An Improved Implementation of Elliptic Curves over $GF(2^n)$ when Using Projective Point Arithmetic

Brian King

Motorola Labs, Schaumburg IL, USA,
Indiana University Purdue University at Indianapolis
briking@iupui.edu

Abstract. Here we provide a comparison of several projective point transformations of an elliptic curve defined over $GF(2^n)$ and rank their performance. We provide strategies to achieve improved implementations of each. Our work shows that under certain conditions, these strategies can alter the ranking of these projective point arithmetic methods.

1 Introduction

In [9,17], Koblitz and Miller independently proposed to use elliptic curves over a finite field to implement cryptographic primitives. One important primitive is the Diffie-Hellman key exchange (the elliptic curve version of the protocol is called the elliptic curve Diffie-Hellman key exchange, abbreviated as ECDH). The underlying task is computing the scalar multiple kP of a point P , where k is the user's private key.

The focus of this paper will be with elliptic curves (EC) defined over fields $GF(2^n)$. For the finite field $GF(2^n)$, the standard equation or Weierstrass equation for a non supersingular elliptic curve is:

$$y^2 + xy = x^3 + ax^2 + b$$

where $a, b \in GF(2^n)$. The points $P = (x, y)$, where $x, y \in GF(2^n)$, that satisfy the equation, together with the point $\mathcal{O}$, called the point of infinity, form an additive abelian group G . Here addition in G is defined by: for all $P \in G$

- $P + \mathcal{O} = P$,
- for $P = (x, y) \neq \mathcal{O}$, $-P = (x, x + y)$
- and for all $P_1 = (x_1, y_1)$, $P_2 = (x_2, y_2)$, both not equal to the identity and $P_1 \neq -P_2$, $P_1 + P_2 = P_3 = (x_3, y_3)$ where $x_3, y_3 \in GF(2^n)$ and satisfy:

$$x_3 = \begin{cases} \left(\frac{y_1 + y_2}{x_1 + x_2} \right)^2 + \frac{y_1 + y_2}{x_1 + x_2} + x_1 + x_2 + a & \text{if } P_1 \neq P_2 \\ x_1^2 + \frac{b}{x_1^2} & \text{if } P_1 = P_2 \end{cases} \quad (1)$$

$$y_3 = \begin{cases} \left(\frac{y_1 + y_2}{x_1 + x_2} \right) (x_1 + x_3) + x_3 + y_1 & \text{if } P_1 \neq P_2 \\ x_1^2 + \left(x_1 + \frac{y_1}{x_1} \right) x_3 + x_3 & \text{if } P_1 = P_2 \end{cases} \quad (2)$$

The computation of a scalar multiple of a point kP can be performed by expressing k in binary form $k = k_r k_{r-1} \dots k_1 k_0$ and applying the “double and add” method. That is,

$$kP = 2(\dots 2((2k_r P) + k_{r-1} P) + \dots) + k_0 P.$$

The “add” operation requires 2 field multiplications, 1 square, and 1 inverse. The “double” operation requires 2 field multiplications, 1 square, and 1 inverse¹.

An alternate method to computing kP is to use projective point coordinates. The use of projective point arithmetic on an elliptic curve, rather than the standard affine arithmetic is such that in projective point arithmetic one delays the computation of an inverse until the very end of the process of computing kP . By doing so one will naturally see a rise in the number of required field multiplications. That is, one inverse will take place during the computation of the key kP , whereas in the affine method, one inverse takes place for each “add” function invoked (and as well, for each “double” function invoked).

The decision “affine arithmetic” vs. “projective point arithmetic” should be decided based on the ratio

$$\frac{\text{time to compute an inverse}}{\text{time to multiply}}.$$

The larger this ratio, the more attractive it is to implement projective point arithmetic. Although there exists improved methods to compute inverses [21,7], the computation of an inverse will take significantly more time than a multiplication. For example, our implementation of field operations in $GF(2^{163})$, using generating polynomial $x^{163} + x^7 + x^6 + x^3 + 1$, was such that the performance of an inverse was over 10 times the time it took to perform a multiplication. This would be equivalent to “add” and “double” functions which requires approximately 13 multiplications each. This led us to investigate projective point arithmetic.

In the following, we discuss alternate methods to develop projective point arithmetic, in an effort to determine the optimal method. We include benchmarks to illustrate which is the optimal method, ranking the methods by performance. We propose strategies which can lead to further improvement in performance, and in some cases lead to different conclusions with regard to such rankings.

2 Some Implementation Strategies That Improve Efficiency

There are a number of resources that discuss efficiency improvements for elliptic curve implementations. Some examples of efficiency improvements include: im-

¹ These requirements reflect the most efficient “add” and the most efficient “double”.

proved key representations [2,24], improved field multiplication algorithms [18,6], the use of projective point coordinates [1,4,14], the use of a halving a point algorithm [22,13], and using the Frobenius map to improve efficiency [8,19,23]. Two resources that provide excellent overviews are [5,3]. For a review of ECC implementations in literature see [15]. In this section we provide a limited list of strategies. This list incorporates strategies considered in our implementations.

