Field Inversion and Point Halving Revisited

Kenny Fong, Darrel Hankerson, Julio López, and Alfred Menezes

Abstract—We present a careful analysis of elliptic curve point multiplication methods that use the point halving technique of Knudsen and Schroeppel and compare these methods to traditional algorithms that use point doubling. The performance advantage of halving methods is clearest in the case of point multiplication kP, where P is not known in advance and smaller field inversion to multiplication ratios generally favor halving. Although halving essentially operates on affine coordinate representations, we adapt an algorithm of Knuth to allow efficient use of projective coordinates with halving-based windowing methods for point multiplication.

Index Terms—Public key cryptosystems, computer arithmetic, efficiency.

1 Introduction

A new method for point multiplication on nonsupersingular elliptic curves over binary fields was proposed independently by Knudsen [11] and Schroeppel [22]. The idea is to replace almost all point doublings in double-and-add methods with a potentially faster operation called point halving. Knudsen [11] presented some rough analysis which suggests that halving methods could be 39 percent faster than doubling methods ([23] claims a 50 percent improvement), but these claims have not been supported by experimental evidence or by detailed analysis.

The purpose of this paper is to carefully analyze point multiplication methods that use halving and to compare them with traditional methods that use doubling. We restrict our attention to implementations on software platforms; some issues with implementing point halving in hardware are discussed in [26]. Furthermore, we restrict our attention to elliptic curves over binary fields $\mathbb{F}_{2^m}$, where m is prime and where the reduction polynomials are trinomials or pentanomials. Such parameters are recommended or mandated by various cryptographic standards, including NIST's FIPS 186-2 [4].

We begin in Section 2 with a description of three variants of the extended Euclidean algorithm for computing inverses in $\mathbb{F}_{2^m}$. A careful analysis of the software implementation of multiplication and inversion is necessary for a fair comparison of halving and doubling methods because a lower relative inversion cost generally favors halving methods over doubling methods. Our extensive experiments suggest that a realistic estimate of the ratio I/M of inversion to multiplication cost is 8 (or higher) rather than the ratio of 3 that is often quoted in the literature [27], [2], [3]. We also analyze algorithms for division in $\mathbb{F}_{2^m}$ and compare them with inversion algorithms.

In Section 3, we review point halving and efficient methods for solving quadratic equations in $\mathbb{F}_{2^m}$. Most of the material in Sections 3.1, 3.2, and 3.3 is from [11] and [23] with the exceptions of an improved method for computing square roots in Section 3.2.3 and an adaptation of an algorithm in Section 3.3 for point multiplication that allows halving to efficiently cooperate with projective coordinate representations. Our analysis of halving methods is presented in Section 3.4. We compare the best halving and doubling methods for performing point multiplication kP in the cases where P is not known in advance and where P is known in advance. The former situation commonly arises in variants of the Diffie-Hellman key agreement protocol, while the latter is encountered in signature generation for ElGamal signature schemes. We also compare halving and doubling methods for performing simultaneous multiple point multiplication $kP + lQ$ that is encountered in signature verification for ElGamal signature schemes. Our analysis suggests that point halving methods are about 29 percent faster than point doubling methods for computing kP when P is not known in advance. The advantage is smaller for simultaneous multiple point multiplication. For point multiplication where P is known in advance, doubling methods outperform halving methods unless I/M is small. As a benchmark, it should be noted that the τ-adic methods for Koblitz curves [29] are significantly faster than halving-based methods, although the latter have the advantage of wider applicability.

2 Field Inversion and Division

When implementing elliptic curve methods, the cost of field inversion to multiplication is of fundamental interest, driving the selection of affine versus projective representations of curve points. As an example, on the NIST-recommended random binary curves over $\mathbb{F}_{2^m}$, the costs (in terms of field multiplications M and inversions I) for

- K. Fong is with the Department of Computer Science, Southern Illinois University, Carbondale, IL 62901. E-mail: kfong@cs.siu.edu.
- D. Hankerson is with the Department of Mathematics, Auburn University, Auburn, AL 36849. E-mail: hankedr@auburn.edu.
- J. López is with the Institute of Computing, University of Campinas, CP 6176 CEP 13084-971, Campinas/SP, Brazil. E-mail: jlopez@ic.unicamp.br.
- A. Menezes is with the Department of Combinatorics and Optimization, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1. E-mail: ajmeneze@uwaterloo.ca.

Manuscript received 31 Oct. 2002; revised 6 Jan. 2004; accepted 28 Jan. 2004. For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 117693.

point addition and doubling are summarized in the following table.

Point operation	Coordinate representation	
	affine	projective[a]
double	$I+2M$	$4M$
add	$I+2M$	$8M$

[a]Formulas appear in the Appendix.

Consider the case that point multiplication kP is to be performed using a method based on double-and-add, where P is not known in advance. The break-even I/M depends on the method used; however, a rough estimate (e.g., if window NAF methods are employed) is obtained by assuming that the cost for each bit of k is approximately $D + A/3$, where D denotes the cost of a point doubling, and A is the cost of a point addition. Under these assumptions, arithmetic using projective (and mixed) coordinates is expected to outperform affine-only arithmetic whenever $I > 3M$.

Goodman and Chandrakasan [7], Chang Shantz [28], and Schroeppel [24] noted that the binary Euclidean algorithm, commonly employed for inversion of field elements, can be modified to do division. This is of particular interest if affine arithmetic is in use, provided that division is cheaper than $I + M$.

In this section, we are interested in realistic estimates of I/M under the assumptions that the processor is general-purpose and can be targeted and that the code may be optimized for specific fields. Since it appears clear that I/M is large (e.g., 40 or more) on such processors for prime fields, the focus will be on binary fields $\mathbb{F}_{2^m}$, where m is prime (e.g., as specified in the NIST-recommended binary curves). A polynomial basis representation will be used for elements of $\mathbb{F}_{2^m}$. Elements of $\mathbb{F}_{2^m}$ are the binary polynomials in $\mathbb{F}_2[z]$ of degree at most $m-1$. The reduction polynomial is denoted by f.

Section 2.1 gives an overview of three variants of the Euclidean algorithm for inversion. As noted, the binary variant can be converted to a division algorithm. Section 2.2 considers computational issues in converting the variants to perform division. Timings and implementation notes on two popular platforms are presented in Section 2.3.

2.1 Inversion Based on the Euclidean Algorithm

The inverse of a nonzero element $a \in \mathbb{F}_{2^m}$ is denoted $a^{-1} \bmod f$ or, simply, a^{-1} if the reduction polynomial f is understood from context. Inverses can be efficiently computed by the extended Euclidean algorithm for polynomials, which uses the fact that $\gcd(a,b) = \gcd(b + ca, a)$ for all binary polynomials c.

Algorithm 2.1 is a variant of the classical Euclidean algorithm. Given invertible a, the algorithm maintains the invariants

$$ag_1 + fh_1 = u$$
$$ag_2 + fh_2 = v$$

for some h_1 and h_2 not explicitly calculated. The algorithm terminates when $u = 1$, in which case, $g_1 = a^{-1}$.

Algorithm 2.1 Extended Euclidean Algorithm (EEA) for inversion in $\mathbb{F}_{2^m}$
INPUT: $a \in \mathbb{F}_{2^m}$, $a \neq 0$.
OUTPUT: $a^{-1} \bmod f$.
1. $u \leftarrow a$, $v \leftarrow f$, $g_1 \leftarrow 1$, $g_2 \leftarrow 0$
2. While $u \neq 1$ do
   2.1 $j \leftarrow \deg(u) - \deg(v)$.
   2.2 If $j < 0$ then: $u \leftrightarrow v$, $g_1 \leftrightarrow g_2$, $j \leftarrow -j$.
   2.3 $u \leftarrow u + z^j v$, $g_1 \leftarrow g_1 + z^j g_2$
3. Return(g_1).

In contrast to Algorithm 2.1 where the bits of u and v are cleared from left to right (high degree terms to low degree terms), the *binary Euclidean algorithm* (BEA) clears bits of u and v from right to left.

Algorithm 2.2 Binary Euclidean Algorithm (BEA) for inversion in $\mathbb{F}_{2^m}$
INPUT: $a \in \mathbb{F}_{2^m}$, $a \neq 0$.
OUTPUT: $a^{-1} \bmod f$.
1. $u \leftarrow a$, $v \leftarrow f$, $g_1 \leftarrow 1$, $g_2 \leftarrow 0$.
2. While z divides u do:
   2.1 $u \leftarrow u/z$.
   2.2 If z divides g_1 then $g_1 \leftarrow g_1/z$; else $g_1 \leftarrow (g_1 + f)/z$.
3. If $u = 1$ then return(g_1).
4. If $\deg(u) < \deg(v)$ then: $u \leftrightarrow v$, $g_1 \leftrightarrow g_2$.
5. $u \leftarrow u + v$, $g_1 \leftarrow g_1 + g_2$.
6. Goto Step 2.

The degree calculations in Step 4 may be replaced by a simpler comparison on the binary representations of the polynomials. This differs from Algorithm 2.1, where explicit degree calculations are required.

The *almost inverse algorithm* (AIA) [27] is a modification of the binary inversion algorithm in which a polynomial g and a positive integer k are first computed satisfying $ag \equiv z^k \pmod{f}$. A reduction is then applied to obtain $a^{-1} = z^{-k} g \bmod f$. The invariants maintained are

$$ag_1 + fh_1 = z^k u$$
$$ag_2 + fh_2 = z^k v$$

for some h_1 and h_2 that are not explicitly calculated.

