Aspects of Hyperelliptic Curves over Large Prime Fields in Software Implementations

Roberto Maria Avanzi[⋆]

Institute for Experimental Mathematics (IEM) – University of Duisburg-Essen
Ellernstraße 29, D-45326 Essen, Germany
mocenigo at exp-math.uni-essen.de

Department of Electrical Engineering and Information Sciences
Ruhr University of Bochum, Universitätsstraße 150, D-44780 Bochum, Germany

Abstract. We present an implementation of elliptic curves and of hyperelliptic curves of genus 2 and 3 over prime fields. To achieve a fair comparison between the different types of groups, we developed an ad-hoc arithmetic library, designed to remove most of the overheads that penalize implementations of curve-based cryptography over prime fields. These overheads get worse for smaller fields, and thus for larger genera for a fixed group size. We also use techniques for delaying modular reductions to reduce the amount of modular reductions 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 14% and 15% respectively.

Keywords: Elliptic and hyperelliptic curves, cryptography, efficient implementation, prime field arithmetic, lazy and incomplete modular reduction.

1 Introduction

In 1988 Koblitz [21] proposed to use hyperelliptic curves (HEC) as an alternative to elliptic curves (EC) for designing cryptosystems based on the *discrete logarithm problem* (DLP). EC are just the genus 1 HEC. Cryptosystems based on EC need a much shorter key than RSA or systems based on the DLP in finite fields: A 160-bit EC key is considered to offer security equivalent to that of a 1024-bit RSA key [25]. Since the best known methods to solve the DLP on EC and on HEC of genus smaller than 4 have the same complexity, these curves offer the same security level, but HEC of genus 4 or higher offer less security [12, 38].

Until recently, HEC have been considered not practical [36] because of the difficulty of finding suitable curves and their poor performance with respect to EC. In the subsequent years the situation changed.

Firstly, it is now possible to efficiently construct genus 2 and 3 HEC whose Jacobian has almost prime order of cryptographic relevance. Over prime fields one can either

[⋆] This research has been supported by the Commission of the European Communities through the IST Programme under Contract IST-2001-32613 (see http://www.arehcc.com).

count points in genus 2 [13], or use the complex multiplication (CM) method for genus 2 [29, 39] and 3 [39].

Secondly, the performance of the HEC group operations has been considerably improved. For genus 2 the first results were due to Harley [17]. The state of the art is now represented by the explicit formulae of Lange: see [23, 24] and further references therein. For genus 3, see [32, 33] (and also [14]).

HEC are attractive to designers of embedded hardware since they require smaller fields than EC: 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 $q \approx 2^{53}$. There has been also research on securing implementations of HEC on embedded devices against differential and Goubin-type power analysis [2].

The purpose of this paper is to present 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 are* not *interested in comparing against* very special *classes of curves or in the use of prime moduli of special form.*

There have been several software implementations of HEC on personal computers and workstations. Most of those are in even characteristic (see [35, 32], [33], and also [40, 41]), but some are over prime fields [22, 35]. 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 [22] are based on general purpose software libraries, such as `gmp` [16]. These libraries introduce overheads which are quite significant for small operands such as those occurring in curve cryptography, and get worse as the fields get smaller. Moreover, `gmp` has no native support for fast modular reduction techniques. In our modular arithmetic library, described in § 2.1, we made every effort to avoid such overheads. On a PC we get a speed-up from 2 to 5 over `gmp` for operations in fields of cryptographic relevance (see Table 1). We also exploit techniques for reducing the number of modular reductions 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 14%, resp. 15%. The gap with genus 3 curves has been significantly reduced too. See Section 3 for more precise results.

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. In fact, a port of our library to the ARM processor has been recently finished and yields similar results.

In Section 2, we describe the implementation of the arithmetic library and of the group operations. In Section 3, we give timings and draw our conclusions.

Acknowledgements. The author thanks Gerhard Frey for his constant support. Tanja Lange greatly influenced this paper at several levels, theoretical and practical, considerably improving its quality. The author acknowledges feedback, support and material from Christophe Doche, Pierrick Gaudry, Johnny Großschädl, Christof Paar, Jean-Jacques Quisquater, Jan Pelzl, Nicolas Thériault and Thomas Wollinger. Thanks also to the anonymous referees for several useful suggestions.

2 Implementation

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 in the field. m and s denote a multiplication and a squaring, respectively, 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.

2.1 Prime Field Library

We already said that standard long integer software libraries introduce several types of overheads. One is the fixed function call overhead. Other ones arise from the processing of operands of variable length in loops, such as branch mispredictions at the beginning and end of the loops, and are negligible for very large operands. For operands of size relevant for curve cryptography the CPU will spend more time performing jumps and paying branch misprediction penalties than doing arithmetic. Memory management overheads can be very costly, too.

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.

2.1.1 Design.

Our software library nuMONGO has been designed to allow efficient reference implementations of EC and HEC over prime fields. It implements arithmetic operations in rings $\mathbb{Z}/N\mathbb{Z}$ with N odd, with the elements stored in Montgomery's representation [31], and the reduction algorithm is Montgomery's REDC function – see § 2.1.3 for some more details. Many optimization techniques employed are similar to those in [6].

nuMONGO is written in C++ to take advantage of inline functions, overloaded functions statically resolved at compile time for clarity of coding, and operator overloading for I/O only. All arithmetic operations are implemented as imperative functions. nuMONGO contains no classes. All data structures are minimalistic. All elements of $\mathbb{Z}/N\mathbb{Z}$ are stored in vectors of fixed length of 32-bit words. All temporary memory is allocated on the stack. No data structure is ever dynamically resized or relocated. This eliminates memory management overheads.