2.1 Field Multiplication Using a Lookup Table

In [6], Hasan described a method which uses lookup tables to improve performance of the field multiplication. The idea is to precompute all 2^g possible g -bit multiples of a multiplicand, and place them in a lookup table. Then you compute the multiplication by sliding over all $\lceil \frac{n}{g} \rceil$ many non overlapping g -bit windows of the other multiplicand. In all of our implementations, we use a four-bit window. This precomputation is placed into a lookup table. The product is then computed by sliding over the $\lceil \frac{n}{4} \rceil$ many non overlapping 4 bit windows of the other multiplicand. Therefore each call to a field multiplication creates a lookup table for one of the multiplicands.

2.2 Alternate Key Representations When Computing kP

Express k in NAF form

To reduce the number of “additions”, one may express k in NAF (non adjacent form) form (see [3,23,2,6,5]). The NAF representation of an integer is a “signed” binary representation such that no two consecutive bits are nonzero. For example, $30 = 16 + 8 + 4 + 2 = 11110_2$, a NAF form for $30 = 1000\bar{1}0_2$, we use $\bar{1}$ to represent -1 . In [2], it was shown that the expected weight of a NAF of length l is approximately $l/3$.

Use a windowing technique on k

Rather than computing kP using the binary representation of $k = k_r \dots k_1 k_0$. One could precompute the first $b - 1$ multiples of P , then express k in base b (see [18,12,3]).

2.3 Koblitz Curves

In [11], Koblitz suggested using anomalous binary curves (or *Koblitz curves*), which possess properties that can lead to improvements in efficiency. A curve described by the equation $y^2 + xy = x^3 + ax^2 + b$, where $b = 1$ and either $a = 0$ or $a = 1$, describes a Koblitz curve. In this setting, the Frobenius map, denoted by τ , is such that $\tau : (x, y) \mapsto (x^2, y^2)$ and satisfies the equation $2 = \tau - \tau^2$. Using this equation, we express the key k in τ -adic form. That is, since $k \in \mathbf{Z}[\tau]$, k which is $(k_i)_2$ can be expressed as $k = (t_j)_\tau$. For example, $010_2 = \bar{1}10_\tau$ describes the equation $2 = \tau - \tau^2$ (we use $\bar{1}$ to represent -1). This allows one to compute kP , as $\sum t_{i_\tau} P$, allowing us to use a “ τ and add” method rather than a “double and add” method [11,19,8]. The efficiency improvement is accomplished because

we are replacing the required 2 multiplications, 1 inverse and 1 square in the “double” by 2 squares (the “ τ ” of a point). Observe that in the “ τ and add” method, the time-consuming operation is the “add”.

2.4 Other Implementation Issues

The goal is to make relevant comparisons between different projective point representations. In particular, we would like to determine the most efficient representations and strategies for both Koblitz curves and “Random” curves. To make these comparisons, we will be using two curves contained in the WAP/WTLS list of curves [26]. One, a Koblitz curve with Weierstrass equation $y^2 + xy = x^3 + x^2 + 1$, where the field is $GF(2^{163})$ defined by generating polynomial $x^{163} + x^7 + x^6 + x^3 + 1$ (this curve has been included in many standards, and is identified in the WTLS standard as Curve 3). The other curve is defined by $y^2 + xy = x^3 + ax^2 + b^2$ and the underlying field is $GF(2^{163})$ defined by the generating polynomial $x^{163} + x^8 + x^2 + x + 1$ (this is identified in WTLS standard by Curve 5). Further assumptions, we will express the key in a NAF form (τ -NAF when the curve is a Koblitz curve). We will not apply any windowing techniques to the key. Our implementation of the field multiplication is such that it will create a four bit lookup table of one of the multiplicands (all possible four bit products of this multiplicand). Remember, the intent of this work is two-fold. To provide an overview of projective point methods, and secondly, to discuss efficiency improvements when using a projective coordinates with a field multiplication which uses lookup tables. The intention is to use benchmarking as a comparison between methods, and the effect of different strategies on these methods.

All benchmarks were created on a HP 9000/782 with a 236 MHz Risc processor (32 bit). The table belows illustrates the performance of the basic field operations in $GF(2^{163})$ with generating polynomial $x^{163} + x^7 + x^6 + x^3 + 1$ on this platform.

