Software Implementation of Elliptic Curve Cryptography Over Binary Fields

Darrel Hankerson[1], Julio López Hernandez[2], and Alfred Menezes[3]

[1] Dept. of Discrete and Statistical Sciences, Auburn University, USA
hankedr@mail.auburn.edu
[2] Dept. of Computer Science, University of Valle, Colombia
jlopez@borabora.univalle.edu.co
[3] Dept. of Combinatorics and Optimization, University of Waterloo, Canada
ajmeneze@uwaterloo.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 $\mathbb{F}_{2^{163}}$, $\mathbb{F}_{2^{233}}$, $\mathbb{F}_{2^{283}}$, $\mathbb{F}_{2^{409}}$, and $\mathbb{F}_{2^{571}}$. 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 in prime field $\mathbb{F}_p$	Dimension m of binary field $\mathbb{F}_{2^m}$
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

$\mathbb{F}_{2^m}$, m was chosen so that there exists a Koblitz curve of almost prime order over $\mathbb{F}_{2^m}$. Since the order $\#E(\mathbb{F}_{2^l})$ divides $\#E(\mathbb{F}_{2^m})$ whenever l divides m, this requirement imposes the condition that m be prime.

Since the NIST binary curves are all defined over fields $\mathbb{F}_{2^m}$ where m is prime, our paper excludes from consideration fields such as $\mathbb{F}_{2^{176}}$ 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 $\mathbb{F}_{2^m}$ 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 $\mathbb{F}_{2^{163}}$, $\mathbb{F}_{2^{233}}$, and $\mathbb{F}_{2^{283}}$. The results can be extrapolated to curves over $\mathbb{F}_{2^{409}}$ and $\mathbb{F}_{2^{571}}$.

Description of the NIST curves over binary fields. The NIST elliptic curves over $\mathbb{F}_{2^{163}}$, $\mathbb{F}_{2^{233}}$ and $\mathbb{F}_{2^{283}}$ are listed in Table 2. The following notation is used. The elements of $\mathbb{F}_{2^m}$ are represented using a polynomial basis representation with reduction polynomial $f(x)$ (see §3.1). The reduction polynomials for the fields $\mathbb{F}_{2^{163}}$, $\mathbb{F}_{2^{233}}$ and $\mathbb{F}_{2^{283}}$ are $f(x) = x^{163} + x^7 + x^6 + x^3 + 1$, $f(x) = x^{233} + x^{74} + 1$, and $f(x) = x^{283} + x^{12} + x^7 + x^5 + 1$, respectively. An elliptic curve E over $\mathbb{F}_{2^m}$ is specified by the coefficients $a, b \in \mathbb{F}_{2^m}$ of its defining equation $y^2 + xy = x^3 + ax^2 + b$. The number of points on E defined over $\mathbb{F}_{2^m}$ is nh, where n is prime, and h is called the co-factor. A random curve over $\mathbb{F}_{2^m}$ is denoted by B-m, while a Koblitz curve over $\mathbb{F}_{2^m}$ is denoted by K-m.

Table 2. NIST-recommended elliptic curves over $\mathbb{F}_{2^{163}}$, $\mathbb{F}_{2^{233}}$ and $\mathbb{F}_{2^{283}}$.

B-163: $a = 1$, $h = 2$,	
$b =$	0x 00000002 0A601907 B8C953CA 1481EB10 512F7874 4A3205FD
$n =$	0x 00000004 00000000 00000000 000292FE 77E70C12 A4234C33
B-233: $a = 1$, $h = 2$,	
$b =$	0x 00000066 647EDE6C 332C7F8C 0923BB58 213B333B 20E9CE42 81FE115F 7D8F90AD
$n =$	0x 00000100 00000000 00000000 00000000 0013E974 E72F8A69 22031D26 03CFE0D7
B-283: $a = 1$, $h = 2$,	
$b =$	0x 027B680A C8B8596D A5A4AF8A 19A0303F CA97FD76 45309FA2 A581485A F6263E31 3B79A2F5
$n =$	0x 03FFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFEF90 399660FC 938A9016 5B042A7C EFADB307
K-163: $a = 1$, $b = 1$, $h = 2$,	
$n =$	0x 00000004 00000000 00000000 00020108 A2E0CC0D 99F8A5EF
K-233: $a = 0$, $b = 1$, $h = 4$,	
$n =$	0x 00000080 00000000 00000000 00000000 00069D5B B915BCD4 6EFB1AD5 F173ABDF
K-283: $a = 0$, $b = 1$, $h = 4$,	
$n =$	0x 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 $\mathbb{F}_{2^m}$, 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 $f(x) = x^m + r(x)$ be an irreducible binary polynomial of degree m. The elements of $\mathbb{F}_{2^m}$ are the binary polynomials of degree at most $m-1$ with addition and multiplication performed modulo $f(x)$. A field element $a(x) = a_{m-1}x^{m-1} + \cdots + a_2x^2 + a_1x + a_0$ is associated with the binary vector $a = (a_{m-1}, \ldots, a_2, a_1, a_0)$ of length m. Let $t = \lceil m/32 \rceil$, and let $s = 32t - m$. In software, we store a in an array of t 32-bit words: $A = (A[t-1], \ldots, A[2], A[1], A[0])$, where the rightmost bit of $A[0]$ is a_0, 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 $a \cdot b = a_{m-1}x^{m-1}b + \cdots + a_2x^2b + a_1xb + a_0b$. Iteration i of the algorithm computes $x^ib \bmod f(x)$ and adds the result to the accumulator c if $a_i = 1$. Note that $b \cdot x \bmod f(x)$ can be easily computed by a left-shift of the vector representation of b, followed by the addition of $r(x)$ to b if $b_m = 1$.

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: $c(x) = a(x) \cdot b(x) \bmod f(x)$.

