Software Implementation of the NIST Elliptic Curves Over Prime Fields

M. Brown[1], D. Hankerson[2,4], J. López[3], and A. Menezes[1,4]

[1] Dept. of C&O, University of Waterloo, Canada
mk3brown@uwaterloo.ca
[2] Dept. of Discrete and Statistical Sciences, Auburn University, USA
hankedr@mail.auburn.edu
[3] Dept. of Computer Science, University of Valle, Colombia
jlopez@borabora.univalle.edu.co
[4] Certicom Research, Canada
amenezes@certicom.com

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

1 Introduction

Elliptic curve cryptography (ECC) was proposed independently in 1985 by Neal Koblitz [17] and Victor Miller [21]. 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) [12], ISO (International Standards Organization) [13, 14], and NIST (National Institute of Standards and Technology) [24].

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 prime 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. We present the results of our implementation on a Pentium II 400 MHz workstation. These results serve to validate our conclusions based primarily on theoretical considerations. Although we make no claims that our implementations are the best possible (they certainly are not), and the optimization techniques used for the two larger fields were restricted to those employed for the smaller fields, we nonetheless 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 elliptic curves and presents some rationale for their selection. In §3, we describe methods for arithmetic in prime 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, and compare the performance of the NIST curves over binary and prime fields. Finally, we draw our conclusions in §7 and discuss avenues for future work in §8.

2 NIST-Recommended Elliptic Curves

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 [24].

FIPS 186-2 has 10 recommended finite fields: 5 prime fields $\mathbb{F}_p$ for $p_{192} = 2^{192} - 2^{64} - 1$, $p_{224} = 2^{224} - 2^{96} + 1$, $p_{256} = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$, $p_{384} = 2^{384} - 2^{128} - 2^{96} + 2^{32} - 1$, and $p_{521} = 2^{521} - 1$, 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 $\mathbb{F}_{2^m}$, m was chosen so that there exists a Koblitz curve of almost prime order over $\mathbb{F}_{2^m}$. In order to allow for efficient modular reduction (see Algorithms 7–11), the primes p for the prime fields $\mathbb{F}_p$ were chosen to either be a Mersenne prime, or a Mersenne-like prime with bitsize a multiple of 32.

The remainder of this paper considers the implementation of the NIST-recommended curves over the prime fields $\mathbb{F}_{p_{192}}$, $\mathbb{F}_{p_{224}}$, $\mathbb{F}_{p_{256}}$, $\mathbb{F}_{p_{384}}$ and $\mathbb{F}_{p_{521}}$.

Description of the NIST curves over prime fields. The NIST elliptic curves over prime fields are listed in Table 2. An elliptic curve E over $\mathbb{F}_p$ is

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 [25]	256	283
192	AES Medium [25]	384	409
256	AES Large [25]	521	571

specified by the coefficients $a, b \in \mathbb{F}_p$ of its defining equation $y^2 = x^3 + ax + b$. The NIST curves all have $a = -3$ because this yields a faster algorithm for point doubling when using Jacobian coordinates (see §4). This choice is without much loss of generality since about half of all isomorphism classes of elliptic curves over $\mathbb{F}_p$ have a representative with $a = -3$. The number of points on E defined over $\mathbb{F}_p$ is nh, where n is prime, and h is called the co-factor. A random curve over $\mathbb{F}_p$, where p is an m-bit prime, is denoted by P-m.

Description of the NIST curves over binary fields. The NIST elliptic curves over $\mathbb{F}_{2^{163}}$, $\mathbb{F}_{2^{233}}$, $\mathbb{F}_{2^{283}}$, $\mathbb{F}_{2^{409}}$ and $\mathbb{F}_{2^{571}}$ are listed in Table 3. The following notation is used. The elements of $\mathbb{F}_{2^m}$ are represented using a polynomial basis representation with reduction polynomial $f(x)$. 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.

3 Prime Field Arithmetic

This section presents algorithms for performing arithmetic in $\mathbb{F}_p$ 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

The elements of $\mathbb{F}_p$ are the integers between 0 and $p-1$, written in binary. Let $m = \lceil \log_2 p \rceil$ and $t = \lceil m/32 \rceil$. In software, we store a field element a in an array of t 32-bit words: $a = (a_{t-1}, \ldots, a_2, a_1, a_0)$. For the NIST primes p_{192}, p_{224}, p_{256}, p_{384} and p_{521}, we have $t = 6$, 7, 8, 12, and 17, respectively.

3.2 Addition and subtraction

Algorithm 1 calculates $a + b \bmod p$ by first finding the sum word-by-word and then subtracting p if the result exceeds $p - 1$. Each word addition produces a 32-bit sum and a 1-bit carry digit which is added to the next higher-order sum. It is assumed that "Add" in step 1 and "Add_with_carry" in step 2 of the algorithm manage the carry digit. On processors such as the Intel Pentium

Table 2. NIST-recommended elliptic curves over prime fields.

P-192: $p = 2^{192} - 2^{64} - 1$, $a = -3$, $h = 1$,	
$b =$ 0x	64210519 E59C80E7 0FA7E9AB 72243049 FEB8DEEC C146B9B1
$n =$ 0x	FFFFFFFF FFFFFFFF FFFFFFFF 99DEF836 146BC9B1 B4D22831
P-224: $p = 2^{224} - 2^{96} + 1$, $a = -3$, $h = 1$,	
$b =$ 0x	B4050A85 0C04B3AB F5413256 5044B0B7 D7BFD8BA 270B3943 2355FFB4
$n =$ 0x	FFFFFFFF FFFFFFFF FFFFFFFF FFFF16A2 E0B8F03E 13DD2945 5C5C2A3D
P-256: $p = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$, $a = -3$, $h = 1$,	
$b =$ 0x	5AC635D8 AA3A93E7 B3EBBD55 769886BC 651D06B0 CC53B0F6 3BCE3C3E
	27D2604B
$n =$ 0x	FFFFFFFF 00000000 FFFFFFFF FFFFFFFF BCE6FAAD A7179E84 F3B9CAC2
	FC632551
P-384: $p = 2^{384} - 2^{128} - 2^{96} + 2^{32} - 1$, $a = -3$, $h = 1$,	
$b =$ 0x	B3312FA7 E23EE7E4 988E056B E3F82D19 181D9C6E FE814112 0314088F
	5013875A C656398D 8A2ED19D 2A85C8ED D3EC2AEF
$n =$ 0x	FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF C7634D81
	F4372DDF 581A0DB2 48B0A77A ECEC196A CCC52973
P-521: $p = 2^{521} - 1$, $a = -3$, $h = 1$,	
$b =$ 0x	00000051 953EB961 8E1C9A1F 929A21A0 B68540EE A2DA725B 99B315F3
	B8B48991 8EF109E1 56193951 EC7E937B 1652C0BD 3BB1BF07 3573DF88
	3D2C34F1 EF451FD4 6B503F00
$n =$ 0x	000001FF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
	FFFFFFFF FFFFFFFA 51868783 BF2F966B 7FCC0148 F709A5D0 3BB5C9B8
	899C47AE BB6FB71E 91386409

Table 3. NIST-recommended elliptic curves over binary fields.