Operation	Time to compute
inverse	6.493 microsec
multiply	0.540 microsec
square	0.046 microsec

3 Projective Point Arithmetic

A projective plane over a field $\mathcal{F}$, can be defined by fixing positive integers α, β and creating an equivalence relation where $(x, y, z) \sim (x', y', z')$ if $(x', y', z') =$

² where in hexadecimal EC parameter

a is 072546B5435234A422E0789675F432C89435DE5242 and

b is 00C9517D06D5240D3CFF38C74B20B6CD4D6F9DD4D9

$(\lambda^\alpha x, \lambda^\beta y, \lambda z)$ where $\lambda \in \mathcal{F}$, $\lambda \neq 0$. An equivalence class is called a projective point in $\mathcal{F}$. Each affine point $(x', y') \in \mathcal{F} \times \mathcal{F}$ can be associated with the equivalence class $(x', y', 1)$. All ordered triples (x, y, z) within this class satisfy $x' = \frac{x}{z^\alpha}$ and $y' = \frac{y}{z^\beta}$. The projective plane can be thought of as the union of the affine plane together with all equivalence classes for which $z = 0$.

Each affine point (x, y) can be mapped into the projective plane by $\phi : (x, y) \rightarrow (x, y, 1)$. This map allows us to make a natural transformation from the affine plane to the projective plane. Then the $Image(\phi)$ consist of all equivalence classes such that $z \neq 0$. Each equivalence class (x, y, z) in the $Image(\phi)$ can be mapped to the affine plane by $x' = \frac{x}{z^\alpha}$ and $y' = \frac{y}{z^\beta}$, let us call this map ψ . Observe then that $(\psi \circ \phi)(x, y) = (x, y)$.

Recall that we are considering an elliptic curve E given by $y^2 + xy = x^3 + ax^2 + b$ defined over $GF(2^n)$. The set of affine points satisfying this equation form an additive abelian group G , and we have denoted this addition by $+$. Together Equations (1) and Equations (2) describe the addition operation in G . (Altogether we have four equations, two to describe x'_3 and two to describe y'_3). The projective point addition will be denoted by $+$. That is, for all $P, Q \in EC$, there exists a $\zeta \in \phi(E)$, such that $\zeta = \phi(P) + \phi(Q)$, and $\psi(\zeta) = P + Q$. The goal is to describe $+$ using the projective point coordinates $\phi(P)$ and $\phi(Q)$ so that a field inverse operation in $GF(2^n)$ is not required. For a brief discussion on how to generate alternate projective point representations see the Appendix.

In practice the concern is to compute kP , where P is a fixed affine point (x_2, y_2) . (In fact if the intention is to compute the ECDH key, then P represents the other user's public key.) This point P is transformed to the projective point $(x_2, y_2, 1)$. Q will represent a projective point (x_1, y_1, z_1) , and will represent the partial computation of kP as we parse the key k . Thus we assume that P is of the form $P = (x_2, y_2, 1)$, that is, $z_2 = 1$. This assumption is adopted for all projective point formulas described here (this assumption is referred to as using mixed coordinates). From now on we will always make this assumption, and we introduce all projective point arithmetic operations under this assumption. Do note that when adding two EC points one needs to test the cases: are the two points equal, is one point equal to the point at infinity, and is one point the negative of the other. However, our intention is to consider the computation of kP where P is a point in a subgroup of prime order. Thus except for the case when $k = \text{subgroup order} - 1$, these cases will never arise. So when discussing the "add", we omit testing for these cases.

3.1 The Homogeneous Projective Point Representation

The Homogeneous projective point transformation [1,16,10] is such that the relationship between affine points (x', y') and projective point (x, y, z) , where $z \neq 0$, is given by $x' = \frac{x}{z}$ and $y' = \frac{y}{z}$. For this setting, the projective point (x, y, z) which belongs to $\phi(E)$ must satisfy $zy^2 + zxy = x^3 + ax^2 + bz^3$.

Assume Q is a projective point (x_1, y_1, z_1) and P is $(x_2, y_2, 1)$, further we assume both P and $Q \neq \mathcal{O}$ and that $P \neq -Q$. There are two cases two consider: the "add" (when $P \neq Q$) and the "double" (when $Q = P$). If $P \neq \pm Q$, then

$$P +_* Q = (x_3, y_3, z_3)$$

$$\begin{aligned} x_3 &= AD \\ y_3 &= CD + A^2(Bx_1 + Ay_1) \\ z_3 &= A^3z_1 \end{aligned} \quad (3)$$

$A = x_2z_1 + x_1$, $B = y_2z_1 + y_1$, $C = A + B$ and $D = A^2(A + az_1) + z_1BC$. In the case where $P = Q$, $2Q = (x_3, y_3, z_3)$

$$\begin{aligned} x_3 &= AB \\ y_3 &= x_1^4A + B(x_1^2 + y_1z_1 + A) \\ z_3 &= A^3 \end{aligned} \quad (4)$$

where $A = x_1z_1$ and $B = bz_1^4 + x_1^4$.