The routines aim to be as simple as possible. The least possible number of routines are implemented which still allow to perform all desired field operations: They are built from elementary operations working on single words, available as generic C macros as well as assembler macros for x86-compatible CPUs. A CPU able to process 32-bit operands is assumed, but not necessarily a 32-bit CPU – the library in fact compiled also on an Alpha. Inlining was used extensively, most loops are unrolled; there are very few conditional branches, hence branch mispredictions are rare. There are separate arithmetic 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-up over the 96 and 128-bit routines).

2.1.2 Multiplication.

We begin with two algorithms to multiply "smallish" multi-precision operands: Schoolbook multiplication and Comba's method [10].

The next two algorithms take as input two ℓ-word integers $u = (u_{\ell-1}, \ldots, u_1, u_0)$ and $v = (v_{\ell-1}, \ldots, v_0)$, and output the 2ℓ-word integer $r = (r_{2\ell-1}, \ldots, r_0)$ such that $r = uv$.

```
Schoolbook multiplication
$r_0 \leftarrow 0, \ldots, r_{2\ell-1} \leftarrow 0$
for $i$ from 0 to $\ell-1$ do {
    $c \leftarrow 0$
    for $j$ from 0 to $\ell-1$ do {
        $(c, r_{j+i}) \leftarrow u_i v_j + r_{j+i} + c$ }
    $r_{i+\ell} \leftarrow c$ }
return($r$)
```

```
Comba's method
$s_0 \leftarrow 0$, $s_1 \leftarrow 0$, $s_2 \leftarrow 0$
for $k$ from 0 to $2(\ell-1)$ do {
    for each pair $(i, j)$ such that $i + j = k$
            and $0 \leqslant i, j < \ell$, do {
        $(s_2, s_1, s_0) \leftarrow (s_2, s_1, s_0) + u_i v_j$ }
    $r_k \leftarrow s_0$, $s_0 \leftarrow s_1$, $s_1 \leftarrow s_2$, $s_2 \leftarrow 0$ }
$r_{2\ell-1} \leftarrow s_0$
return($r$)
```

The schoolbook method multiplies the first factor by each digit of the second factor, and accumulates the results. Comba's method instead, for each digit of the result, say the k^{th} one, computes the partial products $u_i v_j$ on the *diagonals* $i + j = k$, adding these double precision results to a triple precision accumulator. It requires fewer memory writes and more reads than the schoolbook multiplication. This is the method adopted in [6]. For both methods, several optimizations have been done. They can both be used with Karatsuba's trick [20].

In our experience, Comba's method did not perform better than the schoolbook method (on the ARM the situation is different). This may be due to the fact that the Athlon CPU has a write-back level 1 cache [1], hence several close writes to the same memory location cost little more than one generic write. For $n = 192$ and $n = 256$ we reduced a n-bit multiplication to three $n/2$-bit multiplications by means of Karatsuba's trick. For smaller sizes and for 224-bit operands, the schoolbook method was still faster.

For the considered operand sizes, squaring routines did not bring a significant improvement over the multiplication routines, hence they were not included.

2.1.3 Montgomery's reduction without trial division.

Montgomery [31] proposed to speed up modular reduction by replacing the modulus N by a larger integer R coprime to N for which division is faster. In practice, if β is the machine radix (in our case $\beta = 2^{32}$) and N is an odd ℓ-word integer, then $R = \beta^{\ell}$. Division by R and taking remainder are just shift and masking operations.

Let $\mathtt{REDC}(x)$ be a function which, for $0 \leqslant x < NR$, computes $xR^{-1} \bmod N$.

The modular residue x is represented by its *r-sidu* $\overline{x} = xR \bmod N$. Addition, subtraction, negation and testing for equality are performed on r-sidus as usual.

Note that $x = \mathtt{REDC}(\overline{x})$. To get the r-sidu of an integer x, compute $\mathtt{REDC}(xR^2)$, hence $R^2 \bmod N$ should be computed during system initialization. Now $\overline{x}\,\overline{y} \equiv xRyR \equiv$

$\overline{xy}R \bmod N$, so $\overline{xy} = \mathtt{REDC}(xy)$ can be computed without any division by N. We implemented REDC by the following method [5], which requires the inverse n'_0 of N modulo the machine radix $\beta = 2^{32}$.

```
Function REDC(x)
INPUT: A $2\ell$-word integer $x = (x_{2\ell-1}, \ldots, x_1, x_0)$, and $N$, $n'_0$ and $\beta$ as above.
OUTPUT: The $\ell$-word integer $y$ such that $y = xR^{-1} \bmod N$ and $0 \leqslant y < N$.
$y = (y_{2\ell-1}, \ldots, y_1, y_0) \leftarrow x$
for $i$ from 0 to $\ell - 1$ do {
    $t \leftarrow y_0 \cdot n'_0 \bmod \beta$,   $y \leftarrow y + t \cdot N$,   $y \leftarrow y \div \beta$ }
if $y \geqslant N$ then $y \leftarrow y - N$
return($y$)
```

This algorithm is, essentially, Hensel's odd division for computing inverses of 2-adic numbers to a higher base: At each iteration of the loop, a multiple of N is added to y such that the result is divisible by β, and then y is divided by β (a one word shift). After the loop, $y \equiv x/\beta^\ell \equiv xR^{-1} \bmod N$ and $y < 2N$. If $y \geqslant N$, a subtraction corrects the result. The cost of REDC is, at least in theory, that of a schoolbook multiplication of ℓ-word integers, some shifts and some additions; In practice it is somewhat more expensive, but still much faster than the naive reduction involving long divisions. We did not use the interleaved multiplication with reduction [31]: It usually performs better on DSPs [11], but not on general-purpose CPUs with few registers.