B-163: $a = 1,\ h = 2,\ f(x) = x^{163} + x^7 + x^6 + x^3 + 1$	
$b =$	`0x 00000002 0A601907 B8C953CA 1481EB10 512F7874 4A3205FD`
$n =$	`0x 00000004 00000000 00000000 000292FE 77E70C12 A4234C33`
B-233: $a = 1,\ h = 2,\ f(x) = x^{233} + x^{74} + 1$	
$b =$	`0x 00000066 647EDE6C 332C7F8C 0923BB58 213B333B 20E9CE42 81FE115F`
	`7D8F90AD`
$n =$	`0x 00000100 00000000 00000000 00000000 0013E974 E72F8A69 22031D26`
	`03CFE0D7`
B-283: $a = 1,\ h = 2,\ f(x) = x^{283} + x^{12} + x^7 + x^5 + 1$	
$b =$	`0x 027B680A C8B8596D A5A4AF8A 19A0303F CA97FD76 45309FA2 A581485A`
	`F6263E31 3B79A2F5`
$n =$	`0x 03FFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFEF90 399660FC 938A9016`
	`5B042A7C EFADB307`
B-409: $a = 1,\ h = 2,\ f(x) = x^{409} + x^{87} + 1$	
$b =$	`0x 021A5C2 C8EE9FEB 5C4B9A75 3B7B476B 7FD6422E F1F3DD67 4761FA99`
	`D6AC27C8 A9A197B2 72822F6C D57A55AA 4F50AE31 7B13545F`
$n =$	`0x 01000000 00000000 00000000 00000000 00000000 00000000 000001E2`
	`AAD6A612 F33307BE 5FA47C3C 9E052F83 8164CD37 D9A21173`
B-571: $a = 1,\ h = 2,\ f(x) = x^{571} + x^{10} + x^5 + x^2 + 1$	
$b =$	`0x 02F40E7E 2221F295 DE297117 B7F3D62F 5C6A97FF CB8CEFF1 CD6BA8CE`
	`4A9A18AD 84FFABBD 8EFA5933 2BE7AD67 56A66E29 4AFD185A 78FF12AA`
	`520E4DE7 39BACA0C 7FFEFF7F 2955727A`
$n =$	`0x 03FFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF`
	`FFFFFFFF FFFFFFFF E661CE18 FF559873 08059B18 6823851E C7DD9CA1`
	`161DE93D 5174D66E 8382E9BB 2FE84E47`
K-163: $a = 1,\ b = 1,\ h = 2,\ f(x) = x^{163} + x^7 + x^6 + x^3 + 1$	
$n =$	`0x 00000004 00000000 00000000 00020108 A2E0CC0D 99F8A5EF`
K-233: $a = 0,\ b = 1,\ h = 4,\ f(x) = x^{233} + x^{74} + 1$	
$n =$	`0x 00000080 00000000 00000000 00000000 00069D5B B915BCD4 6EFB1AD5`
	`F173ABDF`
K-283: $a = 0,\ b = 1,\ h = 4,\ f(x) = x^{283} + x^{12} + x^7 + x^5 + 1$	
$n =$	`0x 01FFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFE9AE 2ED07577 265DFF7F`
	`265DFF7F 94451E06 1E163C61`
K-409: $a = 0,\ b = 1,\ h = 4,\ f(x) = x^{409} + x^{87} + 1$	
$n =$	`0x 007FFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFE5F`
	`83B2D4EA 20400EC4 557D5ED3 E3E7CA5B 4B5C83B8 E01E5FCF`
K-571: $a = 0,\ b = 1,\ h = 4,\ f(x) = x^{571} + x^{10} + x^5 + x^2 + 1$	
$n =$	`0x 02000000 00000000 00000000 00000000 00000000 00000000 00000000`
	`00000000 00000000 131850E1 F19A63E4 B391A8DB 917F4138 B630D84B`
	`E5D63938 1E91DEB4 5CFE778F 637C1001`

family which offer an "add with carry" as part of the instruction set, these may be fast single-instruction operations.

Algorithm 1. Modular addition

INPUT: A modulus p, and integers $a, b \in [0, p-1]$.
OUTPUT: $c = (a+b) \bmod p$.

1. $c_0 \leftarrow \mathrm{Add}(a_0, b_0)$.
2. For i from 1 to $t-1$ do: $c_i \leftarrow \mathrm{Add\_with\_carry}(a_i, b_i)$.
3. If the carry bit is set, then subtract p from $c = (c_{t-1}, \dots, c_2, c_1, c_0)$.
4. If $c \geq p$ then $c \leftarrow c - p$.
5. Return(c).

Modular subtraction is implemented in a fashion similar to addition; however, the carry is now interpreted as a "borrow." As with addition, the operations in steps 1 and 2 are typically fast in any case, but especially so if they are part of the processor's instruction set.

Algorithm 2. Modular subtraction

INPUT: A modulus p, and integers $a, b \in [0, p-1]$.
OUTPUT: $c = (a-b) \bmod p$.

1. $c_0 \leftarrow \mathrm{Subtract}(a_0, b_0)$.
2. For i from 1 to $t-1$ do: $c_i \leftarrow \mathrm{Subtract\_with\_borrow}(a_i, b_i)$.
3. If the carry bit is set, then add p to $c = (c_{t-1}, \dots, c_2, c_1, c_0)$.
4. Return(c).

3.3 Multiplication and squaring

Algorithm 3 is an elementary multiplication routine which arranges the arithmetic so that the product is calculated right-to-left. Other choices are possible (see [20, Algorithm 14.12], for example). Step 2.1 of the algorithm requires a 64-bit product of two 32-bit operands. Since multiplication is typically much more expensive than addition, a (fast) 32×32 multiply instruction should be used if available. In Algorithm 3, r_0, r_1, r_2, u and v are 32-bit words, and (uv) denotes the 64-bit concatenation of u and v.

Algorithm 3. Integer multiplication

INPUT: Integers $a, b \in [0, p-1]$.
OUTPUT: $c = a \cdot b$.