1. If $a_0 = 1$ then $c \leftarrow b$; else $c \leftarrow 0$.
2. For i from 1 to $m-1$ do
   2.1 $b \leftarrow b \cdot x \bmod f(x)$.
   2.2 If $a_i = 1$ then $c \leftarrow c + b$.
3. 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 $b(x) \cdot x^k$ has been computed for some $k \in [0, 31]$, then $b(x) \cdot x^{32j+k}$ can be easily obtained by appending j zero words to the right of the vector representation of $b(x) \cdot x^k$. 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 $C = (C[n], \ldots, C[2], C[1], C[0])$ is a vector, then $C\{j\}$ denotes the truncated vector $(C[n], \ldots, C[j+1], C[j])$.

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: $c(x) = a(x) \cdot b(x)$.

1. $C \leftarrow 0$.
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 $k \neq 31$ then $B \leftarrow B \cdot x$.
3. 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: $c(x) = a(x) \cdot b(x)$.

1. $C \leftarrow 0$.
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 $k \neq 0$ then $C \leftarrow C \cdot x$.
3. 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 $u(x) \cdot b(x)$ 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: $c(x) = a(x) \cdot b(x)$.

1. Compute $B_u = u(x) \cdot b(x)$ for all polynomials $u(x)$ of degree at most 3.
2. $C \leftarrow 0$.
3. For k from 7 downto 0 do
   3.1 For j from 0 to $t-1$ do
       Let $u = (u_3, u_2, u_1, u_0)$, where u_i is bit $(4k+i)$ of $A[j]$. Add B_u to $C\{j\}$.
   3.2 If $k \neq 0$ then $C \leftarrow C \cdot x^4$.
4. 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 $(m/2)-1$: $a(x) = A_1(x)X + A_0(x)$, $b(x) = B_1(x)X + B_0(x)$, where $X = x^{m/2}$. Then

$$a(x)b(x) = A_1B_1X^2 + [(A_1 + A_0)(B_1 + B_0) + A_1B_1 + A_0B_0]X + A_0B_0,$$

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 $x^i \equiv x^{i-m}r(x) \pmod{f(x)}$ for $i \geq m$. The polynomials $x^k r(x)$, $0 \leq k \leq 31$, 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 $x^k r(x)$ are faster.

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

INPUT: A binary polynomial $c(x)$ of degree at most $2m-2$.
OUTPUT: $c(x) \bmod f(x)$.

1. *Precomputation.* Compute $u_k(x) = x^k r(x)$, $0 \leq k \leq 31$.
2. For i from $2m-2$ downto m do
   2.1 If $c_i = 1$ then
   Let $j = \lfloor (i-m)/32 \rfloor$ and $k = (i-m) - 32j$.
   Add $u_k(x)$ to $C\{j\}$.
3. Return$((C[t-1], \ldots, C[1], C[0]))$.

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 $f(x) = x^{163} + x^7 + x^6 + x^3 + 1$. Here, $m = 163$ and $t = 6$. We have

$$\begin{aligned} x^{288} &\equiv x^{132} + x^{131} + x^{128} + x^{125} \pmod{f(x)} \\ x^{289} &\equiv x^{133} + x^{132} + x^{129} + x^{126} \pmod{f(x)} \\ &\vdots \\ x^{319} &\equiv x^{163} + x^{162} + x^{159} + x^{156} \pmod{f(x)}. \end{aligned}$$

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 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: $c(x) \bmod f(x)$, where $f(x) = x^{163} + x^7 + x^6 + x^3 + 1$.
1. For i from 10 downto 6 do $\{$Reduce $C[i]$ modulo $f(x)\}$
   1.1 $T \leftarrow C[i]$.
   1.2 $C[i-6] \leftarrow C[i-6] \oplus (T \ll 29)$.
   1.3 $C[i-5] \leftarrow C[i-5] \oplus (T \ll 4) \oplus (T \ll 3) \oplus T \oplus (T \gg 3)$.
   1.4 $C[i-4] \leftarrow C[i-4] \oplus (T \gg 28) \oplus (T \gg 29)$.
2. $T \leftarrow C[5]$ AND 0xFFFFFFF8. $\{$Clear bits 0, 1 and 2 of $C[5]\}$
3. $C[0] \leftarrow C[0] \oplus (T \ll 4) \oplus (T \ll 3) \oplus T \oplus (T \gg 3)$.
4. $C[1] \leftarrow C[1] \oplus (T \gg 28) \oplus (T \gg 29)$.
5. $C[5] \leftarrow C[5]$ AND 0x00000007. $\{$Clear the unused bits of $C[5]\}$
6. 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 $\mathbb{F}_{2^m}$; that is, if $a(x) = \sum_{i=0}^{m-1} a_i x^i$, then $a(x)^2 = \sum_{i=0}^{m-1} a_i x^{2i}$. 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: $a \in \mathbb{F}_{2^m}$.
OUTPUT: $a^2 \bmod f(x)$.
1. *Precomputation.* For each byte $v = (v_7, \ldots, v_1, v_0)$, compute the 16-bit quantity $T(v) = (0, v_7, \ldots, 0, v_1, 0, v_0)$.
2. For i from 0 to $t-1$ do
   2.1 Let $A[i] = (u_3, u_2, u_1, u_0)$ where each u_j is a byte.
   2.2 $C[2i] \leftarrow (T(u_1), T(u_0))$, $C[2i+1] \leftarrow (T(u_3), T(u_2))$.
3. Compute $b(x) = c(x) \bmod f(x)$.
4. Return(b).

3.4 Inversion

Algorithm 8 computes the inverse of a non-zero field element $a \in \mathbb{F}_{2^m}$ 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 $\deg(u) \geq \deg(v)$, then a partial division of u by v is performed by subtracting $x^j v$ 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 $x^j c$ from b preserves the invariants. The algorithm terminates when $\deg(u) = 0$, in which case $u = 1$ and $ba + df = 1$; hence $b = a^{-1} \bmod f(x)$.