2.1.4 Inversion. With the exception of 32-bit operands, inversion is based on the extended binary GCD, and uses an almost-inverse technique [19] with final multiplication from a table of precomputed powers of 2 mod N. This was the fastest approach up to about 192 bits. For 32-bit operands we got better performance with the extended Euclidean algorithm and special treatment of small quotients to avoid divisions. Inversion was not sped up further for larger input sizes because of the intended usage of the library: For elliptic curves over prime fields, 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. For hyperelliptic curves, fields are quite small (32 to 128 bits in most cases), hence our inversion routines have optimal performance anyway. Therefore, Lehmer's method or the improvements by Jebelean [18] or Lercier [26] have not been included in the final version of the library.

2.1.5 Performance. In Table 1 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, under the Linux operating system (kernel version 2.4). Our programs have been compiled with the GNU C Compiler (gcc) versions 2.95.3 and 3.3.1. For each test, we took the version that gave the best timings. nuMONGO always performed best with gcc 3.3.1, whereas some gmp tests performed better with gcc 2.95.3[1]. We describe the meaning of the entries. There are two groups of rows, grouped

[1] In some cases gcc 2.95.3 produced the fastest code when optimizing nuMONGO for size (-Os), not for speed! This seems to be a strange but known phenomenon. gcc 3.3.1 had a more orthodox behavior and gave the best code with -O3, i.e. when optimizing aggressively for speed. In both cases, additional compiler flags were used for fine-tuning.

Table 1. *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`	`m`	0.0079	0.0201	0.0267	0.054	0.11	0.146	0.198	0.361	0.392
	`R`	0.0298	0.0536	0.0487	0.097	0.159	0.241	0.319	0.416	0.493
	`I`	0.61	1.85	1.987	4.457	7.6	11.2	16.3	22.3	28.8
	`R/m`	3.77	2.667	1.824	1.796	1.445	1.651	1.61	1.152	1.258
	`I/(R+m)`	16.19	25.102	26.35	29.52	28.25	28.94	31.53	28.7	32.55
`gmp` v. 4.1	`m`	0.094	0.155	0.16	0.206	0.238	0.308	0.354	0.44	0.508
	`R`	0.234	0.419	0.423	0.65	0.81	0.986	1.154	1.264	1.528
	`I`	2.53	4.74	6.41	9.77	13.3	17.2	21.26	25.84	29.6
	`R/m`	2.489	2.703	2.644	3.155	3.403	3.201	3.26	2.873	3.008
	`I/(R+m)`	7.713	8.258	10.99	11.41	12.69	13.29	14.1	15.16	14.54

under the name of library used to benchmark the following operations: multiplication of two integers (`m`), modular or Montgomery reduction (`R`), modular or Montgomery inversion (`I`). The ratios of a reduction to a multiplication and of an inversion to the time of a multiplication together with a reduction are given, too: 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 field inversion in field multiplications. The columns correspond to the bit lengths of the operands. A few remarks:

1. `nuMONGO` can perform better than a far more optimized, but general purpose library. In fact, the kernel of `gmp` is entirely written in assembler for most architectures, including the one considered here.
2. For larger operands `gmp` catches up with `nuMONGO`, the modular reduction remaining slower because it is not based on Montgomery's algorithm.
3. `nuMONGO` has a higher `I/(m+R)` ratio than `gmp`. This shows how big the overheads in general purpose libraries are for such small inputs.

2.2 Lazy and Incomplete reduction

Lazy and incomplete modular reduction are described in [3]. Here, we give a short treatment. Let $p < 2^w$ be a prime, where w is a fixed integer. We consider expressions of the form $\sum_{i=1}^{d} a_i b_i \bmod p$ with $0 \leqslant a_i, b_i < p$. Such expressions occur in the explicit formulae for HEC. To use most modular reduction algorithms, including Montgomery's, at the *end* of the summation, we have to make sure that all partial sums of $\sum a_i b_i$ are smaller than $p\,2^w$. Some authors (for example [27]) suggested to use *small* primes, to guarantee that the condition $\sum a_i b_i < p\,2^w$ is always satisfied. Note that [27] exploited the possibility of accumulating several partial results before reduction for the extension field arithmetic, but not at the group operation level. The use of small primes at the group operation level has been considered also in [14] after the present paper appeared as a preprint. However, "just" using primes which are "small enough" would contradict one of our design principles, which is to have no restriction on p except its word length.

What we do, additionally, 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 $\geqslant p$: if so we subtract p from the number obtained ignoring the w least significant bits of the intermediate result. If the intermediate result is $\geqslant 2^{2w}$, the additional bit can be stored in a carry. Since all intermediate results are bounded by $p2^{w+1} < (p+2^w)2^w$, upon subtraction of $p\,2^w$ the carry will always be zero. This requires as many operations as allowing intermediate results in triple precision, but less memory accesses are needed: In practice this leads to a faster approach, and at the end we have to reduce a number x bounded by $p\,2^w$, making the modular reduction easier.

This technique of course works with any *classical modular reduction algorithm. That it works with Montgomery's r-sidus and with* `REDC` *is a consequence of the linearity of the operator* `REDC` *modulo p.*

`nuMONGO` supports Lazy (i.e. delayed) and Incomplete (i.e. limited to the number obtained by removing the least significant w bits) modular reduction. Thus, an expression of the form $\sum_{i=0}^{d-1} a_i b_i \bmod p$ can be evaluated by d multiplications but only one modular reduction instead of d. A modular reduction is at least as expensive as a multiplication, and often much more, see Table 1.

Remark 1. We *cannot* add a reduced element to an unreduced element in Montgomery's representation. In fact, Montgomery's representation $\overline{a}$ of the integer a is $aR \bmod p$ (R as in § 2.1.3 with $N = p$). Now, $\overline{b}\overline{c}$ is congruent to $bcR^2 \bmod p$, not to $\overline{bc} = bcR \bmod p$. Hence, a and bc have been multiplied by different constants $\bmod p$ to obtain $\overline{a}$ and $\overline{b}\overline{c}$, and $\overline{a} + \overline{b}\overline{c}$ bears no fixed relation to $a + bc$.

