**Software Implementation of Elliptic Curve
  
Cryptography Over Binary Fields**

Darrel Hankersoni, Julio Lopez Hernandez', and Alfred Menezes3

1 Dept. of Discrete and Statistical Sciences, Auburn University, USA
  
hankedrOmail . [auburn. edu](http://auburn.edu)
  
2 Dept. of Computer Science, University of Valle, Colombia
  
[jlopezOborabora.univalle.edu.co](http://jlopezOborabora.univalle.edu.co)
  
3 Dept. of Combinatorics and Optimization, University of Waterloo, Canada
  
ajmenezeOuwaterloo.ca

Abstract. This paper presents an extensive and careful study of the software implementation on workstations of the NIST-recommended el­liptic curves over binary fields. We also present the results of our imple­mentation in C on a Pentium II 400 MHz workstation.

**1 Introduction**

Elliptic curve cryptography (ECC) was proposed independently in 1985 by Neal Koblitz [19] and Victor Miller [29]. Since then a vast amount of research has been done on its secure and efficient implementation. In recent years, ECC has received increased commercial acceptance as evidenced by its inclusion in stan­dards by accredited standards organizations such as ANSI (American National Standards Institute) [1, 2], IEEE (Institute of Electrical and Electronics Engi­neers) [13], ISO (International Standards Organization) [14, 15], and NIST (Na­tional Institute of Standards and Technology) [33].

Before implementing an ECC system, several choices have to be made. These include selection of elliptic curve domain parameters (underlying finite field, field representation, elliptic curve), and algorithms for field arithmetic, elliptic curve arithmetic, and protocol arithmetic. The selections can be influenced by se­curity considerations, application platform (software, firmware, or hardware), constraints of the particular computing environment (e.g., processing speed, code size (ROM), memory size (RAM), gate count, power consumption), and constraints of the particular communications environment (e.g., bandwidth, re­sponse time). Not surprisingly, it is difficult, if not impossible, to decide on a single "best" set of choices—for example, the optimal choices for a PC applica­tion can be quite different from the optimal choice for a smart card application.

Over the past 15 years, numerous papers have been written on various aspects of ECC implementation. Most of these papers do not consider all the factors involved in an efficient implementation. For example, many papers focus only on finite field arithmetic, or only on elliptic curve arithmetic.

The contribution of this paper is an extensive and careful study of the soft­ware implementation on workstations of the NIST-recommended elliptic curves

over binary fields. While the only significant constraint in workstation environ­ments may be processing power, some of our work may also be applicable to other more constrained environments (e.g., see [4] for implementations on a pager and the Palm Pilot). We also present the results of our implementation in C (no hand-coded assembler was used) on a Pentium II 400 MHz workstation. These results serve to validate our conclusions based primarily on theoretical consider­ations. While some effort was made to optimize the code (e.g., loop unrolling), it is likely that significant performance improvements can be obtained especially if the code is tuned for a specific platform. Nonetheless, we hope that our work will serve as a benchmark for future efforts in this area.

The remainder of this paper is organized as follows. §2 describes the NIST curves over binary fields and presents some rationale for their selection. In §3, we describe methods for arithmetic in binary fields. §4 and §5 consider efficient techniques for elliptic curve arithmetic. In §6, we select the best methods for per­forming elliptic curve operations in ECC protocols such as the ECDSA. Finally, we draw our conclusions in §7 and discuss avenues for future work in §8.

2 NIST Curves Over Binary Fields

In February 2000, FIPS 186-1 was revised by NIST to include the elliptic curve digital signature algorithm (ECDSA) as specified in ANSI X9.62 [1] with further recommendations for the selection of underlying finite fields and elliptic curves; the revised standard is called FIPS 186-2 [33].

FIPS 186-2 has 10 recommended finite fields: 5 prime fields, and the binary fields 7

\_2163, 72233, 72283, 72409, and 72571. For each of the prime fields, one ran­domly selected elliptic curve was recommended, while for each of the binary fields one randomly selected elliptic curve and one Koblitz curve was selected.

The fields were selected so that the bitlengths of their orders are at least twice the key lengths of common symmetric-key block ciphers—this is because exhaustive key search of a k-bit block cipher is expected to take roughly the same time as the solution of an instance of the elliptic curve discrete logarithm problem using Pollard's rho algorithm for an appropriately-selected elliptic curve over a finite field whose order has bitlength 2k. The correspondence between symmetric cipher key lengths and field sizes is given in Table 1. For binary fields

Table 1. NIST-recommended field sizes for U.S. Federal Government use.

|  |  |  |  |
| --- | --- | --- | --- |
| Symmetric cipher  key length | Example  algorithm | Bitlength of *p* Dimension m of  in prime field F, binary field g2 m. | |
| 80 | SKIPJACK | 192 | 163 |
| 112 | Triple-DES | 224 | 233 |
| 128 | AES Small [34] | 256 | 283 |
| 192 | AES Medium [34] | 384 | 409 |
| 256 | AES Large [34] | 521 | 571 |

TF2—, m was chosen so that there exists a Koblitz curve of almost prime order over TF2—. Since the order #E(TF21) divides #E(TF2—) whenever *1* divides m, this requirement imposes the condition that m be prime.

Since the NIST binary curves are all defined over fields 72— where m is prime, our paper excludes from consideration fields such as 72178 for which efficient techniques are known for field arithmetic [6, 12]. This exclusion is not a concern in light of recent advances in algorithms for the discrete logarithm problem for elliptic curves over 72— when m has a small non-trivial factor [9, 10].

The remainder of this paper considers the efficient implementation of the NIST-recommended random and Koblitz curves over the fields 72163, 2163, 72233, and 72283. The results can be extrapolated to curves over TF2409 and 72571.

**Description of the NIST curves over binary fields.** The NIST elliptic curves over 72183, 72233 and 72283 are listed in Table 2. The following notation is used. The elements of 72— are represented using a polynomial basis repre­sentation with reduction polynomial *f(x)* (see §3.1). The reduction polynomi­als for the fields 72163, 2163, 72233 and 72283 are *f (x) = x163* + *x7 + x6 + x3 + 1, f (x) = x233* + X" + 1, and *f(x)* = x283 + x12 + x7 + x5 + 1, respectively. An elliptic curve *E* over 72— is specified by the coefficients *a,* b **E** 72— of its defining equation y2 + xy = x3 + *axe +* b. The number of points on *E* defined over 72— is *nh,* where n is prime, and *h* is called the co-factor. A random curve over 72— is denoted by B-m, while a Koblitz curve over 72— is denoted by K-m.

**Table 2.** NIST-recommended elliptic curves over F2163, g2233 and F2283.

B-163: a = 1, *h =* 2,

b = Ox 00000002 0A601907 B8C953CA 1481EB10 512F7874 4A3205FD
  
n = Ox 00000004 00000000 00000000 000292FE 77E70C12 A4234C33

B-233: a = 1, *h =* 2,

b = Ox 00000066 647EDE6C 332C7F8C 0923BB58 213B333B 20E9CE42 81FE115F 7D8F9OAD

n = Ox 00000100 00000000 00000000 00000000 0013E974 E72F8A69 22031D26 03CFE0D7

B-283: a = 1, *h =* 2,

b = Ox 027B680A C8B8596D A5A4AF8A 19A0303F CA97FD76 45309FA2 A581485A F6263E31 3B79A2F5

n = Ox 03FFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFEF90 399660FC 938A9016 5B042A7C EFADB307

K-163: a = 1, b = 1, *h =* 2,

n = Ox 00000004 00000000 00000000 00020108 A2EOCCOD 99F8A5EF

K-233: a = 0, b = 1, *h =* 4,

n = Ox 00000080 00000000 00000000 00000000 00069D5B B915BCD4 6EFB1AD5 F173ABDF

K-283: a = 0, b = 1, *h =* 4,

n = Ox 01FFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFE9AE 2ED07577 265DFF7F 94451E06 1E163C61

**3 Binary Field Arithmetic**

This section presents algorithms that are suitable for performing binary field arithmetic in software. For concreteness, we assume that the implementation platform has a 32-bit architecture. The bits of a word W are numbered from **0** to 31, with the rightmost bit of W designated as bit **0.**

**3.1 Field representation**

Of the many representations of 72- , m prime, that have been studied, it appears that a polynomial basis representation with a trinomial or pentanomial as the reduction polynomial yields the simplest and fastest implementation in software. We will henceforth use a polynomial basis representation.

