulae for elliptic and hyperelliptic curves, would simply replace any arithmetic operation by a sequence of operands between data in registers, as in the following example. Suppose that we want to compute $ab + cd$, and that the registers `a`, `b`, `c`, `d` contain a, b, c and d respectively. The functions `Add`, `Sub`, `Mul` take 3 operands: the third input is added, subtracted, respectively multiplied to/from/by the second one, and the the result is put in the variable indicated by the first parameter.

```
Mul(a, a,b);  Mul(c, c,d);  Add(a, a,c);
```

The original contents of the registers `a` and `c` are overwritten, the result is found in `a`.

There is a faster way of implementing the computation above. First, we allocate two registers `L` and `M` whose width in bits is twice that of `a`. In other words, `L` and `M` are capable of containing

integers up to $2^{2w}-1$. We call such registers *wide*. Registers capable of containing only integers up to 2^w-1 are called *short*.

In `nuMONGO` there are routines for implementing incomplete reduction, namely `AddDbl` and `SubDbl` for adding wide registers while keeping the result smaller than $p2^w$, `MulNoREDC(L, a,b)` for multiplying `a` and `b` without adopting Montgomery's reduction to the product (that is, `MulNoREDC` just computes a product of integers), and `REDC(r, L)` for putting in the short register `r` the reduction of the content of the wide register `L`. A `Mul` is implemented as a `MulNoREDC` followed by a `REDC`.

The example computation above is then implemented as

```
MulNoREDC(L, a,b); MulNoREDC(M, c,d); AddDbl(L, L,M); REDC(a,L);
```

The gain is clear, as we need only one modular reduction instead of two. A modular reduction is at least as expensive as a multiplication, and often much more, see Table 2 on page 10. The overhead introduced by the wide addition in place of the short one is negligible in practice.

Remark 2.1. *We* cannot *add a reduced element to an unreduced element in Montgomery's representation. This is due to the fact that Montgomery's representation $\hat{a}$ of the integer $a \in [0..p-1]$ is $aR \mod p$, where R is a power of 2 larger than p. Now, $\hat{a}\hat{b}$ is congruent to abR^2 modulo p, not to $\widehat{ab}$. Therefore, adding a number in Montgomery's representation to $\hat{a}\hat{b}$ would give meaningless results.*

For most small values of w, all above operations are inlined, so the code size of the second implementation will be actually *smaller* than in the first implementation, due to the reduced number of modular reductions. This makes it even more likely that the code implementing group operations will fit entirely in the CPU's level 1 cache. For large w ($w \leq 192$) the operators `Add`, `Sub`, `Mul`, etc. are *never* inlined to keep the code of the whole group operations in the level 1 cache of the target CPU.

2.3 Implementation of the Explicit Formulae

To derive explicit formulae, one starts with Cantor's algorithm and "unrolls" the steps in order. First, one must consider which cases can occur, which depend on the degrees of the input divisors and, in the case of the addition, whether the supports of the two divisors to be added contain points with the same x–coordinates. The cases that almost always occur (with probability $1-O(p^{-1})$) are those when the divisors have maximal degree g and the points in the supports all have different x–coordinates. The other cases can be handled, e.g. by Cantor's algorithm without loss of performance. The techniques used to turn the addition and doubling algorithms into a procedural list of operations are: Use of the resultant in the inversion of polynomials and computation of the resultant using Bezout's matrix; Montgomery's trick for computing several inverses with a single inversion; using short convolutions for polynomial multiplication, and the computation only of those coefficients which are really used; recovering the second polynomial of the result using one step of Newton iteration or the Chinese Remaindering Theorem; and reorganization of the normalization of polynomials [61]. The usage of these tricks is detailed in [30, 49, 33].

In the next subsubsections, we analyse the extent of applicability of lazy and incomplete reduction in the formulae for the types of curves which we consider.

2.3.1 Genus 2

We now see in a concrete case – namely a particular genus 2 formula – how wide operations are used in practice. Table 3 is derived from results in [30], but restricted to the odd characteristic case. Also we do not distinguish between squarings and multiplications. The detailed breakdown of the

Table 3: Addition, $\deg u_1 = \deg u_2 = 2$

Step	Expression	Cost
	`Input:` $[u_1, v_1], [u_2, v_2]$, with $\deg u_1 = \deg u_2 = 2$, and $f = x^5 + f_3x^3 + f_2x^2 + f_1x + f_0$ `Output:` $[u_3, v_3] = [u_1, v_1] + [u_2, v_2]$ `Notation:` $u_i = x^2 + u_{i1}x + u_{i0}$ and $v_i = v_{i1}x + v_{i0}$	
1	compute resultant r of u_1, u_2: $z_1 = u_{11} - u_{21}$, $z_2 = u_{20} - u_{10}$, $z_3 = u_{11}z_1 + z_2$; $r = z_2z_3 + z_1^2u_{10}$;	4M
2	compute almost inverse of u_2 modulo u_1 ($\iota = \iota_1x + \iota_0 = r/u_2 \bmod u_1$): $\iota_1 = z_1$, $\iota_0 = z_3$;	-
3	compute $s' = rs \equiv (v_1 - v_2)\iota \bmod u_1$: $w_0 = v_{10} - v_{20}$, $w_1 = v_{11} - v_{21}$, $w_2 = \iota_0w_0$, $w_3 = \iota_1w_1$; $s'_1 = (\iota_0 + \iota_1)(w_0 + w_1) - w_2 - w_3(1 + u_{11})$, $s'_0 = w_2 - u_{10}w_3$; If $s_1 = 0$ handle exceptional case (e.g. with Cantor's algorithm)	5M
4	compute $s'' = x + s_0/s_1 = x + s'_0/s'_1$ and s_1: $w_1 = (rs'_1)^{-1}(= 1/r^2s_1)$, $w_2 = rw_1(= 1/s'_1)$, $w_3 = s'^2_1w_1(= s_1)$; $w_4 = rw_2(= 1/s_1)$, $w_5 = w_4^2$; $s''_0 = s'_0w_2$;	I, 7M
5	compute $l' = s''u_2 = x^3 + l'_2x^2 + l'_1x + l'_0$: $l'_2 = u_{21} + s''_0$, $l'_1 = u_{21}s''_0 + u_{20}$, $l'_0 = u_{20}s''_0$	2M
6	compute $u_3 = (s(l + 2v_2) - k)/u_1$: $u_{30} = (s''_0 - u_{11})(s''_0 - z_1) - u_{10} + l'_1 + (2v_{21})w_4 + (2u_{21} + z_1)w_5$; $u_{31} = 2s''_0 - z_1 - w_5$;	3M
7	compute $v_3 \equiv -(l + v_2) \bmod u_3$: $w_1 = l'_2 - u_{31}$, $w_2 = u_{31}w_1 + u_{30} - l'_1$, $v_{31} = w_2w_3 - v_{21}$; $w_2 = u_{30}w_1 - l'_0$, $v_{30} = w_2w_3 - v_{20}$;	4M
total		I, 25M

reductions saved due to the use of wide operands follows:

1. In Step 1 we can save one `REDC` in the computation of r, since we do not need the reduced value of z_2z_3 and $z_1^2u_{10}$ anywhere else.
2. In Step 3 we do not reduce $w_2 = \iota_0w_0$. Since it is used in the computation of s'_1 and s'_0, which are sums of products of two elements. So only 3 `REDC`s are required to implement Step 3: for w_3 and for the final results of s'_1 and s'_0. This is a saving of 2 `REDC`s.

3. In Step 5, it would be desirable to leave the coefficients l'_1 and l'_0 of l' unreduced, since they are used in the following two Steps only in additions with other products of two elements. But $l'_1 = u_{21}s''_0 + u_{20}$ is a problem: we cannot add reduced and unreduced quantities (see Remark 2.1). We circumvent this by computing the wide products $L_1 = u_{21}s''_0$ and $L_0 = u_{20}s''_0$.
4. In Step 6, we have $u'_0 = (s''_0 - u_{11})(s''_0 - z_1) + L_1 + 2v_{21}w_4 + (2u_{21} + z_1)w_5 + z_2$. We need only one `REDC` (instead of four) to compute the (reduced) sum of the first four products. Note that, at this point, L_1 is already known. Then, we add z_2.
5. Step 7: only one further saved `REDC` (the apparent additional ones come from L_1 and L_0).

Summarizing, to implement addition for genus 2 curves in affine coordinates in the most common case, we need 12 `Mul`s, 13 `MulNoREDC`s and 6 `REDC`s. Thus, we save 7 `REDC`s.

We implemented addition and doubling in all coordinate systems. In order to speed-up scalar multiplication, we also implemented addition in the cases where one of the two points to be added is given in $\mathcal{A}$ and the other one in $\mathcal{P}$ or $\mathcal{N}$, with result in $\mathcal{P}$ or $\mathcal{N}$ respectively.

In Table 4 we show the operation counts of the operations implemented, by counting multiplications and squarings together, but separating the number of `REDC`s.

Table 4: Costs of Group Operations

Doubling		Addition	
operation	costs	operation	costs
genus 2			
$2\mathcal{A} = \mathcal{A}$	1I, 27m, 22R	$\mathcal{A} + \mathcal{A} = \mathcal{A}$	1I, 25m, 18R
$2\mathcal{P} = \mathcal{P}$	44m, 38R	$\mathcal{P} + \mathcal{P} = \mathcal{P}$	51m, 42R
		$\mathcal{A} + \mathcal{P} = \mathcal{P}$	43m, 33R
$2\mathcal{N} = \mathcal{N}$	41m, 37R	$\mathcal{N} + \mathcal{N} = \mathcal{N}$	54m, 50R
		$\mathcal{A} + \mathcal{N} = \mathcal{N}$	41m, 37R
genus 3			
$2\mathcal{A} = \mathcal{A}$	1I, 71m, 57R	$\mathcal{A} + \mathcal{A} = \mathcal{A}$	1I, 76m, 55R

The table contains also the counts for the genus 3 case, which we will discuss in the next Paragraph. One sees at once that the number of modular reductions is always significantly smaller than the total number of multiplications.

2.3.2 Genus 3

For the genus three case our starting point are the explicit formulae for the genus 3 affine coordinates developed by Pelzl in [49]. These formulae, as well as those in [50], contain some errors in the most general case if one wants to use them in odd characteristic. This has not been a problem in the reported implementation since it was in even characteristic only, for which the formulae have been verified to be correct. The author corrected these formulae together with Jan Pelzl. After the

resulting formulae have been verified to work correctly – and have been already optimised using lazy reduction and wide operands – they have been compared again to those in [66], which are for general curves of the form $y^2 + h(x)y = f(x)$, and are implemented only in even characteristic with $h(x) = 1$. In Table 5 and Table 6 on the next page we report the correct formulae for addition and doubling in genus 3 in odd characteristic and on curves of equation $y^2 = f(x)$, where the second most significant coefficient of f vanishes.

Table 5: Explicit addition formula on a genus three Jacobian over $\mathbb{F}_q$

Input: $[u_1, v_1], [u_2, v_2]$, with $\deg u_1 = \deg u_2 = 3$, and $f = x^7 + f_5x^5 + f_4x^4 + f_3x^3 + f_2x^2 + f_1x + f_0$
Output: $[u_3, v_3] = [u_1, v_1] + [u_2, v_2]$
Notation: $u_i = x^3 + u_{i2}x^2 + u_{i1}x + u_{i0}$; $\quad v_i = v_{i2}x^2 + v_{i1}x + v_{i0}$; for $1 \le i \le 3$

