PERGAMON Computers and Mathematics with Applications 42 (2001) 91–100

An International Journal
computers & mathematics with applications

www.elsevier.nl/locate/camwa

A Comparison of Different Finite Fields for Elliptic Curve Cryptosystems

N. P. SMART
Computer Science Department
University of Bristol
Woodland Road, Bristol BS8 1UB, UK
nigel@cs.bris.ac.uk

(Received June 2000; revised and accepted November 2000)

Abstract—We examine the relative efficiency of four methods for finite field representation in the context of elliptic curve cryptography (ECC). We conclude that a set of fields called the optimized extension fields (OEFs) give greater performance, even when used with affine coordinates, when compared against the type of fields recommended in the emerging ECC standards. Although this performance advantage is only marginal, and hence, there is probably no need to change the current standards to allow OEF fields in standards compliant implementations.

Keywords—Finite fields, Elliptic curves, Cryptography.

1. INTRODUCTION

The efficient implementation of arithmetic in finite fields is crucial for the high performance of various cryptographic algorithms, such as those based on the difficulty of the discrete logarithm problem in finite fields, elliptic curves or hyperelliptic curves. In all of these schemes, the efficiency of the underlying finite field operations is the dominant performance constraint, any effort spent optimising the field operations is well spent. This has led a number of special choices of field to be used, each with its own performance characteristics. Due to different engineering constraints such as processor type, memory requirements, etc., there is no correct answer to the question: which field should one use?

In this paper, we look in more detail at the choice of finite fields in the case of elliptic curve based systems. It is important, when comparing one parameter choice against another, that we use real world parameter choices and we look not only at the performance of the underlying field arithmetic but also at the performance of the overall cryptographic protocols. This is important since some cryptographic algorithms do not make use of general arithmetic but only require careful optimisation of crucial parts. This is particularly true of modular exponentiation based systems where it makes more sense to spend a lot of time optimising the squaring operation

0898-1221/01/$ - see front matter Typeset by $\mathcal{A}\mathcal{M}\mathcal{S}$-TEX
PII: S0898-1221(01)00133-X

as opposed to the general multiplication operation. A similar situation holds for elliptic curves, where one needs to optimise the point doubling operation more than the general point addition operation.

In this paper, we will concentrate on the choice of finite field and not consider the use of special curves, such as the so-called Koblitz curves, which can provide performance advantages. Hence, our conclusions will not be affected by security considerations as long as the overwhelming majority of curves over the given fields are considered to be secure.

It is also important to compare like for like, for example if one system uses machine code for the main arithmetics whilst the one being compared against uses no machine code then the comparison is not really fair. In addition, one should only compare algorithms using a single computer. Even comparisons on almost identical processors can lead to different conclusions. For example, in fields of characteristic two arithmetic is often implemented via lookup tables, hence, algorithms will behave differently on two processors which are identical bar the fact that one has a faster access time for its cache. Also, comparing two systems which are programmed by different people one could be comparing programming ability rather than actual performance.

Often these considerations are ignored, since complexity theory tells us that implementation details should not matter and that it is the complexity of the algorithm itself which should determine the relative merits. But complexity arguments only hold in the limit, which may not be applicable at the problem size under consideration. For example, when multiplying two integers, complexity theory tells us that Fourier transform techniques or Karatsuba multiplication will work faster than school book multiplication, but when multiplying 200 bit numbers it is often far more efficient in practice to use a school book multiplication algorithm.