1. $r_0 \leftarrow 0$, $r_1 \leftarrow 0$, $r_2 \leftarrow 0$.
2. For k from 0 to $2(t-1)$ do
   2.1 For each element of $\{(i,j) \mid i+j=k,\ 0 \leq i,j < t\}$ do
   $(uv) = a_i \cdot b_j$.
   $r_0 \leftarrow \mathrm{Add}(r_0, v)$, $r_1 \leftarrow \mathrm{Add\_with\_carry}(r_1, u)$, $r_2 \leftarrow \mathrm{Add\_with\_carry}(r_2, 0)$.
   2.2 $c_k \leftarrow r_0$, $r_0 \leftarrow r_1$, $r_1 \leftarrow r_2$, $r_2 \leftarrow 0$.
3. $c_{2t-1} \leftarrow r_0$.
4. Return(c).

The method of Karatsuba [16] can be used to reduce the number of 32×32-bit multiplications at the cost of some complexity in the algorithm. For comparison, Karatsuba was implemented with a depth-2 split for each of the three smaller fields of interest.

A straightforward modification of the multiplication algorithm gives the following algorithm for squaring. There are roughly 1/2 fewer multiplication operations. In step 2.1, the notation "$(uv) \ll 1$" indicates multiplication of the 64-bit quantity by 2, which may be implemented as two shift-through-carry (if available) or as two additions with carry.

Algorithm 4. Classical squaring

INPUT: Integer $a \in [0, p-1]$.
OUTPUT: $c = a^2$.

1. $r_0 \leftarrow 0$, $r_1 \leftarrow 0$, $r_2 \leftarrow 0$.
2. For k from 0 to $2(t-1)$ do
   2.1 For each element of $\{(i,j) \mid i+j=k,\ 0 \le i \le j < t\}$ do
      $(uv) = a_i \cdot a_j$.
      If $(i<j)$ then $(uv) \ll 1$, $r_2 \leftarrow \text{Add\_with\_carry}(r2, 0)$.
      $r_0 \leftarrow \text{Add}(r_0, v)$, $r_1 \leftarrow \text{Add\_with\_carry}(r_1, u)$, $r_2 \leftarrow \text{Add\_with\_carry}(r_2, 0)$.
   2.2 $c_k \leftarrow r_0$, $r_0 \leftarrow r_1$, $r_1 \leftarrow r_2$, $r_2 \leftarrow 0$.
3. $c_{2t-1} \leftarrow r_0$.
4. Return(c).

The following squaring algorithm, based on Algorithm 14.16 of [20] as modified by Guajardo and Paar [9], was also implemented for the timings in Table 4.

Algorithm 5. Squaring

INPUT: Integer $a \in [0, p-1]$.
OUTPUT: $c = a^2$.

1. For i from 0 to $2t-1$ do: $c_i \leftarrow 0$.
2. For i from 0 to $t-1$ do
   2.1 $(uv) \leftarrow c_{2i} + a_i^2$.
   2.2 $c_{2i} \leftarrow v$, $C1 \leftarrow u$, $C2 \leftarrow 0$.
   2.3 For j from $i+1$ to $t-1$ do
      $(uv) \leftarrow c_{i+j} + a_i a_j + C1$, $C1 \leftarrow u$.
      $(uv) \leftarrow v + a_i a_j + C2$, $c_{i+j} \leftarrow v$, $C2 \leftarrow u$.
   2.4 $(uv) \leftarrow C1 + C2$, $C2 \leftarrow u$.
   2.5 $(uv) \leftarrow c_{i+t} + v$, $c_{i+t} \leftarrow v$.
   2.6 $c_{i+t+1} \leftarrow C2 + u$.
3. Return(c).

Despite the simplicity of Algorithms 3 and 4, register allocation and other (platform-dependent) optimizations can greatly influence the performance. For example, the Intel Pentium family of processors have relatively few registers, and the 32×32 multiplication is restrictive in the registers involved. Furthermore, some care in choosing instruction sequences and registers is required in order to cooperate with the processor's ability to "pair" instructions and fully exploit the processor's pipelining capabilities.

3.4 Reduction

The NIST primes are of special form which permits very fast modular reduction. For bitlengths of practical interest, the work of [3] suggests that the methods of Montgomery and Barrett (which do not take advantage of the special form of the prime) are roughly comparable. For comparison with the fast reduction techniques, Barrett reduction was implemented.

Algorithm 6. Barrett reduction

INPUT: $b > 3$, p, $k = \lfloor \log_b p \rfloor + 1$, $0 \le x < b^{2k}$, $\mu = \lfloor b^{2k}/p \rfloor$.
OUTPUT: $x \bmod p$.

1. $\hat{q} \leftarrow \lfloor \lfloor x/b^{k-1} \rfloor \cdot \mu / b^{k+1} \rfloor$.
2. $r \leftarrow (x \bmod b^{k+1}) - (\hat{q} \cdot p \bmod b^{k+1})$.
3. If $r < 0$ then $r \leftarrow r + b^{k+1}$.
4. While $r \ge p$ do: $r \leftarrow r - p$.
5. Return(r).

The arithmetic in Barrett reduction can be reduced by choosing b to be a power of 2. Note that calculation of μ may be done once per field. For the NIST primes Solinas [26] gives the following fast reduction algorithms.

Algorithm 7. Fast reduction modulo $p_{192} = 2^{192} - 2^{64} - 1$

INPUT: Integer $c = (c_5, c_4, c_3, c_2, c_1, c_0)$ where each c_i is a 64-bit word, and $0 \le c < p_{192}^2$.
OUTPUT: $c \bmod p_{192}$.

1. Define 192-bit ints: $s_1 = (c_2, c_1, c_0)$, $s_2 = (0, c_3, c_3)$, $s_3 = (c_4, c_4, 0)$, $s_4 = (c_5, c_5, c_5)$.
2. Return$(s_1 + s_2 + s_3 + s_4 \bmod p_{192})$.

Algorithm 8. Fast reduction modulo $p_{224} = 2^{224} - 2^{96} + 1$

INPUT: Integer $c = (c_{13}, \ldots, c_2, c_1, c_0)$ where each c_i is a 32-bit word, and $0 \le c < p_{224}^2$.
OUTPUT: $c \bmod p_{224}$.

1. Define 224-bit integers: $s_1 = (c_6, c_5, c_4, c_3, c_2, c_1, c_0)$,
   $s_2 = (c_{10}, c_9, c_8, c_7, 0, 0, 0)$, $s_3 = (0, c_{13}, c_{12}, c_{11}, 0, 0, 0)$,
   $s_4 = (c_{13}, c_{12}, c_{11}, c_{10}, c_9, c_8, c_7)$, $s_5 = (0, 0, 0, 0, c_{13}, c_{12}, c_{11})$.
2. Return$(s_1 + s_2 + s_3 - s_4 - s_5 \bmod p_{224})$.