Step	Expression	Cost
1	Compute resultant r of u_1 and u_2 (Bezout): $t_1 = u_{12}u_{21}$; $t_2 = u_{11}u_{22}$; $t_3 = u_{11}u_{20}$; $t_4 = u_{10}u_{21}$; $t_5 = u_{12}u_{20}$; $t_6 = u_{10}u_{22}$; $t_7 = (u_{20} - u_{10})^2$; $t_8 = (u_{21} - u_{11})^2$; $t_9 = (u_{22} - u_{12})(t_3 - t_4)$; $t_{10} = (u_{22} - u_{12})(t_5 - t_6)$; $t_{11} = (u_{21} - u_{11})(u_{20} - u_{10})$; $r = (u_{20} - u_{10} + t_1 - t_2)(t_7 - t_9) + (t_5 - t_6)(t_{10} - 2t_{11}) + t_8(t_3 + t_4)$;	12M, 2S
2	Compute almost inverse $inv = r/u_1 \bmod u_2$: $\iota_2 = (t_1 - t_2 - u_{10} + u_{20})(u_{22} - u_{12}) - t_8$; $\iota_1 = \iota_2 u_{12} - t_{10} + t_{11}$; $\iota_0 = \iota_2 u_{11} + u_{12}(t_{10} - t_{11}) + t_9 - t_7$	4M
3	Compute $s' = rs \equiv (v_2 - v_1)inv \bmod u_2$: $t_{12} = (\iota_1 + \iota_2)(v_{12} - v_{22} + v_{11} - v_{21})$; $t_{13} = (v_{11} - v_{21})\iota_1$; $t_{14} = (\iota_0 + \iota_2)(v_{12} - v_{22} + v_{10} - v_{20})$; $t_{15} = (v_{10} - v_{20})\iota_0$; $t_{16} = (\iota_0 + \iota_1)(v_{11} - v_{21} + v_{10} - v_{20})$; $t_{17} = (v_{12} - v_{22})\iota_2$; $r'_0 = t_{15}$; $r'_1 = t_{13} + t_{15} + t_{16}$; $r'_2 = t_{13} + t_{14} + t_{15} + t_{17}$; $r'_3 = t_{12} + t_{13} + t_{17}$; $r'_4 = t_{17}$; $t_{18} = u_{22}r'_4 - r'_3$; $s'_0 = r'_0 + u_{20}t_{18}$; $s'_1 = r'_1 - (u_{21} + u_{20})(r'_4 - t_{18}) + u_{21}r'_4 - u_{20}t_{18}$; $s'_2 = r'_2 - u_{21}r'_4 + u_{22}t_{18}$; If $s'_2 = 0$ handle exceptional case (e.g. with Cantor's algorithm)	11M
4	Compute $s = (s'/r)$, and make it monic: $w_1 = (rs'_2)^{-1}$; $w_2 = rw_1$; $w_3 = w_1 {s'_2}^2$; $w_4 = rw_2$; $w_5 = w_4^2$; $s_0 = w_2s'_0$; $s_1 = w_2s'_1$;	I, 6M, 2S
5	Compute $z = su_1$: $z_0 = s_0u_{10}$; $z_1 = s_1u_{10} + s_0u_{11}$; $z_2 = s_0u_{12} + s_1u_{11} + u_{10}$; $z_3 = s_1u_{12} + s_0 + u_{11}$; $z_4 = u_{12} + s_1$;	6M
6	Compute $u' = [s(z + 2w_4v_1) - w_5(f - v_1^2)/u_1)]/u_2$: $u'_3 = z_4 + s_1 - u_{22}$; $u'_2 = -u_{22}u'_3 - u_{21} + z_3 + s_0 + w_4 + s_1z_4$; $u'_1 = w_4(2v_{12} + s_1) + s_1z_3 + s_0z_4 + z_2 - w_5 - u_{22}u'_2 - u_{21}u'_3 - u_{20}$; $u'_0 = w_4(2v_{11} + 2s_1v_{12} + s_0) + s_1z_2 + z_1 + s_0z_3 + w_5u_{12} - u_{22}u'_1 - u_{21}u'_2 - u_{20}u'_3$	15M
7	Compute $v' = -(w_3z + v_1) \bmod u'$: $t_1 = u'_3 - z_4$; $v'_0 = w_3(u'_0t_1 + z_0) + v_{10}$; $v'_1 = w_3(u'_1t_1 - u'_0 + z_1) + v_{11}$; $\quad v'_2 = w_3(u'_2t_1 - u'_1 + z_2) + v_{12}$; $\quad v'_3 = w_3(u'_3t_1 - u'_2 + z_3)$;	8M
8	Reduce u', i.e. $u_3 = (f - v'^2)/u'$: $u_{32} = -u'_3 - {v'_3}^2 + v'_3$; $\quad u_{31} = -u'_2 - u_{32}u'_3 + f_5 - 2v'_2v'_3 - v'_2$; $u_{30} = -u'_1 - u_{32}u'_2 - u_{31}u'_3 + f_4 - 2v'_1v'_3 - {v'_2}^2 - v'_1$;	5M, 2S
9	Compute $v_3 = -v' \bmod u_3$: $v_{32} = v'_2 - (v'_3 + 1)u_{32}$; $\quad v_{31} = v'_1 - (v'_3 + 1)u_{31}$; $\quad v_{30} = v'_0 - (v'_3 + 1)u_{30}$;	3M
total		I, 70M, 6S

A pleasant aspect of the genus three formulae is that a large proportion of modular reductions can be saved: at least 21 in the addition and 14 in the doubling (see Table 4 on the facing page).