Let f *(x)* = *r(x)* be an irreducible binary polynomial of degree m. The

elements of 72- are the binary polynomials of degree at most m-1 with addition and multiplication performed modulo f *(x).* A field element *a(x) = ixm-1*

*• • •-ka2x2 -kaix-kao* is associated with the binary vector *a = (a,n\_i, , a2, a1, ao)* of length m. Let *t =* rm/321, and let *s =* 32t - m. In software, we store *a* in an array of *t* 32-bit words: A = *(A[t* - 1], . . . , A[2], A[1], A[0]), where the rightmost bit of A[0] is al), and the leftmost *s* bits of *A[t* - 1] are unused (always set to **0).**

Addition of field elements is performed bitwise, thus requiring only *t* word operations.

**3.2 Multiplication**

The shift-and-add method (Algorithm 1) for field multiplication is based on the observation that *a • b = b + • • • + a2x2b aixb* aob. Iteration *i* of the algorithm computes xi b mod f *(x)* and adds the result to the accumulator *c* if *ai =* 1. Note that b • mod f *(x)* can be easily computed by a left-shift of the vector representation of b, followed by the addition of *r (x)* to b if bm = 1.

Algorithm 1. Right-to-left shift-and-add field multiplication

INPUT: Binary polynomials *a(x)* and *b(x)* of degree at most m — 1. OUTPUT: *c(x) = a(x) b(x)* mod *f (x).*

1. If ao = 1 then *c* b; else *c* 0.
2. For *i* from 1 to m — 1 do

2.1 b *x* mod *f (x).*

2.2 If ai = 1 then *c c +* b.

1. Return(c).

While Algorithm 1 is well-suited for hardware where a vector shift can be performed in one clock cycle, the large number of word shifts make it less de­sirable for software implementation. We next consider faster methods for field multiplication which first multiply the field elements as polynomials, and then reduce the result modulo f *(x).*

**Polynomial multiplication.** The comb method for polynomial multiplication is based on the observation that if *b(X). Xk* has been computed for some *k* E [0, 31], then *b(x)* • *X32i +k* can be easily obtained by appending *j* zero words to the right of the vector representation of *b(x)* • *Xk .* Algorithm 2 considers the bits of the words of A from right to left, while Algorithm 3 considers the bits from left

to right. The following notation is used: if *C* = *(C [n], , C [2] , C [1], C [0])* is a

vector, then *C{j}* denotes the truncated vector *(C [n], , C[i + 1], C[i]).*

**Algorithm 2.** Right-to-left comb method for polynomial multiplication

INPUT: Binary polynomials *a(x)* and *b(x)* of degree at most m — 1. OUTPUT: *c(x) = a(x) b(x).*

1. *C* O.
2. For *k* from 0 to 31 do

2.1 For *j* from 0 to *t* — 1 do

If the kth bit of *A[j]* is 1 then add *B* to *C{j}.* 2.2 If *k* 31 then *B B x.*

1. Return(C).

**Algorithm 3.** Left-to-right comb method for polynomial multiplication

INPUT: Binary polynomials *a(x)* and *b(x)* of degree at most m — 1.

OUTPUT: *c(x) = a(x) • b(x).*

1. *C* 0.
2. For *k* from 31 downto 0 do 2.1 For *j* from 0 to *t* — 1 do If the kth bit of *A[j]* is 1 then add *B* to *C{j}.*

2.2 If *k* 0 then *C • x.*

1. Return(C).

Algorithms 2 and 3 are both faster than Algorithm 1 since there are fewer vector shifts (multiplications by *x).* Algorithm 2 is faster than Algorithm 3 since the vector shifts in the former involve the t-word vector *B,* while the vector shifts in the latter involve the 2t-word vector *C.* In [27] it was observed that Algorithm 3 can be sped up considerably at the expense of some storage overhead by precomputing u*(x)* • *b(x)* for all polynomials u*(x)* of degree less than w, where w divides the word length, and considering the bits of the *A[j]* 's w at a time. The modified method with w = 4 is presented as Algorithm 4.

**Algorithm 4.** Left-to-right comb method with windows of width w = 4

INPUT: Binary polynomials *a(x)* and *b(x)* of degree at most m — 1.

OUTPUT: *c(x) = a(x) • b(x).*

1. Compute *Bt, = u(x) • b(x)* for all polynomials *u(x)* of degree at most 3.
2. *C* 0.
3. For *k* from 7 downto 0 do

3.1 For *j* from 0 to *t* — 1 do

Let u = (u3, u2, ul, u0), where ui is bit (4k + *i)* of *A[j].* Add *Bt,* to *C{j}.* 3.2 If *k* 0 then *C•*x4.

1. Return(C).

The last method we consider for polynomial multiplication was first described by Karatsuba for multiplying integers (see [18]). Suppose that m is even. To multiply two binary polynomials *a(x)* and *b(x)* of degree at most m — 1, we first split up *a(x)* and *b(x)* each into two polynomials of degree at most (m/2) — 1: *a(x) = Ai(x)X A0 (x), b(x) = Bi(x)X + Bo (x),* where *X = xm/2 .* Then

By considering columns on the right side of the above congruences, it follows
  
that reduction of C[9] can be performed by adding C[9] four times to *C,* with

*a(x)b(x) = AiBiX2 + [(Ai + Ao)(Bi + Bo) + AiBi A0B01X*

which can be derived from three products of polynomials of degree (m/2) — 1. These products in turn can be computed recursively. For the case m = 163, we first prepended a 0 bit to the field elements *a* and b so that their bitlength is 164, and then used Karatsuba's method to subdivide the multiplication of *a* and b into multiplications of polynomials of degree at most 40. The latter multiplications were performed using a variant of Algorithm 4. For the case

m = 233 (resp. m = 283), we first prepended twenty-three (five) 0 bits to *a* and b, and then used Karatsuba's method to subdivide the multiplication of *a* and b into multiplications of polynomials of degree at most 63 (71).

**Reduction.** Let *c(x)* be a binary polynomial of degree at most 2m — 2. Algo­rithm 5 reduces *c(x)* modulo *f (x)* one bit at a time, starting with the leftmost bit. It is based on the observation that xi E *m r (x)* (mod *f(x))* for *i* > m. The polynomials *xk r(x),* 0 < *k <* 31, can be precomputed. If *r (x)* is a low-degree polynomial, or if *f(x)* is a trinomial, then the space requirements are smaller, and also the additions involving *xk r (x)* are faster.

**Algorithm 5.** Modular reduction (one bit at a time)

INPUT: A binary polynomial *c(x)* of degree at most 2m — 2. OUTPUT: *c(x)* mod *f (x).*

1. *Precornputation.* Compute *uk(x) = r(x),* 0 < *k <* 31.
2. For *i* from 2m — 2 downto m do

2.1 **If** *ci =* 1 then

Let *j = [(i —* m)/32] and *k = (i — m) —* 32j. Add *uk(x)* to *C{j}.*

1. Return((C[t — 1], , C[1], C[0])).

If *f(x)* is a trinomial, or a pentanomial with middle terms close to each other, then reduction of *c(x)* modulo *f (x)* can be efficiently performed one word at a time. For example, consider reducing the ninth word C[9] of *c(x)* modulo *f(x)* = x163 + *x7 + x6 + x3 +* 1. Here, m = 163 and *t =* 6. We have

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| x288 | = | X132 | + X131 | + X128 | + X125 | (mod *f (x))* |
| *x289* | = | X133 | + X132 | + X129 | + X126 | (mod *f (x))* |
| X319 | = | X163 | + X162 | + X159 | + X156 | (mod f(x)). |

the rightmost bit of C[9] added to bits 132, 131, 128 and 125 of *C.* This leads to Algorithm 6 for modular reduction which can be easily extended to other reduction polynomials. For the reduction polynomials considered in this paper, Algorithm 6 is faster than Algorithm 5 and furthermore has no storage overhead.