Algorithm 9. Fast reduction modulo $p_{256} = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$

INPUT: Integer $c = (c_{15}, \ldots, c_2, c_1, c_0)$ where each c_i is a 32-bit word, and $0 \le c < p_{256}^2$.
OUTPUT: $c \bmod p_{256}$.

1. Define 256-bit integers: $s_1 = (c_7, c_6, c_5, c_4, c_3, c_2, c_1, c_0)$,
   $s_2 = (c_{15}, c_{14}, c_{13}, c_{12}, c_{11}, 0, 0, 0)$, $s_3 = (0, c_{15}, c_{14}, c_{13}, c_{12}, 0, 0, 0)$,
   $s_4 = (c_{15}, c_{14}, 0, 0, 0, c_{10}, c_9, c_8)$, $s_5 = (c_8, c_{13}, c_{15}, c_{14}, c_{13}, c_{11}, c_{10}, c_9)$,
   $s_6 = (c_{10}, c_8, 0, 0, 0, c_{13}, c_{12}, c_{11})$, $s_7 = (c_{11}, c_9, 0, 0, c_{15}, c_{14}, c_{13}, c_{12})$,
   $s_8 = (c_{12}, 0, c_{10}, c_9, c_8, c_{15}, c_{14}, c_{14})$, $s_9 = (c_{13}, 0, c_{11}, c_{10}, c_9, 0, c_{15}, c_{14})$.
2. Return$(s_1 + 2s_2 + 2s_3 + s_4 + s_5 - s_6 - s_7 - s_8 - s_9 \bmod p_{256})$.

Algorithm 10. Fast reduction modulo $p_{384} = 2^{384} - 2^{128} - 2^{96} + 2^{32} - 1$

INPUT: Integer $c = (c_{23}, \ldots, c_2, c_1, c_0)$ where each c_i is a 32-bit word, and $0 \le c < p_{384}^2$.
OUTPUT: $c \bmod p_{384}$.

1. Define 384-bit integers: $s_1 = (c_{11}, c_{10}, c_9, c_8, c_7, c_6, c_5, c_4, c_3, c_2, c_1, c_0)$,
   $s_2 = (0, 0, 0, 0, 0, c_{23}, c_{22}, c_{21}, 0, 0, 0, 0)$,
   $s_3 = (c_{23}, c_{22}, c_{21}, c_{20}, c_{19}, c_{18}, c_{17}, c_{16}, c_{15}, c_{14}, c_{13}, c_{12})$,
   $s_4 = (c_{20}, c_{19}, c_{18}, c_{17}, c_{16}, c_{15}, c_{14}, c_{13}, c_{12}, c_{23}, c_{22}, c_{21})$,
   $s_5 = (c_{19}, c_{18}, c_{17}, c_{16}, c_{15}, c_{14}, c_{13}, c_{12}, c_{20}, 0, c_{23}, 0)$,
   $s_6 = (0, 0, 0, 0, c_{23}, c_{22}, c_{21}, c_{20}, 0, 0, 0, 0)$, $s_7 = (0, 0, 0, 0, 0, 0, c_{23}, c_{22}, c_{21}, 0, 0, c_{20})$,
   $s_8 = (c_{22}, c_{21}, c_{20}, c_{19}, c_{18}, c_{17}, c_{16}, c_{15}, c_{14}, c_{13}, c_{12}, c_{23})$,
   $s_9 = (0, 0, 0, 0, 0, 0, 0, c_{23}, c_{22}, c_{21}, c_{20}, 0)$, $s_{10} = (0, 0, 0, 0, 0, 0, 0, c_{23}, c_{23}, 0, 0, 0)$.
2. Return($s_1 + 2s_2 + s_3 + s_4 + s_5 + s_6 + s_7 - s_8 - s_9 - s_{10} \bmod p_{384}$).

Algorithm 11. Fast reduction modulo $p_{521} = 2^{521} - 1$

INPUT: A binary integer $c = (c_{1041}, \ldots, c_2, c_1, c_0)$.
OUTPUT: $c \bmod p_{521}$.

1. Define 521-bit integers: $s_1 = (c_{1041}, \ldots, c_{514}, c_{513}, c_{512})$, $s_2 = (c_{511}, \ldots, c_2, c_1, c_0)$.
2. Return($s_1 + s_2 \bmod p_{521}$).

3.5 Inversion

Algorithm 12 computes the inverse of a non-zero field element $a \in [1, p-1]$ using a variant of the Extended Euclidean Algorithm (EEA). The algorithm maintains the invariants $Aa + dp = u$ and $Ca + ep = v$ for some d and e which are not explicitly computed. The algorithm terminates when $u = 0$, in which case $v = 1$ and $Ca + ep = 1$; hence $C = a^{-1} \bmod p$.

Algorithm 12. Binary inversion algorithm

INPUT: Prime p, $a \in [1, p-1]$.
OUTPUT: $a^{-1} \bmod p$.

1. $u \leftarrow a$, $v \leftarrow p$, $A \leftarrow 1$, $C \leftarrow 0$.
2. While $u \ne 0$ do
   2.1 While u is even do:
   $u \leftarrow u/2$. If A is even then $A \leftarrow A/2$; else $A \leftarrow (A + p)/2$.
   2.2 While v is even do:
   $v \leftarrow v/2$. If C is even then $C \leftarrow C/2$; else $C \leftarrow (C + p)/2$.
   2.3 If $u \ge v$ then: $u \leftarrow u - v$, $A \leftarrow A - C$; else: $v \leftarrow v - u$, $C \leftarrow C - A$.
3. Return($C \bmod p$).

3.6 Timings

Table 4 presents timing results on a Pentium II 400 MHz workstation for operations in the NIST prime fields. The first column for $\mathbb{F}_{p_{192}}$ indicates times for routines written in C without the aid of hand-coded assembly code[1]; the other

[1] A notable exception was made in that 32×32 multiply (with add) in assembly was used. This was done because standard C does not necessarily support a 32×32 multiply and does not give direct access to the carry bit.

columns show the best times with most code in assembly. The compiler for these timings was Microsoft C (professional edition), with maximal optimizations set; the assembler was the "Netwide Assembler" NASM.

The case for hand-coded assembly is fairly compelling from the table, although timings showed that much of the performance benefit in classical multiplication comes from relatively easy and limited insertion of assembly code. Some assembly coding was driven by the need to work around the relatively poor register-allocation strategy of the Microsoft compiler on some code.