Table 6: Explicit doubling formula on a genus three Jacobian over $\mathbb{F}_q$

Step	Expression	Cost
	`Input:` $[u_1, v_1]$ with $\deg u_1 = 3$, and $f = x^7 + f_5x^5 + f_4x^4 + f_3x^3 + f_2x^2 + f_1x + f_0$ `Output:` $[u_2, v_2] = 2 \cdot [u_1, v_1]$ `Notation:` $u_i = x^3 + u_{i2}x^2 + u_{i1}x + u_{i0}$; $v_i = v_{i2}x^2 + v_{i1}x + v_{i0}$; for $1 \le i \le 2$	
1	Compute resultant r of u_1 and $2v_1$ (Bezout): $t_1 = 2u_{12}v_{11}$; $t_2 = 2u_{11}v_{12}$; $t_3 = 2u_{11}v_{10}$; $t_4 = 2u_{10}v_{11}$; $t_5 = 2u_{12}v_{10}$; $t_6 = 2u_{10}v_{12}$; $t_7 = (2v_{10} - u_{10})^2$; $t_8 = (2v_{11} - u_{11})^2$; $t_9 = (2v_{12} - u_{12})(t_3 - t_4)$; $t_{10} = (2v_{12} - u_{12})(t_5 - t_6)$; $t_{11} = (2v_{11} - u_{11})(2v_{10} - u_{10})$; $r = (2v_{10} - u_{10} + t_1 - t_2)(t_7 - t_9) +$ $+ (t_5 - t_6)[(t_5 - t_6)(2v_{12} - u_{12}) - 2(2v_{11} - u_{11})] + t_8(t_3 - t_4)$;	6M, 2S
2	Compute almost inverse $inv = r/(2v_1) \bmod u_1$: $\iota_2 = -(t_1 - t_2 - u_{10} + 2v_{10})(2v_{12} - u_{12}) - t_8$; $\iota_1 = \iota_2 u_{12} - t_{10} + t_{11}$; $\iota_0 = \iota_2 u_{11} + u_{12}(t_{10} - t_{11}) + t_9 - t_7$	4M
3	Compute $z = ((f - v_1^2)/u_1) \bmod u_1$: $t_{12} = v_{12}^2$; $z_3' = -u_{12}$; $t_{13} = z_3' u_{11}$; $z_2' = f_5 - v_{12} - u_{11} - u_{12}z_3'$; $z_1' = f_4 - v_{11} - t_{12} - u_{10} - t_{13} - z_2' u_{12}$; $z_2 = f_5 - v_{12} - 2u_{11} + u_{12}(u_{12} - 2z_3')$; $z_1 = z_1' - t_{13} + u_{12}u_{11} - u_{10}$; $z_0 = f_3 - 2v_{12}v_{11} - v_{10} - z_3' u_{10} - z_2' u_{11} - z_1' u_{12}$;	7M, 2S
4	Compute $s' = z\iota \bmod u_1$: $t_{12} = (\iota_1 + \iota_2)(z_1 + z_2)$; $t_{13} = z_1\iota_1$; $t_{14} = (\iota_0 + \iota_2)(z_0 + z_2)$; $t_{15} = z_0\iota_0$; $t_{16} = (\iota_0 + \iota_1)(z_0 + z_1)$; $t_{17} = z_2\iota_2$; $r_0' = t_{15}$; $r_1' = t_{13} + t_{15} + t_{16}$; $r_2' = t_{13} + t_{14} + t_{15} + t_{17}$; $r_3' = t_{12} + t_{13} + t_{17}$; $r_4' = t_{17}$; $t_{18} = u_{12}r_4' - r_3'$; $s_0' = r_0' + u_{10}t_{18}$; $s_1' = r_1' - (u_{11} + u_{10})(r_4' - t_{18}) + u_{11}r_4' - u_{10}t_{18}$; $s_2' = r_2' - u_{11}r_4' + u_{12}t_{18}$; If $s_2' = 0$ handle exceptional case (e.g. with Cantor's algorithm)	11 M
5	Compute $s = (s'/r)$ and make s monic: $w_1 = (rs_2')^{-1}$; $w_2 = w_1 r$; $w_3 = w_1(s_2')^2$; $w_4 = w_2 r$; $w_5 = w_4^2$ $s_0 = w_2 s_0'$; $s_1 = w_2 s_1'$;	I, 6M, 2S
6	Compute $G = su_1$: $g_0 = s_0u_{10}$; $g_1 = s_1u_{10} + s_0u_{11}$; $g_2 = s_0u_{12} + s_1u_{11} + u_{10}$; $g_3 = s_1u_{12} + s_0 + u_{11}$; $g_4 = u_{12} + s_1$;	6M
7	Compute $u' = u_1^{-2}[(G + w_4v_1)^2 - w_5 f]$: $u_3' = 2s_1$; $u_2' = s_1^2 + 2s_0 + w_4$; $u_1' = 2s_0s_1 + w_4(2v_{12} + s_1 - u_{12}) - w_5$; $u_0' = w_4[2v_{11} + s_0 - u_{11} + u_{12}(u_{12} - 2v_{12} - s_1)] + 2w_5u_{12} + s_0^2$;	5M, 2S
8	Compute $v' = -(Gw_3 + v_1) \bmod u'$: $t_1 = u_3' - g_4$; $v_3' = (t_1u_3' - u_2' + g_3)w_3$; $v_2' = (t_1u_2' - u_1' + g_2)w_3 + v_{12}$; $v_1' = (t_1u_1' - u_0' + g_1)w_3 + v_{11}$; $v_0' = (t_1u_0' - g_0)w_3 + v_{10}$;	8M
9	Reduce u', i.e. $u_2 = (f - v'^2)/u'$: $u_{22} = -u_3' - {v_3'}^2 + v_3'$; $u_{21} = -u_2' - u_{22}u_3' + f_5 - 2v_2'v_3' - v_2'$; $u_{20} = -u_1' - u_{22}u_2' - u_{21}u_3' + f_4 - 2v_1'v_3' - {v_2'}^2 - v_1'$;	5M, 2S
10	Compute $v_2 = -v' \bmod u_2$: $v_{22} = v_2' - (v_3' + 1)u_{22}$; $v_{21} = v_1' - (v_3' + 1)u_{21}$; $v_{20} = v_0' - (v_3' + 1)u_{20}$;	3M
total		I, 61M, 10S

2.3.3 Elliptic Curves

For EC explicit formulae, no savings in REDCs are possible in the cases which we consider. This is due to the fact that the formulae are much simpler and whenever expressions of the form $ab + cd$ appear, then ab, say, must be later multiplied by another field element – or is "reused" in an expression involving other products which must, sooner or later, be reduced anyway. This implies that it can as well be reduced immediately, and one gains more by reducing also cd and adding ab and cd as short operands instead of wide ones.

Let us work out an example, namely, the addition of a point in $\mathcal{A}$ coordinates to a point in $\mathcal{P}$, with result in $\mathcal{P}$. Let $P_1 = (X_1, Y_1)$, $P_2 = (X_2 : Y_2 : Z_2)$ and $P_3 = (X_3 : Y_3 : Z_3)$. The addition $P_3 = P_1 + P_2$ is done by performing the following operations in the given order

$$u = Y_2 - Y_1 Z_2,\ v = X_2 - X_1 Z_2,\ A = u^2 Z_2 - v^3 - 2v^2 X_1 Z_2 \ ,$$
$$X_3 = vA,\ Y_3 = u(v^2 X_1 Z_2 - A) - v^3 Y_1 Z_2,\ Z_3 = v^3 Z_2 \ .$$

For the computation of u and v no savings are possible. The expression $A = u^2 Z_2 - v^3 - 2v^2 X_1 Z_2$ is a bit more complex. We could hope to save reduction in its computation, but: We need v^3 reduced anyway for Z_3, and also A must be reduced to get X_3. In the computation of Y_3 we have the subexpression $v^2 X_1 Z_2 - A$: Since A will be reduced anyway for other computations, there is no saving here from keeping $\ell := v^2 X_1 Z_2$ unreduced (say, multiplying reduced v^2 with $X_1 Z_2$), subtracting an unreduced copy of A and then reducing. At this point we can either keep ℓ unreduced and reduce A and $\ell - A$ separately, or reduce it right away. But then there is no saving in computing A using lazy reduction.