Algorithm 2.3 Almost Inverse Algorithm (AIA) for inversion in $\mathbb{F}_{2^m}$
INPUT: $a \in \mathbb{F}_{2^m}$, $a \neq 0$.
OUTPUT: $a^{-1} \bmod f$.
1. $u \leftarrow a$, $v \leftarrow f$, $g_1 \leftarrow 1$, $g_2 \leftarrow 0$, $k \leftarrow 0$.
2. While z divides u do:
   2.1 $u \leftarrow u/z$, $g_2 \leftarrow zg_2$, $k \leftarrow k + 1$.
3. If $u = 1$ then return($z^{-k} g_1$).
4. If $\deg(u) < \deg(v)$ then: $u \leftrightarrow v$, $g_1 \leftrightarrow g_2$.
5. $u \leftarrow u + v$, $g_1 \leftarrow g_1 + g_2$.
6. Goto Step 2.

A reduction of the form $z^{-k} g$ is required in Step 3 and can be performed as follows: Let $l = \min\{i \geq 1 \mid f_i = 1\}$, where $f(x) = f_m z^m + \cdots + f_1 z + f_0$. Let s be the polynomial formed by the l rightmost bits of g. Then, $sf + g$ is divisible by z^l and $t = (sf + g)/z^l$ has degree less than m; thus, $t = gx^{-l} \bmod f$.

This process can be repeated to finally obtain $z^{-k}g \bmod f$. The reduction polynomial is said to be *suitable* if l is above some threshold (which may depend on the implementation, e.g., $l \geq 32$ is desirable with 32-bit words) since, then, less effort is required in the reduction step.

Two strategies can be applied to enlarge the class of "suitable" polynomials. The method of the preceding paragraph can be extended to arbitrary $l \leq m$ at relatively low cost [12]. Let $q(z) = f_{l-1}z^{l-1} + \cdots + f_1 z + 1$ and precompute Q satisfying $Qq \equiv 1 \pmod{z^l}$ with $\deg Q < l$. If $S \equiv sQ \pmod{z^l}$ with $\deg S < l$, then $Sf + g$ is divisible by z^l. If $f(z) = z^m + q(z)$, then division by z^l requires two $l \times l$ polynomial multiplications. As an alternative, the reduction in Step 3 can be replaced by pre and postalgorithm multiplications [25]. The revised method finds $c = 1/a$ via $a' \leftarrow z^{2m}a \bmod f$, $c' \leftarrow z^k/a' \bmod f$, $c \leftarrow z^{2m-k}c' \bmod f$, that is, the revised algorithm processes $z^{2m}a$ rather than a, and Step 3 is modified to find $z^{2m-k}g$.

Step 2 of AIA is simpler than that in Algorithm 2.2. In addition, the g_1 and g_2 appearing in these algorithms grow more slowly in almost inverse. Thus, one can expect AIA to outperform BEA if the reduction polynomial is suitable and conversely. As with BEA, the explicit degree calculations may be replaced with simpler comparisons.

2.2 Division

The binary Euclidean algorithm can be easily modified to perform division $b/a = ba^{-1}$ [7], [28], [24]. In cases where I/M is small, this could be especially significant in elliptic curve schemes since an affine point operation could use division rather than an inversion and multiplication.

2.2.1 Division Using BEA

To obtain b/a, Algorithm 2.2 is modified at Step 1, replacing $g_1 \leftarrow 1$ with $g_1 \leftarrow b$. The associated invariants are

$$ag_1 + fh_1 = ub$$
$$ag_2 + fh_2 = vb.$$

On termination with $u = 1$, it follows that $g_1 = ba^{-1}$. The division algorithm is expected to have the same running time as BEA since g_1 in BEA goes to full-length in a few iterations at Step 2.2 (i.e., the difference in initialization of g_1 does not contribute significantly to the time for division versus inversion).

If BEA is the inversion method of choice, then affine point operations would benefit from the use of division since the cost of a point double or addition changes from $I + 2M$ to $I + M$. If I/M is small, then this represents a significant improvement, e.g., if I/M is indeed 3, then the use of a division algorithm variant of BEA provides a 20 percent reduction in the time to perform an affine point double or addition. However, if $I/M > 7$, then the savings is less than 12 percent. Note that, unless I/M is very small, it is likely that schemes are used which reduce the number of inversions required (e.g., halving and projective coordinates) so that point multiplication involves relatively few field inversions, diluting any savings from the use of a division algorithm.

2.2.2 Division Using EEA

Algorithm 2.1 can be transformed to a division algorithm in a similar fashion. However, the change in the initialization step may have significant impact on the implementation of a division algorithm based on EEA. There are two performance issues: tracking of the lengths of variables and implementing the addition to g_1 at Step 2.3.

In EEA, it is relatively easy to track the length of u and v efficiently (the lengths shrink), especially if the number of words t representing a field element is (roughly) four or more. In EEA, it is also possible to track the lengths of g_1 and g_2. However, the change in initialization for division means that g_1 goes to full-length immediately and tracking the lengths of g_1 and g_2 is no longer effective.

The second performance issue concerns the addition to g_1 at Step 2.3 of EEA. An implementation of EEA may assume that the addition may be done as ordinary polynomial addition with no reduction, i.e., the degrees of g_1 and g_2 never exceed $m - 1$. In adapting for division, Step 2.3 may be less efficiently implemented since g_1 is full-length on initialization.

2.2.3 Division Using AIA

Although Algorithm 2.3 is similar to the binary Euclidean algorithm, the ability to efficiently track the lengths of g_1 and g_2 (in addition to the lengths of u and v) may be an implementation advantage of AIA over BEA. As with EEA, this advantage is lost in a division algorithm variant of AIA.

It should be noted that efficient tracking of the lengths of g_1 and g_2 (in addition to the lengths of u and v) in AIA may involve significant code expansion (perhaps t^2 fragments rather than the t fragments in BEA). If this code expansion cannot be tolerated (because of application constraints or platform characteristics), then AIA may not be preferable to the other inversion algorithms (even if the reduction polynomial is suitable).[1]

2.3 Timings

Table 1 gives some comparative timings on two popular platforms: the Intel Pentium III and Sun UltraSPARC. The example fields are from the NIST recommendations, with reduction polynomials $f(z) = z^{163} + z^7 + z^6 + z^3 + 1$ and $f(z) = z^{233} + z^{74} + 1$, respectively. Field multiplication based on the comb method [17] appears to be fastest on these platforms. A width-4 comb was used and the times include reduction. Other than the MMX code and a one-line assembler fragment for EEA, algorithms were coded entirely in C.

Some table entries are as expected, e.g., the relatively good times for almost inverse in $\mathbb{F}_{2^{233}}$. Other entries illustrate the significant differences between platforms or between compilers on a single platform. To obtain acceptable multiplication times with gcc on the Sun SPARC, code was tuned to be more "gcc-friendly." Limited tuning

1. Most of the performance of AIA can be obtained with modest code expansion [25]. The lengths of the variables u and v decrease, while the lengths of g_1 and g_2 increase. If $l = \max\{\text{len}\, u, \text{len}\, v\}$, then AIA can be expanded under the assumption that the lengths of g_1 and g_2 are bounded by $t + 1 - l$, with a fall back generic inversion routine used in exceptional cases. Experimentally, we observed a performance penalty of roughly 15 percent compared to the times in Table 1 for $\mathbb{F}_{2^{233}}$ on the SPARC.

TABLE 1
Multiplication and Inversion Times (in μ sec) for the Intel Pentium III and Sun UltraSPARC IIe

	Pentium III (800 MHz)			SPARC (500 MHz)		
	32-bit		64-bit	32-bit		64-bit
Algorithm	gcc	icc	mmx	gcc	cc	cc
Arithmetic in $\mathbb{F}_{2^{163}}$						
multiplication	1.8	1.3	.7	1.9	1.8	.9
Euclidean algorithm	10.9	10.6	7.1	21.4	14.8	—
binary Euclidean algorithm	20.7	16.0	—	16.8	14.9	10.6
almost inverse ($a^{-1}=z^{2m-k}(z^k/z^{2m}a)$)	12.9	11.1	—	14.3	11.4	7.8
I/M	6.1	8.0	9.8	7.9	6.7	8.9
Arithmetic in $\mathbb{F}_{2^{233}}$						
multiplication	3.0	2.3	—	4.0	2.9	1.7
Euclidean algorithm	18.3	18.8	—	45.5	25.7	—
binary Euclidean algorithm	36.2	28.9	—	42.0	34.0	16.9
almost inverse	22.7	20.1	—	36.8	24.7	12.9
I/M	6.1	8.2		9.2	8.5	7.7

The compilers are GNU C 2.95 (gcc), Intel 6 (icc), and Sun Workshop 6U2 (cc). The 64-bit "multimedia" registers were employed for the entries under "mmx." Inversion to multiplication (I/M) uses the best inversion time.

for gcc was also performed on the inversion code. Optimizing the inversion code is tedious, in part because rough operation counts at this level often fail to capture processor or compiler characteristics adequately. There are apparent inconsistencies remaining in Table 1, but we believe that the fastest times provide meaningful estimates of inversion and multiplication costs on these platforms.

The timings do not make a very strong case for division using a modification of the BEA. Unless EEA or AIA can be converted to efficiently perform division, then it appears that division will be fastest via inversion followed by multiplication. Furthermore, the ratio I/M is at least 8 in most cases and, hence, the savings from use of a division algorithm would be less than 10 percent. With such a ratio, elliptic curve methods will be chosen to reduce the number of inversions, so the savings on a point multiplication kP would be significantly less than 10 percent.

On the other hand, if affine-only arithmetic is in use in a point multiplication method based on double-and-add, then a fast division would be especially welcomed even if I/M is significantly larger than 5. If BEA is the algorithm of choice, then division has essentially the same cost as inversion.

2.3.1 Implementation Notes

In addition to the special tuning required for gcc, there were other troublesome compiler differences and flaws. A small code change triggered an apparent optimization flaw in the Sun Workshop (6U2) compiler, causing shifts to be processed as multiplication, a much slower operation on that platform. The only workarounds were to postprocess the assembler output or use a weaker optimization setting.