As expected, fast reduction for the NIST primes was much faster than Barrett. Despite our best efforts, we could not make Karatsuba multiplication competitive with the classical version (but the situation was different on some platforms where primarily C was used). It is likely that the overhead in the Karatsuba code can be reduced by additional hand-tuning; however, it appears from the timings that such tuning is unlikely to be sufficient to change the conclusions for these fields on the given platform. The implementation of the squaring algorithm (Algorithm 5) is slower than classical squaring, in part due to the repeated accesses of the output array. The ratio of inversion to multiplication (with fast reduction) is roughly 80 to 1.

Table 4. Timings (in μs) for operations in the NIST prime fields.

	$\mathbb{F}_{p_{192}}$ [a]	$\mathbb{F}_{p_{192}}$	$\mathbb{F}_{p_{224}}$	$\mathbb{F}_{p_{256}}$	$\mathbb{F}_{p_{384}}$	$\mathbb{F}_{p_{521}}$
Addition (Algorithm 1)	0.235	0.097	0.114	0.123	0.169	0.162
Subtraction (Algorithm 2)	0.243	0.094	0.112	0.125	0.158	0.150
Modular reduction						
Barrett reduction (Algorithm 6)	3.645	1.021	1.462	1.543	3.004	5.448
Fast reduction (Algorithms 7–11)	0.223	0.203	0.261	0.522	0.728	0.503
Multiplication (including fast reduction)						
Classical (Algorithm 3)	1.268 [b]	0.823	1.074	1.568	2.884	4.771
Karatsuba	2.654 [c]	1.758	2.347	2.844	—	—
Squaring (including fast reduction)						
Classical (Algorithm 4)	—	0.705	0.913	1.358	2.438	3.864
Algorithm 5	1.951 [c]	1.005	1.284	1.867	3.409	5.628
Inversion (Algorithm 12)	146.21	66.30	88.26	115.90	249.69	423.21

[a] Coded primarily in C.
[b] Uses a 32×32 multiply-and-add.
[c] Uses a 32×32 multiply.

4 Elliptic Curve Point Representation

Affine coordinates. Let E be an elliptic curve over $\mathbb{F}_p$ given by the (affine) equation $y^2 = x^3 - 3x + b$. 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:

$$x_3 = \lambda^2 - x_1 - x_2, \;\; y_3 = \lambda(x_1 - x_3) - y_1, \text{ where}$$
$$\lambda = \frac{y_2 - y_1}{x_2 - x_1} \text{ if } P_1 \neq P_2, \;\; \text{and } \lambda = \frac{3x_1^2 - 3}{2y_1} \text{ if } P_1 = P_2. \tag{1}$$

When $P_1 \neq P_2$ (general addition) the formulas for computing P_3 require 1 inversion, 2 multiplications, and 1 squaring—as justified in §3.6, we can ignore the cost of field additions and subtractions. When $P_1 = P_2$ (doubling) the formulas for computing P_3 require 1 inversion, 2 multiplications, and 2 squarings.

Projective coordinates. Since inversion in $\mathbb{F}_p$ is significantly more expensive than multiplication (see §3.6), it is 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 $Y^2Z = X^3 - 3XZ^2 + bZ^3$. 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 $Y^2 = X^3 - 3XZ^4 + bZ^6$. In *Chudnovsky* Jacobian coordinates [5], the Jacobian point $(X : Y : Z)$ is represented as $(X : Y : Z : Z^2 : Z^3)$.

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 and §5.2) is the addition of two points using mixed coordinates—where the two points are given in different coordinates [6].

The field operation counts for point addition and doubling in various coordinate systems are listed in Table 5. From Table 5 we see that Jacobian coordinates yield the fastest point doubling, while mixed Jacobian-affine coordinates yield the fastest point addition. Also useful in some point multiplication algorithms (see Algorithm 17) are mixed Jacobian-Chudnovsky coordinates and mixed Chudnovsky-affine coordinates for point addition. We note that the modified Jacobian coordinates presented in [6] do not yield any speedups over ordinary Jacobian coordinates for curves with $a = -3$.

Table 5. Operation counts for elliptic curve point addition and doubling. A = affine, P = standard projective, J = Jacobian, C = Chudnovsky.

Doubling		General addition		Mixed coordinates	
$2A \to A$	$1I$, $2M$, $2S$	$A + A \to A$	$1I$, $2M$, $1S$	$J + A \to J$	$8M$, $3S$
$2P \to P$	$7M$, $3S$	$P + P \to P$	$12M$, $2S$	$J + C \to J$	$11M$, $3S$
$2J \to J$	$4M$, $4S$	$J + J \to J$	$12M$, $4S$	$C + A \to C$	$8M$, $3S$
$2C \to C$	$5M$, $4S$	$C + C \to C$	$11M$, $3S$		

Formulas for doubling in Jacobian coordinates are: $2(X_1 : Y_1 : Z_1) = (X_3 : Y_3 : Z_3)$, where

$$A = 4X_1 \cdot Y_1^2, \quad B = 8Y_1^4, \quad C = 3(X_1 - Z_1^2) \cdot (X_1 + Z_1^2), \quad D = -2A + C^2,$$
$$X_3 = D, \quad Y_3 = C \cdot (A - D) - B, \quad Z_3 = 2Y_1 \cdot Z_1. \tag{2}$$

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

$$A = X_2 \cdot Z_1^2, \quad B = Y_2 \cdot Z_1^3, \quad C = A - X_1, \quad D = B - Y_1,$$
$$X_3 = D^2 - (C^3 + 2X_1 \cdot C^2), \quad Y_3 = D \cdot (X_1 \cdot C^2 - X_3) - Y_1 \cdot C^3, \quad Z_3 = Z_1 \cdot C. \tag{3}$$

Formulas for addition in mixed Jacobian-Chudnovsky coordinates are: $(X_1 : Y_1 : Z_1) + (X_2 : Y_2 : Z_2 : Z_2^2 : Z_2^3) = (X_3 : Y_3 : Z_3)$, where

$$A = X_1 \cdot Z_2^2, \quad B = Y_1 \cdot Z_2^3, \quad C = X_2 \cdot Z_1^2 - A, \quad D = Y_2 \cdot Z_1^3 - B,$$
$$X_3 = D^2 - 2A \cdot C^2 - C^3, \quad Y_3 = D \cdot (A \cdot C^2 - X_3) - B \cdot C^3, \quad Z_3 = Z_1 \cdot Z_2 \cdot C. \tag{4}$$

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* and dominates the execution time of elliptic curve cryptographic schemes. We will assume that $\#E(\mathbb{F}_p) = nh$ where n is prime and h is small (so $n \approx p$), P has order n, and $k \in_R [1, n-1]$. §5.1 covers the case where P is not known a priori. 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; this case is covered in §5.2.