Algorithm 8. Extended Euclidean Algorithm for inversion in $\mathbb{F}_{2^m}$

INPUT: $a \in \mathbb{F}_{2^m}$, $a \neq 0$.
OUTPUT: $a^{-1} \bmod f(x)$.

1. $b \leftarrow 1$, $c \leftarrow 0$, $u \leftarrow a$, $v \leftarrow f$.
2. While $\deg(u) \neq 0$ do
   2.1 $j \leftarrow \deg(u) - \deg(v)$.
   2.2 If $j < 0$ then: $u \leftrightarrow v$, $b \leftrightarrow c$, $j \leftarrow -j$.
   2.3 $u \leftarrow u + x^j v$, $b \leftarrow b + x^j c$.
3. Return(b).

The Almost Inverse Algorithm (AIA, Algorithm 9) is from [36]. For $a \in \mathbb{F}_{2^m}$, $a \neq 0$, a pair (b, k) is returned where $ba \equiv x^k \pmod{f(x)}$. A reduction is then applied to obtain $a^{-1} = bx^{-k} \bmod f(x)$. The invariants are $ba + df = ux^k$ and $ca + ef = vx^k$ 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 $ba + df = x^k$. 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 $s = \min\{i \geq 1 \mid f_i = 1\}$, where $f(x) = f_m x^m + \cdots + f_1 x + f_0$. Let b' be the polynomial formed by the s rightmost bits of b. Then $b'f + b$ is divisible by x^s and $b'' = (b'f + b)/x^s$ has degree less than m; thus $b'' = bx^{-s} \bmod f(x)$. This process can be repeated to finally obtain $bx^{-k} \bmod f(x)$. The reduction polynomial is said to be *suitable* if $s \geq 32$, since then fewer iterations are required in the reduction step.

Algorithm 9. Almost Inverse Algorithm for inversion in $\mathbb{F}_{2^m}$

INPUT: $a \in \mathbb{F}_{2^m}$, $a \neq 0$.
OUTPUT: $b \in \mathbb{F}_{2^m}$ and $k \in [0, 2m-1]$ such that $ba \equiv x^k \pmod{f(x)}$.

1. $b \leftarrow 1$, $c \leftarrow 0$, $u \leftarrow a$, $v \leftarrow f$, $k \leftarrow 0$.
2. While x divides u do:
   2.1 $u \leftarrow u/x$, $c \leftarrow cx$, $k \leftarrow k + 1$.
3. If $u = 1$ then return(b,k).
4. If $\deg(u) < \deg(v)$ then: $u \leftrightarrow v$, $b \leftrightarrow c$.
5. $u \leftarrow u + v$, $b \leftarrow b + c$.
6. 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 $b = a^{-1} \bmod f(x)$.

Algorithm 10. Modified Almost Inverse Algorithm for inversion in $\mathbb{F}_{2^m}$

INPUT: $a \in \mathbb{F}_{2^m}$, $a \neq 0$.
OUTPUT: $a^{-1} \bmod f(x)$.

1. $b \leftarrow 1$, $c \leftarrow 0$, $u \leftarrow a$, $v \leftarrow f$.
2. While x divides u do:
   2.1 $u \leftarrow u/x$.
   2.2 If x divides b then $b \leftarrow b/x$; else $b \leftarrow (b+f)/x$.
3. If $u = 1$ then return(b).
4. If $\deg(u) < \deg(v)$ then: $u \leftrightarrow v$, $b \leftrightarrow c$.
5. $u \leftarrow u + v$, $b \leftarrow b + c$.
6. 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 $\mathbb{F}_{2^{163}}$, $\mathbb{F}_{2^{233}}$ and $\mathbb{F}_{2^{283}}$. The field arithmetic was implemented in C and the timings obtained on a Pentium II 400 MHz workstation.

Table 3. Timings (in μs) for operations in $\mathbb{F}_{2^{163}}$, $\mathbb{F}_{2^{233}}$ and $\mathbb{F}_{2^{283}}$. The reduction polynomials are, respectively, $f(x) = x^{163} + x^7 + x^6 + x^3 + 1$, $f(x) = x^{233} + x^{74} + 1$, and $f(x) = x^{283} + x^{12} + x^7 + x^5 + 1$.

	$m = 163$	$m = 233$	$m = 283$
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.

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 $\mathbb{F}_{2^m}$ given by the (affine) equation $y^2 + xy = x^3 + ax^2 + b$, where $a \in \{0, 1\}$. Let $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ be two points on E with $P_1 \neq -P_2$. Then the coordinates of $P_3 = P_1 + P_2 = (x_3, y_3)$ can be computed as follows:

$$x_3 = \lambda^2 + \lambda + x_1 + x_2 + a, \quad y_3 = (x_1 + x_3)\lambda + x_3 + y_1, \text{ where}$$
$$\lambda = \frac{y_1 + y_2}{x_1 + x_2} \text{ if } P_1 \neq P_2, \quad \text{and } \lambda = \frac{y_1}{x_1} + x_1 \text{ if } P_1 = P_2. \tag{1}$$

In either case, when $P_1 \neq P_2$ (general addition) and $P_1 = P_2$ (doubling), the formulas for computing P_3 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 $\mathbb{F}_{2^m}$ 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 $(X : Y : Z)$, $Z \neq 0$, corresponds to the affine point $(X/Z, Y/Z)$. The projective equation of the elliptic curve is $Y^2Z + XYZ = X^3 + aX^2Z + bZ^3$. In *Jacobian* projective coordinates [5], the projective point $(X : Y : Z)$, $Z \neq 0$, corresponds to the affine point $(X/Z^2, Y/Z^3)$ and the projective equation of the curve is $Y^2 + XYZ = X^3 + aX^2Z^2 + bZ^6$. In [25], a new set of projective coordinates was introduced. Here, a projective point $(X : Y : Z)$, $Z \neq 0$, corresponds to the affine point $(X/Z, Y/Z^2)$, and the projective equation of the curve is

$$Y^2 + XYZ = X^3Z + aX^2Z^2 + bZ^4. \tag{2}$$

Formulas which do not require inversions for adding and doubling points in projective coordinates can be derived by first converting the points to affine 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: $2(X_1 : Y_1 : Z_1) = (X_3 : Y_3 : Z_3)$, where

$$Z_3 = X_1^2 \cdot Z_1^2, \quad X_3 = X_1^4 + b \cdot Z_1^4, \quad Y_3 = bZ_1^4 \cdot Z_3 + X_3 \cdot (aZ_3 + Y_1^2 + bZ_1^4). \tag{3}$$