We note that the Microsoft compiler (Visual C 6) gives times comparable to that produced by the Intel compiler (icc, on Linux in our case). However, the insertion of short inline assembly fragments is less effective than with icc or gcc since there is only limited ability in the Microsoft product to direct the cooperation with the surrounding C code. We also found significant optimization problems with the Microsoft compiler concerning inlining of C code, although this was not an issue for the algorithms in this section.

Multimedia registers. The Intel Pentium family (all but the original and the Pentium-Pro) and AMD processors possess eight 64-bit "multimedia" registers that were employed for the times in the column marked "mmx" [1], [9]. Use of these capabilities for field arithmetic is discussed in [5].

Field multiplication. The GNU C compiler (gcc) is weak at instruction scheduling on these platforms, but can be coerced into producing somewhat better sequences by relatively small changes to the source. The times in the table for multiplication with gcc on SPARC are for code that has received such tuning.

We believe that the commonly cited ratio of $I/M \approx 3$ [27], [2], [3] is too optimistic for processors such as the Pentium and SPARC and is due, in part, to use of a suboptimal field multiplication.

EEA. Algorithm 2.1 requires polynomial degree calculations. A relatively fast method uses a binary search and table lookup once the nonzero word of interest is located. Some processors have instruction sets from which a fast "bit scan" may be built. As an example, the Intel x86 has single instructions (*bsr* and *bsf*) for finding the position of the most or least significant bit in a word. A one-line assembler fragment for bit scan was used for the Intel EEA timings, resulting in an improvement of approximately 15 percent. The SPARC has a Hamming weight (population) instruction which Sun suggests using for building a fast bit scan from the right; unfortunately, our field representation needed a bit scan from the left.

The code tracks the lengths of u and v using t fragments of similar code, each fragment corresponding to the current "top" of u and v. Here, t was chosen to be the number of words required to represent field elements.

BEA. Algorithm 2.2 was implemented with a t-fragment split to track the lengths of u and v efficiently. Rather than the degree calculation indicated in Step 4, a simpler comparison on the appropriate words was used.

AIA. Algorithm 2.3 allows efficient tracking of the lengths of g_1 and g_2 (in addition to the lengths of u and v). A total of t^2 similar fragments of code were used, a significant amount of code expansion unless t is small. As with BEA, a simple comparison replaces the degree calculations. An optimization flaw in the Sun compiler for 64-bit code was corrected by replacing expensive multiplications with shifts in the compiler output.

3 Point Multiplication Using Point Halving

Let E be an elliptic curve over $\mathbb{F}_{2^m}$ defined by the equation $y^2+xy=x^3+ax^2+b$, where $a,b\in\mathbb{F}_{2^m}$, $b\neq 0$. Let $P=(x,y)$ be a point on E with $P\neq -P$. Then, the (affine) coordinates of $Q=2P=(u,v)$ can be computed as follows:

$$\lambda = x + y/x, \tag{1}$$

$$u = \lambda^2+\lambda+a, \tag{2}$$

$$v = x^2+u(\lambda+1). \tag{3}$$

Affine point doubling requires one field multiplication and one field division. With projective coordinates and $a\in\{0,1\}$, point doubling can be done in four field multiplications.

Point halving is the following operation: Given $Q=(u,v)$, compute $P=(x,y)$ such that $Q=2P$. Since halving is the reverse operation of doubling, the basic idea for halving is to solve (2) for λ, (3) for x, and, finally, (1) for y. That is, solve $\lambda^2+\lambda=u+a$ for λ and $x^2=v+u(\lambda+1)$ for x. Finally, compute $y=\lambda x+x^2$.

Let G be a point of odd order n on E. It can be proven that point doubling and point halving are automorphisms of $\langle G\rangle$. Therefore, given a point $Q\in\langle G\rangle$, one can always find a unique point $P\in\langle G\rangle$ such that $Q=2P$. Sections 3.1 and 3.2 describe an efficient algorithm for point halving in $\langle G\rangle$. In Section 3.3, point halving is used to obtain efficient *halve-and-add* methods for point multiplication in cryptographic schemes based on elliptic curves over binary fields. Section 3.4 compares the point halving methods and the traditional point doubling methods.

3.1 Point Halving

The notion of *trace* plays a central role in deriving an efficient algorithm for point halving. Let $\mathrm{Tr}:\mathbb{F}_{2^m}\to\mathbb{F}_{2^m}$ be defined by $\mathrm{Tr}(c)=c+c^2+c^{2^2}+\cdots+c^{2^{m-1}}$.

Lemma 3.1. *Let* $c,d\in\mathbb{F}_{2^m}$.

1. $\mathrm{Tr}(c)=\mathrm{Tr}(c^2)=\mathrm{Tr}(c)^2$; *in particular,*
$$\mathrm{Tr}(c)\in\{0,1\}.$$
2. *Trace is linear, i.e.,* $\mathrm{Tr}(c+d)=\mathrm{Tr}(c)+\mathrm{Tr}(d)$.
3. *The NIST-recommended random curves [4] over binary fields have* $\mathrm{Tr}(a)=1$.
4. *If* $(x,y)\in\langle G\rangle$, *then* $\mathrm{Tr}(x)=\mathrm{Tr}(a)$.

Given $Q=(u,v)\in\langle G\rangle$, point halving seeks the unique point $P=(x,y)\in\langle G\rangle$ such that $Q=2P$. The first step of halving is to find $\lambda=x+y/x$ by solving the equation

$$\hat\lambda^2+\hat\lambda=u+a \tag{4}$$

for $\hat\lambda\in\mathbb{F}_{2^m}$. An efficient algorithm for solving (4) is presented in Section 3.2. Let $\hat\lambda$ denote the solution of (4) obtained from this algorithm. It is easily verified that $\hat\lambda=\lambda$ or $\hat\lambda=\lambda+1$. If $\mathrm{Tr}(a)=1$, the following result [11] can be used to identify λ.

Theorem 3.2. *Let* $P=(x,y),Q=(u,v)\in\langle G\rangle$ *be such that* $Q=2P$ *and denote* $\lambda=x+y/x$. *Let* $\hat\lambda$ *be a solution to (4) and* $t=v+u\hat\lambda$. *Suppose that* $\mathrm{Tr}(a)=1$. *Then,* $\hat\lambda=\lambda$ *if and only if* $\mathrm{Tr}(t)=0$.

Proof. Recall that $x^2=v+u(\lambda+1)$. By Lemma 3.1.4, we get $\mathrm{Tr}(x)=\mathrm{Tr}(a)$ since $P=(x,y)\in\langle G\rangle$. Thus,

$$\mathrm{Tr}(v+u(\lambda+1))=\mathrm{Tr}(x^2)=\mathrm{Tr}(x)=\mathrm{Tr}(a)=1.$$

Hence, if $\hat\lambda=\lambda+1$, then $\mathrm{Tr}(t)=\mathrm{Tr}(v+u(\lambda+1))=1$ as required. Otherwise, we must have $\hat\lambda=\lambda$, which gives $\mathrm{Tr}(t)=\mathrm{Tr}(v+u\lambda)=\mathrm{Tr}(v+u((\lambda+1)+1))$. Since the trace is linear,

$$\begin{aligned}\mathrm{Tr}(v+u((\lambda+1)+1)) &= \mathrm{Tr}(v+u(\lambda+1))+\mathrm{Tr}(u)\\ &= 1+\mathrm{Tr}(u)=0.\end{aligned}$$

Hence, we conclude that $\hat\lambda=\lambda$ if and only if $\mathrm{Tr}(t)=0$. □

Theorem 3.2 suggests a simple algorithm for identifying λ in the case that $\mathrm{Tr}(a)=1$.[2] We can then solve $x^2=v+u(\lambda+1)$ for the unique root x. Section 3.2 presents efficient algorithms for finding traces and square roots in $\mathbb{F}_{2^m}$. Finally, if needed, $y=\lambda x+x^2$ may be recovered with one field multiplication.

Let the λ-representation of a point $Q=(u,v)$ be (u,λ_Q), where $\lambda_Q=u+v/u$. Given the λ-representation of Q as the input to point halving, we may compute t in Theorem 3.2 without converting to affine coordinates since

$$t=v+u\hat\lambda=u\Big(u+u+\frac{v}{u}\Big)+u\hat\lambda=u(u+\lambda_Q+\hat\lambda).$$

In point multiplication, repeated halvings may be performed directly on the λ-representation of a point, with conversion to affine coordinates only when a point addition is required.

Algorithm 3.3 Point halving

INPUT: λ-representation (u,λ_Q) or affine representation (u,v) of $Q\in\langle G\rangle$.

OUTPUT: λ-representation (x,λ_P) of $P=(x,y)\in\langle G\rangle$, where $\lambda_P=x+y/x$ and $Q=2P$.

1. Find a solution $\hat\lambda$ of $\hat\lambda^2+\hat\lambda=u+a$.
2. If the input is in λ-representation, then compute $t=u(u+\lambda_Q+\hat\lambda)$;
   else, compute $t=v+u\hat\lambda$.
3. If $\mathrm{Tr}(t)=0$, then $\lambda_P\leftarrow\hat\lambda$, $x\leftarrow\sqrt{t+u}$;
   else $\lambda_P\leftarrow\hat\lambda+1$, $x\leftarrow\sqrt{t}$.
4. Return (x,λ_P).