**Algorithm 6.** Modular reduction (one word at a time)

INPUT: A binary polynomial *c(x)* of degree at most 324.

OUTPUT: *c(x)* mod *f (x),* where *f (x) = x163 + x7 + x6 +x3+* 1.

1. For *i* from 10 downto 6 do {Reduce *C[i]* modulo *f (x)}*

1.1 *T C[i].*

1.2 *C[i* - 6] *C[i* - 6] ED *(T <* 29).

1.3 *C[i* - 5] *C[i* - 5] ED *(T* < 4) ED *(T* < 3) *ED T (T* 3).

1.4 *C[i* - 4] *C[i* - 4] ED *(T >>* 28) ED *(T >>* 29).

1. *T* C[5] AND OxFFFFFFF8. {Clear bits 0, 1 and 2 of C[5]}
2. C[0] C[0] *(T <* 4) el) *(T <* 3) el) *T* el) *(T* 3).
3. C[1] C[1] ED *(T >>* 28) ED *(T >>* 29).
4. C[5] C[5] AND 0x00000007. {Clear the unused bits of C[5]}
5. Return((C[5], C[4], C[3], C[2], C[1], C[0])).

**3.3 Squaring**

Squaring a polynomial is much faster than multiplying two arbitrary polynomials
  
since squaring is a linear operation in 72m; that is, if *a(x) =* of *,* then

*a(x)2* = *ai X2i .* The binary representation of *a(x)2* is obtained by inserting a 0 bit between consecutive bits of the binary representation of *a(x).* To facilitate this process, a table of size 512 bytes can be precomputed for converting 8-bit polynomials into their expanded 16-bit counterparts [36].

**Algorithm 7.** Squaring

INPUT: a E g2m •

OUTPUT: a2 mod *f(x).*

1. *Precornputation.* For each byte v = (v7, , v1, vo), compute the 16-bit quantity

T(v) = (0, v7, ... ,O, v1,0, vo)•

1. For *i* from 0 to *t* - I\_ do

2.1 Let *A[i]* = (u3, u2, ul, uo) where each ui is a byte. 2.2 *C[2i] (T (4) , T(u0)), C[2i* (T(u3),T(u2)).

1. Compute *b(x) = c(x)* mod *f (x).*
2. Return(b).

**3.4 Inversion**

Algorithm 8 computes the inverse of a non-zero field element *a* E 72m using

a variant of the Extended Euclidean Algorithm (EEA) for polynomials. The algorithm maintains the invariants *ba df* = u and *ca* ef = v for some *d* and e which are not explicitly computed. At each iteration, if deg(u) > deg (v), then a partial division of u by v is performed by subtracting *xi* v from u, where *j =* deg(u) - deg(v). In this way the degree of u is decreased by at least 1, and on average by 2. Subtracting *xi c* from b preserves the invariants. The algorithm terminates when deg(u) = 0, in which case u = 1 and *ba df = 1;* hence

b = a-1 mod *f(x).*

**Algorithm 8.** Extended Euclidean Algorithm for inversion in F2m.

INPUT: a E F2m, a 0. OUTPUT: a-1 mod *f(x).*

1. b1,c0,ua,vf.
2. While deg(u) 0 do

2.1 *j* deg(u) - deg(v).

2.2 If *j* < 0 then: u v, c, -*j.*

2.3 u *u xiv, b -b x3c.*

1. Return(b).

The Almost Inverse Algorithm (AIA, Algorithm 9) is from [36]. For *a* E IF2m, *a 0* 0, a pair *(b, k)* is returned where *ba = Xk* (mod *f (x)).* A reduction is then applied to obtain *a-1 = bx-k* mod *f (x).* The invariants are *ba df = Uxk* and *ca f = vxk* for some *d* and e which are not explicitly calculated. After step 2, both u and v have a constant term of 1; after step 5, u is divisible by *x* and hence the degree of u is always reduced at each iteration. The value of *k* is incremented in step 2.1 to preserve the invariants. The algorithm terminates when u = 1, giving *ba df = xk .* While EEA eliminates bits of u and v from left to right (high degree to low degree), AIA eliminates bits from right to left. In addition, in AIA some bits are also lost on the left in the case deg(u) = deg(v) before step 5. Consequently, AIA is expected to take fewer iterations than EEA.

The reduction step can be performed as follows. Let *s =* min{i > 1 f2 = 1}, where *f (x) = fmxm + • • • + fix + fo.* Let b' be the polynomial formed by the *s* rightmost bits of b. Then *VI' b* is divisible by *xs* and b" = *(bif +b)Ixs* has degree less than m; thus b" = *bx- s* mod *f(x).* This process can be repeated to finally obtain *b7-k* mod *f (x).* The reduction polynomial is said to be *suitable if s >* 32, since then fewer iterations are required in the reduction step.

**Algorithm 9.** Almost Inverse Algorithm for inversion in F2m.

INPUT: a E F2m, a 0.

OUTPUT: b E F2m and *k E* [0, 2m - 1] such that ba = (mod *f(x)).*

1. b1,c0,ua,vf,k0.
2. While *x* divides u do:

2.1 u *k +* 1.

1. If u = 1 then return(b,k).
2. If deg(u) < deg(v) then: u v, b *c.*
3. *uu-Fv,bb-Fc.*
4. Goto step 2.

Algorithm 10 is a modification of Algorithm 9, producing the inverse directly. Rather than maintaining the integer *k,* the algorithm performs a division of b whenever u is divided by *x.* Note that if b is not divisible by *x,* then b is replaced by b *f* (and *d* by *d - a)* in step 2.2 before the division. On termination, *ba df =* 1, whence b = a-1 mod *f (x).*

**Algorithm 10.** Modified Almost Inverse Algorithm for inversion in g2m.

INPUT: a E F2m, a 0. OUTPUT: a-1 mod *f(x).*

1. b1,e0,ua, vf.
2. While *x* divides u do:

2.1 u *u/x.*

2.2 If *x* divides b then *b* +- *b/x;* else b (b + *f)/x.*

1. If u = 1 then return(b).
2. If deg(u) < deg(v) then: u -)- v, b -)- *c.*
3. *uu-Fv,bb-Fc.*
4. Goto step 2.

Step 2 of AIA is simpler than that in MAIA. In addition, the b and *c* appearing in these algorithms grows more slowly in AIA. Thus one can expect AIA to outperform MAIA if the reduction polynomial is suitable, and conversely.

3.5 Timings

Table 3 presents timing results for operations in the fields 7

\_ 2163, 72233 and 72283. The field arithmetic was implemented in C and the timings obtained on a Pen­tium II 400 MHz workstation.

**Table 3.** Timings (in *ps)* for operations in F2163, F2233 and F2283. The reduction polynomials are, respectively, *f(x)* = *x163* + *x7* + X6 + X3 + 1, *f(X)* = *x233* + *x74* + 1, and *f(x)* = x283 + x12 + *x7 + x5 + 1.*

|  |  |  |  |
| --- | --- | --- | --- |
|  | m = 163 | m = 233 | m = 283 |
| *Addition* | 0.10 | 0.12 | 0.13 |
| *Modular reduction* (Algorithm 6) | 0.18 | 0.22 | 0.35 |
| *Multiplication* (including reduction) |  |  |  |
| Shift-and-add (Algorithm 1) | 16.36 | 27.14 | 37.95 |
| Right-to-left comb (Algorithm 2) | 6.87 | 12.01 | 14.74 |
| Left-to-right comb (Algorithm 3) | 8.40 | 12.93 | 15.81 |
| LR comb with windows of size 4 (Algorithm 4) | 3.00 | 5.07 | 6.23 |
| Karatsuba | 3.92 | 7.04 | 8.01 |
| *Squaring* (Algorithm 7) | 0.40 | 0.55 | 0.75 |
| *Inversion* |  |  |  |
| Extended Euclidean Algorithm (Algorithm 8) | 30.99 | 53.22 | 70.32 |
| Almost Inverse Algorithm (Algorithm 9) | 42.49 | 68.63 | 104.28 |
| Modified Almost Inverse Algorithm (Algorithm 10) | 40.26 | 73.05 | 96.49 |

As expected, addition, modular reduction, and squaring are relatively inex­pensive compared to multiplication and inversion. The left-to-right comb method with windows of size 4 is the fastest multiplication algorithm, however it requires a modest amount of extra storage (e.g., 336 bytes for 14 polynomials in the case

m = 163). Our implementation of Karatsuba's algorithm is competitive and re­quires a similar amount of storage since the base multiplications were performed using the left-to-right comb method with windows of size 4.