5.1 Unknown Point

Algorithm 13 is the additive version of the basic repeated-square-and-multiply method for exponentiation.

Algorithm 13. (Left-to-right) binary method for point multiplication

INPUT: $k = (k_{m-1}, \ldots, k_1, k_0)_2$, $P \in E(\mathbb{F}_p)$.
OUTPUT: kP.
1. $Q \leftarrow \mathcal{O}$.
2. For i from $m-1$ 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 $m/2$, whence the expected running time of Algorithm 13 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 $1.5mI + 3mM + 2.5mS$, where I denotes an inversion, M a multiplication, and

S a squaring. If mixed Jacobian-affine coordinates (see §4) are used, then Q is stored in Jacobian coordinates, while P is stored in affine coordinates. Thus the doubling in step 2.1 can be performed using (2), while the addition in step 2.2 can be performed using (3). The field operation count of Algorithm 13 is then $8mM + 5.5mS + (1I + 3M + 1S)$ (1 inversion, 3 multiplications and 1 squaring are required to convert back to affine coordinates).

If $P = (x, y) \in E(\mathbb{F}_p)$ then $-P = (x, -y)$. Thus point subtraction is as efficient as addition. This motivates using a *signed digit representation* $k = \sum k_i 2^i$, 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, and can be efficiently computed using Algorithm 14 [27].

Algorithm 14. 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 15 modifies Algorithm 13 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 $1/3$ [23]. It follows that the expected running time of Algorithm 15 is approximately $(m/3)A + mD$.

Algorithm 15. Binary NAF method for point multiplication

INPUT: $\text{NAF}(k) = \sum_{i=0}^{l-1} k_i 2^i$, $P \in E(\mathbb{F}_p)$.
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 15 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., [18] or [22]), and then process the digits using a sliding window of width w. Algorithm 16 from [27], 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 14 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 $-2^{w-1} \leq u < 2^{w-1}$. 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 $1/(w+1)$ [27]. It follows that the expected running time of Algorithm 16 is approximately $(1D + (2^{w-2}-1)A) + (m/(w+1)A + mD)$. When using mixed Jacobian-Chudnovsky coordinates, the running time is minimized when $w = 5$ for P-192, P-224, and P-256, while $w = 6$ is optimal for P-384 and P-521.

Algorithm 16. Window NAF method for point multiplication

INPUT: Window width w, $\mathrm{NAF}_w(k) = \sum_{i=0}^{l-1} k_i 2^i$, $P \in E(\mathbb{F}_p)$.
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).

5.2 Fixed Point

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 $2P, 2^2P, \ldots, 2^{m-1}P$ are precomputed, then the right-to-left binary method has expected running time $(m/2)A$ (all doublings are eliminated). In [4], 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 $d = \lceil m/w \rceil$, and let $Q_j = \sum_{i:k_i=j} 2^{wi}P$. Then

$$\begin{aligned} kP &= \sum_{i=0}^{d-1} k_i(2^{wi}P) = \sum_{j=1}^{2^w-1} \left(j \sum_{i:k_i=j} 2^{wi}P \right) = \sum_{j=1}^{2^w-1} jQ_j \\ &= Q_{2^w-1} + (Q_{2^w-1} + Q_{2^w-2}) + \cdots + (Q_{2^w-1} + Q_{2^w-2} + \cdots + Q_1). \quad (5) \end{aligned}$$

Algorithm 17 is based on this observation. Its expected running time is approximately $((d(2^w-1)/2^w - 1) + (2^w - 2))A$. The optimum choice of coordinates is affine in step 1, mixed Chudnovsky-affine in step 3.1, and mixed Jacobian-Chudnovsky in step 3.2.

Algorithm 17. Fixed-base windowing method

INPUT: Window width w, $d = \lceil m/w \rceil$, $k = (k_{d-1}, \ldots, k_1, k_0)_{2^w}$, $P \in E(\mathbb{F}_p)$.
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 a variation of the comb method proposed in [19], the binary representation of k is written in w rows, and the columns of the resulting rectangle are processed two columns at a time. We define $[a_{w-1}, \ldots, a_2, a_1, a_0]P = a_{w-1}2^{(w-1)d}P + \cdots + a_2 2^{2d}P + a_1 2^d P + a_0 P$, where $d = \lceil m/w \rceil$ and $a_i \in \{0, 1\}$. The expected running time of Algorithm 18 is $((d-1)(2^w-1)/2^w)A + ((d/2)-1)D$. The optimum choice of coordinates is affine in step 1, Jacobian in step 4.1, and mixed Jacobian-affine in step 4.2.

Algorithm 18. Fixed-base comb method with two tables

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

1. *Precomputation.* Let $e = \lceil d/2 \rceil$. Compute $[a_{w-1}, \ldots, a_0]P$ and $2^e[a_{w-1}, \ldots, a_0]P$ for all $(a_{w-1}, \ldots, a_1, a_0) \in \{0,1\}^w$.
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 $e-1$ downto 0 do
   4.1 $Q \leftarrow 2Q$.
   4.2 $Q \leftarrow Q + [K_i^{w-1}, \ldots, K_i^1, K_i^0]P + 2^e[K_{i+e}^{w-1}, \ldots, K_{i+e}^1, K_{i+e}^0]P$
5. Return(Q).

From Table 6 we see that the fixed-base comb method is expected to slightly outperform the fixed-base window method for similar amounts of storage. For our implementation, we chose $w = 4$ for the comb method and $w = 5$ for fixed-base window for curves over $\mathbb{F}_{p_{192}}$, $\mathbb{F}_{p_{224}}$, and $\mathbb{F}_{p_{256}}$; the curves over the larger fields $\mathbb{F}_{p_{384}}$ and $\mathbb{F}_{p_{521}}$ used $w = 5$ for comb and $w = 6$ in fixed-base window.

Table 6. Comparison of fixed-base window and fixed-base comb methods for $\mathbb{F}_{p_{192}}$. w is the window width, S denotes the number of points stored in the precomputation phase, and T denotes the number of field operations.

	$w=2$		$w=3$		$w=4$		$w=5$		$w=6$		$w=7$		$w=8$	
Method	S	T	S	T	S	T	S	T	S	T	S	T	S	T
Fixed-base window	95	860	63	745	47	737	38	876	31	1246	27	2073	23	3767
Fixed-base comb	6	1188	14	900	30	725	62	632	126	529	254	472	510	415

5.3 Timings

Table 7 presents rough estimates of costs in terms of both elliptic curve operations and field operations for point multiplication methods for the P-192 elliptic curve. These estimates serve as a guideline for comparing point multiplication algorithms without concern for platform or implementation specifics.