3.2 Performing Point Halving Efficiently

Point halving requires a field multiplication and three main steps: Computing the trace of t, solving the quadratic equation (4), and computing a square root. In a normal basis, field elements are represented in terms of a basis of

2. The algorithm can be modified for binary curves with $\mathrm{Tr}(a)=0$; however, it is comparatively complicated since $\mathrm{Tr}(v+u\lambda)$ and $\mathrm{Tr}(v+u(\lambda+1))$ may not necessarily be distinct. See [11], [23].

the form $\{\beta, \beta^2, \ldots, \beta^{2^{m-1}}\}$. The trace of an element $c = \sum c_i \beta^{2^i} = (c_{m-1}, \cdots, c_0)$ is given by $\mathrm{Tr}(c) = \sum c_i$. The square root computation is a right rotation: $\sqrt{c} = (c_0, c_{m-1}, \ldots, c_1)$. Squaring is a left rotation and $x^2 + x = c$ can be solved bitwise. These operations are expected to be inexpensive relative to field multiplication. However, field multiplication in software for normal basis representations is very slow in comparison to multiplication with a polynomial basis [21], [20]. Conversion between polynomial and normal bases at each halving is likely too slow to give a competitive method, even if significant storage is used [10]. For these reasons, we restrict our discussion to computations in a polynomial basis representation.

3.2.1 Computing the Trace

Let $c = \sum_{i=0}^{m-1} c_i z^i \in \mathbb{F}_{2^m}$, with $c_i \in \{0, 1\}$, represented as the vector $c = (c_{m-1}, \ldots, c_1, c_0)$. A primitive method for computing $\mathrm{Tr}(c)$ uses the definition of trace, requiring $m-1$ field squarings and $m-1$ field additions. A much more efficient method makes use of the property that the trace is linear: $\mathrm{Tr}(c) = \mathrm{Tr}(\sum_{i=0}^{m-1} c_i z^i) = \sum_{i=0}^{m-1} c_i \mathrm{Tr}(z^i)$. The values $\mathrm{Tr}(z^i)$ may be precomputed, allowing the trace of an element to be found efficiently, especially if $\mathrm{Tr}(z^i) = 0$ for most i.

Example 3.4. Consider $\mathbb{F}_{2^{163}}$ with reduction polynomial $f(z) = z^{163} + z^7 + z^6 + z^3 + 1$. A routine calculation shows that $\mathrm{Tr}(z^i) = 1$ if and only if $i \in \{0, 157\}$. As examples, $\mathrm{Tr}(z^{160} + z^{46}) = 0$, $\mathrm{Tr}(z^{157} + z^{46}) = 1$, and $\mathrm{Tr}(z^{157} + z^{46} + 1) = 0$. For $\mathbb{F}_{2^{233}}$ with reduction polynomial $f(z) = z^{233} + z^{74} + 1$, $\mathrm{Tr}(z^i) = 1$ if and only if $i \in \{0, 159\}$.

3.2.2 Solving the Quadratic Equation

The first step of point halving seeks a solution x of a quadratic equation of the form $x^2 + x = c$ over $\mathbb{F}_{2^m}$. The performance of this step is crucial in point halving.

Lemma 3.5. *Assume m is odd and let the* half-trace $H : \mathbb{F}_{2^m} \to \mathbb{F}_{2^m}$ *be defined by*

$$H(c) = \sum_{i=0}^{(m-1)/2} c^{2^{2i}}.$$

1. $H(c + d) = H(c) + H(d)$ *for all* $c, d \in \mathbb{F}[2^m]$.
2. $H(c)$ *is a solution of the equation* $x^2 + x = c + \mathrm{Tr}(c)$.
3. $H(c) = H(c^2) + c + \mathrm{Tr}(c)$ *for all* $c \in \mathbb{F}_{2^m}$.

Let $c = \sum_{i=0}^{m-1} c_i z^i \in \mathbb{F}_{2^m}$ with $\mathrm{Tr}(c) = 0$; in particular, $H(c)$ is a solution of $x^2 + x = c$. A simple method for finding $H(c)$ directly from the definition requires $m-1$ squarings and $(m-1)/2$ additions. If storage for $\{H(z^i) : 0 \le i < m\}$ is available, then Lemma 3.5.1 may be applied to obtain

$$H(c) = H\left(\sum_{i=0}^{m-1} c_i z^i\right) = \sum_{i=0}^{m-1} c_i H(z^i).$$

However, this requires storage for m field elements and the associated method requires an average of $m/2$ field additions.

Lemma 3.5 can be used to significantly reduce the storage required as well as the time needed to solve the quadratic equation. The basic strategy is to write $H(c) = H(c') + s$, where c' has fewer nonzero coefficients than c. For even i, note that $H(z^i) = H(z^{i/2}) + z^{i/2} + \mathrm{Tr}(z^i)$. Algorithm 3.6 is based on this observation, eliminating storage of $H(z^i)$ for all even i. Precomputation builds a table of $(m-1)/2$ field elements $H(z^i)$ for odd i and the algorithm is expected to have approximately $m/4$ field additions at Step 4. The terms involving $\mathrm{Tr}(z^i)$ and $H(1)$ have been discarded since it suffices to produce a solution $s \in \{H(c), H(c) + 1\}$ of $x^2 + x = c$.

Algorithm 3.6 Solve $x^2 + x = c$ (basic version)
INPUT: $c = \sum_{i=0}^{m-1} c_i z^i \in \mathbb{F}_{2^m}$ with $\mathrm{Tr}(c) = 0$.
OUTPUT: A solution s of $x^2 + x = c$.

1. Precompute $H(z^i)$ for odd i, $1 \le i \le m-2$.
2. $s \leftarrow 0$.
3. For i from $(m-1)/2$ downto 1 do
   3.1 If $c_{2i} = 1$ then do: $c \leftarrow c + z^i$, $s \leftarrow s + z^i$.
4. $s \leftarrow s + \sum_{i=1}^{(m-1)/2} c_{2i-1} H(z^{2i-1})$.
5. Return(s).

Further improvements are possible by use of Lemma 3.5 together with the reduction polynomial [11, Appendix B]. Let i be odd and define j and s by $m \le 2^j i = m + s < 2m$. The basic idea is to apply Lemma 3.5.3 j times, obtaining

$$H(z^i) = H(z^{2^j i}) + z^{2^{j-1} i} + \cdots + z^{4i} + z^{2i} + z^i + j\mathrm{Tr}(z^i). \quad (5)$$

Let $f(z) = z^m + r(z)$, where $r(z) = z^{b_\ell} + \cdots + z^{b_1} + 1$ and $0 < b_1 < \cdots < b_\ell < m$. Then,

$$\begin{aligned} H(z^{2^j i}) &= H(z^s r(z)) \\ &= H(z^{s+b_\ell}) + H(z^{s+b_{\ell-1}}) + \cdots + H(z^{s+b_1}) + H(z^s). \end{aligned}$$

Thus, storage for $H(z^i)$ may be exchanged for storage of $H(z^{s+e})$ for $e \in \{0, b_1, \ldots, b_\ell\}$ (some of which may be further reduced). The amount of storage reduction is limited by dependencies among elements $H(z^i)$.

If $\deg r < m/2$, the strategy can be applied in an especially straightforward fashion to eliminate some of the storage for $H(z^i)$ in Algorithm 3.6. For $m/2 < i < m - \deg r$,

$$\begin{aligned} H(z^i) &= H(z^{2i}) + z^i + \mathrm{Tr}(z^i) \\ &= H(r(z) z^{2i-m}) + z^i + \mathrm{Tr}(z^i) \\ &= H(z^{2i-m+b_\ell} + \cdots + z^{2i-m+b_1} + z^{2i-m}) + z^i + \mathrm{Tr}(z^i). \end{aligned}$$

Since $2i - m + \deg r < i$, the reduction may be applied to eliminate storage of $H(z^i)$ for odd i, $m/2 < i < m - \deg r$. If $\deg r$ is small, Algorithm 3.7 requires approximately $m/4$ elements of storage.

Algorithm 3.7 Solve $x^2 + x = c$
INPUT: $c = \sum_{i=0}^{m-1} c_i z^i \in \mathbb{F}_{2^m}$ with $\mathrm{Tr}(c) = 0$ and reduction polynomial $f(z) = z^m + r(z)$.
OUTPUT: A solution s of $x^2 + x = c$.

1. Precompute $H(z^i)$ for $i \in I_0 \cup I_1$, where
   $I_0 = [1, (m-1)/2] \setminus 2\mathbb{Z}$ and
   $I_1 = [m - \deg r, m-2] \setminus 2\mathbb{Z}$.

2. $s \leftarrow 0$.
3. For each odd $i \in ((m-1)/2, m - \deg r)$, processed in decreasing order, do:
   3.1 If $c_i = 1$, then do: $c \leftarrow c + z^{2i-m+b_\ell} + \cdots + z^{2i-m}$, $s \leftarrow s + z^i$.
4. For i from $(m-1)/2$ downto 1 do:
   4.1 If $c_{2i} = 1$, then do: $c \leftarrow c + z^i$, $s \leftarrow s + z^i$.
5. $s \leftarrow s + \sum_{i \in I_0 \cup I_1} c_i H(z^i)$.
6. Return(s).

The technique may also reduce the time required for solving the quadratic equation since the cost of reducing each $H(z^i)$ may be less than the cost of adding a precomputed value of $H(z^i)$ to the accumulator. Elimination of the even terms (Step 4) can be implemented efficiently. Processing odd terms (as in Step 3) is more involved, but will be less expensive than a field addition if only a few words must be updated.

Example 3.8. Consider $\mathbb{F}_{2^{163}}$ with reduction polynomial $f(z) = z^{163} + z^7 + z^6 + z^3 + 1$. Step 3 of Algorithm 3.7 begins with $i = 155$. By Lemma 3.5,

$$\begin{aligned} H(z^{155}) &= H(z^{310}) + z^{155} + \mathrm{Tr}(z^{155}) \\ &= H(z^{147}z^{163}) + z^{155} \\ &= H(z^{147}(z^7 + z^6 + z^3 + 1)) + z^{155}. \end{aligned}$$

If $c_{155} = 1$, then $z^{154} + z^{153} + z^{150} + z^{147}$ is added to c and z^{155} is added to s. In this fashion, storage for $H(z^i)$ is eliminated for $i \in \{83, 85, \cdots, 155\}$, the odd integers in $((m-1)/2, m - \deg r)$.