This case, together with $\mathcal{P} + \mathcal{P} = \mathcal{P}$, is perhaps the one where it is most difficult to see that lazy reduction cannot be applied to EC.

2.4 Scalar Multiplication

There are many methods for performing a scalar multiplication in a generic group, which can be used for EC and HEC. See [18] for a survey.

A simple method for computing $s \cdot D$ for an integer s and a divisor D is based on the binary representation of s. If $s = \sum_{i=0}^{n-1} s_i 2^i$ where each $s_i = 0$ or 1, then $n \cdot D$ can be computed as

$$sD = 2(2(\cdots 2(2(s_{n-1}D) + s_{n-2}D) + \cdots) + s_1 D) + s_0 D. \qquad (4)$$

This requires $n-1$ doublings and on average $n/2$ additions on the curve.

On EC and HEC, the computation of the inverse of a given group element is for free. In other words, adding and subtracting a point has the same cost. Hence one can use the *non adjacent form* (NAF) [52], which is an expansion $s = \sum_{i=0}^{n} s_i 2^i$ where each $s_i \in \{0, \pm 1\}$ and $s_i s_{i+1} = 0$. This leads to a method needing n doublings and on average $n/3$ additions or subtractions.

A generalization of the NAF comes from using "sliding windows". The wNAF [60, 8] of the integer s is a representation $s = \sum_{j=0}^{n} s_j 2^j$ where the integers s_j satisfy the following two conditions: (i) either $s_j = 0$ or s_j is odd and $|s_j| \leq 2^w$; (ii) of any $w+1$ consecutive coefficients $s_{j+w}, \ldots, s_j$ at most one is nonzero.

The 1NAF coincides with the NAF. The wNAF has average density $1/(w+2)$. To compute a scalar multiplication based on the wNAF one must first precompute the divisors D, $3D, \ldots, (2^w - 1)D$, and then perform a double-and-add step like (4).

3 Results and Comparisons

Table 7 on page 20 reports the timings of our implementation. Since `nuMONGO` provides support only for moduli up to 256 bits, EC are tested only on fields up to that size. For genus 2 curves on a 256 bit field, a group up to 512 bits is possible: We choose this group size as a limit also for our tests of genus 3 curves.

All benchmarks were performed on a 1 Ghz AMD Athlon (Model 4) PC, under the Linux operating system (kernel version 2.4.18). The compiler used was the GNU Compiler Collection (`gcc`) version 2.95.3. The elliptic curves all have almost prime order and have been found by point counting on random curves. The genus 2 and 3 curves have rational divisor class groups of almost prime order and have been costructed by complex multiplication.

For each combination of group type (EC and HEC of genus 2 and 3), coordinate system and bit size of the group (not of the field!), we averaged the timings of several thousands scalar multiplications with random scalars, and performed the scalar multiplication using three different recodings of the scalar: the binary representation, the NAF, and the wNAF. For the wNAF we report only the best timing and the value of w for which is was attained. When using the various coordinate systems, we always keep the base divisor *and* its multiples in affine coordinates, since adding an affine point to a point in *any* coordinate system is faster than adding two points in that coordinate system. In the timings for the wNAF the timings for the precomputations are always included in the results.

For comparison with our timings, Lange [30] reported timings of 8.232 and 9.121 milliseconds for genus 2 curves with group order $\approx 2^{160}$ and 2^{180} respectively on a `gmp`-based implementation of affine coordinates on a Pentium IV processor running at 1.5 Ghz, and for a `ntl`-based the timings are 11.326 and 16.324 for the two curves above. The scalar multiplication in [30] is based on the double-and-add algorithm based on the unsigned binary representation. In [53], a timing of 98 milliseconds for a genus 3 curve of about 180 bits ($p \approx 2^{60}$) on an Alpha 21164A CPU running at 600MHz is reported. The speed of these two CPUs is close to that of the machine we used for our tests.

In Table 8 on page 20 we provide timings for `ecc` and `hec` using `gmp` (with the highest optimization possible) and the double-and-add scalar multiplication is based on the unsigned binary representation. In Table 9 on page 21 we show the timings of the `nuMONGO` genus 2 and 3 `hec` implementation without lazy and incomplete reduction.

A summary of the results follows:

(1) Using a specialised software library one can get a speed-up by a factor of 3 to 4.5 for EC with respect to a traditional implementation. The speed-up for genus 2 and 3 curves is up to 8.

(2) Lazy and incomplete modular reduction alone brings a speed-up from 3% to 7%.

  (a) The speed-up is larger for genus 3 curves, since a larger proportion of reductions is saved.

  (b) For genus 2, the amount of modular reductions saved with new coordinates is smaller than with projective coordinates. As a result, the performance difference between the two coordinate systems is smaller than without lazy and incomplete reduction (for example,

for 256-bit curves, the gain is 5.3% instead of 9.7%). It would be interesting to rewrite the explicit formulae for $\mathcal{N}$ to allow more savings in reduction.

(3) HEC over prime fields are still slower than EC, but the gap has been narrowed.

  (a) Affine coordinates for genus 2 HEC are significantly faster than those for EC. Those for genus 3 are faster from 144 bits upwards.

  (b) Comparing the best coordinate systems and scalar multiplication algorithms for genus 2 HEC and EC, we see that:

    (i) For 192 bit, resp. 256 bit curves, HEC is slower than EC by only 18%, resp. 15%.

    (ii) For other group sizes the difference is often around 50%.

  (c) Genus 3 curves are slower than genus 2 ones. Whereas with `gmp` the difference is 80% to 100% for 160 to 512 bit groups, using `nuMONGO` the difference is often as small as 50%.

(4) Using `nuMONGO` we can successfully eliminate most of the overheads associated to function calls and to the processing of short operands. This proves the soundness of our approach.