Various other authors have compared implementation details for elliptic curve systems, but we would contend that such comparisons are to be taken with a pinch of salt. For example Bailey and Paar [1] compare their OEF based elliptic curve implementation against other peoples implementations of characteristic two fields of composite degrees. This is bad practice for a number of reasons. First, composite fields of characteristic two are not recommended for use in cryptographic standards (a view which has been reinforced by recent work in [2]). Second, the authors of [1] have a reason to prefer OEF fields since they are putting them forward as a replacement for standard implementations, although in our comparison we give independent verification of their conclusion that OEF fields offer significant advantages. Third, the comparison was performed on 64 bit architectures, which although these are now common in high end workstations, are not likely to be common in the small devices which are the target for elliptic curve based systems. Our comparison on a 32 bit RISC system is likely to be more indicative since such processors, like the StrongArm, are likely to be used in a number of such devices in the coming years. A comparison of various elliptic curve algorithms on an 8 bit smartcard without coprocessor was given in [3], although this paper concentrated mainly on OEF fields. For those interested in constrained environments we recommend that the current paper is read in conjunction with [3].

In another comparison, in [4] the authors give a comparison between even characteristic fields and odd characteristic fields, the later being implemented using Barrett reduction. This is slightly flawed since standards compliant elliptic curve systems in odd characteristic are more likely to be implemented over fields defined by a Generalised Mersenne prime which provide different performance characteristics than general primes.

The main reason for our interest was to compare the new idea of OEF fields with the fields defined by GM-primes which occur in the standards documents. Hence, all our implementations were coded from scratch and shared many core subprocedures. A similar length of time was spent optimising each implementation, and so, the relative differences in performance should be indicative of completely optimised implementations. Although the resulting comparisons are not completely scientific, we do hope that they are on a better foundation than previous ones. Hence, hopefully they can be used to make further commercial considerations as to which fields are to be preferred.

2. METHODOLOGY

In this section, we outline the rough methodology we used to compare the running times of the various field choices. This is done to enable others to compare their own timings against ours, or our design decisions with their own.

First, we decided to compare on two processor types, one a RISC processor and one a CISC processor. These were an Ultra Sparc IIi running at 440 Mhz with the Solaris operating system, the other was a Pentium Pro running at 166 Mhz with the Linux operating system. Although both processors support a limited amount of special purpose 64 bit registers, for example the MMX instructions on the Pentium can use 64 bit data, we decided our implementation should concentrate on the 32 bit core. This decision was also done to aid interoperability, since it is easier to write portable C++ code which uses 32 bit data rather than 64 bit data. In addition, since elliptic curves are more likely to be used in constrained environments, it is more likely that they will be implemented on 32 bit processors rather than 64 bit processors.

The programming language used was C++ and the compiler was the GNU g++ compiler, version 2.95.1. The GNU compiler enables a quite powerful inline assembly option. We decided to only implement three small routines in assembler, mainly adding functionality present on most modern processors to the C++ language. These three pieces of assembler implemented the following (inlined) functions, the entities on the left denote the output which depends on the input on the right,

```
(c,a)=a+b+c
(c,a)=a-b-c
(h,l)=a*b+t+c
```

where c denotes a one bit value and (h, l) denotes the 64 bit quantity $h2^{32} + l$. The multiplication used may seem a little strange at first, but it turns out to be very useful. Since we only implemented a small amount in assembler our code was highly portable, on the other hand the code could be made considerably more efficient using greater use of assembler.

The code was implemented using as much common shared code between the various field representations as possible, hence, the comparison would be of the fields and not the different implementations of other nonimportant functions.

The code was also designed to work with any number of field sizes, hence, although special tricks were used for each field type these were coded in as general a way as possible. One could argue that this is unlikely to be the case in real life, since one may wish to hardwire a field into a particular application. This would result in even greater efficiency gains, since the exact field would be decided at compile time. However, one could also argue that some applications would need to keep the size of field open until run time, since that way one maintains greater chances of being interoperable with other applications. Since we tried to keep the code general and did not use a great deal of machine code this meant aggressive optimisations such as software pipelining would only be available via the compiler's own optimisation routines.

Using C++ also allowed us to direct a number of 'optimisation hints' to the compiler, by for example hinting as to when a function should be inlined or when a passed argument should be considered a constant for the function call. However, it is widely believed that object code produced from C source is more efficient than that produced from C++ source. We did not consider this a problem since we were mainly interested in the relative performance of the various arithmetics.