Algorithm 3.7 uses 44 field elements of precomputation. While this is roughly half that required by the basic algorithm, it is not minimal. For example, storage for $H(z^{51})$ may be eliminated since

$$\begin{aligned} H(z^{51}) &= H(z^{102}) + z^{51} + \mathrm{Tr}(z^{51}) \\ &= H(z^{204}) + z^{102} + z^{51} + \mathrm{Tr}(z^{102}) + \mathrm{Tr}(z^{51}) \\ &= H(z^{163}z^{41}) + z^{102} + z^{51} \\ &= H(z^{48} + z^{47} + z^{44} + z^{41}) + z^{102} + z^{51}, \end{aligned}$$

which corresponds to (5) with $j = 2$. The same technique eliminates storage for $H(z^i)$, $i \in \{51, 49, \ldots, 41\}$. Similarly, if (5) is applied with $i = 21$ and $j = 3$, then

$$H(z^{21}) = H(z^{12} + z^{11} + z^8 + z^5) + z^{84} + z^{42} + z^{21}.$$

Note that the odd exponents 11 and 5 are less than 21 and, hence, storage for $H(z^{21})$ may be eliminated.

In summary, the use of (5) with $j \in \{1, 2, 3\}$ eliminates storage for odd values of $i \in \{21, 41, \ldots, 51, 83, \ldots, 155\}$ and a corresponding algorithm for solving the quadratic equation requires 37 elements of precomputation. Further reductions are possible, but there are some complications since the formula for $H(z^i)$ involves $H(z^j)$ for $j > i$. As an example,

$$H(z^{23}) = H(z^{28} + z^{27} + z^{24} + z^{21}) + z^{92} + z^{46} + z^{23}$$

and storage for $H(z^{23})$ may be exchanged for storage on $H(z^{27})$. Our implementation uses these strategies to reduce the precomputation to 30 field elements, significantly less than the 44 used in Algorithm 3.7. In fact, use of

$$z^n = z^{157+n} + z^{n+1} + z^{n-3} + z^{n-6}$$

together with the previous techniques reduces the storage to 21 field elements $H(z^i)$ for

$$\begin{aligned} i \in \{&157, 73, 69, 65, 61, 57, 53, 39, 37, 33, 29, 27, \\ &17, 15, 13, 11, 9, 7, 5, 3, 1\}. \end{aligned}$$

However, this final reduction comes at a somewhat higher cost in required code compared with the 30-element version.

Experimentally, the algorithm for solving the quadratic equation (with 21 or 30 elements of precomputation) requires approximately $2/3$ the time of a field multiplication. Special care should be given to branch misprediction factors as this algorithm performs many bit tests.

Example 3.9. Consider $\mathbb{F}_{2^{233}}$ with reduction trinomial $f(z) = z^{233} + r(z) = z^{233} + z^{74} + 1$. In comparison with the reduction polynomial for $\mathbb{F}_{2^{163}}$ in the preceding example, $\deg r$ is relatively large. Algorithm 3.7 requires 95 field elements of precomputation, significantly more than the approximately $m/4 \approx 59$ elements required by the algorithm when $\deg r$ is small.

The amount of precomputation can be reduced to the 43 elements $H(z^i)$ for $i \in \{1, 3, \cdots, 79, 155, 157, 159\}$ by direct application of the relation $z^n = z^{n+159} + z^{n-74}$ together with Lemma 3.5.3. Using a slightly different order of computation, the entries for $i \in \{75, 77, 155, 157\}$ are eliminated (but at somewhat higher cost) and the corresponding algorithm uses 39 elements of precomputation. Experimentally, the algorithm solves the quadratic equation in approximately half the time of a field multiplication.

3.2.3 *Computing Square Roots in* $\mathbb{F}_{2^m}$

The basic method for computing $\sqrt{c}$, $c \in \mathbb{F}_{2^m}$, is based on the little theorem of Fermat: $c^{2^m} = c$. Then, $\sqrt{c}$ can be computed as $\sqrt{c} = c^{2^{m-1}}$, requiring $m - 1$ squarings. A more efficient method can be obtained from the observation that $\sqrt{c}$ can be expressed in terms of the square root of the element z. Let $c = \sum_{i=0}^{m-1} c_i z^i \in \mathbb{F}_{2^m}$, $c_i \in \{0, 1\}$. Since squaring is a linear operation in $\mathbb{F}_{2^m}$, the square root of c can be written as

$$\sqrt{c} = \left(\sum_{i=0}^{m-1} c_i z^i\right)^{2^{m-1}} = \sum_{i=0}^{m-1} c_i (z^{2^{m-1}})^i.$$

Splitting c into even and odd powers, we have

$$\begin{aligned}\sqrt{c} &= \sum_{i=0}^{(m-1)/2} c_{2i}(z^{2^{m-1}})^{2i} + \sum_{i=0}^{(m-3)/2} c_{2i+1}(z^{2^{m-1}})^{2i+1} \\ &= \sum_{i=0}^{(m-1)/2} c_{2i} z^i + \sum_{i=0}^{(m-3)/2} c_{2i+1} z^{2^{m-1}} z^i \\ &= \sum_{i \text{ even}} c_i z^{\frac{i}{2}} + \sqrt{z} \sum_{i \text{ odd}} c_i z^{\frac{i-1}{2}}.\end{aligned}$$

This reveals an efficient method for computing $\sqrt{c}$: Extract the two half-length vectors $c_{\text{even}} = (c_{m-1}, \ldots, c_4, c_2, c_0)$ and $c_{\text{odd}} = (c_{m-2}, \ldots, c_5, c_3, c_1)$ from c (assuming m is odd), perform a field multiplication of c_{odd} of length $\lfloor m/2 \rfloor$ with the precomputed value $\sqrt{z}$, and, finally, add this result with c_{even}. The computation is expected to require approximately half the time of a field multiplication.

An Improved Method for Trinomials. In the case that the reduction polynomial f is a trinomial, we can further speed the computation of $\sqrt{c}$ by the observation that an efficient formula for $\sqrt{z}$ can be derived directly from f. Let $f(z) = z^m + z^k + 1$ be an irreducible trinomial of degree m, where $m > 2$ is prime.

Consider the case that k is odd. Note that $1 \equiv z^m + z^k \pmod{f(z)}$. Then, multiplying by z and taking the square root, we get

$$\sqrt{z} \equiv z^{\frac{m+1}{2}} + z^{\frac{k+1}{2}} \pmod{f(z)}.$$

Thus, the product $\sqrt{z} \cdot c_{\text{odd}}$ requires two shift-left operations and one modular reduction.

Now, suppose k is even. Observe that $z^m \equiv z^k + 1 \pmod{f(z)}$. Then, dividing by z^{m-1} and taking the square root, we get

$$\sqrt{z} \equiv z^{-\frac{m-1}{2}}(z^{\frac{k}{2}} + 1) \pmod{f(z)}.$$

In order to compute z^{-s} modulo $f(z)$, where $s = \frac{m-1}{2}$, one can use the congruences $z^{-t} \equiv z^{k-t} + z^{m-t} \pmod{f(z)}$ for $1 \le t \le k$ for writing z^{-s} as a sum of few positive powers of z. Hence, the product $\sqrt{z} \cdot c_{\text{odd}}$ can be performed with a few shift-left operations and one modular reduction.

Example 3.10. The trinomial for the NIST-recommended finite field $\mathbb{F}_{2^{409}}$ is $f(z) = z^{409} + z^{87} + 1$. Then, the new formula for computing the square root of $c \in \mathbb{F}_{2^{409}}$ is

$$\sqrt{c} = c_{\text{even}} + z^{205} \cdot c_{\text{odd}} + z^{44} \cdot c_{\text{odd}} \bmod f(z).$$

Example 3.11. The trinomial for the NIST-recommended finite field $\mathbb{F}_{2^{233}}$ is $f(z) = z^{233} + z^{74} + 1$. Since $k = 74$ is even, we have $\sqrt{z} = z^{-116} \cdot (z^{37} + 1) \bmod f(z)$. Notice that $z^{-74} \equiv 1 + z^{159} \pmod{f(z)}$ and

$$z^{-42} \equiv z^{32} + z^{191} \pmod{f(z)}.$$

It follows that $z^{-116} \equiv z^{32} + z^{117} + z^{191} \pmod{f(z)}$. Hence, the new method for computing the square root of $c \in \mathbb{F}_{2^{233}}$ is

$$\sqrt{c} = c_{\text{even}} + (z^{32} + z^{117} + z^{191})(z^{37} + 1) \cdot c_{\text{odd}} \bmod f(z).$$

Compared to the standard method of computing square roots, the proposed technique eliminates the need for storage and replaces the required field multiplication by a faster operation. Experimentally, finding a root in Example 3.11 requires roughly $1/8$ the time of a field multiplication.

3.3 Point Multiplication

Let $P = (x, y) \in \langle G \rangle$ and k be an integer with $0 \le k < n$. Furthermore, let $\mathcal{O}$ denote the point at infinity and $t = \lfloor \log_2 n \rfloor + 1$. Point multiplication kP dominates the execution time of elliptic curve cryptographic schemes. The basic technique for point multiplication is the *double-and-add method*, also known as the *binary method*, which is the additive version of the repeated-square-and-multiply method for exponentiation. The expected number of ones in the binary representation of k is $t/2$, whence the expected running time of this method is approximately $(t/2)A + tD$, where A denotes a point addition and D denotes a point doubling.