3.2 The Jacobian Projective Point Representation

In [4], G. Chudnovsky and D. Chudnovsky described the Jacobian projective point representation. The IEEE working group P1363 [25] is developing a public-key cryptography standard and has incorporated the Jacobian transformation as their recommended manner to perform elliptic curve arithmetic when using projective point representation. The Jacobian transformation is given by $x' = \frac{x}{z^2}$ and $y' = \frac{y}{z^3}$ to perform elliptic curve addition when using projective point coordinates. For this projective to affine transformation, the projective point (x, y, z) which belongs to $\phi(E)$ must satisfy $y^2 + zxy = x^3 + az^2x^2 + bz^6$.

To compute $P +_* Q = (x_3, y_3, z_3)$, with $Q = (x_1, y_1, z_1)$ and $P = (x_2, y_2, 1)$ where both P and $Q \neq \mathcal{O}$: if $P \neq \pm Q$

$$\begin{aligned} U_1 &= x_1 & S_2 &= y_2z_1^3 & z_3 &= Lz_1 \\ S_1 &= y_1 & R &= S_1 + S_2 & T &= R + z_3 \\ U_2 &= x_2z_1^2 & L &= z_1W & x_3 &= az_3^2 + TR + W^3 \\ W &= U_1 + U_2 & V &= Rx_2 + Ly_1 & y_3 &= Tx_3 + VL^2. \end{aligned} \quad (5)$$

In the case where $P = Q$, $2Q = (x_3, y_3, z_3)$

$$\begin{aligned} c &= b^{2^{n-2}} & U &= z_3 + x_1^2 + y_1z_1 \\ z_3 &= x_1z_1^2 & y_3 &= x_1^4z_3 + Ux_1. \\ x_3 &= (x_1 + cz_1^2)^4 \end{aligned} \quad (6)$$

Here $c = \sqrt{b}$ in $GF(2^n)$. The equations collectively numbered (5) are performed columnwise left to right, top to bottom.

3.3 Lopez & Dahab

In [14], Lopez and Dahab described an alternate projective point representation. Here the relationship between affine point (x', y') and projective point (x, y, z)

is $x' = \frac{x}{z^2}$ and $y' = \frac{y}{z}$. Then all projective points (x, y, z) satisfy $y^2 + xyz = zx^3 + az^2x^2 + bz^4$. To compute $P +_* Q = (x_3, y_3, z_3)$, with $Q = (x_1, y_1, z_1)$ and $P = (x_2, y_2, 1)$ where both P and $Q \neq \mathcal{O}$: if $P \neq \pm Q$

$$\begin{aligned} A &= y_2 z_1^2 + y_1 & z_3 &= C^2 & G &= x_3 + y_2 Z_3 \\ B &= x_2 z_1 + x_1 & E &= AC & y_3 &= EF + z_3 G \\ C &= z_1 B & x_3 &= A^2 + D + E \\ D &= B^2(C + az_1^2) & F &= x_3 + x_2 z_3 \end{aligned} \quad (7)$$

In the case where $P = Q$, $2Q = (x_3, y_3, z_3)$

$$\begin{aligned} z_3 &= z_1^2 x_1^2 \\ x_3 &= x_1^4 + bz_1^4 \\ y_3 &= bz_1^4 z_3 + x_3(az_3 + y_1^2 + bz_1^4) \end{aligned} \quad (8)$$

3.4 A Comparison of the Projective Point Representations

Table 1 describes the computational requirements for each of the projective point methods. Note, if the EC parameter a is “sparse” and if it has to be multiplied to the field element t , then the time to compute at is not equivalent to a field multiplication of two arbitrary elements. In many sources, for example [14], they disregard the number of field multiplications performed with the EC parameters a and b , assuming that one can choose an elliptic curve with sparse parameters. However, in practice one may have to implement curves which have been defined in standards (although it maybe possible to make a transformation so that one or more of the transformed EC parameters is sparse). For example as the WAP/WTLS standard developed, there were initially only two strong elliptic curves defined over $GF(2^n)$, one a Koblitz curve (Curve 3) which has parameters $a = b = 1$, and the other a random curve (Curve 5) where both a and b are not sparse. And so to disregard field multiplications with EC parameters is not realistic. In Table 1 we have counted all field multiplications.

Table 1.

		no. of mult.	no. of squares
Homogeneous	Add	13	1
	Double	7	5
Jacobian	Add	11	4
	Double	5	5
Lopez& Dahab	Add	10	5
	Double	5	5

3.5 Are There More Efficient Projective Point Representations?

Of the three primary field operations in $GF(2^n)$ needed to perform projective point arithmetic: add, multiply and square; the field multiply is the time consuming operation. Although algebraically, a square is a field multiplication, as an implementation, the square can be performed very efficiently in $GF(2^n)$. For example, if you are using a normal basis, a square is a cyclic shift. If you are using a polynomial basis, then the square can be implemented by inserting 0's between terms (see [3,21]), then reducing. That is, if $\zeta = (\zeta_0, \dots, \zeta_{n-1}) \in GF(2^n)$ then $\zeta^2 = (\zeta_0, 0, \zeta_1, 0, \dots, \zeta_{n-1}, 0, \zeta_n)$. The representation of ζ^2 uses $n+n-1 = 2n-1$ terms, so a significant reduction needs to take place. But this can be achieved very efficiently.