  (a) In the `gmp`-based implementation (see Table 8) the timings with different coordinate systems are closer to each other than with `nuMONGO` because of the big amount of time lost in the overheads. For HEC we even have the paradoxical result that $\mathcal{P}$ and $\mathcal{N}$ are slower than $\mathcal{A}$, because of they require significantly more function calls for each group operation than $\mathcal{A}$. Therefore, with `gmp` in some cases the overheads dominate the running time.

  (b) For affine coordinates the dominant part of the operation is the field inversion, hence the speed-up given by `nuMONGO` is not big, and is close to that in Table 2 on page 10 for the inversion alone. For the other coordinate systems, the speed-up becomes significant.

(5) If the *field* size for a given group is not close to a multiple of the machine word size, there is a relative drop in performance with respect to other groups where the field size is nicer.

  (a) This is seen, for example, for 160-bit groups. For genus 2 this means a 80-bit field, but then 96-bit arithmetic must be used on a 32-bit CPU. For genus 3 (53-bit field) this is even worse. For 144-bit groups, genus 3 curves can exploit 48-bit arithmetic, which has been made faster by suitable implementation tricks (an approach which did not work for 80 and 112 bit fields), and thus the gap to genus 2 is only 50%.

  (b) A similar phenomenon occurs when comparing 224-bit EC and genus 2 HEC groups. The performance loss of HEC is about 50%, due to the 112-bit field arithmetic, which is not a multiple of the native word size of the CPU. However, 192 bit and 256 bit HEC perform much better in comparison with EC.

In practice, the performance of hyperelliptic curves is satisfactory enough to be considered as a valid alternative to elliptic curves, especially if relatively large point groups are desired.

Table 7: Comparison of running times, in msec (1 Ghz AMD Athlon PC)

curve	coord.	scalar mult.	Bitlength of group order							
			128	144	160	192	224	256	320	512
`ec`	$\mathcal{A}$	binary	1.82	2.824	3.396	5.946	9.27	13.638		
		NAF	1.61	2.504	3.044	5.31	8.283	12.147		
		wNAF	1.49 ($w=3$)	2.272 ($w=3$)	2.768 ($w=4$)	4.788 ($w=4$)	7.451 ($w=4$)	10.795 ($w=4$)		
	$\mathcal{P}$	binary	0.72	1.04	1.224	2.094	3.001	4.592		
		NAF	0.635	0.92	1.108	1.88	2.713	4.123		
		wNAF	0.615 ($w=3$)	0.884 ($w=3$)	1.056 ($w=3$)	1.781 ($w=3$)	2.59 ($w=4$)	3.876 ($w=3$)		
	$\mathcal{J}$	binary	0.65	0.916	1.12	1.897	2.708	4.095		
		NAF	0.57	0.817	1.04	1.683	2.425	3.598		
		wNAF	0.535 ($w=3$)	0.78 ($w=3$)	0.936 ($w=2$)	1.567 ($w=3$)	2.22 ($w=3$)	3.399 ($w=3$)		
	$\mathcal{J}^c$	binary	0.675	0.976	1.172	2.001	2.805	4.354		
		NAF	0.63	0.864	1.06	1.752	2.579	3.916		
		wNAF	0.585 ($w=2$)	0.816 ($w=2$)	1.004 ($w=3$)	1.688 ($w=3$)	2.456 ($w=4$)	3.631 ($w=3$)		
	$\mathcal{J}^m$	binary	0.545	0.912	1.096	1.816	2.631	4.036		
		NAF	0.505	0.764	0.924	1.555	2.25	3.399		
		wNAF	0.5 ($w=3$)	0.7 ($w=3$)	0.844 ($w=3$)	1.423 ($w=3$)	2.053 ($w=3$)	3.06 ($w=4$)		
`hec` g=2	$\mathcal{A}$	binary	0.99	1.78	2.08	2.696	4.884	5.783	11.675	42.463
		NAF	0.907	1.57	1.94	2.399	4.365	5.212	10.325	38.457
		wNAF	0.82 ($w=3$)	1.44 ($w=4$)	1.72 ($w=4$)	2.166 ($w=4$)	3.92 ($w=4$)	4.662 ($w=4$)	9.27 ($w=5$)	33.709 ($w=5$)
	$\mathcal{P}$	binary	1.073	1.55	1.82	2.174	4.095	4.718	9.225	31.928
		NAF	0.966	1.37	1.62	1.945	3.677	4.245	8.275	28.219
		wNAF	0.906 ($w=3$)	1.29 ($w=4$)	1.533 ($w=4$)	1.786 ($w=4$)	3.339 ($w=4$)	3.841 ($w=4$)	7.37 ($w=4$)	25.104 ($w=5$)
	$\mathcal{N}$	binary	1.051	1.5	1.72	2.075	3.891	4.447	8.6	29.981
		NAF	0.946	1.3	1.51	1.846	3.447	3.959	7.25	26.468
		wNAF	0.867 ($w=3$)	1.21 ($w=4$)	1.41 ($w=4$)	1.676 ($w=4$)	3.085 ($w=4$)	3.528 ($w=4$)	6.85 ($w=4$)	23.323 ($w=5$)
`hec` g=3	$\mathcal{A}$	binary	2.332	2.612	4.7	5.653	5.67	6.82	13.22	47.843
		NAF	1.936	2.16	3.807	4.577	5.06	6.008	11.72	42.348
		wNAF	1.604 ($w=4$)	1.808 ($w=4$)	2.792 ($w=5$)	3.538 ($w=5$)	4.53 ($w=5$)	5.343 ($w=5$)	10.28 ($w=5$)	36.209 ($w=5$)

Table 8: Timings with `gmp`, in msec (1 Ghz AMD Athlon PC)

`ec`	160	192	256
$\mathcal{A}$	5.468	8.305	15.354
$\mathcal{P}$	4.306	5.845	9.16
$\mathcal{J}$	3.775	5.4	8.878
$\mathcal{J}^c$	4.029	5.75	9.67
$\mathcal{J}^m$	3.75	5.182	9.075

`hec`		160	192	256	320	512
g=2	$\mathcal{A}$	9.292	12.082	18.873	29.5	72.09
	$\mathcal{P}$	12.15	14.961	23.442	32.212	81.586
	$\mathcal{N}$	11.349	13.278	20.4	28.93	74.389
g=3	$\mathcal{A}$	19.799	22.452	40.39	59.691	129.541

Table 9: Timings with nuMONGO without lazy and incomplete reduction, in msec (1 Ghz AMD Athlon PC)

hec		160	192	256	320	512
g=2	$\mathcal{A}$	2.22	2.88	6.253	11.832	44.596
	$\mathcal{P}$	1.967	2.393	4.93	9.625	33.915
	$\mathcal{N}$	1.77	2.267	4.493	8.772	31.073
g=3	$\mathcal{A}$	5.961	7.138	7.29	13.959	49.977

References

[1] L. Adleman and J. DeMarrais. *A subexponential algorithm for discrete logarithms over all finite fields*, Mathematics of Computation, **61** (1993), pp. 1–15.