Formulas for addition in mixed coordinates are: $(X_1 : Y_1 : Z_1) + (X_2 : Y_2 : 1) = (X_3 : Y_3 : Z_3)$, where

$$\begin{gathered} A = Y_2 \cdot Z_1^2 + Y_1, \quad B = X_2 \cdot Z_1 + X_1, \quad C = Z_1 \cdot B, \quad D = B^2 \cdot (C + aZ_1^2), \\ Z_3 = C^2, \quad E = A \cdot C, \quad X_3 = A^2 + D + E, \quad F = X_3 + X_2 \cdot Z_3, \\ G = X_3 + Y_2 \cdot Z_3, \quad Y_3 = E \cdot F + Z_3 \cdot G. \end{gathered} \tag{4}$$

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	$1I$, $2M$	—	$1I$, $2M$
Standard projective $(X/Z, Y/Z)$	$13M$	$12M$	$7M$
Jacobian projective $(X/Z^2, Y/Z^3)$	$14M$	$10M$	$5M$
Projective $(X/Z, Y/Z^2)$	$14M$	$9M$	$4M$

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 $\#E(\mathbb{F}_{2^m}) = nh$ where n is prime and h is small (so $n \approx 2^m$), P has order n, and $k \in_R [1, n-1]$. 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 is the additive version of the basic repeated-square-and-multiply method for exponentiation.

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

INPUT: $k = (k_{t-1}, \ldots, k_1, k_0)_2$, $P \in E(\mathbb{F}_{2^m})$.
OUTPUT: kP.

1. $Q \leftarrow \mathcal{O}$.
2. For i from $t-1$ downto 0 do
   2.1 $Q \leftarrow 2Q$.
   2.2 If $k_i = 1$ then $Q \leftarrow Q + P$.
3. Return(Q).

The expected number of ones in the binary representation of k is $t/2 \approx m/2$, 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 $3mM + 1.5mI$, 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 $8.5mM + (2M + 1I)$ (1 inversion and 2 multiplications are required to convert back to affine coordinates).

If $P = (x, y) \in E(\mathbb{F}_{2^m})$ 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* $k = \sum_{i=0}^{l-1} k_i 2^i$, where $k_i \in \{0, \pm 1\}$. A particularly useful signed digit representation is the *non-adjacent form* (NAF) which has the property that no two consecutive coefficients k_i 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. $i \leftarrow 0$.
2. While $k \geq 1$ do
   2.1 If k is odd then: $k_i \leftarrow 2 - (k \bmod 4)$, $k \leftarrow k - k_i$;
   2.2 Else: $k_i \leftarrow 0$.
   2.3 $k \leftarrow k/2$, $i \leftarrow i + 1$.
3. Return($(k_{i-1}, k_{i-2}, \ldots, k_1, k_0)$).

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 l 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: $\mathrm{NAF}(k) = \sum_{i=0}^{l-1} k_i 2^i$, $P \in E(\mathbb{F}_{2^m})$.
OUTPUT: kP.

1. $Q \leftarrow \mathcal{O}$.
2. For i from $l-1$ downto 0 do
   2.1 $Q \leftarrow 2Q$.
   2.2 If $k_i = 1$ then $Q \leftarrow Q + P$.
   2.3 If $k_i = -1$ then $Q \leftarrow Q - P$.
3. 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 $\mathrm{NAF}(k)$ 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 $k = \sum_{i=0}^{l-1} k_i 2^i$, where each non-zero coefficient k_i is odd, $|k_i| < 2^{w-1}$, and at most one of any w consecutive coefficients is nonzero. Every positive integer has a unique width-w NAF, denoted $\mathrm{NAF}_w(k)$. Note that $\mathrm{NAF}_2(k) = \mathrm{NAF}(k)$. $\mathrm{NAF}_w(k)$ can be efficiently computed using Algorithm 12 modified as follows: in step 2.1 replace "$k_i \leftarrow 2-(k \bmod 4)$" by "$k_i \leftarrow k \text{ mods } 2^w$", where $k \text{ mods } 2^w$ denotes the integer u satisfying $u \equiv k \pmod{2^w}$ and $-2^{w-1} \le u < 2^{w-1}$. It is known that the length of $\mathrm{NAF}_w(k)$ 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 l is approximately $1/(w+1)$ [37]. It follows that the expected running time of Algorithm 14 is approximately $(1D + (2^{w-2}-1)A) + (m/(w+1)A + mD)$. 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, $\mathrm{NAF}_w(k) = \sum_{i=0}^{l-1} k_i 2^i$, $P \in E(\mathbb{F}_{2^m})$.
OUTPUT: kP.

1. Compute $P_i = iP$, for $i \in \{1, 3, 5, \ldots, 2^{w-1}-1\}$.
2. $Q \leftarrow \mathcal{O}$.
3. For i from $l-1$ downto 0 do
   3.1 $Q \leftarrow 2Q$.
   3.2 If $k_i \neq 0$ then:
   If $k_i > 0$ then $Q \leftarrow Q + P_{k_i}$;
   Else $Q \leftarrow Q - P_{k_i}$.
4. Return(Q).