Our interest in efficient projective point representations, led us to pursue alternate equations in an effort to reduce the time-consuming field multiplication. Consider the set of equations described by the Jacobian transformation, although they are algebraically efficient, the equations force an increase in the number of multiplications by the choice of an odd power. That is, to compute ζ raised to an odd power ensures the need of a multiplication, whereas, if one needs to compute ζ raised to 2, 4, 8, ... all that is required is a series of squares.

Consider the projective point to affine point transformation $x' = \frac{x}{z^\alpha}$ and $y' = \frac{y}{z^\beta}$. In the Jacobian, $\alpha = 2$ and $\beta = 3$. The odd power in the denominator for y' forces additional field multiplications. In the Homogeneous transformation, both α and β are 1. This is inefficient in that both of the affine equations that generate y' , Equation (2), include the term x' . Hence β should be greater than α . (This is why the Homogeneous representation has so many more field multiplications than the Jacobian.) It would appear that the “best” implementation of ECC using projective point arithmetic with fields of the form $GF(2^n)$ would satisfy α , β and $\beta - \alpha$ are powers of 2. A solution is to have $\alpha = 2^j$ and $\beta = 2^{j+1}$ for non-negative integer j . This incorporates the suggestion that $\beta > \alpha$ and the observation that the computation of z^{2^j} and $z^{2^{j+1}}$ can be done by performing a series of squares. Of course this selection criteria determines a family of projective point representations. It is trivial to show that the most efficient of this family of projective point representations is the case when $j = 0$, which is the Lopez & Dahab representation. Also note that the number of field multiplications³ for the case $j = 1$ is equal to the number of field multiplications for the case $j = 0$ (the case $j = 1$ will require one more square operation than the $j = 0$ case). This observation could have been generated solely from manipulating the Lopez & Dahab representation.

4 Efficiency Improvements

Clearly Table 1 shows that the most efficient projective point representation is the one developed by Lopez & Dahab, next the Jacobian, and last the Homogeneous projective point representation. However we have found that there can

³ Our count refers to the number of field operations required to perform an “add” and a “double” when you use this projective point representations.

be other factors that may influence performance, in addition to the number of field operations. Further, one must consider the type of curve that one is implementing. For example, if the curve is a Koblitz curve, then only the “add” must be gauged. Whereas, if the curve is a random curve then both the “double” and the “add” must be gauged. In this case, further factors that may play a role are whether the EC parameters a and/or b are sparse?

4.1 Create a Pipeline Effect—Reusing Constructed Lookup Table

A technique used to speed up computations is to pipeline common instructions together. Recall that our field multiplication is utilizing lookup tables to generate the product. To achieve a pipeline effect we examined all products and looked for similar multiplicands, allowing us to share lookup table for more than one multiply. We applied this technique to all three projective point methods. There are various strategies, the first strategy we employed was to allow at most one lookup table to be created in RAM at a time.

We illustrate the multiplications that would take place in the “add” of a Koblitz curve using the Lopez & Dahab method. Each rectangle represents a lookup table to be generated.

$z_1^2 \cdot y_2$	$z_1 \cdot x_2$	$\sqrt{z_3} \cdot (x_1 + z_1 x_2)^2$
	$z_1 \cdot (x_1 + z_1 x_2)$	$\sqrt{z_3} \cdot (y_1 + z_1^2 y_2)$
$z_3 \cdot x_2$	$(z_3 \cdot (y_1 + z_1^2 y_2)) \cdot (z_3 \cdot x_2 + x_3)$	
$z_3 \cdot y_2$		
$z_3 \cdot (x_3 + z_3 y_2)$		

The result is that the nine required field multiplications within the “add” can be arranged so that only 5 lookup tables are required. A word of warning, the existence of common multiplicands does not imply that a lookup table reduction can take place, for there may be a dependency between two multiplications. We have implemented a pipe which allowed us to add an output with a field element and take it as the next input (which is why we used the same lookup table for $z_3 \cdot y_2$ and $z_3(x_3 + z_3 \cdot y_2)$). All our piping was restricted to allowing at most one field addition.

In our implementation, we saw a dramatic improvement (in proportion to the previous benchmark) when we incorporated a pipelined multiplication (which minimizes the number of multiplication lookup tables created within the “add”). The following table highlights our benchmarks for the pipelined version of the Jacobian, Homogeneous, and Lopez & Dahab. We point out that the Homogeneous transformation performed slightly better than the Jacobian transformation (a reverse of what one may infer from Table 1). Within the Jacobian transformation we found many of the multiplications were dependent upon each other which limited our reduction of lookup tables.