2.3 Implementation of the Explicit Formulae

We assume that the reader is acquainted with elliptic and hyperelliptic curves.

2.3.1 Elliptic Curves. We consider elliptic curves defined over a field F of odd characteristic greater than 3 given by a Weierstrass equation

$$E \ : \ y^2 = x^3 + a_4 x + a_6 \tag{1}$$

where the polynomial $x^3 + a_4 x + a_6$ has no multiple roots. The set of points of E over (any extension of) the field F and the point at infinity O form a group.

There are 5 different coordinate systems [9]: *affine* ($\mathcal{A}$), the finite points "being" the pairs (x,y) that satisfy (1); *projective* ($\mathcal{P}$), also called *homogeneous*, where a point $[X,Y,Z]$ corresponds to $(X/Z, Y/Z)$ in affine coordinates; *Jacobian* ($\mathcal{J}$), where a point (X,Y,Z) corresponds to $(X/Z^2, Y/Z^3)$; and two variants of $\mathcal{J}$, namely, *Chudnowski Jacobian* ($\mathcal{J}^c$), with coordinates (X,Y,Z,Z^2,Z^3), and *modified Jacobian* ($\mathcal{J}^m$), with coordinates (X,Y,Z,a_4Z^4). They are accurately described in [9], where the formulae for all group operations are given. It is possible to add two points in any two different coordinate systems and get a result in a third system. For example, when doing a scalar multiplication, it is a good idea to keep the base point and all precomputed points in $\mathcal{A}$, since adding those points will be less expensive than using other coordinate systems.

Table 2. *Costs of Group Operations on EC and HEC*

	Doubling		**Addition**			
	operation	costs	operation	costs	operation	costs
EC	$2\mathcal{A} = \mathcal{A}$	I, 2m, 2s, 4R	$\mathcal{A}+\mathcal{A} = \mathcal{A}$	I, 2m, 1s, 3R		
	$2\mathcal{P} = \mathcal{P}$	7m, 5s, 10R	$\mathcal{P}+\mathcal{P} = \mathcal{P}$	12m, 2s, 13R	$\mathcal{A}+\mathcal{P} = \mathcal{P}$	9m, 2s, 10R
	$2\mathcal{J} = \mathcal{J}$	4m, 6s, 8R	$\mathcal{J}+\mathcal{J} = \mathcal{J}$	12m, 4s, 16R	$\mathcal{A}+\mathcal{J} = \mathcal{J}$	8m, 3s, 11R
	$2\mathcal{J}^c = \mathcal{J}^c$	5m, 6s, 9R	$\mathcal{J}^c+\mathcal{J}^c = \mathcal{J}^c$	11m, 3s, 14R	$\mathcal{A}+\mathcal{J}^c = \mathcal{J}^c$	8m, 3s, 11R
	$2\mathcal{J}^m = \mathcal{J}^m$	4m, 4s, 8R	$\mathcal{J}^m+\mathcal{J}^m = \mathcal{J}^m$	13m, 6s, 19R	$\mathcal{A}+\mathcal{J}^m = \mathcal{J}^m$	9m, 5s, 14R
g=2	$2\mathcal{A} = \mathcal{A}$	I, 22m, 5s, 22R	$\mathcal{A}+\mathcal{A} = \mathcal{A}$	I, 22m, 3s, 18R		
	$2\mathcal{P} = \mathcal{P}$	38m, 6s, 38R	$\mathcal{P}+\mathcal{P} = \mathcal{P}$	45m, 5s, 42R	$\mathcal{A}+\mathcal{P} = \mathcal{P}$	40m, 3s, 33R
	$2\mathcal{N} = \mathcal{N}$	34m, 7s, 37R	$\mathcal{N}+\mathcal{N} = \mathcal{N}$	47m, 7s, 50R	$\mathcal{A}+\mathcal{N} = \mathcal{N}$	36m, 5s, 37R
g=3	$2\mathcal{A} = \mathcal{A}$	I, 71(m/s), 57R	$\mathcal{A}+\mathcal{A} = \mathcal{A}$	I, 76(m/s), 55R		

For EC, only few savings in REDCs are possible.

Let us work out an example, namely, how many REDCs can be saved in the addition $\mathcal{A}+\mathcal{P} = \mathcal{P}$. Let $P_1 = (X_1, Y_1)$, $P_2 = [X_2, Y_2, Z_2]$ and $P_3 = [X_3, Y_3, Z_3]$. Then, $P_3 = P_1 + P_2$ is computed as follows [9]:

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

For the computation of u and v no savings are possible. We cannot save any reductions in the computation of $A = u^2 Z_2 - v^3 - 2v^2 X_1 Z_2$ because: We need v^3 reduced anyway for Z_3, A must be available also in reduced form to compute X_3, and from $v^2 X_1 Z_2$ we subtract A in the computation of Y_3; It is then easy to see that here no gain is obtained by delaying reduction. But Y_3 can be computed by first multiplying u by $v^2 X_1 Z_2 - A$, then v^3 by $Y_1 Z_2$, adding these two products and reducing the sum. Hence, one REDC can be saved in the addition formula.

For affine coordinates, no REDCs can be saved. Additions in $\mathcal{P}$ allow saving of 1 REDC, even if one of the two points is in $\mathcal{A}$. With no other addition formula we can save reductions. For all doublings we can save 2 REDCs, except for the doubling in $\mathcal{J}^m$, where no savings can be done due to the differences in the formulae depending on the introduction of $a_4 Z^4$.

In Table 2, we write the operation counts of the implemented operations. Results for genus 2 and 3 curves are given, too. The shorthand $\mathcal{C}_1 + \mathcal{C}_2 = \mathcal{C}_3$ means that two points in the coordinate systems $\mathcal{C}_1$ and $\mathcal{C}_2$ are added and the result is given in $\mathcal{C}_3$, where any of the $\mathcal{C}_i$ can be one of the applicable coordinate systems. Doubling a point in $\mathcal{C}_1$ with result in $\mathcal{C}_2$ is denoted by $2\mathcal{C}_1 = \mathcal{C}_2$. The number of REDCs is given separately from the multiplications and squarings.