Algorithm 15 is from [26] and is based on an idea of Montgomery [31]. Let $Q_1 = (x_1, y_1)$, $Q_2 = (x_2, y_2)$ with $Q_1 \neq \pm Q_2$. Let $Q_1 + Q_2 = (x_3, y_3)$ and $Q_1 - Q_2 = (x_4, y_4)$. Then using the addition formulas (1), it can be verified that

$$x_3 = x_4 + \frac{x_1}{x_1 + x_2} + \left(\frac{x_1}{x_1 + x_2}\right)^2. \tag{5}$$

Thus, the x-coordinate of $Q_1 + Q_2$ can be computed from the x-coordinates of Q_1, Q_2 and $Q_1 - Q_2$. Iteration j of Algorithm 15 for determining kP computes $T_j = (lP, (l+1)P)$, where l is the integer given by the j leftmost bits of k. Then $T_{j+1} = (2lP, (2l+1)P)$ or $((2l+1)P, (2l+2)P)$ 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 $kP = (x_1, y_1)$ and $(k+1)P = (x_2, y_2)$, the y-coordinate of kP can be recovered as:

$$y_1 = x^{-1}(x_1 + x)[(x_1 + x)(x_2 + x) + x^2 + y] + y. \tag{6}$$

Equation (6) is derived using the addition formula (1) for computing the x-coordinate x_2 of $(k+1)P$ from $kP = (x_1, y_1)$ and $P = (x, y)$. Algorithm 15 is presented using standard projective coordinates (see §4). The approximate running time is $6mM + (1I + 10M)$. One advantage of Algorithm 15 is that it does not have any extra storage requirements.

Algorithm 15. Montgomery point multiplication

INPUT: $k = (k_{t-1}, \ldots, k_1, k_0)_2$ with $k_{t-1} = 1$, $P = (x, y) \in E(\mathbb{F}_{2^m})$.
OUTPUT: kP.

1. $X_1 \leftarrow x$, $Z_1 \leftarrow 1$, $X_2 \leftarrow x^4 + b$, $Z_2 \leftarrow x^2$. {Compute $(P, 2P)$}
2. For i from $t-2$ downto 0 do
   - 2.1 If $k_i = 1$ then
     - $T \leftarrow Z_1$, $Z_1 \leftarrow (X_1 Z_2 + X_2 Z_1)^2$, $X_1 \leftarrow x Z_1 + X_1 X_2 T Z_2$.
     - $T \leftarrow X_2$, $X_2 \leftarrow X_2^4 + b Z_2^4$, $Z_2 \leftarrow T^2 Z_2^2$.
   - 2.2 Else
     - $T \leftarrow Z_2$, $Z_2 \leftarrow (X_1 Z_2 + X_2 Z_1)^2$, $X_2 \leftarrow x Z_2 + X_1 X_2 Z_1 T$.
     - $T \leftarrow X_1$, $X_1 \leftarrow X_1^4 + b Z_1^4$, $Z_1 \leftarrow T^2 Z_1^2$.
3. $x_3 \leftarrow X_1 / Z_1$.
4. $y_3 \leftarrow (x + X_1/Z_1)[(X_1 + xZ_1)(X_2 + xZ_2) + (x^2 + y)(Z_1 Z_2)](x Z_1 Z_2)^{-1} + y$.
5. Return((x_3, y_3)).

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 $2P, 2^2P, \ldots, 2^{t-1}P$ 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 $(k_{d-1}, \ldots, k_1, k_0)_{2^w}$ be the 2^w-ary representation of k, where $d = \lceil t/w \rceil$, and let $Q_j = \sum_{i:k_i=j} 2^{wi}P$. Then

$$\begin{aligned} kP &= \sum_{i=0}^{d-1} k_i(2^{wi}P) = \sum_{j=1}^{2^w-1}\left(j \sum_{i:k_i=j} 2^{wi}P\right) = \sum_{j=1}^{2^w-1} jQ_j \\ &= Q_{2^w-1} + (Q_{2^w-1} + Q_{2^w-2}) + \cdots + (Q_{2^w-1} + Q_{2^w-2} + \cdots + Q_1). \end{aligned} \tag{7}$$

Algorithm 16 is based on this observation. Its expected running time is approximately $((d(2^w-1)/2^w-1)+(2^w-2))A$. 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, $d=\lceil t/w \rceil$, $k=(k_{d-1},\dots,k_1,k_0)_{2^w}$, $P\in E(\mathbb{F}_{2^m})$.
OUTPUT: kP.

1. *Precomputation.* Compute $P_i=2^{wi}P$, $0\le i\le d-1$.
2. $A\leftarrow\mathcal{O}$, $B\leftarrow\mathcal{O}$.
3. For j from 2^w-1 downto 1 do
   3.1 For each i for which $k_i=j$ do: $B\leftarrow B+P_i$. {Add Q_j to B}
   3.2 $A\leftarrow A+B$.
4. 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 $[a_{w-1},\dots,a_2,a_1,a_0]P = a_{w-1}2^{(w-1)d}P+\cdots+ +a_2 2^{2d}P+a_1 2^d P+a_0P$, where $d=\lceil t/w\rceil$ and $a_i\in\mathbb{Z}_2$. The expected running time of Algorithm 17 is $((d-1)(2^w-1)/2^w)A+(d-1)D$.

Algorithm 17. Fixed-base comb method

INPUT: Window width w, $d=\lceil t/w \rceil$, $k=(k_{t-1},\dots,k_1,k_0)_2$, $P\in E(\mathbb{F}_{2^m})$.
OUTPUT: kP.