To avoid the influence of programming ability on the performance of different field types, all code was implemented from scratch by the author with no routines 'borrowed' from existing pieces of software.

We end this section by stressing that the absolute run times we give below could clearly be improved. However, we believe the relative run times should be indicative of what should hold in any reasonable implementation.

3. THE CHOICES FOR FIELDS

Currently, there are a number of field choices for elliptic curve systems that are mentioned in the literature. These can be divided into two classes.

3.1. Prime Fields

Fields of large prime characteristic, $\mathbb{F}_p$, are very popular since they can be efficiently implemented using techniques borrowed from other finite field based cryptographic systems such as DSA and RSA. However, using standard modular arithmetic is not very efficient since multiprecision remaindering operations are very expensive. Hence, when used in elliptic curve systems there are various choices that are often made.

General Primes

For general primes the most efficient implementation technique is almost always to use Montgomery arithmetic, [5]. Although the authors of [4] use Barrett reduction, the timing difference between Montgomery arithmetic and Barrett reduction is usually negligible.

Montgomery arithmetic uses a special representation to perform efficient arithmetic, the division and remaindering essentially being performed by bit, or word, shifting. We do not cover this arithmetic here since it is covered in a number of text books, e.g., [6,7]. In [8] various ways of implementing Montgomery arithmetic are described. We tried both the FIOS and CIOS methods and decided that in our situation the FIOS method gave the greatest efficiency.

Field inversion was performed using a standard modification of the binary extended Euclidean algorithm for Montgomery arithmetic.

Generalized Mersenne Primes

Certain primes are highly suited for efficient reduction techniques, the most simple form of such primes being the Mersenne primes, which are primes of the form $p = 2^k - 1$. However, the number of Mersenne primes of the correct size for cryptography is limited. This has led a number of authors to propose generalisations on the Mersenne primes.

Crandall [9] proposed the use of primes of the form $p = 2^k - c$ where c is a small integer, which is usually chosen to fit into a single word. Primes of the form $p = 2^k \pm c$ for a small value of c (in comparison to 2^k) are often called pseudo-Mersenne primes.

In another direction, Solinas [10] introduced the concept of generalized Mersenne primes (GM-primes) which are primes of the form

$$p = f\left(2^k\right),$$

where f is a polynomial of small degree and weight and k is a multiple of the computer word size.

The use of GM-primes has become popular due to the adoption of these primes in the recommend curves in standards from such bodies as ANSI [11], NIST [12], SECG [13], and WAP [14]. As an example we take the following example from Solinas' paper

$$f(t) = t^3 - t - 1.$$

Then we obtain the 192 bit prime

$$p = 2^{192} - 2^{64} - 1 = f\left(2^{64}\right).$$

Reducing the result of a multiprecision multiplication is then a simple matter: after performing the multiprecision multiplication (via school book methods) of two 192 bit integers we obtain a number of the form

$$N = (A_5||A_4||A_3||A_2||A_1||A_0),$$

where A_i is a 64 bit integer. The value of N modulo p can then be computed with three additions modulo p,

$$N \equiv T + S_1 + S_2 + S_3 \pmod{p},$$

where

$$T = (A_2||A_1||A_0), \qquad S_1 = (0||A_3||A_3),$$
$$S_2 = (A_4||A_4||0), \qquad S_3 = (A_5||A_5||A_5).$$

Inversion was again performed using a variant of the binary extended Euclidean algorithm.

3.2. Nonprime Fields

Again there are a number of choices here for the fields $\mathbb{F}_q$ with $q = p^n$. These are usually implemented via a polynomial basis where

$$\mathbb{F}_q = \frac{\mathbb{F}_p[x]}{f(x)}$$

and $f(x)$ is an irreducible polynomial of degree n over $\mathbb{F}_p$.