Point subtraction on an elliptic curve is as efficient as point addition, motivating use of the *nonadjacent form* of k, $\text{NAF}(k) = \sum_{i=0}^{l-1} k_i 2^i$ with $k_i \in \{0, \pm 1\}$, which has the property that no two consecutive coefficients k_i are nonzero [29]. The *width-w NAF* is a generalization, where each nonzero coefficient k_i is odd, $|k_i| < 2^{w-1}$, and at most one of any consecutive w digits is nonzero. NAFs are used to reduce the number of point additions required in finding kP and have the following properties:

1. k has a unique width-w NAF, denoted $\text{NAF}_w(k)$.
2. $\text{NAF}_2(k) = \text{NAF}(k)$.
3. The length of $\text{NAF}_w(k)$ is at most one more than the length of the binary representation of k.
4. The average density of nonzero digits among all width-w NAFs of length l is approximately $1/(w+1)$.

Algorithm 3.12 modifies the binary method by using $\text{NAF}_w(k)$ instead of the binary representation of k. The expected running time is approximately

$$((w > 2) \cdot D + (2^{w-2} - 1)A) + (t/(w+1)A + tD),$$

where $(w > 2)$ is understood to be 1 if $w > 2$ and 0 otherwise. If P is known a priori, then the 2^{w-2} points calculated in Step 1 of Algorithm 3.12 can be precomputed statically and the expected running time of this algorithm will then be approximately $t/(w+1)A + tD$. If affine coordinates are used, then both point addition and point doubling cost $M + V$, where M denotes a field multiplication and V denotes a field division; for $w = 2$, this translates to a field operation count of $(4/3)tM + (4/3)tV$. The accumulator Q may be stored in projective coordinates, in which case a point addition costs $8M$ and a point doubling costs $4M$. The field operation count in the $w = 2$ case is then $(20/3)tM + (2M + I)$.

Algorithm 3.12 Window NAF method for point multiplication

INPUT: Window width w, $\text{NAF}_w(k) = \sum_{i=0}^{l-1} k_i 2^i$, $P \in \langle G \rangle$.
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 $k_i > 0$, then $Q \leftarrow Q + P_{k_i}$

3.3 If $k_i < 0$, then $Q \leftarrow Q - P_{-k_i}$.
4. Return(Q).

The *halve-and-add* method for point multiplication proposed by Knudsen and Schroeppel replaces almost all point doublings in double-and-add methods with point halvings. However, it may be necessary (depending on the application) to convert the representation of k.

Lemma 3.13. *Let $\sum_{i=0}^{t} k_i' 2^i$ be the w-NAF representation of $2^t k \bmod n$. Then,*

$$k \equiv \sum_{i=0}^{t} \frac{k'_{t-i}}{2^i} \pmod n.$$

Proof. We have $2^t k \equiv \sum_{i=0}^{t} k_i' 2^i \pmod n$. Since n is odd, we can divide the congruence by 2^t to obtain

$$k \equiv \sum_{i=0}^{t} \frac{k'_i}{2^{t-i}} \equiv \sum_{i=0}^{t} \frac{k'_{t-i}}{2^i} \pmod n.$$

□

Algorithm 3.14 presents a right-to-left version of the halve-and-add method with the input $2^t k \bmod n$ in w-NAF representation. Point halving occurs on the input P rather than on accumulators. The expected running time is approximately $(\text{step 3 cost}) + (t/(w+1) - 2^{w-2})A' + tH$, where H denotes a point halving and A' is the cost of a point addition when one of the inputs is in λ-representation. If projective coordinates are used for Q_i, then the additions in Step 2 are mixed-coordinate. Step 3 may be performed by conversion of Q_i to affine (with cost $I + (5 \cdot 2^{w-2} - 3)M$ if inverses are obtained by a simultaneous method) and then the sum is obtained by interleaving with appropriate signed-digit representations of the odd multipliers i. The cost for $2 \le w \le 5$ is approximately $w-2$ point doublings and 0, 2, 6, or 16 point additions, respectively.[3]

Algorithm 3.14 Halve-and-add w-NAF (right-to-left) method for point multiplication
INPUT: Window width w, $\mathrm{NAF}_w(2^t k \bmod n) = \sum_{i=0}^{t} k_i' 2^i$, $P \in \langle G \rangle$.
OUTPUT: kP. (Note: $k = k_0'/2^t + \cdots + k_{t-1}'/2 + k_t' \bmod n$.)
1. $Q_i \leftarrow \mathcal{O}$, $i \in I = \{1, 3, \ldots, 2^{w-1} - 1\}$.
2. For i from t downto 0 do:
   2.1 If $k_i' > 0$, then $Q_{k_i'} \leftarrow Q_{k_i'} + P$.
   2.2 If $k_i' < 0$ then $Q_{-k_i'} \leftarrow Q_{-k_i'} - P$.
   2.3 $P \leftarrow P/2$.
3. $Q \leftarrow \sum_{i \in I} i Q_i$.
4. Return(Q).

Consider the case $w = 2$. The expected running time of Algorithm 3.14 is then approximately $(1/3)tA' + tH$. If affine coordinates are used, then a point halving costs approximately $2M$, while a point addition costs $2M + V$ since the λ-representation of P must be converted to affine with one field multiplication. It follows that the field operation count with affine coordinates is approximately $(8/3)tM + (1/3)tV$. However, if Q is stored in projective coordinates, then a point addition requires $9M$. The field operation count of a mixed-coordinate Algorithm 3.14 with $w = 2$ is then approximately $5tM + (2M + I)$.

Algorithm 3.15 is a left-to-right method. Point halving occurs on the accumulator Q, whence projective coordinates cannot be used. The expected running time is approximately $(D + (2^{w-2} - 1)A) + (t/(w+1)A' + tH)$.

Algorithm 3.15 Halve-and-add w-NAF (left-to-right) method for point multiplication
INPUT: Window width w, $\mathrm{NAF}_w(2^t k \bmod n) = \sum_{i=0}^{t} k_i' 2^i$, $P \in \langle G \rangle$.
OUTPUT: kP. (Note: $k = k_0'/2^t + \cdots + k_{t-1}'/2 + k_t' \bmod n$.)
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 0 to t do
   3.1 $Q \leftarrow Q/2$.
   3.2 If $k_i' > 0$ then $Q \leftarrow Q + P_{k_i'}$.
   3.3 If $k_i' < 0$, then $Q \leftarrow Q - P_{-k_i'}$.
4. Return(Q).

3.4 Analysis

In comparison to methods based on doubling, point halving looks best when I/M is small and kP is to be computed for P not known in advance. In applications, the operations kP and $kP + lQ$ with P known in advance are also of interest and this section provides comparative results. The concrete examples used are the NIST random curves over $\mathbb{F}_{2^{163}}$ and $\mathbb{F}_{2^{233}}$ (known as B-163 and B-233, respectively), although the general conclusions apply more widely.

Example 3.16. Table 2 provides an operation count comparison between double-and-add and halve-and-add methods for the NIST random curve over $\mathbb{F}_{2^{163}}$. For the field operations, the assumption is that $I/M = 8$ and that a field division has cost $I + M$.

The basic NAF halving method is expected to outperform the w-NAF doubling methods. However, the halving method has 46 field elements of precomputation. In contrast, Algorithm 3.12 with $w = 4$ (which runs in approximately the same time as with $w = 5$) requires only six field elements of extra storage.

The left-to-right w-NAF halving method requires that the accumulator be in affine coordinates and point additions have cost $2M + V$ (since a conversion from λ-representation is required). For sufficiently large I/M, the right-to-left algorithm will be preferred; in the example, Algorithm 3.14 with $w = 2$ will outperform Algorithm 3.15 at roughly $I/M = 11$. Table 3 gives timings on an Intel Pentium III. Only general-purpose registers are used and all code is in C except for a one-line assembler fragment for computing polynomial degree during inversion. The observed inversion to multiplication ratio is $I/M \approx 8$. On this platform, field division is fastest by performing an inversion and multiplication, i.e., $V = I + M$.

The timing for solving $x^2 + x = c$ in $\mathbb{F}_{2^{163}}$ is with a routine that uses an 8-word table to assist in processing z^i for odd i, reducing the number of conditional expressions. (Branch misprediction penalties are a significant factor in

3. Knuth [13, Exercise 4.6.3-9] suggests calculating $Q_i \leftarrow Q_i + Q_{i+2}$ for i from $2^{w-1}-3$ to 1 and then the result is given by $Q_1 + 2\sum_{i \in I \setminus \{1\}} Q_i$. The cost is comparable in the projective point case. See also [18], [19].

TABLE 2
Point and Field Operation Counts for Point Multiplication for the NIST Random Curve over $\mathbb{F}_{2^{163}}$

Method	Storage (field elts)	Point operations	Field operations ($H = 2M$, $I/M = 8$) affine	projective
NAF, doubling (Algorithm 3.12)	0	$163D+54A$	$217(M+V)=2173$	$1089M+I=1097$
NAF, halving (Algorithm 3.14)	46	$163H+54A'$	$435M+54V=\ 924$	$817M+I=\ 825$
5-NAF, doubling (Algorithm 3.12)	14	$[D+7A]+163D+27A$	$198(M+V)=1982$	$879M+8V+I=\ 959$
4-NAF, halving (Algorithm 3.14)	55	$[3D+6A]+163H+30A'$	—	$671M+2I=\ 687$
5-NAF, halving (Algorithm 3.15)	60	$[D+7A]+163H+27A'$	$388M+35V=\ 705$	—

Halving uses 30 field elements of precomputation in solving $x^2 + x = c$ and 16 elements for square root. $A' = A + M$, the cost of a point addition when one of the inputs is in $\lambda\text{-}representation$. Field operation counts assume that a division V costs $I + M$.

the implementation.) On some platforms, incremental improvements in halving may be obtained by using a larger table of precomputation in the square root routine. Improvements in the routine to solve $x^2 + x = c$ were observed with limited use of assembly-language coding (essentially to improve on register allocation).