2.3.2 Hyperelliptic Curves.

An excellent, low brow, introduction to hyperelliptic curves is given in [28], including proofs of the facts used below.

A hyperelliptic curve $\mathcal{C}$ of genus g over a finite field $\mathbb{F}_q$ of odd characteristic is defined by a Weierstrass equation $y^2 = f(x)$, where f is a monic, square-free polynomial of degree $2g+1$. In general, the points on $\mathcal{C}$ do *not* form a group. Instead, the *ideal class group* is used, which is isomorphic to the Jacobian variety of $\mathcal{C}$. Its elements are represented by pairs of polynomials and [7] showed how to compute with group elements in this form. A generic ideal class is represented by a pair of polynomials $U(x) = x^g + \sum_{i=0}^{d-1} U_i x^i,\ V(x) = \sum_{i=0}^{d-1} V_i x^i \in \mathbb{F}_q[x]$ such that for each root ξ of $U(x)$, $(\xi, V(\xi))$ is a point on $\mathcal{C}$ (equivalently, $U(x)$ divides $V(x)^2 - f(x)$). The *affine coordinates* are the $2g$-tuple $[U_{g-1}, \ldots, U_1, U_0, V_{g-1}, \ldots, V_1, V_0]$.

2.3.2.1 Genus 2. For genus 2 there are two more coordinate systems besides affine ($\mathcal{A}$): in *projective coordinates* ($\mathcal{P}$) [30]: a quintuple $[U_1, U_0, V_1, V_0, Z]$ corresponds to the ideal class represented by $[x^2 + U_1/Z\,x + U_0/Z, V_1/Z\,x + V_0/Z]$; with Lange's *new* coordinates ($\mathcal{N}$) [24], the sextuple $[U_1, U_0, V_1, V_0, Z_1, Z_2]$ corresponds to the ideal class $[x^2 + U_1/Z_1^2\,x + U_0/Z_1^2, V_1/Z_1^3 Z_2\,x + V_0/Z_1^3 Z_2]$. The system $\mathcal{N}$ is important in scalar multiplications since it has the fastest doubling. We refer to [24] for the formulae.

Table 3. *Addition in genus* 2, $\deg u_1 = \deg u_2 = 2$

INPUT: $[u_1, v_1], [u_2, v_2]$, with $\deg u_1 = \deg u_2 = 2$, and $f = x^5 + f_3 x^3 + f_2 x^2 + f_1 x + f_0$
OUTPUT: $[u_3, v_3] = [u_1, v_1] + [u_2, v_2]$
NOTATION: $u_i = x^2 + u_{i1} x + u_{i0}$ and $v_i = v_{i1} x + v_{i0}$

Step	Expression	Cost
1	compute resultant r of u_1, u_2: $z_1 = u_{11} - u_{21}$, $z_2 = u_{20} - u_{10}$, $z_3 = u_{11} z_1 + z_2$; $r = z_2 z_3 + z_1^2 u_{10}$;	1S, 3M
2	compute almost inverse of u_2 modulo u_1 ($\imath = \imath_1 x + \imath_0 = r/u_2 \bmod u_1$): $\imath_1 = z_1$, $\imath_0 = z_3$;	-
3	compute $s' = rs \equiv (v_1 - v_2)\imath \bmod u_1$: $w_0 = v_{10} - v_{20}$, $w_1 = v_{11} - v_{21}$, $w_2 = \imath_0 w_0$, $w_3 = \imath_1 w_1$; $s'_1 = (\imath_0 + \imath_1)(w_0 + w_1) - w_2 - w_3(1 + u_{11})$, $s'_0 = w_2 - u_{10} w_3$; 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: $w_1 = (rs'_1)^{-1} (= 1/r^2 s_1)$, $w_2 = r w_1 (= 1/s'_1)$, $w_3 = s'^2_1 w_1 (= s_1)$; $w_4 = r w_2 (= 1/s_1)$, $w_5 = w_4^2$; $s''_0 = s'_0 w_2$;	I, 2S, 5M
5	compute $l' = s'' u_2 = x^3 + l'_2 x^2 + l'_1 x + l'_0$: $l'_2 = u_{21} + s''_0$, $l'_1 = u_{21} s''_0 + u_{20}$, $l'_0 = u_{20} s''_0$	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 $v_3 \equiv -(l + v_2) \bmod u_3$: $w_1 = l'_2 - u_{31}$, $w_2 = u_{31} w_1 + u_{30} - l'_1$, $v_{31} = w_2 w_3 - v_{21}$; $w_2 = u_{30} w_1 - l'_0$, $v_{30} = w_2 w_3 - v_{20}$;	4M
total		I, 3S, 22M

We now see in an example – the addition formula in affine coordinates – how lazy and incomplete reductions are used in practice. Table 3 is derived from results in [24], but restricted to the odd characteristic case. The detailed breakdown of the REDCs we can save follows:

1. In Step 1 we can save one REDC in the computation of r, since we do not need the reduced value of $z_2 z_3$ and $z_1^2 u_{10}$ anywhere else.

2. In Step 3 we do not reduce $w_2 = \iota_0 w_0$, 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 two `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 1). We circumvent this by computing the unreduced products $L_1 = u_{21}s''_0$ (in place of ℓ'_1) and $L_0 = u_{20}s''_0$. Two `REDC`s are saved.
4. In Step 6, it is $u_{30} = (s''_0 - u_{11})(s''_0 - z_1) + L_1 + 2v_{21}w_4 + (2u_{21} + z_1)w_5 + z_2$. We need only one `REDC` to compute the (reduced) sum of the first four products: Note that, at this point, L_1 is already known and we already counted the saving of one `REDC` associated to it. So, we save a total of two `REDC`s.