Table 2. For a Koblitz curve

VERSION	Number of mult.	Compute kP nonpipeline	Lookup tables pipeline-add	Compute kP pipeline	Improvement
Lopez & Dahab	9	3.550ms	5	2.982ms	16%
Jacobian	10	3.926ms	7	3.525ms	10%
Homogeneous	12	4.240ms	6	3.486ms	18%

4.2 Applying the Pipelining to a Random Curve

We then applied the technique of pipelining multiplications with a common multiplicand in our implementation when using a random curve. To achieve a minimal amount of lookup tables, the piping choices made for a random curve will differ with the choices made for a Koblitz curve. For example, pipelines for lookup tables for the “add” and “double”, respectively, using the Lopez and Dahab implementation is such that the “add” required 5 lookup tables and the “double” required 4 lookup tables. (See the appendix for an illustration of how to construct a minimum number of lookup tables for the add and the double of a Random curve).

Table 3. For a Random curve

VERSION	Number of mult. add	Number of mult. double	Compute kP nonpipeline	Lookup tables pipeline add	Lookup tables pipeline double	Compute kP pipeline
L & D	10	5	7.987ms	5	4	6.908ms
Jacobian	11	5	8.276ms	8	5	7.910ms
Homog.	13	7	10.445 ms	6	4	8.259ms

As expected the Lopez & Dahab method performed the best. Again we see a dramatic improvement in the Homogeneous method when we pipeline like multiplicands together. Although it doesn’t in this case, out-perform the Jacobian method. When implementing all three methods on a Koblitz curve, the time to compute “ τ ” of a point is the same for all three methods. Thus the comparison of the three methods for a Koblitz curve is really a comparison of the “add” for each. Whereas in the implementation for the Random curve, the “double” is computed for each bit of the key, and so this comparison illustrates how “double” affects performance.

5 Other Strategies

Although we have achieved an implementation which requires a minimum number of lookup tables for both the “double” and “add”, there are further improve-

ments that can be achieved, however some may require more memory. That is, presently we have created a elliptic curve implementation over binary fields which will utilize lookup tables for field multiplications such that only one lookup table will exist in RAM at a time. We can improve on this implementation slightly at a cost of memory by allowing more than one lookup table to exist.

Choices we made earlier were based on the requirement that only one lookup table existed at a given time. Thus the RAM requirement for this implementation is the same as the RAM requirement for a nonpipelined implementation of the computation of kP . The pipeline was developed to meet this requirement and minimize the number of lookup tables generated. Suppose we relax this requirement and allow multiple lookup tables to exist at a given time. Observe that a number of field multiplications will contain an EC parameter as a multiplicand (i.e. a and/or b). Consequently, we can generate lookup tables for both a and b during the initial stage and save them throughout the computation. Normally a lookup table represent all 4 bit multiples of the field element. A field element in $GF(2^{163})$ requires at most 21 bytes. The size of a lookup table is at least 336 bytes. (In practice our lookup tables were such that each multiple was expressed using 6 words requiring 384 bytes). Because a table for a and/or b is such that it is used throughout the entire kP computation, the expected number of times a 4-bit multiple of a is used (say from the “add”) is heuristically $160 \cdot \frac{1}{2} \cdot \frac{1}{2^4}$ for each occurrence of a multiply of an a in an “add” and $160 \cdot \frac{1}{2^4}$ for each occurrence of a multiply of an a in the “double”. A similar analysis is true for multiples of b . Thus one finds an argument for increasing the window size from 4-bit to 6-bit or 8-bit, consequently we generated larger lookup tables for a and b , saved them and used these tables throughout the computation of kP .

Note that in the “add”, there exist products which have a multiplicand x_2 and/or y_2 , where P of the kP computation is such that $P = (x_2, y_2)$. Fortunately, we were not implementing any windowing. However, because we are working with a NAF form, we will utilize both P and $-P$. Consequently, there are two possible y_2 and one possible x_2 . Again we can generate lookup tables for x_2 and both y_2 of P and the y_2 of $-P$ during the initial stage, save them and use these tables throughout the computation of kP . (Do note that if one uses a b -bit window in their implementation, and opts for the strategy of generating a precomputed lookup table for y_2 and saving it throughout the computation, then it is required to compute and save $2^b - 1$ lookup tables.)

Lastly, observe that in the Lopez & Dahab method when we compute the “add”, a required multiplication is z_3x_3 (this is required to compute y_3). Further note that a required computation in the “double” is x_1z_1 . Of course in every case, but the case when we are processing the last bit of the key, a “double” will follow an “add”. Also, in every such case the product x_3z_3 in the “add” is the same as x_1z_1 in the following “double”. Consequently, if we save the product x_3z_3 from the “add” workspace and use it in the following “double”, then we reduce the number of field multiplications for the “double” by one. Note we are suggesting saving a computation from a workspace, thus if we are computing the “double” after a “double” we will not have this product saved and so we must

compute $x_1 z_1$. Consequently when we discuss the number of field multiplications for the “double”, we must use “expected number of field multiplications”. This “expected number” is of course correlated to the Hamming weight of the key. Perhaps a better way to incorporate this reduction of multiplications is to report this reduction for a multiplication as a reduction in the “add” (even though it occurs in the “double,” the following table reports this reduction as a reduction within the “add” function).