For point multiplication kP where P is not known in advance, the example case in Table 2 predicts that use of halving gives roughly 25 percent improvement over a similar method based on doubling, when $I/M = 8$. (On the test platform in Table 3, the observed improvement was 29 percent for B-163.) The improvement is less than the 39 percent estimate in [11], where the comparison was based on the use of methods similar to Algorithm 3.12 and Algorithm 3.14 with $w = 2$ and $I/M = 3$. The small ratio favors halving—if Table 2 is modified to use $I/M = 3$, then the predicted improvement using Algorithm 3.14 over Algorithm 3.12 with $w = 2$ matches that in [11]. The trinomial in $\mathbb{F}_{2^{233}}$ also favors halving, in part because the cost of finding a square root is significantly less than the estimate used to obtain Table 2.

TABLE 3
Curve and Field Timings (in μ sec) for the NIST Curves B-163 and B-223 on an 800 MHz Intel Pentium III, Using General-Purpose Registers Only

Algorithm	B-163	B-233
Field operations		
multiplication (width-4 comb [17], with reduction)	1.32	2.28
inversion (Alg. 2.1)	10.55	18.75
square root	.69[a]	.26[b]
solve $x^2 + x = c$	.89[c]	1.17[d]
Curve operations		
double (projective)	6.40	10.4
halve (Alg 3.3)	3.08	3.95
Point multiplication kP (random point)		
NAF, halving (Alg 3.14, mixed coords)	1262	2675
4-NAF, halving (Alg 3.14, mixed coords)	1062	2150
5-NAF, halving (Alg 3.15, affine coords)	1046	2200
5-NAF, doubling (Alg 3.12, mixed coords)	1477	3375
Point multiplication $kP + lQ$		
6-NAF interleaved with 5-NAF, halving (affine coords)	1431	3100
6-NAF interleaved with 5-NAF, doubling (mixed coords)	1769	4075

[a]16 elements of precomputation. [b]Example 3.11. [c]30 elements of precomputation. [d]39 elements of precomputation.

Multiple random elements are used to obtain realistic branch-misprediction penalties in routines such as solve. The Intel compiler version 6 was used on Linux 2.2.

The comparison is unbalanced in terms of storage required since halving was permitted 39-46 field elements of precomputation for solving $x^2 + x = c$ and finding square roots. The amount of storage in the square root routine (for $\mathbb{F}_{2^{163}}$) can be reduced at tolerable cost to halving; significant storage (e.g., 30 elements) for solving $x^2 + x = c$ appears to be essential. In addition to the routines specific to halving, most of the support for methods based on doubling will be required, giving some code expansion.

3.4.1 Random Curves versus Koblitz Curves

The $\tau\text{-adic}$ methods on Koblitz curves [29] (curves defined over $\mathbb{F}_2$) share strategy with halving in the sense that point doubling is replaced by a less-expensive operation. In the Koblitz curve case, the replacement is the Frobenius map $\tau : (x, y) \mapsto (x^2, y^2)$, an inexpensive operation compared to field multiplication. Point multiplication on Koblitz curves using $\tau\text{-adic}$ methods will be faster than those based on halving, with approximate cost for kP given by

$$\left(2^{w-2} - 1 + \frac{t}{w+1}\right) A + t \cdot (\text{cost of } \tau)$$

when using a width-w $\tau\text{-adic}$ NAF in a scheme similar to that described by Algorithm 3.12. To compare with Table 2, assume that mixed coordinates are used, $w = 5$, and that field squaring has approximate cost $M/6$. In this case, the operation count is approximately $379M$, significantly less than the $687M$ required by the halving method.

3.4.2 Known Point versus Unknown Point

In the case that P is known in advance (e.g., signature generation in ECDSA) and storage is available for precomputation, halving loses some of its performance advantages. For our case, and for relatively modest amounts of storage, the single-table comb method [8, Algorithm 17] is among the fastest and can be used to obtain meaningful operation count comparisons. The multiplier k is split into $w \geq 2$ rows and then columns are processed left to right; a total of $2^w - 1$ points of precomputation are required. The operation counts for kP using methods based on doubling and halving are approximately

$$\frac{t}{w}\left(D + \frac{2^w - 1}{2^w}A\right) \quad \text{and} \quad \frac{t}{w}\left(H + \frac{2^w - 1}{2^w}A'\right),$$

respectively. In contrast to the random point case, roughly half the operations are point additions. Note that the method based on doubling may use mixed-coordinate arithmetic (in which case, $D = 4M$, $A = 8M$, and there is a final conversion to affine), while the method based on halving must work in affine coordinates (with $H = 2M$ and $A' = V + 2M$). If $V = I + M$, then values of t and w of practical interest give a threshold I/M between 7 and 8, above which the method based on doubling is expected to be superior (e.g., for $w = 4$ and $t = 163$, the threshold is roughly 7.4).

3.4.3 Simultaneous Multiple Point Multiplication

In ECDSA signature verification, the computationally expensive step is a calculation $kP + lQ$ where only P is known in advance. The times in Table 3 for $kP + lQ$ use an interleaving method [6], [18] with width-w NAFs. Given widths w_1 and w_2, the points iP for odd $i < 2^{w_1-1}$ and iQ for odd $i < 2^{w_2-1}$ are computed; since P is known in advance, the precomputation involving P may be stored for repeated use. The expansions $\mathrm{NAF}_{w_1}(k)$ and $\mathrm{NAF}_{w_2}(l)$ are processed jointly, left to right, with a single double or halving of the accumulator at each stage. The expected operation count for the method based on doubling is approximately

$$[(w_2 > 2) \cdot D + (2^{w_2-2} - 1)A] + t\left[D + \left(\frac{1}{w_1 + 1} + \frac{1}{w_2 + 1}\right)A\right],$$

where the precomputation involving P is not included. (The expected count for the method using halving can be estimated by a similar formula; however, a more precise estimate must distinguish the case where consecutive additions occur since the cost is $A' + V + M$ rather than $2A'$.)

In the example case presented in Table 3, the interleaving method for $kP + lQ$ with halving is superior to the method based on doubling, although the difference is less pronounced than in the case of a random point multiplication kP due to the larger number of point additions relative to halvings. Note that the interleaving method cannot be efficiently converted to a right-to-left algorithm (where $w_1 = w_2 = 2$) since the halving or doubling operation would be required on two points at each step. For sufficiently large I/M, the method based on doubling will be superior; in the example, this occurs at roughly $I/M = 11.7$.

3.4.4 Constrained Environments

For workstations (e.g., the example platforms based on the SPARC and Pentium), the memory consumption of the algorithms and supporting routines described in this paper is relatively modest. Exceeding processor cache size may be a serious concern in some routines, but the memory consumed by a few dozen field elements may be inconsequential. The analysis is more complicated if there are significant memory constraints.

Point multiplication methods based on halving require most of the support used in methods based on doubling and there are also the routines for solving $x^2 + x = c$ and finding square roots. It appears that a significant number of field elements of precomputation (e.g., 21-30 for $\mathbb{F}_{2^{163}}$) are necessary for halving to be efficient. In comparison, the method of Montgomery point multiplication [16] can be coded compactly, requiring storage for only a few temporary field elements, and has running time approximately $6tM$ (which is competitive with Algorithm 3.12 with optimal w).

For $\mathbb{F}_{2^{163}}$, the field-dependent precomputation specific to halving includes 30 field elements for solving $x^2 + x = c$, 16 elements for square root, and 8 words to reduce the number of conditionals in solving $x^2 + x = c$; there is also a 256-byte table supporting extraction of even and odd bits of a word. For a fixed field, these tables are static. If dynamic storage is the principal constraint and the platform provides (fast) access to a sufficient amount of static data, then methods based on halving use roughly the same amount of the scarce resource as methods based on doubling.

Constraints on code and data size for field routines are likely to affect the inversion to multiplication ratio. (Squaring would also be affected if the static 8-to-16 expansion table of size 512 bytes must be shortened.) The scenario of interest here is where static storage is relatively abundant, but dynamic memory is scarce. If the 15 elements of data-dependent precomputation in the width-4 comb method must be reduced, then a reasonable choice is a right-to-left comb, requiring only a single field element (and some temporary storage comparable to that in the $w = 4$ comb), with performance degradation by a factor between 2 and 3. The penalty for inversion in the case that code size is limited is more difficult to estimate. (On the Pentium, for example, the Euclidean Algorithm 2.1 with limited code expansion incurs only a small penalty relative to the times in Table 1.) Constraints which give a smaller I/M will favor affine coordinates and halving methods.

In summary, methods based on halving are likely to retain their advantages in the constrained case over methods based on doubling, under the assumption that a threshold amount of static storage is available for solving $x^2 + x = c$. The advantages would, in fact, extend to the known-point case if constraints limit the number of points of precomputation. However, if processor speed is also limited, then there is a strong incentive to use Koblitz curves, provided that the cost of support for τ-adic NAFs is not prohibitive.

4 Conclusions

Point multiplication methods based on halving are straightforward to implement, although some extra static storage (per field) is required over methods based on doubling. The performance advantage of halving methods is clearest in the case of point multiplication kP, where P is not known in advance, and smaller inversion to multiplication ratios generally favor halving. Algorithm 3.14 partially addresses the challenge presented in Knudsen [11] to derive "an efficient halving algorithm for projective coordinates." While the algorithm does not provide halving on a projective point, it does illustrate an efficient windowing method with halving and projective coordinates, especially applicable in the case of larger I/M.

The analysis in [11] gives halving methods a 39 percent advantage for the unknown point case, under the assumption that $I/M \approx 3$. The results in Section 2 suggest that this ratio is too optimistic on common SPARC and Pentium platforms, where the fastest times give $I/M > 8$. The larger ratio reduces the advantage to approximately 25 percent in the unknown-point case under a similar analysis; if P is known in advance and storage for a modest amount of precomputation is available, then methods based on halving are inferior. For $kP + lQ$, where only P is known in advance, the differences between methods based on halving and methods based on doubling are smaller, with halving methods faster for ratios I/M commonly reported.