Characteristic Two

In this case, due to work described in [2], we need to choose n to be prime. One chooses $f(x)$ to be a trinomial or pentanomial for efficiency. In other words, we choose either

$$f(x) = x^n + x^k + 1$$

or

$$f(x) = x^n + x^k + x^l + x^m + 1.$$

These have been a popular choice in standards bodies and for implementors due to the advantages that they offer in hardware and on some RISC processors. The literature on these is quite extensive so we just refer the reader to [6] or [15] for more mathematical details. Normal bases can be used, but these are more usual when considering hardware implementations of fields in characteristic two, hence, we shall not consider normal bases further here.

To implement characteristic two arithmetic a 256×256 look up table was computed which gave the 16 bit result of multiplying two 8 bit quantities, when one considered the integers representing a binary polynomial. Using this look up table, an inlined function was created to multiply two 32 bit quantities and return a 64 bit result. Then the multiplication of two n bit polynomials was reduced to various calls of the 32 bit multiplication routines using Karatsuba style techniques.

Squaring an n bit polynomial is particularly easy since there are no 'cross terms' in characteristic two. Hence, all that one needs to do is divide the n bit input into 8 bit chunks and then compute the 16 bit square using the previously mentioned lookup table.

After having squared an n bit polynomial, or multiplied two n bit polynomials, we obtain a $2n$ bit result. This needs to be reduced to an n bit final answer by taking the remainder on division by f. This is done using a word-oriented version of [6, Algorithm II.9].

Field inversion was performed using a variant of the binary Euclidean algorithm.

Optimized Extension Fields

These fields have been proposed by Bailey and Paar in [1,16]. They appear to offer a number of advantages which we shall outline below. An optimized extension field (OEF) is one of the form

- $p = 2^k \pm c$ is a pseudo-Mersenne prime with $\log_2 c \leq k/2$ and such that p fits into a computer word,
- $f(x) = x^n - \omega$ is irreducible.

For efficiency reasons it is often sensible to insist that $\omega = 2$.

In OEF fields, addition of elements in $\mathbb{F}_q$ is relatively simple and can be accomplished without carries propagating, since elements of $\mathbb{F}_q$ are implemented as polynomials modulo $f(x)$. Multiplication is also very simple since reduction of a polynomial modulo $f(x)$ is particular simple. Multiplication can also be simplified using Karatsuba multiplication, which provides a performance advantage for polynomial multiplication for very small degree polynomials.

Finally, inversion is particularly easy in OEF fields since one can use a technique due to Itoh and Tsujii [17] combined with an efficient method to compute the action of the Frobenius mapping, see [16] for more details on this.

It should be noted that the technique of Weil descent which is described in [2] could be applied to curves defined over OEFs, since n is typically small. However, the resulting curve does not seem to have the nice properties that one observes in the even characteristic case. This is because the function field extensions are not Artin-Schreier in nature. Hence, to the best current knowledge there are no security concerns with using OEFs. However, this could change given the rapid progress made in studying the EC-DLP in recent years.

As stated in the introduction, there has been no comprehensive comparison of the above choices of finite fields in the literature. Since the use of Montgomery arithmetic and even characteristic finite fields are quite standard [4] these are not the most interesting cases.

However, the comparison of OEFs against GM-prime fields has not to our knowledge been carried out. But this is the most important comparison to make, since GM-prime fields are those which are being used by various standards bodies, e.g., ANSI, NIST, and SECG.

4. FIELD OPERATIONS

The fields we used for comparison where the following.

$K_1 = \mathbb{F}_p$, where $p = 2^{192} - 2^{64} - 1$. This was used for the GM-prime implementation and the Montgomery implementation.

$K_2 = \mathbb{F}_{p^n} = \mathbb{F}_p[z]/(z^6 - 2)$, where $p = 2^{31} - 19$. This was used for the OEF implementation.

$K_3 = \mathbb{F}_{2^n} = \mathbb{F}_2[z]/(z^{191} + z^9 + 1)$.

Notice that roughly the same bitlength was used for all fields, namely 192 and 186. Tables 1 and 2 describe the timings we obtained for our two different processors.

Table 1. Field operation timings on the Sparc.