In the table that follows, we have included the before and after benchmark performance of our Lopez & Dahab implementation (here the after refers to the use of the above strategies).

One can opt for any or all of the RAM increasing pipelining strategies. However, it is redundant to try to utilize all three strategies to improve performance. In fact for a Koblitz curve, only strategy 2 is relevant (precomputing lookup tables for x_2 and/or y_2). For a Random curve, we suggest strategies 1 and 3 and to increase the lookup table window for parameters a and b . In the appendix we describe an implementation for an “add” and a “double” which utilizes only strategy 1. The following table describes an implementation of a Random curve utilizes strategies 1 and 3, and uses an increased window for the lookup tables for a and b .

Random Curve using RAM increasing pipelining strategies

VERSION	Number of mult. add/double	Lookup tables add/double	Compute kP pipeline	Compute kP improved pipeline
Lopez & Dahab	9 / 5	4 / 3	6.908 ms	6.342 ms

Thus by using the Lopez & Dahab method, pipelining, and using the RAM increasing pipeline strategies that we have outlined, we have reduced a scalar multiplication from the original time of 7.983 ms to 6.342 ms. (Note that in table above we have reduced the number of multiplies in an “add” by one due to the strategy of saving the computation of $x_3 z_3$ in an “add” and using it as the $x_1 z_1$ in the following “double”.)

6 A Suggestion on Squaring in $GF(2^m)$

Let $\mu \in GF(2^m)$, then $\mu = \mu_0 + \mu_1 x + \mu_2 x^2 + \cdots + \mu_{m-1} x^{m-1}$. We will also use $(\mu_0, \dots, \mu_{m-1})$ to represent μ . Further, if $\mu_i = 0$ for all $i > j$ then we may represent μ by $(\mu_0, \dots, \mu_j)$. Let $x^m + p(x)$ represent the generating polynomial of the field.

Several sources [21,3], have observed that a square can be computed by inserting 0’s between terms and reducing. That is, $\mu^2 = \mu_0 + \mu_1 x^2 + \mu_2 x^4 + \cdots +$

$\mu_{m-1}x^{2(m-1)} = (\mu_0, 0, \mu_1, 0, \mu_2, \dots, 0, \mu_{m-1}, 0)$. which needs to be reduced to compute the square. The required reduction can be inefficient, for it involves performing a series of shifts and adds. The shifts are determined by the terms of the generating polynomial, and they are performed on the terms of μ^2 . This inefficiency exists because every other term of μ^2 is zero, thus performing an xor (an add in $GF(2^m)$) with a shift can be a waste. For example if the shift was an even length, we will see that we are wastefully computing an xor of a 0 with a 0. If the shift is an odd length then we are performing an xor with a value and a zero, rather than simply resetting a term.

Our suggestion involves using the odd and even “parts” of a polynomial and performing the shifting and adds in place. That is, performing shifts and adds on either the odd or even “parts” of a polynomial.

Consider the calculation of μ^2 . Let $A = (\mu_0, 0, \mu_1, \dots, \mu_{\frac{m-1}{2}})$ and $B = (0, \mu_{\frac{m-1}{2}+1}, 0, \mu_{\frac{m-1}{2}+2}, 0, \dots, 0, \mu_{m-1})$, then both A and B are of degree $m-1$. Here A is an even polynomial and B is an odd polynomial. Note that $\mu^2 = A + x^m B$. So what remains is to reduce $x^m B$.

We will explain our algorithm using an example. Consider the field $GF(2^m)$ with generating polynomial $x^{163} + x^7 + x^6 + x^3 + 1$ (i.e. $m = 163$). So

$$x^m B = p(x)B = (x^7 + x^6 + x^3 + 1)B = x^7 B + x^6 B + x^3 B + 1B.$$

Since odd $\cdot$ odd is even and odd $\cdot$ even is odd, we see that $x^6 B$ and $1B$ are odd polynomials, and $x^7 B$ and $x^3 B$ are even polynomials. However, $x^6 B$, $x^7 B$, and $x^3 B$ are of degree greater than 162 (so they will require a multiplication with $p(x)$ but only for a few coefficients).