Summarizing, for one addition 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. To speed up scalar multiplication, we also implemented addition in the cases where one of the two group elements to be added is given in $\mathcal{A}$ and the second summand and the result are both given either in $\mathcal{P}$ or $\mathcal{N}$.

In Table 2 we write the operation counts of the implemented operations. The table contains also the counts for EC and genus 3 curves (see the next paragraph). The number of modular reductions is always significantly smaller than the number of multiplications.

2.3.2.2 Genus 3. Affine coordinates are the only coordinate system currently available for genus 3 curves. The formulae in [32, 33] contain some errors in odd characteristic. We took the formulae of [40] – which are for general curves of the form $y^2 + h(x)y = f(x)$, and have been implemented only in even characteristic with $h(x) = 1$ – and simplified them for the case of odd characteristic, $h(x) = 0$, and vanishing second most significant coefficient of $f(x)$. A pleasant aspect of these 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 2).

2.4 Scalar Multiplication

There are many methods for computing a scalar multiplication in a generic group, which can be used for EC and HEC. See [15] for a survey.

A simple method for computing $s \cdot D$ for an integer s and a ideal class 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

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

This requires $n-1$ doublings and on average $n/2-1$ additions on the curve (the first addition is replaced by an assignment).

On EC and HEC, adding and subtracting an element have the same cost. Hence one can use the *non adjacent form* (NAF) [34], which is an expansion $s = \sum_{i=0}^{n} s_i 2^i$ with $s_i \in \{0, \pm 1\}$ and $s_i s_{i+1} = 0$. This leads to a method needing n doublings and on average $n/3 - 1$ additions or subtractions.

A generalization of the NAF uses "sliding windows": The wNAF [37, 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 $|s_j| \leqslant 2^w$; (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 first precomputes the ideal classes D, $3D, \ldots, (2^w - 1)D$, and then performs a double-and-add step like (2). A left-to-right recoding with the same density as the wNAF can be found in [4].

3 Results, Comparisons and Conclusions

Table 4 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 513 bits is possible: We choose this group size as a limit also for the 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). The compilers used were the GNU C Compiler (`gcc`), versions 2.95.3 and 3.3.1 and all the performance considerations made in § 2.1.5 apply.

All groups have prime or almost prime order. The elliptic curves up to 256 bits have been found by point counting on random curves, the larger ones as well as the genus 2 and 3 curves have been constructed with the CM method.

For each combination of curve type, coordinate system and group size, we averaged the timings of several thousands scalar multiplications with random scalars, 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 corresponding value of w. We always keep the base ideal class *and* its multiples in affine coordinates, since adding an affine point to a point in *any* coordinate system other than affine is faster than adding two points in that coordinate system. The timings always include the precomputations.

In Table 5 we provide timings for `ecc` and `hec` using `gmp` and the double-and-add scalar multiplication based on the unsigned binary representation. We also provide in Table 6 timings with `nuMONGO` but without lazy and incomplete reduction. For comparison with our timings, Lange [23] 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 1.5 GHz Pentium 4 PC. In [23] the double-and-add algorithm based on the unsigned binary representation is used. In [35], 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.

A summary of the results follows:

Table 4. *Comparison of running times, in msec (1 GHz AMD Athlon PC)*

curve	coord.	scalar mult.	Bitlength of group order (approximate)							
			128	144	160	192	224	256	320	512
`ec`	$\mathcal{A}$	binary	1.671	2.521	3.074	5.385	8.536	12.619		
		NAF	1.488	2.252	2.701	4.809	7.596	11.315		
		*w*NAF	1.363 $(w=4)$	2.205 $(w=3)$	2.489 $(w=4)$	4.335 $(w=4)$	6.841 $(w=4)$	10.099 $(w=4)$		
	$\mathcal{P}$	binary	0.643	0.94	1.152	1.879	3.22	4.243		
		NAF	0.575	0.841	1.017	1.685	2.881	3.747		
		*w*NAF	0.551 $(w=3)$	0.808 $(w=3)$	0.982 $(w=3)$	1.591 $(w=3)$	2.711 $(w=4)$	3.523 $(w=4)$		
	$\mathcal{J}$	binary	0.584	0.856	1.05	1.702	2.912	3.876		
		NAF	0.517	0.776	0.907	1.499	2.558	3.325		
		*w*NAF	0.492 $(w=3)$	0.713 $(w=3)$	0.864 $(w=3)$	1.397 $(w=3)$	2.357 $(w=3)$	3.086 $(w=4)$		
	$\mathcal{J}^c$	binary	0.614	0.901	1.109	1.812	3.081	3.995		
		NAF	0.546	0.802	0.965	1.6	2.727	3.583		
		*w*NAF	0.517 $(w=3)$	0.756 $(w=3)$	0.922 $(w=3)$	1.499 $(w=3)$	2.527 $(w=3)$	3.275 $(w=3)$		
	$\mathcal{J}^m$	binary	0.607	0.872	1.076	1.782	3.005	3.945		
		NAF	0.512	0.748	0.906	1.515	2.592	3.35		
		*w*NAF	0.474 $(w=3)$	0.684 $(w=3)$	0.838 $(w=3)$	1.395 $(w=3)$	2.296 $(w=3)$	3.048 $(w=3)$		

`hec`
`g=2`
	$\mathcal{A}$	binary	0.888	1.614	1.899	2.546	4.612	5.514	10.409	39.673