Field Type	Addition	Multiplication	Square	Inversion	I/M
Montgomery	0.72 μs	6.06 μs	6.04 ms	0.16 ms	26.40
GM-prime	0.76 μs	4.56 μs	4.44 ms	0.16 ms	35.09
OEF	0.84 μs	4.08 μs	4.04 ms	0.02 ms	4.90
$\mathbf{F}_{2^{191}}$	0.16 μs	11.20 μs	1.36 ms	0.14 ms	12.50

Table 2. Field operation timings on the Pentium.

Field Type	Addition	Multiplication	Square	Inversion	I/M
Montgomery	2.25 μs	11.40 μs	11.36 ms	0.60 ms	52.63
GM-prime	2.25 μs	11.40 μs	11.24 ms	0.58 ms	50.88
OEF	1.72 μs	9.36 μs	8.8 ms	0.08 ms	8.54
$\mathbf{F}_{2^{191}}$	0.40 μs	20.88 μs	5.40 ms	0.42 ms	20.11

Notice that the even characteristic case appears to be the worst since multiplication is almost twice as slow as the next slowest field type, namely the Montgomery representation. However, this does not translate into a 100% increase in the required CPU time for the final cryptographic operation since for fields of even characteristic the squaring operation comes almost for free. In addition, elliptic curve point doubling formulae in characteristic two are simpler than those in the odd characteristic case.

The use of multiplication is slightly faster for the OEF field compared to the GM-prime field. But the most striking improvement is in the time required to perform an inversion in the field. As we shall comment later, this leads to important decisions on how one actually implements an elliptic curve cryptographic system. It is important to look at the ratio, $r = I/M$, of the time needed to perform an inversion, I, to the time to compute a multiplication, M. This figure is given in the last column of the previous tables.

5. CURVE OPERATIONS

The basic elliptic curve operation required in cryptography is point multiplication. That is given $P \in E(\mathbb{F}_q)$ and $k \in_R [1, \ldots, \#E(\mathbb{F}_q) - 1]$ compute $[k]P$. There are various techniques to perform this which are described in [6,7].

A first observation is that if P is a fixed point which is required to be multiplied by a large number of values, k, then one can use a great deal of precomputation, see [7, Algorithm 14.109]. Such a point, P, is often the generator of the group $\#E(\mathbb{F}_q)$, and is hence, called a base point.

However, sometimes we do not know the value of P in advance and so different optimisations need to be performed, in such a situation we call P a general point. In this case, we used the signed window method, see [6, Algorithm IV.7].

In standard implementations for the EC-DH protocol, each party needs to perform one multiplication of a general point and one multiplication of the fixed base point. In the EC-DSA protocol, the signer needs to perform one multiplication of the base point and the verifier needs to perform a multiplication of the base point and a multiplication of a general point.

A second observation is that multiplication, of both a general point and of the base point, can be done in either affine or 'mixed' coordinates. 'Mixed' coordinates refers to the fact that although we may use a projective representation of the points, we take into account that some of the intermediate points may be in affine representation. Mixed coordinates are to be preferred when the ratio, r, of inversion to multiplication is large, since one is trading off inversions for a larger number of multiplications. On the other hand, affine coordinates require 33% less storage.

For our timings we used the following curves which are suitably strong for cryptographic use:
$\underline{E(K_1)}$. We used the curve labelled $P-192$ by NIST, [12]. This is the curve *secp192r1* in SECG and *prime192v1* in ANSI X9.62. This curve is given by

$$E_1 : Y^2 = X^3 - 3X + b,$$

where

$$b = 2455155546008943817740293915197451784769108058161191238065.$$

This curve has group order

$$6277101735386680763835789423176059013767194773182842284081,$$

which is a prime. The use of a curve with a coefficient of -3 for the X term in the equation provides a certain performance advantage, whilst a curve of prime group order is clearly a security advantage.