1. *Precomputation.* Compute $[a_{w-1},\dots,a_1,a_0]P\ \forall(a_{w-1},\dots,a_1,a_0)\in\mathbb{Z}_2^w$.
2. By padding k on the left with 0's if necessary, write $k=K^{w-1}\|\cdots\|K^1\|K^0$, where each K^j is a bit string of length d. Let K_i^j denote the ith bit of K^j.
3. $Q\leftarrow\mathcal{O}$.
4. For i from $d-1$ downto 0 do
   4.1 $Q\leftarrow 2Q$.
   4.2 $Q\leftarrow Q+[K_i^{w-1},\dots,K_i^1,K_i^0]P$.
5. 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

Koblitz curves are elliptic curves defined over $\mathbb{F}_2$, 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: $E_0 : y^2 + xy = x^3 + 1$ and $E_1 : y^2 + xy = x^3 + x^2 + 1$. Let $\mu = (-1)^{1-a}$. We have $\#E_a(\mathbb{F}_2) = 3 - \mu$. We assume that $\#E_a(\mathbb{F}_{2^m})$ is almost prime, i.e., $\#E_a(\mathbb{F}_{2^m}) = hn$, where n is prime and $h = 3-\mu$. The number of points is given by $\#E_a(\mathbb{F}_{2^m}) = 2^m + 1 - V_m$, where $\{V_k\}$ is the Lucas sequence defined by $V_0 = 2$, $V_1 = \mu$, $V_{k+1} = \mu V_k - 2V_{k-1}$ for $k \geq 1$.

Since E_a is defined over $\mathbb{F}_{2^m}$, the *Frobenius map* $\tau : E_a(\mathbb{F}_{2^m}) \rightarrow E_a(\mathbb{F}_{2^m})$ defined by $\tau(\mathcal{O}) = \mathcal{O}$, $\tau((x, y)) = (x^2, y^2)$ is well-defined. Moreover, it can be efficiently computed since squaring in $\mathbb{F}_{2^m}$ is relatively inexpensive (see §3.5). It is known that $(\tau^2 + 2)P = \mu\tau P$ for all $P \in E_a(\mathbb{F}_{2^m})$. Hence the Frobenius map can be regarded as the complex number τ satisfying $\tau^2 + 2 = \mu\tau$, i.e., $\tau = (\mu + \sqrt{-7})/2$. It now makes sense to multiply points in $E_a(\mathbb{F}_{2^m})$ by elements of the ring $\mathbb{Z}[\tau]$: if $u_{l-1}\tau^{l-1} + \cdots + u_1\tau + u_0 \in \mathbb{Z}[\tau]$ and $P \in E_a(\mathbb{F}_{2^m})$, then

$$(u_{l-1}\tau^{l-1} + \cdots + u_1\tau + u_0)P = u_{l-1}\tau^{l-1}(P) + \cdots + u_1\tau(P) + u_0P. \qquad (8)$$

The strategy for developing an efficient point multiplication algorithm is find a "nice" expression for k of the form $k = \sum_{i=0}^{l-1} u_i\tau^i$, and then use (8) to compute kP. Here, "nice" means that l is relatively small and the non-zero coefficients u_i are small (e.g., ± 1) and sparse.

Since $\tau^2 + 2 = \mu\tau$, every element in $\mathbb{Z}[\tau]$ can be expressed in canonical form $r_0 + r_1\tau$, where $r_0, r_1 \in \mathbb{Z}$. $\mathbb{Z}[\tau]$ is a Euclidean domain, and hence also a unique factorization domain, with respect to the norm function $N(r_0 + r_1\tau) = r_0^2 + \mu r_0 r_1 + 2r_1^2$. The norm function is multiplicative. We have $N(\tau) = 2$, $N(\tau - 1) = h$, $N(\tau^m - 1) = \#E_a(\mathbb{F}_{2^m})$, and $N(\delta) = n$ where $\delta = (\tau^m - 1)/(\tau - 1)$.

A *τ-adic NAF* or *TNAF* of an element $\kappa \in \mathbb{Z}[\tau]$ is an expression $\kappa = \sum_{i=0}^{l-1} u_i\tau^i$ where $u_i \in \{0, \pm 1\}$, and no two consecutive coefficients u_i are nonzero. Every $\kappa \in \mathbb{Z}[\tau]$ has a unique TNAF, denoted TNAF(κ), which can be efficiently computed using Algorithm 18.

Algorithm 18. Computing the TNAF of an element in $\mathbb{Z}[\tau]$

INPUT: $\kappa = r_0 + r_1\tau \in \mathbb{Z}[\tau]$.
OUTPUT: TNAF(κ).

1. $i \leftarrow 0$.
2. While $r_0 \neq 0$ or $r_1 \neq 0$ do
   2.1 If r_0 is odd then: $u_i \leftarrow 2 - (r_0 - 2r_1 \bmod 4)$, $r_0 \leftarrow r_0 - u_i$;
   2.2 Else: $u_i \leftarrow 0$.
   2.3 $t \leftarrow r_0$, $r_0 \leftarrow r_1 + \mu r_0/2$, $r_1 \leftarrow -t/2$, $i \leftarrow i + 1$.
3. Return($(u_{i-1}, u_{i-2}, \ldots, u_1, u_0)$).

To compute kP, one can find TNAF(k) using Algorithm 18, and then use (8). Now, the length $l(\alpha)$ of TNAF(α) satisfies $\log_2(N(\alpha)) - 0.55 < l(\alpha) <$

$\log_2(N(\alpha)) + 3.52$ when $l \ge 30$. It follows that $l(k) \approx 2\log_2 k$, which is twice as long as the length of NAF(k). To circumvent the problem of a long TNAF, notice that if $\rho = k \bmod \delta$ then $kP = \rho P$ for all points P of order n (because $\delta P = \mathcal{O}$). Since $N(\rho) < N(\delta) = n$, it follows that $l(\rho) \approx m$, which suggests that TNAF(ρ) should be used instead of TNAF(k) for computing kP. Algorithm 19 is an efficient method for computing an element $\rho' \in \mathbb{Z}[\tau]$ such that $\rho' \equiv k \pmod{\delta}$; we write $\rho' = k$ partmod δ. The parameter C ensures that TNAF(ρ') is not much longer than TNAF(ρ). In fact, $l(\rho) \le m + a$, and if $C \ge 2$ then $l(\rho') \le m + a + 3$. Also, the probability that $\rho' \ne \rho$ is less than $2^{-(C-5)}$.