Our analysis using windowing methods estimates that point multiplication with halving is about 29 percent faster than doubling-based methods, under the assumptions that a field division costs roughly the same as inversion followed by multiplication, $I \approx 8M$, and $H \approx 2M$. In our experiments on an Intel Pentium III, we obtained $H \approx 2.3M$ for B-163 and $H \approx 1.7M$ for B-233 and the corresponding observed improvements in point multiplication times were 29 percent and 36 percent, respectively. For simultaneous point multiplication under similar assumptions, the analysis gives halving-based methods a 15 percent edge over those based on doubling. Experimentally, we observed improvements of 19 percent and 24 percent for B-163 and B-233, respectively.

Our work has focused on methods using relatively modest amounts of precomputation. However, the routines for solving quadratic equations benefit from per-field precomputation and are fundamental to the performance of halving-based methods. A practical comparison under more generous memory ceilings would be of interest.

Finally, it should be noted that methods based on halving will be significantly slower than τ-adic methods for Koblitz curves. However, the halving methods apply to all curves and finding a τ-adic NAF for a given k involves some extra code [29].

APPENDIX

In the projective coordinates of López-Dahab [15], the projective point $(X : Y : Z)$, $Z \neq 0$, corresponds to the affine point $(X/Z, Y/Z^2)$. The projective form of the elliptic curve equation $y^2 + xy = x^3 + ax^2 + b$ is

$$Y^2 + XYZ = X^3Z + aX^2Z^2 + bZ^4.$$

The point at infinity corresponds to $(1 : 0 : 0)$, while the negative of $(X : Y : Z)$ is $(X : X + Y : Z)$. The double $(X_3 : Y_3 : Z_3)$ of $(X_1 : Y_1 : Z_1)$ is given by

$$Z_3 \leftarrow X_1^2 \cdot Z_1^2, \;\; X_3 \leftarrow X_1^4 + b \cdot Z_1^4,$$
$$Y_3 \leftarrow bZ_1^4 \cdot Z_3 + X_3 \cdot (aZ_3 + Y_1^2 + bZ_1^4).$$

The mixed-coordinate sum $(X_3 : Y_3 : Z_3)$ of $(X_1 : Y_1 : Z_1)$ and $(X_2 : Y_2 : 1)$ is given by

$$A \leftarrow Y_2 \cdot Z_1^2 + Y_1, \;\; B \leftarrow X_2 \cdot Z_1 + X_1, \;\; C \leftarrow Z_1 \cdot B,$$
$$D \leftarrow B^2 \cdot (C + aZ_1^2), \;\; Z_3 \leftarrow C^2, \;\; E \leftarrow A \cdot C,$$
$$X_3 \leftarrow A^2 + D + E, \;\; F \leftarrow X_3 + X_2 \cdot Z_3,$$
$$G \leftarrow (X_2 + Y_2) \cdot Z_3^2, \;\; Y_3 \leftarrow (E + Z_3) \cdot F + G.$$

If $a \in \{0, 1\}$, then doubling in projective coordinates requires four field multiplications and addition (with mixed coordinates) requires eight multiplications [15], [14].

ACKNOWLEDGMENTS

The authors are indebted to Erik Knudsen, Richard Schroeppel, and the anonymous referees for numerous suggestions improving this paper.

REFERENCES

[1] Advanced Micro Devices, *AMD-K6 Processor Multimedia Technology*, Publication 20726, http://www.amd.com, 2000.

[2] E. De Win, A. Bosselaers, S. Vandenberghe, P. De Gersem, and J. Vandewalle, "A Fast Software Implementation for Arithmetic Operations in $GF(2^n)$," *Proc. Advances in Cryptology—ASIACRYPT '96*, pp. 65-76, 1996.

[3] E. De Win, S. Mister, B. Preneel, and M. Wiener, "On the Performance of Signature Schemes Based on Elliptic Curves," *Proc. Algorithmic Number Theory—ANTS-III*, pp. 252-266, 1998.

[4] FIPS 186-2, Digital Signature Standard (DSS), Federal Information Processing Standards Publication 186-2, Nat'l Inst. Standards and Technology, 2000.

[5] K. Fong, D. Hankerson, J. López, and A. Menezes, "Field Inversion and Point Halving Revisited," Technical Report CORR 2003-18, Dept. of Combinatorics and Optimization, Univ. of Waterloo, Canada, 2003, http://www.cacr.math.uwaterloo.ca.

[6] R. Gallant, R. Lambert, and S. Vanstone, "Faster Point Multiplication on Elliptic Curves with Efficient Endomorphisms," *Proc. Advances in Cryptology—CRYPTO 2001*, pp. 190-200, 2001.

[7] J. Goodman and A. Chandrakasan, "An Energy Efficient Reconfigurable Public-Key Cryptography Processor Architecture," *Proc. Cryptographic Hardware and Embedded Systems—CHES 2000*, pp. 175-190, 2000.

[8] D. Hankerson, J. López, and A. Menezes, "Software Implementation of Elliptic Curve Cryptography over Binary Fields," *Proc. Cryptographic Hardware and Embedded Systems—CHES 2000*, pp. 1-24, 2000.

[9] Intel Corp., *Intel Pentium 4 and Intel Xeon Processor Optimization Reference Manual*, Number 248966-04, http://developer.intel.com, 2001.

[10] B. Kaliski and Y. Yin, "Storage-Efficient Finite Field Basis Conversion," *Proc. Selected Areas in Cryptography—SAC '98*, pp. 81-93, 1999.

[11] E. Knudsen, "Elliptic Scalar Multiplication Using Point Halving," *Proc. Advances in Cryptology—ASIACRYPT '99*, pp. 135-149, 1999.

[12] E. Knudsen, personal communication, Aug. 2003.

[13] D. Knuth, *The Art of Computer Programming—Seminumerical Algorithms*, third ed. Addison-Wesley, 1998.

[14] C. Lim and H. Hwang, "Speeding Up Elliptic Scalar Multiplication with Precomputation," *Proc. Information Security and Cryptology—ICISC '99*, pp. 102-119, 2000.

[15] J. López and R. Dahab, "Improved Algorithms for Elliptic Curve Arithmetic in $GF(2^n)$," *Proc. Selected Areas in Cryptography—SAC '98*, pp. 201-212, 1999.

[16] J. López and R. Dahab, "Fast Multiplication on Elliptic Curves over $GF(2^m)$ without Precomputation," *Proc. Cryptographic Hardware and Embedded Systems—CHES '99*, pp. 316-327, 1999.

[17] J. López and R. Dahab, "High-Speed Software Multiplication in $\mathbb{F}_{2^m}$," *Proc. Progress in Cryptology—INDOCRYPT 2000*, pp. 203-212, 2000.

[18] B. Möller, "Algorithms for Multi-Exponentiation," *Proc. Selected Areas in Cryptography—SAC 2001*, pp. 165-180, 2001.

[19] B. Möller, "Improved Techniques for Fast Exponentiation," *Proc. Information Security and Cryptology (ICISC) 2002*, P. Lee and C. Lim, eds., pp. 298-312, 2003.

[20] P. Ning and Y. Yin, "Efficient Software Implementation for Finite Field Multiplication in Normal Basis," *Proc. Information and Comm. Security 2001*, pp. 177-189, 2001.
[21] A. Reyhani-Masoleh and M.A. Hasan, "Fast Normal Basis Multiplication Using General Purpose Processors (extended abstract)," *Proc. Selected Areas in Cryptography—SAC 2001*, pp. 230-244, 2001.
[22] R. Schroeppel, "Elliptic Curve Point Halving Wins Big," *Second Midwest Arithmetical Geometry in Cryptography Workshop*, Nov. 2000.
[23] R. Schroeppel, "Elliptic Curve Point Ambiguity Resolution Apparatus and Method," Int'l Application Number PCT/US00/31014, filed 9 Nov. 2000, publication number WO 01/35573 A1, 17 May 2001.
[24] R. Schroeppel, "Automatically Solving Equations in Finite Fields," US Patent Application No. 09/834,363, filed 12 Apr. 2001, publication number US 2002/0055962 A1, 9 May 2002.
[25] R. Schroeppel, personal communication, Oct. 2003.
[26] R. Schroeppel, C. Beaver, R. Gonzales, R. Miller, and T. Draelos, "A Low-Power Design for an Elliptic Curve Digital Signature Chip," *Proc. Cryptographic Hardware and Embedded Systems—CHES 2002*, pp. 366-380, 2002.
[27] R. Schroeppel, H. Orman, S. O'Malley, and O. Spatscheck, "Fast Key Exchange with Elliptic Curve Systems," *Proc. Advances in Cryptology—CRYPTO '95*, pp. 43-56, 1995.
[28] S. Chang Shantz, "From Euclid's GCD to Montgomery Multiplication to the Great Divide," SML Technical Report SMLI TR-2001-95, Sun Microsystems Laboratories, 2001.
[29] J. Solinas, "Efficient Arithmetic on Koblitz Curves," *Designs, Codes and Cryptography*, vol. 19, pp. 195-249, 2000.

Kenny Fong received the BMath degree in computer science and combinatorics and optimization from the University of Waterloo, Canada, in 2002, and the MS degree in computer science from Harvard University, in 2003. His area of interest is cryptography and information security.

Darrel Hankerson is a professor in the Department of Mathematics at Auburn University. His recent research interests are in cryptography, in particular, efficient implementation. He is coauthor of the *Guide to Elliptic Curve Cryptography*.

Julio López is a professor in the Institute of Computing at the University of Campinas, Brazil. His research interests are in cryptography, in particular, implementation of cryptographic algorithms.

Alfred Menezes is a professor in the Department of Combinatorics and Optimization at the University of Waterloo, Canada. His research interests are in cryptography and algorithmic number theory. He is coauthor of *Handbook of Applied Cryptography* and *Guide to Elliptic Curve Cryptography*