Table 7. Rough estimates of point multiplication costs for P-192, with $S = .85M$.

Method	Coordinates	w	Points stored	EC operations		Field operations		
				A	D	M	I	Total[a]
Binary (Algorithm 13)	affine	—	0	96	191	980	287	23940
	Jacobian-affine	—	0	96	191	2430	1	2510
Binary NAF (Algorithm 15)	affine	—	0	64	191	889	255	21289
	Jacobian-affine	—	0	64	191	2092	1	2172
Window NAF (Algorithm 16)	Jacobian-affine	4	3	42	192	1844	4	2164
	Jacobian-Chudnovsky	5	7	39	192	1949	1	2029
Fixed-base window (Algorithm 17)	Chudnovsky-affine & Jacobian-Chudnovsky	5	38	30[b]+37[c]	0	796	1	876
Fixed-base comb (Algorithm 18)	Jacobian-affine	4	30	45	23	645	1	725

[a] Total cost in field multiplications assuming $1I = 80M$.
[b] Jacobian-Chudnovsky.
[c] Chudnovsky-affine.

Table 8 presents timing results for the NIST curves over prime fields, obtained on a Pentium II 400 MHz workstation. The field arithmetic is largely in assembly, while the curve arithmetic is in C.

The timings in Table 8 are consistent with the estimates in Table 7. The large inverse to multiplication ratio gives a slight edge to the use of Chudnovsky over affine in Window NAF. As predicted, the simpler binary NAF with Jacobian coordinates obtains fairly comparable speeds with less code. The first column in Table 8 illustrates the rather steep performance penalty for using C over assembly in the field operations.

6 ECDSA Elliptic Curve Operations

The execution times of elliptic curve cryptographic schemes such as the ECDSA [1] 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 potentially speed the computation of $kP + lQ$ is simultaneous multiple point multiplication (Algorithm 20), also known as Shamir's trick [8]. Algorithm 20 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.

Table 8. Timings (in μs) for point multiplication on the NIST curves over prime fields.

	P-192[a]	P-192	P-224	P-256	P-384	P-521
Binary (Algorithm 13)						
Affine	44,604	20,570	31,646	47,568	153,340	347,478
Jacobian-affine	4,847	2,443	3,686	6,038	20,570	35,171
Binary NAF (Algorithm 15)						
Affine	39,838	18,306	26,260	42,402	136,376	310,386
Jacobian-affine	4,386	2,144	3,255	5,298	17,896	30,484
Window NAF (Algorithm 16)						
Jacobian-affine[b]	4,346	2,103	3,144	5,058	16,374	27,830
Jacobian-Chudnovsky[c]	4,016	1,962	2,954	4,816	16,163	27,189
Fixed-base window (Algorithm 17)						
Chud-affine & Jacobian-Chud[c]	1,563	812	1,161	1,773	6,389	9,533
Fixed-base comb (Algorithm 18)						
Jacobian-affine[b]	1,402	681	1,052	1,672	4,656	8,032

[a] Field ops coded primarily in C except for 32×32 multiply-and-add instructions.
[b] $w = 4$ in P-192, P-224, and P-256; $w = 5$ in P-384 and P-521.
[c] $w = 5$ in P-192, P-224, and P-256; $w = 6$ in P-384 and P-521.

Algorithm 20. Simultaneous multiple point multiplication

INPUT: Window width w, $k = (k_{m-1}, \ldots, k_1, k_0)_2$, $l = (l_{m-1}, \ldots, l_1, l_0)_2$, $P, Q \in E(\mathbb{F}_p)$.
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 9 lists the most efficient methods for computing kP, P fixed, for the NIST random prime curves, random binary curves, and Koblitz binary curves. The timings for the binary curves are from [10]. For each type of curve, two cases are distinguished—when there is no extra memory available and when memory is not heavily constrained. Table 10 does the same for computing $kP + lQ$ where P is fixed and Q is not known a priori. We should note that no special effort was expended in optimizing our field arithmetic over the larger fields $\mathbb{F}_{p_{384}}$, $\mathbb{F}_{p_{521}}$, $\mathbb{F}_{2^{409}}$ and $\mathbb{F}_{2^{571}}$—the optimization techniques used for these fields were restricted to those employed in the smaller fields.

Table 11 presents timings for these operations for the P-192 curve when the field arithmetic is implemented primarily in assembly, when Barrett reduction

Table 9. Timings (in μs) of the fastest methods for point multiplication kP, P fixed, in ECDSA signature generation.

Curve type	Memory constrained?	Fastest method	NIST curve				
			P-192	P-224	P-256	P-384	P-521
Random prime	No	Fixed-base comb[a]	681	1,052	1,672	4,656	8,032
	Yes	Binary NAF Jacobian	2,144	3,255	5,298	17,896	30,484
			B-163	B-233	B-283	B-409	B-571
Random binary	No	Fixed-base comb[b]	1,683	3,966	5,919	12,448	30,120
	Yes	Montgomery	3,240	7,697	11,602	29,535	71,132
			K-163	K-233	K-283	K-409	K-571
Koblitz binary	No	FBW TNAF (w=6)	1,176	2,243	3,330	7,611	18,118
	Yes	TNAF	1,946	4,349	6,612	15,762	37,685

[a] $w = 4$ for P-192, P-224, and P-256; $w = 5$ for P-384 and P-521.

[b] $w = 4$ for B-163, B-233, and B-283; $w = 5$ for B-409 and B-571. A "single table" comb method was used, which has half the points of precomputation for a given w compared with Algorithm 18.

Table 10. 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	NIST curve				
			P-192	P-224	P-256	P-384	P-521
Random prime	No	Fixed-base comb[a] + Window NAF Jac-Chud[b]	2,594	3,965	6,400	20,610	34,850
	No	Simultaneous (w=2)	2,663	4,898	7,510	22,192	40,048
	Yes	Binary NAF Jacobian	4,288	6,510	10,596	35,792	60,968
			B-163	B-233	B-283	B-409	B-571
Random binary	No	Simultaneous (w=2)	4,969	11,332	16,868	42,481	100,963
	No	Fixed-base comb (w=5) + Window NAF (w=5)	—	—	—	41,322	98,647
	Yes	Montgomery	6,564	15,531	23,346	59,254	142,547
			K-163	K-233	K-283	K-409	K-571
Koblitz binary	No	Window TNAF (w=5) + FBW TNAF (w=6)	2,702	5,348	7,826	17,621	40,814
	Yes	TNAF	3,971	8,832	13,374	31,618	75,610

[a] $w = 4$ for P-192, P-224, and P-256; $w = 5$ for P-384 and P-521.