Algorithm 19. Partial reduction modulo δ

INPUT: $k \in [1, n-1]$, $C \ge 2$, $s_0 = d_0 + \mu d_1$, $s_1 = -d_1$, where $\delta = d_0 + d_1\tau$.
OUTPUT: $\rho' = k$ partmod δ.

1. $k' \leftarrow \lfloor k/2^{a-C+(m-9)/2} \rfloor$.
2. For i from 0 to 1 do
   - 2.1 $g' \leftarrow s_i \cdot k'$, $j' \leftarrow V_m \cdot \lfloor g'/2^m \rfloor$, $\lambda_i \leftarrow \lfloor (g' + j')/2^{(m+5)/2} + \frac{1}{2} \rfloor / 2^C$.
   - 2.2 $f_i \leftarrow \lfloor \lambda_i + \frac{1}{2} \rfloor$, $\eta_i \leftarrow \lambda_i - f_i$, $h_i \leftarrow 0$.
3. $\eta \leftarrow 2\eta_0 + \mu\eta_1$.
4. If $\eta \ge 1$ then
   - 4.1 If $\eta_0 - 3\mu\eta_1 < -1$ then $h_1 \leftarrow \mu$; else $h_0 \leftarrow 1$.

   Else
   - 4.2 If $\eta_0 + 4\mu\eta_1 \ge 2$ then $h_1 \leftarrow \mu$.
5. If $\eta < -1$ then
   - 5.1 If $\eta_0 - 3\mu\eta_1 \ge 1$ then $h_1 \leftarrow -\mu$; else $h_0 \leftarrow -1$.

   Else
   - 5.2 If $\eta_0 + 4\mu\eta_1 < -2$ then $h_1 \leftarrow -\mu$.
6. $q_0 \leftarrow f_0 + h_0$, $q_1 \leftarrow f_1 + h_1$, $r_0 \leftarrow k - (s_0 + \mu s_1)q_0 - 2s_1q_1$, $r_1 \leftarrow s_1q_0 - s_0q_1$.
7. Return($r_0 + r_1\tau$).

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

Algorithm 20. TNAF method for point multiplication

INPUT: TNAF(ρ') $= \sum_{i=0}^{l-1} u_i\tau^i$ where $\rho' = k$ partmod δ, $P \in E_a(\mathbb{F}_{2^m})$.
OUTPUT: kP.

1. $Q \leftarrow \mathcal{O}$.
2. For i from $l-1$ downto 0 do
   - 2.1 $Q \leftarrow \tau Q$.
   - 2.2 If $u_i = 1$ then $Q \leftarrow Q + P$.
   - 2.3 If $u_i = -1$ then $Q \leftarrow Q - P$.
3. Return(Q).

We now extend Algorithm 20 to a window method analogous to Algorithm 14. Let $t_w = 2U_{w-1}U_w^{-1} \bmod 2^w$, where $\{U_k\}$ is the Lucas sequence defined by $U_0 = 0$, $U_1 = 1$, $U_{k+1} = \mu U_k - 2U_{k-1}$ for $k \ge 1$. Then the map $\phi_w : \mathbb{Z}[\tau] \to \mathbb{Z}_{2^w}$ induced by $\tau \mapsto t_w$ is a surjective ring homomorphism with kernel $\{\alpha \in \mathbb{Z}[\tau] : \tau^w | \alpha\}$. It follows that a set of distinct representatives of the congruence classes modulo τ^w whose elements are not divisible by τ is $\{\pm 1, \pm 3, \ldots, \pm(2^{w-1} - 1)\}$. Define $\alpha_i = i \bmod \tau^w$ for $i \in \{1, 3, \ldots, 2^{w-1} - 1\}$. A *width-w TNAF* of $\kappa \in \mathbb{Z}[\tau]$, denoted $\mathrm{TNAF}_w(\kappa)$, is an expression $\kappa = \sum_{i=0}^{l-1} u_i \tau^i$, where $u_i \in \{0, \pm\alpha_1, \pm\alpha_3, \ldots, \pm\alpha_{2^{w-1}-1}\}$, and at most one of any w consecutive coefficients is nonzero. Algorithm 21 is an efficient method for computing $\mathrm{TNAF}_w(\kappa)$.

Algorithm 21. Computing a width-w TNAF of an element in $\mathbb{Z}[\tau]$

INPUT: w, t_w, $\alpha_u = \beta_u + \gamma_u \tau$ for $u \in \{1, 3, \ldots, 2^{w-1} - 1\}$, $\rho = r_0 + r_1\tau \in \mathbb{Z}[\tau]$.
OUTPUT: $\mathrm{TNAF}_w(\rho)$.

1. $i \leftarrow 0$.
2. While $r_0 \neq 0$ or $r_1 \neq 0$ do
   2.1 If r_0 is odd then
   $u \leftarrow r_0 + r_1 t_w \text{ mods } 2^w$.
   If $u > 0$ then $s \leftarrow 1$; else $s \leftarrow -1$, $u \leftarrow -u$.
   $r_0 \leftarrow r_0 - s\beta_u$, $r_1 \leftarrow r_1 - s\gamma_u$, $u_i \leftarrow s\alpha_u$.
   2.2 Else: $u_i \leftarrow 0$.
   2.3 $t \leftarrow r_0$, $r_0 \leftarrow r_1 + \mu r_0/2$, $r_1 \leftarrow -t/2$, $i \leftarrow i + 1$.
3. Return$((u_{i-1}, u_{i-2}, \ldots, u_1, u_0))$.