Formulas which do not require inversions for adding and doubling points in projective coordinates can be derived by first converting the points to affine

We found the Extended Euclidean Algorithm to be faster than the Almost Inverse Algorithm and the Modified Almost Inverse Algorithm, contrary to the findings of [36] and [7]. This discrepancy is partially explained by the unsuitable form of the reduction polynomial for m = 163 and m = 283 (see [7]). Also, we found that AIA and MAIA were more difficult to optimize than EEA without resorting to hand-coded assembler. In any case, the ratio of the fastest inversion method to the fastest multiplication method was found to be roughly 10 to 1, again contrary to the roughly 3 to 1 ratio reported in [36], [6] and [7]. This discrepancy could be attributed to a considerably faster implementation of mul­tiplication in our work. As a result, we chose to represent elliptic curve points in projective coordinates instead of affine coordinates as was done in [36] and [7] (see §4).

**4 Elliptic Curve Point Representation**

**Affine coordinates.** Let *E* be an elliptic curve over 72– given by the (affine)
  
equation y2 + xy = x3 + *axe + b,* where *a* E {0, 1}. Let Pi = (xi, yi) and

P2 = (X2, y2) be two points on *E* with Pi A -P2. Then the coordinates of

P3 = P1 + P2 = (x3, y3) can be computed as follows:

x3 = A2 + A + xi + X2 + a, )3=(1+3)+3+y1, where

*A=*

yi + Y2 i yi

f Pl A P2, and A= = — + xi if Pi = P2. (1)

Xi + X2 Xi

In either case, when Pi A P2 (general addition) and Pi = P2 (doubling), the formulas for computing P3 require 1 field inversion and 2 field multiplications—as justified in §3.5, we can ignore the cost of field additions and squarings.

**Projective coordinates.** In situations where inversion in 72– is expensive relative to multiplication, it may be advantageous to represent points using projective coordinates of which several types have been proposed. In *standard* projective coordinates, the projective point *(X : Y : Z), Z* A 0, corresponds to the affine point *(X/Z,* Y/Z). The projective equation of the elliptic curve is Y2Z+XY *Z* = *X3 +aX2 Z -kbZ3 .* In *Jacobian* projective coordinates [5], the pro­jective point *(X : Y : Z), Z* A 0, corresponds to the affine point *(X/Z2,* Y/Z3) and the projective equation of the curve is Y2 + *XYZ = X3 + aX2Z2 + bZ6 .* In [25], a new set of projective coordinates was introduced. Here, a projective point *(X : Y : Z), Z* A 0, corresponds to the affine point *(X/Z,* Y/Z2), and the projective equation of the curve is

y2 *+ XYZ = X3 Z + aX2Z2 + bZ4 .* (2)

coordinates, then using the formulas (1) to add the affine points, and finally clearing denominators. Also of use in left-to-right point multiplication methods (see §5.1) is the addition of two points using mixed coordinates—one point given in affine coordinates and the other in projective coordinates. Doubling formulas for the projective equation (2) are: 2(X1 : Vi : Z1) = (X3 : Y3 : Z3), where

Algorithm 11 is the additive version of the basic repeated-square-and-multiply method for exponentiation.

Z3 = X? • *Z?,* X3 = XLii + b • Z`il, Y3 = *bZ'il •* Z3 + X3 • *(aZ3+Yi2 +b4).* (3)

Formulas for addition in mixed coordinates are: (Xi : 171 : Z1) + (X2 : Y2 : 1) = (X3 : Y3 : Z3 ), where

A = Y2 • Zi2 + Yi , *B =* X2 • Zi + Xi , *C* = Zi • *B, D = B2 • (C + aZ?),*Z3 = C2, *E = A • C,* X3 = A2 + *D + E, F =* X3 + X2 • Z3,

*G* — X3 + Y2 • Z3, Y3 = *E • F +* Z3 • *G.* (4)

The field operation counts for point addition and doubling in the various coordinate systems are listed in Table 4. Since our implementation of inversion is at least 10 times as expensive as multiplication (see §3.5), unless otherwise stated, all our elliptic curve operations will use projective coordinates.

**Table 4.** Operation counts for point addition and doubling.

|  |  |  |  |
| --- | --- | --- | --- |
| Coordinate system | General addition | General addition  (mixed coordinates) | Doubling[ |
| Affine  Standard projective *(X/Z,* Y/Z) Jacobian projective *(X/Z2,* Y/Z3) Projective *(X/Z,* Y/Z2) | 1/, 2M 13M 14M 14M | 12M  10M  9M | 1/, 2M *7M* 5M 4M |

5 **Point Multiplication**

This section considers methods for computing *kP,* where *k* is an integer and *P* is an elliptic curve point. This operation is called *point multiplication* or *scalar multiplication,* and dominates the execution time of elliptic curve cryptographic schemes. We will assume that #E(IF2— ) = *nh* where n is prime and *h* is small (so n ,,-,-,; 2m), *P* has order n, and *k* ER [1, n —1]. In §5.1 we consider techniques which do not exploit any special structure of the curve. In §5.2 we study techniques for Koblitz curves which use the Frobenius endomorphism. In both cases, one can take advantage of the situation where *P* is a fixed point (e.g., the base point in elliptic curve domain parameters) by precomputing some data which depends only on *P.* For surveys of exponentiation methods, see [11] and [28].

**5.1 Random Curves**

**Algorithm 11.** (Left-to-right) binary method for point multiplication

INPUT: *k = (kt\_1,... P* E E(g2m).

OUTPUT: *kP.*

1. *Q4-O.*
2. For *i* from *t -1* downto 0 do 2.1 *Q* 2Q.

2.2 If *=* 1 then *Q P.*

1. Return(Q).

The expected number of ones in the binary representation of *k is t/2* m/2, whence the expected running time of Algorithm 11 is approximately m/2 point additions and m point doublings, denoted 0.5mA + *mD.* If affine coordinates (see §4) are used, then the running time expressed in terms of field operations is 3mM +1.5m/, where I denotes an inversion and *M* a field multiplication. If pro­jective coordinates (see §4) are used, then *Q* is stored in projective coordinates, while *P* can be stored in affine coordinates. Thus the doubling in step 2.1 can be performed using (3), and the addition in step 2.2 can be performed using (4). The field operation count of Algorithm 11 is then 8.5mM + (2M + 1/) (1 inversion and 2 multiplications are required to convert back to affine coordinates).

If *P = (x, y)* E E(72,-.) then *-P = (x,x y).* Thus subtraction of points on an elliptic curve over a binary field is just as efficient as addition. This mo­tivates using a *signed digit representation k =Ei=*/-

*okiT ,* where *ki* E {0, +1}. A particularly useful signed digit representation is the *non-adjacent form* (NAF) which has the property that no two consecutive coefficients *ki* are nonzero. Every positive integer *k* has a unique NAF, denoted NAF(k). Moreover, NAF*(k)* has the fewest non-zero coefficients of any signed digit representation of *k.* NAF(k) can be efficiently computed using Algorithm 12 [37].

**Algorithm 12.** Computing the NAF of a positive integer

INPUT: A positive integer *k.* OUTPUT: NAF(k).

1. *i* 0.
2. While *k >* 1 do

2.1 If *k* is odd then: 2 - *(k* mod 4), *k -ki;*

2.2 Else: *ki* 0.

2.3 *kk12,ii-F1.*

1. Return((ki\_i,ki-2,...

Algorithm 13 modifies Algorithm 11 by using NAF(k) instead of the binary representation of *k.* It is known that the length of NAF(k) is at most one longer than the binary representation of *k.* Also, the average density of non-zero coeffi­cients among all NAFs of length *1* is approximately 1/3 [32]. It follows that the expected running time of Algorithm 13 is approximately (m/3)A + *mD.*

**Algorithm 13.** Binary NAF method for point multiplication

INPUT: NAF(k) = ki2i, *P* E E(g2m.).