[b] $w = 5$ for P-192, P-224, and P-256; $w = 6$ for P-384 and P-521.

is used instead of fast reduction[2], and when the field arithmetic is implemented primarily in C.

Table 11. Timings (in μs) of the fastest methods for point multiplication kP, P fixed, and for $kP + lQ$, P fixed and Q not known a priori on the P-192 curve.

Point multiplication method	Field arithmetic primarily in assembly	Barrett[a] reduction	Field arithmetic primarily in C
For kP:			
Fixed-base comb ($w = 4$)	681	1,211	1,402
Binary NAF Jacobian	2,144	3,906	4,386
For $kP + lQ$:			
Fixed-base comb ($w = 4$) + Window NAF Jac-Chud ($w = 5$)	2,594	4,767	5,278
Simultaneous ($w = 2$)	2,663	4,907	5,407
Binary NAF Jacobian	4,288	7,812	8,772

[a] Fast reduction is replaced by an assembler version of Barrett reduction (Alg. 6).

Finally, to give an indication of which field operations are worthy of further optimization efforts, Table 12 presents the percentage of the total time spent in Algorithm 15 on the operations of addition, subtraction, integer multiplication, integer squaring, fast reduction, and inversion. Note that 95.4% of the total execution time was spent on these basic operations.

Table 12. Average number of function calls and percentage of time spent on the basic field operations in executions of the binary NAF Jacobian method (Algorithm 15) for the P-192 curve.

Field operation	Number of function calls	Percentage of total time
Addition (Algorithm 1)	1,137	5.8%
Subtraction (Algorithm 2)	1,385	7.4%
Integer multiplication (Algorithm 3)	1,213	38.3%
Integer squaring (Algorithm 4)	934	28.2%
Fast reduction (Algorithm 7)	2,147	14.8%
Modular inversion (Algorithm 12)	1	0.9%

7 Conclusions

Significant performance improvements are obtained when using Jacobian and Chudnovsky coordinates, primarily due to the high inversion to multiplication

[2] Since Barrett reduction does not exploit the special nature of the NIST primes, the Barrett column of Table 11 can be interpreted as rough timings for ECDSA operations over a random 192-bit prime.

ratio observed in our implementation. The high cost of inversion also favored precomputation in Chudnovsky coordinates for point multiplication (in the case of a point which is not known a priori), although some extra storage was also required.

As a rough comparison with curves over binary fields, times for the curves over the smaller fields in ECDSA operations show that known-point multiplications were significantly faster in the Koblitz (binary) and random prime cases than for the random binary case. For the point multiplication $kP + lQ$ where only P is known a priori, the random prime timings were somewhat faster than the Koblitz binary times, and both were significantly faster than the random binary times.

In our environment, hand-coded algorithms in assembly for field arithmetic gave significant performance improvements. It should be noted that the routines for curves over binary fields in the ECDSA tables were written entirely in C; some performance improvements would be obtained if segments were optimized with assembly, although it is expected that these would be less than in the prime-field case.

As expected, the special form of the NIST primes makes modular reduction very fast; the times for reduction with the Barrett method were generally much larger than the fast reduction by a factor of more than 2.5.

8 Future Work

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, 11, 15].

Acknowledgements

The authors would like to thank 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, October 2000.
3. A. Bosselaers, R. Govaerts and J. Vandewalle, "Comparison of three modular reduction functions", *Advances in Cryptology–Crypto '93*, LNCS **773**, 1994, 175-186.
4. E. Brickell, D. Gordon, K. McCurley and D. Wilson, "Fast exponentiation with precomputation", *Advances in Cryptology–Eurocrypt '92*, LNCS **658**, 1993, 200-207.
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. H. Cohen, A. Miyaji and T. Ono, "Efficient elliptic curve exponentiation using mixed coordinates", *Advances in Cryptology–Asiacrypt '98*, LNCS **1514**, 1998, 51-65.
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. J. Guajardo and C. Paar, "Modified squaring algorithm", preprint, 1999.
10. D. Hankerson, J. Hernandez and A. Menezes, "Software implementation of elliptic curve cryptography over binary fields", *Cryptographic Hardware and Embedded Systems–CHES 2000*, to appear.
11. T. Hasegawa, J. Nakajima and M. Matsui, "A practical implementation of elliptic curve cryptosystems over $GF(p)$ on a 16-bit microcomputer", *Public Key Cryptography–Proceedings of PKC '98*, LNCS **1431**, 1998, 182-194.
12. IEEE 1363-2000, *Standard Specifications for Public-Key Cryptography*, 2000.
13. ISO/IEC 14888-3, *Information Technology – Security Techniques – Digital Signatures with Appendix – Part 3: Certificate Based-Mechanisms*, 1998.
14. ISO/IEC 15946, *Information Technology – Security Techniques – Cryptographic Techniques Based on Elliptic Curves*, Committee Draft (CD), 1999.
15. K. Itoh, M. Takenaka, N. Torii, S. Temma and Y. Kurihara, "Fast implementation of public-key cryptography on a DSP TMS320C6201", *Cryptographic Hardware and Embedded Systems–CHES '99*, LNCS **1717**, 1999, 61-72.
16. D. Knuth, *The Art of Computer Programming–Seminumerical Algorithms*, Addison-Wesley, 3rd edition, 1998.
17. N. Koblitz, "Elliptic curve cryptosystems", *Mathematics of Computation*, **48** (1987), 203-209.
18. 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.
19. C. Lim and P. Lee, "More flexible exponentiation with precomputation", *Advances in Cryptology–Crypto '94*, LNCS **839**, 1994, 95-107.
20. A. Menezes, P. van Oorschot and S. Vanstone, *Handbook of Applied Cryptography*, CRC Press, 1997.
21. V. Miller, "Uses of elliptic curves in cryptography", *Advances in Cryptology–Crypto '85*, LNCS **218**, 1986, 417-426.
22. A. Miyaji, T. Ono and H. Cohen, "Efficient elliptic curve exponentiation", *Proceedings of ICICS '97*, LNCS **1334**, 1997, 282-290.
23. 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.
24. National Institute of Standards and Technology, *Digital Signature Standard*, FIPS Publication 186-2, February 2000.
25. National Institute of Standards and Technology, *Advanced Encryption Standard*, work in progress.
26. J. Solinas, "Generalized Mersenne numbers", Technical Report CORR 99-39, Dept. of C&O, University of Waterloo, 1999.
27. J. Solinas, "Efficient arithmetic on Koblitz curves", *Designs, Codes and Cryptography*, **19** (2000), 195-249.