The average density of non-zero coefficients among all TNAF_ws of length l is approximately $1/(w+1)$. Since the length of $\mathrm{TNAF}_w(\rho')$ is approximately $l(\rho')$, it follows that Algorithm 22 which uses $\mathrm{TNAF}(\rho')$ for computing kP has an expected running time of approximately $(2^{w-2} - 1 + m/(w+1))A$.

Algorithm 22. Window TNAF method for point multiplication

INPUT: $\mathrm{TNAF}_w(\rho') = \sum_{i=0}^{l-1} u_i \tau^i$, where $\rho' = k \text{ partmod } \delta$, $P \in E_a(\mathbb{F}_{2^m})$.
OUTPUT: kP.

1. Compute $P_u = \alpha_u P$, for $u \in \{1, 3, 5, \ldots, 2^{w-1} - 1\}$.
2. $Q \leftarrow \mathcal{O}$.
3. For i from $l-1$ downto 0 do
   3.1 $Q \leftarrow \tau Q$.
   3.2 If $u_i \neq 0$ then:
   Let u be such that $\alpha_u = u_i$ or $\alpha_{-u} = -u_i$.
   If $u > 0$ then $Q \leftarrow Q + P_u$;
   Else $Q \leftarrow Q - P_{-u}$.
4. Return(Q).

If the point P is fixed, then the points P_u 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[a]
Binary (Algorithm 11)	affine	—	0	82	163	490	245	2940
	projective	—	0	82	163	1390	1	1400
Binary NAF (Algorithm 13)	affine	—	0	54	163	434	217	2604
	projective	—	0	54	163	1140	1	1150
Window NAF (Algorithm 14)	affine	4	3	36	164	400	200	2400
	projective	4	3	3^b+33	164	955	5	1005
Montgomery (Algorithm 15)	affine	—	0	163^c	163	329	327	3600
	projective	—	0	163^c	163	988	1	998
Fixed-base window (Algorithm 16)	affine	6	27	89	0	178	89	1068
	projective	6	27	27+62^d	0	1113	1	1123
Fixed-base comb (Algorithm 17)	affine	4	14	38	40	156	78	936
	projective	4	14	38	40	504	1	514
TNAF (Algorithm 20)	affine	—	0	54	0	108	54	648
	projective	—	0	54	0	488	1	498
Window TNAF (Algorithm 22)	affine	5	7	34	0	68	34	408
	projective	5	7	7^b+27	0	261	8	341
Fixed-base window TNAF (Algorithm 23)	affine	6	15	23	0	46	23	276
	projective	6	15	23	0	209	1	219

[a] Total cost in field multiplications assuming $1I = 10M$.
[b] Additions are in affine coordinates
[c] Additions using formula (5).
[d] Additions are not in mixed coordinates.

Table 7. Timings (in μs) for point multiplication on random and Koblitz curves over $\mathbb{F}_{2^{163}}$, $\mathbb{F}_{2^{233}}$ and $\mathbb{F}_{2^{283}}$. 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 μs for partial reduction (Algorithm 19), 126μs for width-w TNAF computation (Algorithm 21), and 1266μs 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 $kP+lQ$ where P is fixed and Q is not known a priori (signature verification). One method to speed the computation of $kP+lQ$ is simultaneous multiple point multiplication (Algorithm 24), also known as Shamir's trick [8]. Algorithm 24 has an expected running time of $(2^{2w}-3)A+((d-1)(2^{2w}-1)/2^{2w}A+(d-1)wD)$, and requires storage for 2^{2w} points.

Algorithm 24. Simultaneous multiple point multiplication

INPUT: Window width w, $k = (k_{t-1}, \ldots, k_1, k_0)_2$, $l = (l_{t-1}, \ldots, l_1, l_0)_2$, P, Q.
OUTPUT: $kP + lQ$.

1. Compute $iP + jQ$ for all $i, j \in [0, 2^w - 1]$.
2. Write $k = (k^{d-1}, \ldots, k^1, k^0)$ and $l = (l^{d-1}, \ldots, l^1, l^0)$ where each k^i and l^i is a bitstring of length w, and $d = \lceil t/w \rceil$.
3. $R \leftarrow \mathcal{O}$.
4. For i from $d-1$ downto 0 do
   4.1 $R \leftarrow 2^w R$.
   4.2 $R \leftarrow R + (k^i P + l^i Q)$.
5. 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 $kP + lQ$ 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 μs) of the fastest methods for point multiplication kP, P fixed, in ECDSA signature generation.

Curve type	Memory 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 μs) of the fastest methods for point multiplications $kP + lQ$, P fixed and Q not known a priori, in ECDSA signature verification.

Curve type	Memory constrained?	Fastest method	m=163	m=233	m=283
Random	No	Montgomery + Fixed-base comb ($w = 4$)	5005	11798	17659
	No	Simultaneous ($w = 2$)	4969	11332	16868
	Yes	Montgomery	6564	15531	23346
Koblitz	No	Window TNAF ($w = 5$) + Fixed-base window TNAF (w=6)	2702	5348	7826
	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 $\mathbb{F}_{2^m}$ 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 NIST-recommended prime fields, and a comparison with the NIST-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, 200-207.
4. M. Brown, D. Cheung, D. Hankerson, J. Hernandez, M. Kirkup and A. Menezes, "PGP 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 $GF(2^n)$", *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. Koç and T. Acar, "Montgomery multiplication in $GF(2^k)$", *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. López and R. Dahab, "Improved algorithms for elliptic curve arithmetic in $GF(2^n)$", *Selected Areas in Cryptography – SAC '98*, LNCS **1556**, 1999, 201-212.
26. J. López and R. Dahab, "Fast multiplication on elliptic curves over $GF(2^n)$ without precomputation", *Cryptographic Hardware and Embedded Systems – CHES '99*, LNCS **1717**, 1999, 316-327.
27. J. López and R. Dahab, "High-speed software multiplication in $\mathbb{F}_{2^m}$", 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 théorique 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 O. 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.