OUTPUT: *kP.*

1. *Q4-O.*
2. For *i* from / - 1 downto 0 do 2.1 *Q 2Q.*

2.2 If *ki =* 1 then *Q + P.*

2.3 If *ki =* -1 then *QQ - P.*

1. Return(Q).

If some extra memory is available, the running time of Algorithm 13 can be decreased by using a window method which processes w digits of *k* at a time. One approach we did not implement is to first compute NAF(k) or some other signed digit representation of *k* (e.g., [23] or [30]), and then process the digits using a sliding window of width w. Algorithm 14 from [37], described next, is another window method.

A *width-w NAF* of an integer *k* is an expression *k =* Eli-o k222, where each non-zero coefficient *ki* is odd, k2 *<* 2w-1, and at most one of any w con­secutive coefficients is nonzero. Every positive integer has a unique width-w NAF, denoted NAFw *(k).* Note that NAF2(k) = NAF(k). NAFw *(k)* can be ef­ficiently computed using Algorithm 12 modified as follows: in step 2.1 replace *"ki* 2- *(k* mod 4)" by *"kik* mods 221", where *k* mods 221 denotes the integer u satisfying u = *k* (mod 2w) and -2w-1 < u < 2w-1. It is known that the length of NAFw *(k)* is at most one longer than the binary representation of *k.* Also, the average density of non-zero coefficients among all width-w NAFs of length *1* is approximately 1/(w + 1) [37]. It follows that the expected running time of Al­gorithm 14 is approximately (1D + (2'2 - 1)A) (m/(w + 1)A + *mD).* When using projective coordinates, the running time in the case m = 163 is minimized when w = 4. For the cases m = 233 and m = 283, the minimum is attained when w = 5; however, since the running times are only slightly greater when w = 4, we selected w = 4 for our implementation.

**Algorithm 14.** Window NAF method for point multiplication

INPUT: Window width w, NAF„(k) = Eli:01 *ki2i, P* E E(g2m). OUTPUT: *kP.*

1. Compute *Pi* = *iP,* for *i* E {1,3, 5, , 2w-1 - 1}.
2. *Q4-O.*
3. For *i* from / - 1 downto 0 do

3.1 *Q 2Q.*

3.2 If *ki* 0 then:

If *ki >* 0 then *Q Q + Ph, ;*

Else *Q - Ph,*

1. Return(Q).

Algorithm 15 is from [26] and is based on an idea of Montgomery [31]. Let Qi = yl), Q2 = (x2, y2) with Qi A +Q2. Let Qi + Q2 = (X3, y3) and Qi - Q2 = (x4, y4). Then using the addition formulas (1), it can be verified that

2w-

*d-1*

*kP =*

i=0

*j=1*

*j=1*

2w-1

E 2-p)

*ki(2w =*

= Q2w-1+ (Q2w-1+ Q2w-2) + • • • + (Q2w—l+Q2w-2 + • • • + Q1)• (7)

( xi )2 X1+ X2 •

*X3 =* X4 + *Xi*

+ X2

(5)

Thus, the x-coordinate of Qi + Q2 can be computed from the x-coordinates of Ql, Q2 and Qi - Q2. Iteration *j* of Algorithm 15 for determining *kP* computes *Ti* = *(1P, (1 + 1)P),* where *1* is the integer given by the *j* leftmost bits of *k.* Then Ti+i = (21P, (21 +1)P) or ((21 +1)P, (21 +2)P) if the *(j* + 1)st leftmost bit of *k is* 0 or 1, respectively. Each iteration requires one doubling and one addition using (5). After the last iteration, having computed the x-coordinates of *kP =* yi) and *(k* + 1)P = (x2, y2), the y-coordinate of *kP* can be recovered as:

= *x-1(x1 x)[(xi x)(x2 x)* + x2 + y] + y. (6)

Equation (6) is derived using the addition formula (1) for computing the *x-*coordinate x2 of *(k* + 1)P from *kP =* yi) and *P =* y). Algorithm 15 is presented using standard projective coordinates (see §4). The approximate running time is 6mM + (1/ + 10M). One advantage of Algorithm 15 is that it does not have any extra storage requirements.

**Algorithm 15.** Montgomery point multiplication

INPUT: *k = (kt\_1, ,k0)2* with kt\_i = 1, *P = (x,y)* E E(g2m).

OUTPUT: *kP.*

1. Xi 1, X2 X4 b, Z2 X2. {Compute (P, 2P)}
2. For *i* from *t -* 2 downto 0 do

2.1 If *ki =* 1 then

*T* (X1 Z2 + X2 Z1 )2 , X1*xZi* + X2TZ2 •

*T* X2, X2 *bZj,* Z2 *T24.*

2.2 Else

*T* Z2*,* Z2 (X1 Z2 + X2 Z1)2, X2 *X* Z2 + X1 X2 *ZIT. T X1, X1 X`11 +b4, Z1 -T2Z,2.*

1. x3 Xi /Z1 .
2. y3 *(X* + X1 /Z1 )[(X1 *XZ1)(X2 XZ2* + (X2 + Y)(Z1 Z2 )1(XZ1 Z2)-1 y.
3. Return((x3, y3)).

If the point *P* is fixed and some storage is available, then point multiplication can be sped up by precomputing some data which depends only on *P.* For example, if the points *2P,* 22P, , 2t-IP are precomputed, then the right-to-left binary method has expected running time (m/2)A (all doublings are eliminated). In [3], a refinement of this idea was proposed. Let *(kd\_* 1, , ki, ko)2w be the 2w-ary representation of *k,* where *d = rt/wl,* and let *Qi* = Ei:k *=i 2wi P.* Then

Algorithm 16 is based on this observation. Its expected running time is approxi­mately ((d(2w — 1)/2w — 1) + (2w — 2))A. Note that if projective coordinates are used, then only the additions in step 3.1 are in mixed coordinates.

**Algorithm 16.** Fixed-base windowing method

INPUT: Window width w, d = Ft/wl, *k = P* E E(g2m)•

OUTPUT: *kP.*

1. *Precornputation.* Compute *Pi* = 2w2P, 0 < *i* < d — 1.
2. *A+-O, B+-O.*
3. For *j* from 2u) — 1 downto 1 do

3.1 For each *i* for which *ki = j* do: *B B Pi.* {Add *Qi* to B} 3.2 *AA-FB.*

1. Return(A).

In the comb method, proposed in [24], the binary representation of *k* is written in w rows, and the columns of the resulting rectangle are processed one column at a time. We define *[aw\_* 1, , a2, al, *ao]P = aw\_12(w-1) dP + • • • +* +a222dP + *a* 12dP *aoP,* where *d =* [t/w1 and *ai* E 7Z2. The expected running time of Algorithm 17 is *((d — 1) (2W —* 1)/2w)A *(d — 1)D.*

**Algorithm 17.** Fixed-base comb method

INPUT: Window width w, d = Ft/wl, *k = (kt\_1,...,k1,ko)2, P* E E(g2m)• OUTPUT: *kP.*

1. *Precornputation.* Compute *[aw\_1,...* ,al,ao]P V(aw-1,...,al,ao) E Z;v.
2. By padding *k* on the left with 0's if necessary, write *k = K W 11 .101 K °* where
     
   each *Ki* is a bit string of length d. Let R7 denote the ith bit of /(3.
3. *Q O.*
4. For *i* from d — 1 downto 0 do

4.1 *Q* 2Q.

4.2 *Q [Kr-1,*

1. Return(Q).

From Table 5 we see that the fixed-base comb method is expected to out­perform the fixed-base window method for similar amounts of storage. For our implementation, we chose w = 4 for the fixed-base comb method.