		NAF	0.797	1.449	1.706	2.265	4.139	4.952	9.298	35.430
		*w*NAF	0.73 $(w=4)$	1.421 $(w=4)$	1.558 $(w=4)$	2.053 $(w=4)$	3.73 $(w=4)$	4.464 $(w=4)$	8.343 $(w=4)$	31.246 $(w=5)$
	$\mathcal{P}$	binary	0.839	1.473	1.642	2.102	3.996	4.712	8.653	30.564
		NAF	0.755	1.325	1.48	1.901	3.588	4.203	7.758	27.359
		*w*NAF	0.703 $(w=4)$	1.211 $(w=4)$	1.352 $(w=4)$	1.742 $(w=4)$	3.256 $(w=4)$	3.842 $(w=4)$	6.998 $(w=4)$	24.451 $(w=5)$
	$\mathcal{N}$	binary	0.844	1.395	1.564	2.038	3.777	4.413	8.265	29.11
		NAF	0.746	1.247	1.391	1.778	3.357	4.002	7.329	25.816
		*w*NAF	0.675 $(w=4)$	1.14 $(w=4)$	1.262 $(w=4)$	1.623 $(w=3)$	3.02 $(w=4)$	3.575 $(w=4)$	6.53 $(w=4)$	22.73 $(w=5)$

`hec`
`g=3`
	$\mathcal{A}$	binary	1.896	1.984	2.992	3.597	5.39	6.001	12.66	42.907

		NAF	1.64	1.744	2.538	3.085	4.82	5.39	11.24	38.326
		*w*NAF	1.424 $(w=4)$	1.528 $(w=4)$	2.077 $(w=5)$	2.584 $(w=5)$	4.33 $(w=4)$	4.86 $(w=5)$	9.92 $(w=4)$	34.117 $(w=4)$

Table 5. *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 6. *Timings with* `nuMONGO` *without lazy and incomplete reduction, in msec (1 GHz AMD Athlon PC)*

`ec`	160	192	256
$\mathcal{A}$	3.074	5.385	12.619
$\mathcal{P}$	1.227	2.041	4.541
$\mathcal{J}$	1.109	1.829	4.069
$\mathcal{J}^c$	1.176	1.939	4.292
$\mathcal{J}^m$	1.076	1.782	3.945

`hec`		160	192	256	320	512
g=2	$\mathcal{A}$	2.234	2.708	5.788	11.112	41.691
	$\mathcal{P}$	2.02	2.352	4.894	9.415	33.23
	$\mathcal{N}$	1.831	2.113	4.494	8.731	30.653
g=3	$\mathcal{A}$	4.469	5.184	6.52	13.54	47.372

1. Using a specialized 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 reduction bring a speed-up from 3% to 10%.
3. For EC, the performance of the systems $\mathcal{J}$ and $\mathcal{J}^m$ is almost identical. The reason lies in the fact that with $\mathcal{J}^m$ no modular reductions can be saved.
4. HEC 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 groups, EC is only 14%, resp. 15% faster than HEC. In fact, consider the best timings for EC and HEC with genus 2 with 192 bits: $(1.623 - 1.395)/1.623 = 0.1405 \approx 14\%$.
      (ii) For other group sizes the difference is often around 50%.
   (c) Genus 3 curves are slower than genus 2 ones. With `gmp` the difference is 80% to 100% for 160 to 512 bit groups, but using `nuMONGO` the gap is often as small as 50%.
5. Using `nuMONGO` we can successfully eliminate most of the overheads, thus proving the soundness of our approach.
   (a) In the `gmp`-based implementation, 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 have the paradoxical result that $\mathcal{P}$ and $\mathcal{N}$ are slower than $\mathcal{A}$, because they require more function calls for each group operation than $\mathcal{A}$. Therefore, with standard libraries the overheads can dominate the running time.
   (b) For affine coordinates the most expensive 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 1 for the inversion alone.
6. If the *field* size for a given group is not close to a multiple of the machine word size b, there is a relative drop in performance with respect to other groups where the field size is almost a multiple of b. For example, a 160-bit group can be given by a genus 2 curve over a 80-bit field, but then 96-bit arithmetic must be used on a 32-bit CPU. Similarly, with 224-bit groups, a genus 2 HEC is penalized by the 112-bit field arithmetic. 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), hence the gap to genus 2 is only 50%.

We conclude that the performance of hyperelliptic curves over prime fields is satisfactory enough to be considered as a valid alternative to elliptic curves, especially when large point groups are desired, and the bit length of the characteristic is close to (but smaller than) a multiple of the machine word length.

In software implementations not only should we employ a custom software library, as done for elliptic curves in [6], but for a further speed-up the use of lazy and incomplete reduction is recommended. Development of new explicit formulae should take into account the possibility of delaying modular reductions.