$\underline{E(K_2)}$. In this case, we needed to generate our own curve so we took the curve given by

$$E_2 : Y^2 = X^3 - 3X + 131072z^5.$$

This curve has group order

$$9807970940881741910790475986522413956726171926140144424 4,$$

which is four times a prime. Again notice the coefficient of X in the curve equation is minus three.

$E(K_3)$. In this case, we use the curve in Example 16 [6, p. 187]. This is given by

$$E_3 : Y^2 + XY = X^3 + X^2 + b,$$

where

$$b = 7BC86E2102902EC4D5890E8B6B4981FF27E0482750FEFC03.$$

This curve has group order

$$2 \times 1569275433846670190958947355834614995815261150867795429199,$$

which is two times a prime. Here we have chosen a curve with coefficient of X^2 equal to one, this gives greater performance in characteristic two. In addition, for characteristic two fields the best type of group order we can use is one which is twice a prime. Hence, this curve is typical of ones used in real life systems, although NIST does not have a curve in characteristic two at this level of security.

In Tables 3 and 4, we see that for OEF fields we obtain a improvement when using affine coordinates. We also reduce the amount of memory required for the tables when using only affine coordinates in the window multiplication methods, since we no longer need to store the z-coordinates.

Table 3. Curve operation timings on the Sparc.

Operation	Montgomery	GM-prime	OEF	$\mathbb{F}_{2^{191}}$
Addition: Affine	212 μs	179 μs	40 μs	179 μs
Addition: Mixed	114 μs	81 μs	74 μs	168 μs
Doubling: Affine	212 μs	191 μs	48 μs	174 μs
Doubling: Mixed	69 μs	48 μs	40 μs	67 μs
Base Point Mult.: Affine	13 ms	12 ms	3 ms	12 ms
Base Point Mult.: Mixed	7 ms	5 ms	4 ms	9 ms
General Point Mult.: Affine	48 ms	43 ms	10 ms	40 ms
General Point Mult.: Mixed	18 ms	12 ms	10 ms	19 ms

Table 4. Curve operation timings on the Pentium.

Operation	Montgomery	GM-prime	OEF	$\mathbb{F}_{2^{191}}$
Addition: Affine	650 μs	635 μs	104 μs	490 μs
Addition: Mixed	234 μs	221 μs	179 μs	414 μs
Doubling: Affine	700 μs	665 μs	128 μs	488 μs
Doubling: Mixed	146 μs	134 μs	108 μs	164 μs
Base Point Mult.: Affine	44 ms	43 ms	7 ms	33 ms
Base Point Mult.: Mixed	14 ms	14 ms	11 ms	23 ms
General Point Mult.: Affine	160 ms	151 ms	28 ms	111 ms
General Point Mult.: Mixed	37 ms	35 ms	27 ms	46 ms

We also notice that for general point multiplications the performance of curves over fields of even characteristic is not as bad as one would be led to believe from just looking at the timings for the field arithmetic. In some small systems, to avoid attacks like DPA [18] one often alters the base point on every run of the protocol. Hence, one never actually uses the special optimisations for multiplying a base point and all point multiplications become general ones.

6. CRYPTOGRAPHIC OPERATIONS

Finally, we timed three basic cryptographic operations which are popular using ECC namely unsigned Diffie-Hellman (EC-DH), the EC variant of the digital signature algorithm (EC-DSA),

Table 5. Cryptographic operation times on the Sparc.

Operation	Montgomery	GM-prime	OEF	$\mathbf{F}_{2^{191}}$
EC-DSA Sign	6 ms	4 ms	3 ms	10 ms
EC-DSA Verify	27 ms	18 ms	12 ms	29 ms
EC-DH	24 ms	15 ms	13 ms	29 ms
EC-MQV	34 ms	23 ms	19 ms	39 ms

Table 6. Cryptographic operation times on the Pentium.