**Table 5.** Comparison of fixed-base window and fixed-base comb methods. w is the window width, *S* denotes the number of points stored in the precomputation phase, and *T* denotes the number of field operations. Affine coordinates were used for fixed-base window, and projective coordinates were used for fixed-base comb.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | w = 2 | | w = 3 | | w = 4 | | w = 5 | | w = 6 | | *w = 7* | | w = 8 | |
| Method | S | T | S | T | S | T | S | T | S | T | S | T | S | T |
| Fixed-base window | 81 | 756 | 54 | 648 | 40 | 624 | 32 | 732 | 27 | 1068 | 23 | 1788 | 20 | 3288 |
| Fixed-base comb | 2 | 885 | 6 | 660 | 14 | 514 | 30 | 419 | 62 | 363 | 126 | 311 | 254 | 272 |

**5.2 Koblitz Curves**

To compute *kP,* one can find TNAF(k) using Algorithm 18, and then use (8). Now, the length *1(a)* of TNAF *(a)* satisfies log2(N(a)) - 0.55 < /(a) <

Koblitz curves are elliptic curves defined over 72, and were first proposed for cryptographic use in [20]. The primary advantage of Koblitz curves is that point multiplication algorithms can be devised that do not use any point doublings. All the algorithms and facts stated in this section are due to Solinas [37].

There are two Koblitz curves: *E0* : y2 + xy = x3 + 1 and E1 : y2 + xy = x3 + x2 + 1. Let *p = (-1)1'.* We have *#E,(72) =* 3 - *p.* We assume that *#E,(72-)* is almost prime, i.e., *#E,(72-) = hn,* where n is prime and *h =*

The number of points is given by *#E,(72-) =* 2' + 1 - Vri, where {V, } is the Lucas sequence defined by Vo = 2, V1 = Ec, *171,+1 = - 2171,\_1* for *k > 1.*

Since *Ea* is defined over TF2-, the *Frobenius map* T : Ea (72,-.) Ea (72m) defined by *7(0)* = *0, 7((x,* y)) = (x2, y2) is well-defined. Moreover, it can be efficiently computed since squaring in 72- is relatively inexpensive (see §3.5). It is known that (72 + 2)P = *,urP* for all *P* E *Ea(72m).* Hence the Frobenius map can be regarded as the complex number T satisfying 72 + 2 = *pr,* i.e., T = *(p+ A/-7)/2.* It now makes sense to multiply points in *E,(72-)* by elements of the ring Z[7]: if ui\_171-1 + • • • + uir uo E E[r] and *P* E *E,(72m),* then

(u1\_171-1 *+ • • • + uir uo)P = u1\_17-1-1(P) + • • • + uir(P) + u0P.* (8)

The strategy for developing an efficient point multiplication algorithm is find a "nice" expression for *k* of the form *k =* Eli-o UiTi, and then use (8) to compute *kP.* Here, "nice" means that *1* is relatively small and the non-zero coefficients ui are small (e.g., +1) and sparse.

Since 72 + 2 = *pr,* every element in E[7] can be expressed in canonical form r0 rir, where r0, r1 E E. E[r] is a Euclidean domain, and hence also a unique factorization domain, with respect to the norm function N(ro 7-17) = r(2) prori +2r?. The norm function is multiplicative. We have *N(7)* = 2, *N(7-* 1) = *h, N(rm - 1) = #E,(72m),* and N(S) = n where S = (7m - 1)/(7 - 1).

A *7-ache NAF* or *TNAF* of an element *K* E E[r] is an expression *K = Eli—o u2 Ti* where ui E {0, +1}, and no two consecutive coefficients ui are nonzero. Every *K* E E[r] has a unique TNAF, denoted TNAF *(K),* which can be efficiently computed using Algorithm 18.

**Algorithm 18.** Computing the TNAF of an element in Z[7]

INPUT: *i =* 7'0 ri 7 E Z[7].

OUTPUT: TNAF(K).

1. *i* O.
2. While ro 0 or ri 0 do

2.1 If ro is odd then: ui 2 — (ro — 2r1 mod 4), ro ro — ui;

2.2 Else: ui 0.

2.3 tro, ro 7-1 +pro/2, ri *i* +1.

1. Return((ui\_i , ui-2, , u1, uo}}.

log2 *(N(a)) +* 3.52 when *1 >* 30. It follows that *1(k)* 2 log2 *k,* which is twice

as long as the length of NAF *(k).* To circumvent the problem of a long TNAF, notice that if *p = k* mod S then *kP = pP* for all points *P* of order n (because *SP = 0).* Since *N(p) < N(S) = n,* it follows that *l(p) m,* which suggests that TNAF(p) should be used instead of TNAF(k) for computing *kP.* Algorithm 19 is an efficient method for computing an element *p'* E E[r] such that *p' = k* (mod 5); we write *p' = k* partmod S. The parameter *C* ensures that TNAF(p') is not much longer than TNAF(p). In fact, *l(p) < m + a,* and if *C* > 2 then *l(p')* < m + *a +* 3. Also, the probability that *p' p* is less than 2-(C-5).

**Algorithm 19.** Partial reduction modulo

INPUT: *k* E [1, n - 1], *C* > 2, so = do *+Pdi,* si = -d1, where S = do + d17.

OUTPUT: *p' = k* partmod 5.

1. *k'* Lk/2a-c+(71-9)/2].
2. For *i* from 0 to 1 do 2.1 *g' si • k', j' V.* [g72"1], *Ai* L(g` +j`)/2("1+5)/2 2J /2c.

2.2 *fi +12-],* Ai - *fi, hi* O.

1. *ri* 2/70 +
2. If *ri* > 1 then

4.1 If *rio* - *3pr11* < -1 then h1 i\_c; else ho 1.

Else

4.2 If 770 *4pril* > 2 then h1 *p.*

1. If *ri* < -1 then

5.1 If *rio* - *3pr11* > 1 then h1 *-p;* else ho -1.

Else

5.2 If 770 4pr11 < -2 then h1 *-p.*

1. qo *Jo ho,* h1, ro *k - (so + psi )qo -* si qo - soqi.
2. Return(ro riT)•

The average density of non-zero coefficients among all TNAFs of length *1* is approximately 1/3. Hence Algorithm 20 which uses TNAF(p') for computing *kP* has an expected running time of approximately (m/3)A.

**Algorithm 20.** TNAF method for point multiplication

INPUT: TNAF(p`) = Eli:01 ui T2 where *p' = k* partmod *5, P* E Ea(g2m)• OUTPUT: *kP.*

1. *Q4-O.*
2. For *i* from / - 1 downto 0 do 2.1 *Q 7Q.*

2.2 If ui = 1 then *Q + P.*

2.3 If ui = -1 then *Q - P.*

1. Return(Q).

We now extend Algorithm 20 to a window method analogous to Algorithm 14. Let *tt, =* 2Utu\_1U,71 mod 221, where {Uk } is the Lucas sequence defined by U0 = 0, = 1, *Uk+1 = itUf, - 2Uk\_i* for *k >* 1. Then the map Ow : E[r] E2w

induced by T *tt,* is a surjective ring homomorphism with kernel *fa* E E[r] :
  
T21 *cc}.* It follows that a set of distinct representatives of the congruence classes

modulo Tw whose elements are not divisible by T is {+1, +3, , ±(2w-1 -

1)}. Define *ai* = *i* mod Tw for *i* E {1, 3, ... , 2w' - 1}. A *width-w TNAF* of *K* E E[7], denoted TNAFw (K), is an expression *K = Eli-o UiTi* where ui E {0, *+ai,* +a3, , +a2.\_i\_ I}, and at most one of any w consecutive coefficients is nonzero. Algorithm 21 is an efficient method for computing TNAFw *(K).*

**Algorithm 21.** Computing a width-w TNAF of an element in 7Z[7]

INPUT: w, tw, at, = /3„ 7„7 for u E {1,3,... ,2w-1 -1}, *p = ro ri7 E Z[7].*

OUTPUT: TNAF„ (p).

1. *i* 0.
2. While ro 0 or ri 0 do

2.1 If ro is odd then

u ro ttu mods 2").

If a> 0 then *s* 1; else s+--1, a--u.

ro ro - - s7„, ui *sat,.*

2.2 Else: ui 0.

2.3 ro, ro )"'l +pro/2, ri *i* 1.

1. Return((u2\_i,ui-2,... ,tii,uo))•

The average density of non-zero coefficients among all TNAFws of length *1* is approximately 1/(w + 1). Since the length of TNAFw *(p')* is approximately */(p'),* it follows that Algorithm 22 which uses TNAF(p') for computing *kP* has an expected running time of approximately (2w-2 - 1 + m/(w 1))A.