[2] L. Adleman, J. DeMarrais and M. Huang. *A subexponential algorithm for discrete logarithms over the rational subgroup of the Jacobians of large genus hyperelliptic curves over finite fields*. Algorithmic Number Theory, LNCS 877, pp. 28-40. Springer, 1994.

[3] R.M. Avanzi. *Countermeasures against differential power analysis for hyperelliptic curve cryptosystems*. In: *Proceedings of CHES 2003*, Cologne, Germany. LNCS 2779, pp. 366–381. Springer, 2003.

[4] R.M. Avanzi. nuMONGO and *Description and Use of the* nuMONGO *Library*. A software library and related documentation. Available from the author.

[5] R.M. Avanzi and P.M. Mihăilescu. *Generic Efficient Arithmetic Algorithms for PAFFs (Processor Adequate Finite Fields) and Related Algebraic Structures*. To appear in proceedings of: Selected Areas in Cryptography 2003, Ottawa, August 14–15, 2003.

[6] M. Bauer. *A subexponential algorithm for solving the discrete logarithm problem in the Jacobian of high genus hyperelliptic curves over arbitrary finite fields*. Preprint, 1999.

[7] D. Cantor. *Computing in the Jacobian of a Hyperelliptic Curve.* Mathematics of Computation, **48** (1987), pp. 95–101.

[8] H. Cohen, A. Miyaji and T. Ono. *Efficient elliptic curve exponentiation*. In *Proceedings ICICS'97*, LNCS 1334, pp. 282–290. Springer, 1997.

[9] H. Cohen, A. Miyaji and T. Ono. *Efficient Elliptic Curve Exponentiation Using Mixed Coordinates*, In: *ASIACRYPT: Advances in Cryptology*. LNCS 1514, pp. 51–65. Springer, 1998.

[10] D. Coppersmith. *Fast evaluation of logarithms in fields of characteristic two*. IEEE Transactions on Information Theory, **30** (1984), pp. 587–594.

[11] A. Enge and P. Gaudry. *A general framework for subexponential discrete logarithm algorithms*. *Acta Arithmetica* **102**, pp. 83–103, 2002.

[12] M. Fouquet, P. Gaudry, and R. Harley. *On Satoh's algorithm and its implementation*. Lix/RR/00/06, 2000. Preprint.

[13] P. Gaudry, *An algorithm for solving the discrete log problem on hyperelliptic curves*. In Proceedings of: Eurocrypt 2000, pp. 19–34. Springer LNCS, 2000.

[14] P. Gaudry and R. Harley. *Counting points on hyperelliptic curves over finite fields.* Proceedings of: ANTS-IV. LNCS 1838, pp. 313–332. Springer–Verlag, 2000.

[15] P. Gaudry and E. Schost. *Cardinality of a genus 2 hyperelliptic curve over* $\mathrm{GF}(5\cdot 10^{24}+41)$. Email message to the NMBRTHRY mailing list, September 2002.

[16] P. Gaudry and E. Schost. *Construction of Secure Random Curves of Genus 2 over Prime Fields*. Submitted.

[17] R. Gerkmann. *The p-adic Cohomology of Varieties over Finite Fields and Applications on the Computation of Zeta Functions*. Ph.D. Thesis, University Duisburg-Essen, Campus Essen, 2003.

[18] D.M. Gordon. *A survey of fast exponentiation methods*. Journal of Algorithms **27**, pp. 129–146, 1998.

[19] D. Gordon. *Discrete logarithms in GF(p) using the number field sieve*, SIAM Journal on Discrete Mathematics, 6 (1993), pp. 124–138.

[20] T. Grandlund. *GMP. A software library for arbitrary precision integers*.
Available from: `http://www.swox.com/gmp/`

[21] R. Harley. *Fast Arithmetic on Genus Two Curves.*
Available at `http://cristal.inria.fr/~harley/hyper/`

[22] T. Jebelean. *A Generalization of the Binary GCD Algorithm.* ISSAC 1993: pp. 111–116.

[23] T. Jebelean. *A Double-Digit Lehmer-Euclid Algorithm for Finding the GCD of Long Integers.* Journal of Symbolic Computation 19(1-3), pp. 145–157 (1995)

[24] B.S. Kaliski Jr.. *The Montgomery inverse and its applications*. IEEE Transactions on Computers, 44(8), pp. 1064–1065, August 1995.

[25] K.S. Kedlaya. *Counting Points on Hyperelliptic Curves using Monsky-Washnitzer Cohomology*. Journal of the Ramanujan Mathematical Society **16**, pp. 323–338, 2001.

[26] N. Koblitz. *Hyperelliptic Cryptosystems*. Journal of Cryptology **1**, pp. 139–150, 1989.

[27] N. Koblitz. *Algebraic aspects of cryptography.* Springer, 1998.

[28] U. Krieger. `signature.c`: *Anwendung hyperelliptischer Kurven in der Kryptographie*. M.S. Thesis, Mathematik und Informatik, Universität Essen, Fachbereich 6, Essen, Germany.

[29] J. Kuroki, M. Gonda, K. Matsuo, J. Chao and S. Tsujii. *Fast Genus Three Hyperelliptic Curve Cryptosystems.* In: *The 2002 Symposium on Cryptography and Information Security, Japan - SCIS 2002*, Jan. 29–Feb. 1 2002.