Operation	Montgomery	GM-prime	OEF	$\mathbf{F}_{2^{191}}$
EC-DSA Sign	16 ms	16 ms	9 ms	22 ms
EC-DSA Verify	52 ms	49 ms	35 ms	70 ms
EC-DH	53 ms	50 ms	40 ms	72 ms
EC-MQV	73 ms	70 ms	50 ms	97 ms

and the EC variant of the MQV primitive (EC-MQV). The timings we given in Tables 5 and 6 for our four specimen fields. The times for EC-DH and EC-MQV are the times required by one of the parties to perform their calculations. We assumed that any base point multiplication was done using the optimisations alluded to above.

7. CONCLUSION

We have shown that OEF fields appear to offer performance advantages over other field representations used in ECC. This is not only in terms of overall performance but also in terms of storage memory requirements. The present author has no vested interests in any of the four field types under consideration and hopefully the results can be taken as completely independent of commercial bias or the use of aggressive optimisation techniques applied to one of the cases only.

On the other hand, it should be noted that the performance difference between OEF fields and fields based on generalized Mersenne numbers is probably not large enough to warrant additions to the various standards since addition of OEF fields would degrade attempts to obtain interoperability between various implementations.

REFERENCES

1. D.V. Bailey and C. Paar, Optimal extension fields for fast arithmetic in public-key algorithms, In *Advances in Cryptology-CRYPTO 98*, pp. 472–485, Springer-Verlag LNCS 1462, (1998).
2. P. Gaudry, F. Hess and N.P. Smart, Constructive and destructive facets of Weil descent on elliptic curves, Preprint, (2000).
3. A.D. Woodbury, D.V. Bailey and C. Paar, Elliptic curve cryptography on smart cards without coprocessors, In *Smart Card and Advanced Applications, CARDIS 2000*, pp. 71–92, Kluwer, (2000).
4. E. De Win, S. Mister, B. Preneel and M. Wiener, On the performance of signature schemes based on elliptic curves, In *Algorithmic Number Theory-ANTS-III*, pp. 252–266, Springer-Verlag LNCS 423, (1998).
5. P.L. Montgomery, Modular multiplication without trial division, *Math. Comp.* **44**, 519–521, (1985).
6. I.F. Blake, G. Seroussi and N.P. Smart, *Elliptic Curves in Cryptography*, Cambridge University Press, (1999).
7. A.J. Menezes, P.C. van Oorschot and S.A. Vanstone, *Handbook of Applied Cryptography*, CRC Press, (1996).
8. C.K. Koc, T. Acer and B.S. Kaliski, Jr., Analyzing and comparing Montgomery multiplication algorithm, *IEEE Micro* **16**, 26–33, (June 1996).
9. R. Crandall, Method and apparatus for public key exchange in a cryptographic system, U.S. Patent Number 5159632, (1992).
10. J.A. Solinas, *Generalised Mersenne Numbers*, Preprint, (1999).
11. ANSI, *X9.62: Public Key Cryptography For The Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)*, American National Standards Institute, (1999).
12. NIST, *FIPS PUB 186-2: Digital Signature Standard (DSS)*, National Institute for Standards and Technology, (2000).
13. SECG, *SEC 2: Recommended Elliptic Curve Domain Parameters*, Standards for Efficient Cryptography Group, (1999).
14. WAP, *WTLS: Wireless Transport Layer Security Specification*, Wireless Application Forum Ltd, (1999).

15. R. Lidl and H. Niederreiter, Finite fields, In *Encyclopedia of Mathematics and Its Applications*, (Edited by G.-C. Rota), Addison-Wesley, (1983).
16. D.V. Bailey and C. Paar, Efficient arithmetic in finite field extensions with applications in elliptic curve cryptography, *J. Cryptology* (to appear).
17. T. Itoh and S. Tsujii, A fast algorithm for computing multiplicative inverses in $GF(2^m)$ using normal bases, *Information and Computation* **78**, 171–177, (1988).
18. P. Kocher, J. Jaffe and B. Jun, Differential power analysis, In *Advances in Cryptology, CRYPTO '99*, pp. 388–397, Springer LNCS 1666, (1999).