**Algorithm 22.** Window TNAF method for point multiplication

INPUT: TNAF,v(p`) *=* where *p' =k* partmod 5, *P E Ea(g2m)•*

OUTPUT: *kP.*

1. Compute Pt, = *at,P,* for u E {1, 3, 5, , 2")-1 - 1}.
2. *Q4-O.*
3. For *i* from / - 1 downto 0 do

3.1 *Q 7Q.*

3.2 If ui 0 then:

Let u be such that at, = ui or cf\_t, = -ui. If u > 0 then *Q P„;*

Else *Q -*

1. Return(Q).

If the point *P is* fixed, then the points Pt, in step 1 of Algorithm 22 can be precomputed. The resulting method, which we call fixed-base window TNAF (or Algorithm 23) has an expected running time of (m/(w 1))A.

5.3 **Timings**

In Table 6 we present rough estimates of costs in terms of both elliptic curve operations and field operations for the various point multiplication methods in the case m = 163. These estimates serve as a guideline for comparing point mul­tiplication algorithms without concern for platform or implementation specifics.

**Table 6.** Rough estimates of point multiplication costs for m = 163.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Method | Coordinates | w | Points  stored | EC operations | | Field operations | | |
| A | *D* | *M* | I | Total' |
| Binary | affine | — | 0 | 82 | 163 | 490 | 245 | 2940 |
| (Algorithm 11) | projective | — | 0 | 82 | 163 | 1390 | 1 | 1400 |
| Binary NAF | affine | — | 0 | 54 | 163 | 434 | 217 | 2604 |
| (Algorithm 13) | projective | — | 0 | 54 | 163 | 1140 | 1 | 1150 |
| Window NAF | affine |  | 3 | 36 | 164 | 400 | 200 | 2400 |
| (Algorithm 14) | projective |  | 3 | 3b+33 | 164 | 955 | 5 | 1005 |
| Montgomery | affine | — | 0 | 163G | 163 | 329 | 327 | 3600 |
| (Algorithm 15) | projective | — | 0 | 163G | 163 | 988 | 1 | 998 |
| Fixed-base window | affine |  | 27 | 89 | 0 | 178 | 89 | 1068 |
| (Algorithm 16) | projective | CO | 27 | 27+62d | 0 | 1113 | 1 | 1123 |
| Fixed-base comb | affine |  | 14 | 38 | 40 | 156 | 78 | 936 |
| (Algorithm 17) | projective | '71-1 | 14 | 38 | 40 | 504 | 1 | 514 |
| TNAF | affine | — | 0 | 54 | 0 | 108 | 54 | 648 |
| (Algorithm 20) | projective | — | 0 | 54 | 0 | 488 | 1 | 498 |
| Window TNAF | affine |  | 7 | 34 | 0 | 68 | 34 | 408 |
| (Algorithm 22) | projective | LfJ | 7 | 71)+27 | 0 | 261 | 8 | 341 |
| Fixed-base window TNAF | affine | CO CO | 15 | 23 | 0 | 46 | 23 | 276 |
| (Algorithm 23) | projective | 15 | 23 | 0 | 209 | 1 | 219 |

a Total cost in field multiplications assuming 1/ = 10M.

*b* Additions are in affine coordinates

Additions using formula (5).

*d* Additions are not in mixed coordinates.

**Table 7.** Timings (in *ps)* for point multiplication on random and Koblitz curves over g2163 g2233 and g2283. Unless otherwise stated, projective coordinates were used.

|  |  |  |  |
| --- | --- | --- | --- |
|  | m = 163 | m = 233 | m = 283 |
| *Random curves* |  |  |  |
| Binary (Alg 11, affine coordinates) | 9178 | 21891 | 34845 |
| Binary (Alg 11) | 4716 | 10775 | 16123 |
| Binary NAF (Alg 13) | 4002 | 9303 | 13896 |
| Window NAF with w = 4 (Alg 14) | 3440 | 7971 | 11997 |
| Montgomery (Alg 15) | 3240 | 7697 | 11602 |
| Fixed-base comb with w = 4 (Alg 17) | 1683 | 3966 | 5919 |
| *Koblitz curves* |  |  |  |
| TNAF (Alg 20) | 1946 | 4349 | 6612 |
| Window TNAF with w = 5 (Alg 22) | 1442 | 2965 | 4351 |
| Fixed-base window TNAF with w = 6 (Alg 23) | 1176 | 2243 | 3330 |

Table 7 presents timing results for the NIST curves B-163, B-233, B-283, K-163, K-233 and K-283. The implementation was done in C and the timings were obtained on a Pentium II 400 MHz workstation. The big number library in OpenSSL [35] was used to perform multiprecision integer arithmetic.

The timings in Table 7 are consistent with the estimates in Table 6. In gen­eral, point multiplication on Koblitz curves is significantly faster than on random curves. The difference is especially pronounced in the case where *P* is not known a priori (Montgomery vs. window TNAF). For the window TNAF method with w = 5 and m = 163, the timings for the three components were 50 *ps* for partial reduction (Algorithm 19), 126ps for width-w TNAF computation (Algo­rithm 21), and 1266/is for elliptic curve operations (Algorithm 22).

6 ECDSA Elliptic Curve Operations

The execution times of elliptic curve cryptographic schemes such as the ECDSA [16, 21] are typically dominated by point multiplications. In ECDSA, there are two types of point multiplications, *kP* where *P* is fixed (signature generation), and *kP +1Q* where *P* is fixed and *Q* is not known a priori (signature verification). One method to speed the computation of *kP +1Q* is simultaneous multiple point multiplication (Algorithm 24), also known as Shamir's trick [8]. Algorithm 24 has an expected running time of (22w -3)A+ *((d-* 1)(22w -1)/22wA+ *(d-1)wD),* and requires storage for 22w points.

**Algorithm 24.** Simultaneous multiple point multiplication

INPUT: Window width w, *k = (kt\_1, ... ,ki,ko)2, I = (4-1, • • • ,11,102, P, Q.* OUTPUT: *kP + 1Q.*

1. Compute *iP + jQ* for all *i, j* E [0, 2W - 1].
2. Write *k = (kd-1,k1 ,k°)* and / = *(/d-1,... XX)* where each *ki* and /i is a bitstring of length w, and d = [t/wl.
3. *R O.*
4. For *i* from d - 1 downto 0 do

4.1 *R 2") R.*

4.2 *R R + (ki P -FliQ).*

1. Return(R).

Table 8 lists the most efficient methods for computing *kP, P* fixed, for random curves and Koblitz curves. For each type of curve, two cases are distinguished—when there is no extra memory available and when memory is not heavily con­strained. Table 9 does the same for computing *kP + 1Q* where *P* is fixed and *Q* is not known a priori.

7 Conclusions

We found that significant performance improvements can be achieved by the use of projective coordinates over affine coordinates due to the high inversion to multiplication ratio observed in our implementation.

**Table 8.** Timings (in *ps)* of the fastest methods for point multiplication *kP, P* fixed, in ECDSA signature generation.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Curve Memory  type constrained? | Fastest  method | m=163 | m=233 | m=283 |
| Random No | Fixed-base comb (w = 4) | 1683 | 3966 | 5919 |
| Yes | Montgomery | 3240 | 7697 | 11602 |
| Koblitz No | Fixed-base window TNAF (w=6) | 1176 | 2243 | 3330 |
| Yes | TNAF | 1946 | 4349 | 6612 |

**Table 9.** Timings (in *ps)* of the fastest methods for point multiplications *kP + 1Q, P* fixed and *Q* not known a priori, in ECDSA signature verification.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Curve Memory  type constrained? | Fastest  method | m=163 | m=233 | m=283 |
| Random No | Montgomery + | 5005 | 11798 | 17659 |
|  | Fixed-base comb (w = 4) |  |  |  |
| No | Simultaneous (w = 2) | 4969 | 11332 | 16868 |
| Yes | Montgomery | 6564 | 15531 | 23346 |
| Koblitz No | Window TNAF (w = 5) + | 2702 | 5348 | 7826 |
|  | Fixed-base window TNAF (w=6) |  |  |  |
| Yes | TNAF | 3971 | 8832 | 13374 |

Implementing the specialized algorithms for Koblitz curves is straightfor­ward. Point multiplication for Koblitz curves is considerably faster than on random curves, yielding faster implementations of elliptic curve cryptographic schemes. For both random and Koblitz curves, substantial performance improve­ments can be obtained with only a modest commitment of memory for storage of tables and precomputed data.