References

1. AMD Corporation. *AMD-K6-2 Processor Data Sheet*.
   `http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/21850.pdf`
2. R.M. Avanzi. *Countermeasures against differential power analysis for hyperelliptic curve cryptosystems*. Proc. CHES 2003. LNCS 2779, 366–381. Springer, 2003.
3. R.M. Avanzi and P.M. Mihăilescu. *Generic Efficient Arithmetic Algorithms for PAFFs (Processor Adequate Finite Fields) and Related Algebraic Structures*. Proc. SAC 2003. LNCS 3006, 320–334. Springer 2004.
4. R.M. Avanzi. *A note on the sliding window integer recoding and its left-to-right analogue*. Submitted.
5. A. Bosselaers, R. Govaerts and J. Vandewalle. *Comparison of three modular reduction functions*. Proc. Crypto '93. LNCS 773, 175-186. Springer, 1994.
6. M.K. Brown, D. Hankerson, J, Lopez and A. Menezes. *Software implementation of the NIST elliptic curves over prime fields.* Proc. CT-RSA 2001. LNCS 2020, 250–265. Springer, 2001.
7. D. Cantor. *Computing in the Jacobian of a Hyperelliptic Curve.* Math. Comp. **48** (1987), 95–101.
8. H. Cohen, A. Miyaji and T. Ono. *Efficient elliptic curve exponentiation*. Proc. ICICS 1997, LNCS 1334, 282–290. Springer, 1997.
9. H. Cohen, A. Miyaji and T. Ono. *Efficient Elliptic Curve Exponentiation Using Mixed Coordinates*, Proc. ASIACRYPT 1998. LNCS 1514, 51–65. Springer, 1998.
10. P.G. Comba. *Exponentiation cryptosystems on the IBM PC.* IBM Systems Journal, **29** (Oct. 1990), 526–538.
11. S.R. Dussé and B.S. Kaliski. *A cryptographic library for the Motorola DSP56000*. Proc. EUROCRYPT '90. LNCS 473, 230–244. Springer, 1991.
12. P. Gaudry, *An algorithm for solving the discrete log problem on hyperelliptic curves*. Proc. EUROCRYPT 2000. LNCS 1807, 19–34. Springer, 2000.
13. P. Gaudry and E. Schost. *Construction of Secure Random Curves of Genus 2 over Prime Fields*. Proc. EUROCRYPT 2004. LNCS 3027, 239–256. Springer, 2004.
14. M. Gonda, K. Matsuo, K. Aoki, J. Chao, and S. Tsuji. *Improvements of addition algorithm on genus 3 hyperelliptic curves and their implementations*. Proc. SCIS 2004, 995–1000.
15. D.M. Gordon. *A survey of fast exponentiation methods*. J. of Algorithms **27** (1998), 129–146.
16. T. Grandlund. *GMP. A software library for arbitrary precision integers*.
    Available from: `http://www.swox.com/gmp/`
17. R. Harley. *Fast Arithmetic on Genus Two Curves.*
    Available at `http://cristal.inria.fr/~harley/hyper/`
18. T. Jebelean. *A Generalization of the Binary GCD Algorithm.* Proc. ISSAC 1993, 111–116.
19. B.S. Kaliski Jr.. *The Montgomery inverse and its applications*. IEEE Transactions on Computers, 44(8), 1064–1065, August 1995.
20. A. Karatsuba and Y. Ofman. Multiplication of Multidigit Numbers on Automata, *Soviet Physics - Doklady*, **7** (1963), 595–596.
21. N. Koblitz. *Hyperelliptic Cryptosystems*. J. of Cryptology **1** (1989), 139–150.
22. U. Krieger. `signature.c`: *Anwendung hyperelliptischer Kurven in der Kryptographie*. M.S. Thesis, Mathematik und Informatik, Universität Essen, Fachbereich 6, Essen, Germany.
23. 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/`
24. T. Lange. *Formulae for Arithmetic on Genus 2 Hyperelliptic Curves.* To appear in: J. AAECC.
25. A.K. Lenstra and E.R. Verheul. *Selecting Cryptographic Key Sizes*. J. of Cryptology **14** (2001), 255–293.

26. R. Lercier. *Algorithmique des courbes elliptiques dans les corps finis.* These. Available from `http://www.medicis.polytechnique.fr/~lercier/`
27. C.H. Lim, H.S. Hwang. *Fast implementation of Elliptic Curve Arithmetic in GF(p^m)*. Proc. PKC 2000, LNCS 1751, 405–421. Springer 2000.
28. A. Menezes, Y.-H. Wu and R. Zuccherato. *An Elementary Introduction to Hyperelliptic Curves.* In N. Koblitz, *Algebraic aspects of cryptography.* Springer, 1998.
29. J.-F. Mestre. *Construction des courbes de genre 2 a partir de leurs modules.* Progr. Math. **94** (1991), 313–334.
30. Y. Miyamoto, H. Doi, K. Matsuo, J. Chao, and S. Tsuji. *A Fast Addition Algorithm of Genus Two Hyperelliptic Curve.* Proc. SCIS 2002, IEICE Japan, 497–502, 2002. In Japanese.
31. P.L. Montgomery. *Modular multiplication without trial division.* Math. Comp. **44** (1985), 519–521.
32. J. Pelzl. *Fast Hyperelliptic Curve Cryptosystems for Embedded Processors*. Master's Thesis. Dept. of Elec. Eng. and Infor. Sci., Ruhr-University of Bochum, 2002.
33. J. Pelzl, T. Wollinger, J. Guajardo, J. and C. Paar. *Hyperelliptic Curve Cryptosystems: Closing the Performance Gap to Elliptic Curves*. CHES 2003, LNCS 2779, 351–365. Springer, 2003.
34. G.W. Reitwiesner. *Binary arithmetic.* Advances in Computers **1** (1960), 231–308.
35. Y. Sakai, and K. Sakurai. *On the Practical Performance of Hyperelliptic Curve Cryptosystems in Software Implementation.* IEICE-Tran. Fund. Elec., Comm. and Comp. Sci. Vol. E83-A No.4., 692–703.
36. N.P. Smart. *On the Performance of Hyperelliptic Cryptosystems*. Proc. EUROCRYPT '99, LNCS 1592, 165–175. Springer, 1999.
37. J.A. Solinas. *An improved algorithm for arithmetic on a family of elliptic curves.* Proc. CRYPTO '97, LNCS 1294, 357–371. Springer, 1997.
38. N. Thériault. *Index calculus attack for hyperelliptic curves of small genus.* Proc. Asiacrypt 2003. LNCS 2894, 75–92. Springer, 2003.
39. A. Weng. *Konstruktion kryptographisch geeigneter Kurven mit komplexer Multiplikation*. PhD thesis, Universität Gesamthochschule Essen, 2001.
40. T. Wollinger, J. Pelzl, V. Wittelsberger, C. Paar, G. Saldamli, and Ç.K. Koç. *Elliptic & Hyperelliptic Curves on Embedded μP.* Special issue on Embedded Systems and Security of the ACM Transactions in Embedded Computing Systems.
41. T. Wollinger. *Engineering Aspects of Hyperelliptic Curves*. Ph.D. Thesis. Dept. of Elec. Eng. and Infor. Sci., Ruhr-University of Bochum. July 2004.