[30] 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/`

[31] T. Lange. *Inversion-Free Arithmetic on Genus 2 Hyperelliptic Curves*. Cryptology ePrint Archive, Report 2002/147, 2002. `http://eprint.iacr.org/`

[32] T. Lange. *Weighted Coordinates on Genus 2 Hyperelliptic Curves.* Cryptology ePrint Archive, Report 2002/153, 2002. `http://eprint.iacr.org/`

[33] T. Lange. *Formulae for Arithmetic on Genus 2 Hyperelliptic Curves.* Preprint.
Available from: `http://www.ruhr-uni-bochum.de/itsc/tanja/`
It partially contains and extends the material of the previous three papers [30, 31, 32].

[34] A. Lauder and D. Wan. *Counting points on varieties over finite fields of small characteristic*. Submitted.

[35] A.K. Lenstra and H.W. Lenstra Jr. and M.S. Manasse and J.M. Pollard. *The Number Field Sieve*, In: *ACM Symposium on Theory of Computing"*, pp. 564–572, 1990.

[36] A.K. Lenstra, and H.W. Lenstra, Jr. (eds.). *The development of the number field sieve*, LNCS 1554. Springer, 1993

[37] A.K. Lenstra and E.R. Verheul. *Selecting Cryptographic Key Sizes*. Journal of Cryptology **14**, no. 4, pp. 255–293 (2001).

[38] R. Lercier. *Algorithmique des courbes elliptiques dans les corps finis.* These.
Available from `http://www.medicis.polytechnique.fr/~lercier/`

[39] C.H. Lim, H.S. Hwang. *Fast implementation of Elliptic Curve Arithmetic in GF(p^m).* Proc. PKC 00, LNCS 1751, pp. 405–421. Springer 2000.

[40] K. Matsuo, J. Chao, and S. Tsujii. *Fast Genus Two Hyperelliptic Curve Cryptosystems.* Tech Report ISEC 2001–23, pp. 89–96. IEICE Japan.

[41] A. Menezes, Y.-H. Wu and R. Zuccherato. *An Elementary Introduction to Hyperelliptic Curves.* In [27].

[42] J.-F. Mestre. *Construction des courbes de genre 2 a partir de leurs modules.* Progr. Math. **94**, pp. 313–334, 1991.

[43] J.-F. Mestre. *Lettre adressé à Gaudry et Harley.* December 2000.
`http://www.math.jussieu.fr/~mestre/lettreGaudryHarley.ps`.

[44] K. Matsuo, and J. Chao, and S. Tsujii. *An improved baby step giant step algorithm for point counting of hyperelliptic curves over finite fields.* Proc. of SCIS 2002, IEICE Japan.

[45] Y. Miyamoto, H. Doi, K. Matsuo, J. Chao, and S. Tsuji. *A Fast Addition Algorithm of Genus Two Hyperelliptic Curve.* In SCIS, IEICE Japan, pp. 497–502, 2002. in Japanese.

[46] P.L. Montgomery. *Modular multiplication without trial division*, Math. Comp. **44** (1985),pp. 519–521.

[47] D. Mumford. *Tata Lectures on Theta II.* Birkhäuser 1984.

[48] P. van Oorschot and M. Wiener. *Parallel collision search with cryptanalytic applications*, Journal of Cryptology, **12** (1999), pp. 1–28.

[49] J. Pelzl. *Fast Hyperelliptic Curve Cryptosystems for Embedded Processors*. Master's thesis, Ruhr-University of Bochum, 2002.

[50] J. Pelzl, T. Wollinger, J. Guajardo, J. and C. Paar. *Hyperelliptic Curve Cryptosystems: Closing the Performance Gap to Elliptic Curves*. In: *Proceedings of CHES 2003*. LNCS 2779, pp. 351–365.

[51] J. Pollard. *Monte Carlo methods for index computation mod p*. Mathematics of Computation, **32** (1978), pp. 918–924.

[52] G.W. Reitwiesner. *Binary arithmetic.* Advances in Computers **1**, 231–308, 1960.

[53] 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. 692–703. IEICE Trans.

[54] Y. Sakai, K. Sakurai, and H. Ishizuka. *Secure Hyperelliptic Cryptosystems and their Performance.* In Public Key Cryptography. LNCS 1431, pp. 164–181. Springer, Berlin.

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

[56] T. Satoh, B. Skjernaa and Y. Taguchi. *Fast computation of canonical lifts of elliptic curves and its application to point counting*. 2001, Preprint available by email to `pprtserv@rimath.saitama-u.ac.jp` with `u0108.1.1` as subject.

[57] E. Savacs, Ç. K. Koç. *The Montgomery Modular Inverse – Revisited*. IEEE Transactions on Computers **49**, No. 7, July 2000, pp. 763–766.

[58] V. Shoup. *NTL: A Library for doing Number Theory*.
Available from: `http://www.shoup.net/ntl/`

[59] N.P. Smart. *On the Performance of Hyperelliptic Cryptosystems*. In Advances in Cryptology – EUROCRYPT '99, LNCS 1592, pp. 165–175, Berlin, 1999. Springer.

[60] J.A. Solinas. *An improved algorithm for arithmetic on a family of elliptic curves.* In: *Advances in Cryptology – CRYPTO '97 (1997)*, LNCS 1294, pp. 357–371.

[61] M. Takahashi. *Improving Harley Algorithms for Jacobians of Genus 2 Hyperelliptic Curves*. In SCIS, IEICE Japan, 2002. In Japanese.

[62] N. Thériault. *Index calculus attack for hyperelliptic curves of small genus.* Proceedings of Asiacrypt 2003. To appear.

[63] F. Vercauteren, B. Preneel, J. Vandewalle. *A Memory Efficient Version of Satoh's Algorithm*. In: Advances in Cryptology - EUROCRYPT 2001, LNCS 2045, pp. 1–13. Springer 2001.

[64] F. Vercauteren. *Zeta Functions of Hyperelliptic Curves over Finite Fields of Characteristic 2*. In: Advances in cryptology – Crypto' 2002, pp. 373–387. Springer 2002.

[65] A. Weng. *Konstruktion kryptographisch geeigneter Kurven mit komplexer Multiplikation*. PhD thesis, Universität Gesamthochschule Essen, 2001.

[66] T. Wollinger, J. Pelzl, V. Wittelsberger, C. Paar, G. Saldamli, and Ç.K. Koç. *Elliptic & Hyperelliptic Curves on Embedded μP*. To appear in: Special issue on Embedded Systems and Security of the ACM Transactions in Embedded Computing Systems.