While some effort was made to optimize the code, it is likely that considerable performance enhancements can be obtained especially if the code is tuned for a specific platform. For example, the times for the AIA and MAIA methods (see §3.5) compared with inversion using EEA require some explanation. Even with optimization efforts (but in C only) and a suitable reduction trinomial in the m = 233 case, we found that the EEA implementation was significantly faster on the Pentium II. Non-optimal register allocation may have contributed to the relatively poor showing of AIA and MAIA, suggesting that a few hand-coded assembly sections may be desirable. Even with the same source code, compiler and hardware differences are apparent. On a Sun Ultra, for example, we found that EEA required roughly 9 times as long as multiplication using the same code as on the Pentium II, and AIA and MAIA required approximately the same time as inversion using the EEA.

Despite the limitations of our analysis and implementation, we nonetheless hope that our work will serve as a benchmark for future efforts in this area.

**8 Future Work**

We did not implement the variant of Montgomery integer multiplication for 72—presented in [22]. We also did not implement the point multiplication method of [17] which uses point halvings instead of doublings since this method appears to be advantageous only when affine coordinates are employed.

We are currently investigating the software implementation of ECC over the MST-recommended prime fields, and a comparison with the MST-recommended binary fields. A careful and extensive study of ECC implementation in software for constrained devices such as smart cards, and in hardware, would be beneficial to practitioners. Also needed is a thorough comparison of the implementation of ECC, RSA, and discrete logarithm systems on various platforms, continuing the work reported in [7].

**Acknowledgements**

The authors would like to thank Mike Brown, Donny Cheung, Eric Fung, and Mike Kirkup for numerous fruitful discussions and for help with the implemen­tation and timings.

**References**

1. ANSI X9.62, *Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA),* 1999.
2. ANSI X9.63, *Public Key Cryptography for the Financial Services Industry: Elliptic Curve Key Agreement and Key Transport Protocols,* working draft, August 1999.
3. E. Brickell, D. Gordon, K. McCurley and D. Wilson, "Fast exponentiation with precomputation", *Advances in Cryptology — Eurocrypt '92,* LNCS **658,** 1993, 200­207.
4. M. Brown, D. Cheung, D. Hankerson, J. Hernandez, M. Kirkup and A. Menezes, "POP in constrained wireless devices", *Proceedings of the Ninth USENIX Security Symposium,* 2000.
5. D. Chudnovsky and G. Chudnovsky, "Sequences of numbers generated by addition in formal groups and new primality and factoring tests", *Advances in Applied Mathematics,* **7** (1987), 385-434.
6. E. De Win, A. Bosselaers, S. Vandenberghe, P. De Gersem and J. Vandewalle, "A fast software implementation for arithmetic operations in *GF(r)", Advances in Cryptology — Asiacrypt '96,* LNCS **1163,** 1996, 65-76.
7. E. De Win, S. Mister, B. Preneel and M. Wiener, "On the performance of signature schemes based on elliptic curves", *Algorithmic Number Theory, Proceedings Third Intern. Symp., ANTS-III,* LNCS **1423,** 1998, 252-266.
8. T. ElGamal, "A public key cryptosystem and a signature scheme based on discrete logarithms", *IEEE Transactions on Information Theory,* **31** (1985), 469-472.
9. S. Galbraith and N. Smart, "A cryptographic application of Weil descent", *Codes and Cryptography,* LNCS **1746,** 1999, 191-200.
10. P. Gaudry, F. Hess and N. Smart, "Constructive and destructive facets of Weil descent on elliptic curves", preprint, January 2000.
11. D. Gordon, "A survey of fast exponentiation methods", *Journal of Algorithms,* **27** (1998), 129-146.
12. J. Guajardo and C. Paar, "Efficient algorithms for elliptic curve cryptosystems", *Advances in Cryptology - Crypto '97,* LNCS **1294,** 1997, 342-356.
13. IEEE P1363, *Standard Specifications for Public-Key Cryptography,* 2000.
14. ISO/IEC 14888-3, *Information Technology - Security Techniques - Digital Signa­tures with Appendix - Part 3: Certificate Based-Mechanisms,* 1998.
15. ISO/IEC 15946, *Information Technology - Security Techniques - Cryptographic Techniques Based on Elliptic Curves,* Committee Draft (CD), 1999.
16. D. Johnson and A. Menezes, "The elliptic curve digital signature algorithm (ECDSA)", Technical report CORR 99-34, Dept. of C&O, University of Water­loo, 1999.
17. E. Knudsen, "Elliptic scalar multiplication using point halving", *Advances in Cryp­tology - Asiacrypt '99,* LNCS **1716,** 1999, 135-149.
18. D. Knuth, *The Art of Computer Programming - Seminumerical Algorithms,* Addison-Wesley, 3rd edition, 1998.
19. N. Koblitz, "Elliptic curve cryptosystems", *Mathematics of Computation,* **48** (1987), 203-209.
20. N. Koblitz, "CM-curves with good cryptographic properties", *Advances in Cryp­tology - Crypto '91,* LNCS **576,** 1992, 279-287.
21. N. Koblitz, A. Menezes and S. Vanstone, "The state of elliptic curve cryptography", *Designs, Codes and Cryptography,* **19** (2000), 173-193.
22. C. Koc and T. Acar, "Montgomery multiplication in *CF(2k)", Designs, Codes and Cryptography,* **14** (1998), 57-69.
23. K. Koyama and Y. Tsuruoka, "Speeding up elliptic cryptosystems by using a signed binary window method", *Advances in Cryptology - Crypto '92,* LNCS **740,** 1993, 345-357.
24. C. Lim and P. Lee, "More flexible exponentiation with precomputation", *Advances in Cryptology - Crypto '94,* LNCS **839,** 1994, 95-107.
25. J. Lopez and R. Dahab, "Improved algorithms for elliptic curve arithmetic in *CF(r)" , Selected Areas in Cryptography - SAC '98,* LNCS **1556,** 1999, 201-212.
26. J. Lopez and R. Dahab, "Fast multiplication on elliptic curves over *GF(2n)* without precomputation", *Cryptographic Hardware and Embedded Systems - CHES '99,* LNCS **1717,** 1999, 316-327.
27. J. Lopez and R. Dahab, "High-speed software multiplication in g2m" , preprint, 2000.
28. A. Menezes, P. van Oorschot and S. Vanstone, *Handbook of Applied Cryptography,* CRC Press, 1997.
29. V. Miller, "Uses of elliptic curves in cryptography", *Advances in Cryptology -Crypto '85,* LNCS **218,** 1986, 417-426.
30. A. Miyaji, T. Ono and H. Cohen, "Efficient elliptic curve exponentiation", *Pro­ceedings of ICICS '97,* LNCS **1334,** 1997, 282-290.
31. P. Montgomery, "Speeding up the Pollard and elliptic curve methods of factoriza­tion", *Mathematics of Computation,* **48** (1987), 243-264.
32. F. Morain and J. Olivos, "Speeding up the computations on an elliptic curve using addition-subtraction chains", *Informatique theorique et Applications,* **24** (1990), 531-544.
33. National Institute of Standards and Technology, *Digital Signature Standard,* FIPS Publication 186-2, February 2000.
34. National Institute of Standards and Technology, *Advanced Encryption Standard,* work in progress.
35. OpenSSL, <http://www.openssl.org>
36. R. Schroeppel, H. Orman, S. O'Malley and 0. Spatscheck, "Fast key exchange with elliptic curve systems", *Advances in Cryptology — Crypto '95,* LNCS 963, 1995, 43-56.
37. J. Solinas, "Efficient arithmetic on Koblitz curves", *Designs, Codes and Cryptog­raphy,* 19 (2000), 195-249.