Next observe that A and B have alternating zeros, as well $x^6 B$, $x^7 B$, and $x^3 B$. We can efficiently encode this representation: encode A by $\mathcal{A} = (\mu_0, \mu_1, \dots, \mu_{\frac{n-1}{2}})_{EVEN} = (\mu_0, \mu_1, \dots, \mu_{81})_{EVEN}$ and encode B by $\mathcal{B} = (\mu_{\frac{n+1}{2}}, \dots, \mu_{n-1})_{ODD} = (\mu_{82}, \dots, \mu_{162})_{ODD}$. The subscript *EVEN* and *ODD* refer to whether the polynomials are even or odd. Also note that $\mathcal{A}$ and $\mathcal{B}$ have different lengths (although we may view them as the same length but that the coefficient of the leading term of $\mathcal{B}$ is 0). Multiplication of $\mathcal{B}$ by x^i is a right shift, introducing zeros on the left and possibly generating a polynomial of degree $> m-1$ (i.e. in this case $m-1 = 162$). If this latter case occurs then we will call it an “overflow”, the result is that these terms will need to be multiplied by $p(x)$. We will use SH_i to represent this right shift caused by multiplying by x^i . Then $xB = SH(\mathcal{B}) = (0, \mu_{82}, \mu_{83}, \dots, \mu_{162})_{EVEN}$. $x^2 B = (0, \mu_{82}, \mu_{83}, \dots, \mu_{161})_{ODD} + \mu_{162}p(x) = SH_2(\mathcal{B}) + \mu_{162}p(x)$. $x^3 B = (0, 0, \mu_{82}, \mu_{83}, \dots, \mu_{161})_{EVEN} + \mu_{162}xp(x) = SH_3(\mathcal{B}) + \mu_{162}xp(x)$, and so forth. Note $x^6 B = (0, 0, 0, 0, \mu_{82}, \mu_{83}, \dots, \mu_{159})_{ODD} + (\mu_{162}x^4 + \mu_{161}x^2 + \mu_{160})p(x) = SH_6(\mathcal{B}) + (\mu_{162}x^4 + \mu_{161}x^2 + \mu_{160})p(x)$, and $x^7 B = (0, 0, 0, 0, \mu_{82}, \mu_{83}, \dots, \mu_{159})_{EVEN} + (\mu_{162}x^4 + \mu_{161}x^2 + \mu_{160})p(x) = SH_7(\mathcal{B}) + (\mu_{162}x^5 + \mu_{161}x^3 + \mu_{160}x)p(x)$.

We can compute μ^2 by computing the even part, the odd part and the overflow part.

$$\text{even part} = \mathcal{A} + SH_3(\mathcal{B}) + SH_7(\mathcal{B})$$

$$\text{odd part} = \mathcal{B} + SH_6(\mathcal{B})$$

We write the overflow as:

$$overflow = p(x) (\mu_{162}(x^5 + x^4 + x + 1) + \mu_{161}(x^3 + x^2) + \mu_{160}(x + 1))$$

The *overflow* is not decomposed in odd and even form.

What remains is to transfer even part and odd part to correct form. If f represents some algorithm which takes as input even part and odd part polynomial and outputs the polynomial, then $\mu^2 = overflow + f(even\ part, odd\ part)$. This algorithm f is very similar to the algorithm which inserts zeros between terms, except it will insert odd coefficients between even coefficients.

Consider the algorithm to determine a square by the method of inserting zeros between terms and reducing. The reduction consists of a multiplication of the generating polynomial to the higher degree terms, and creates a series of xor (addition) operations (1 less than the number of terms in the generating polynomial). When performing the xor operation, one would perform the operation on the entire multi precision integer, however every other bit is zero. Thus one does not need to perform the xor operation on the entire multi precision integer, and if one is, then they are performing twice the number of bitwise-xor operations than what is really needed. Our algorithm computes the square without performing the wasted bitwise-xor operations (essentially it will halve the number of bitwise-xor operations performed). However, in our algorithm there still is a need to compute the polynomial given its even and odd parts (this is comparable to inserting zeros between terms).

In practice one should find that this algorithm will perform better than the method that inserts zeros and reduces. Although the improvement will be marginal for small field sizes.

7 Conclusion

Our work has provided a survey and comparison of several projective point representations. In addition we have provided strategies that lead to efficiency improvements, and these strategies may alter the comparison rankings of these projective point representations. We have provided further strategies that one can implement at a cost of memory. Lastly we have illustrated how to utilize the odd and even “parts” of a polynomial, to compute a square in $GF(2^n)$. The author wishes to thank the reviewers for their valuable and constructive suggestions.

References

1. Agnew, G., R. Mullin, S. Vanstone, “On the development of a fast elliptic curve processor chip”, In *Advances in Cryptology - Crypto '91*, Springer-Verlag, 1991, pp 482-487.
2. Steven Arno and Ferrell S. Wheeler. “Signed Digit Representations of Minimal Hamming Weight”. In *IEEE Transactions on Computers*. 42(8), 1993, p. 1007-1009.

3. I.F. Blake, Nigel Smart, and G. Seroussi, *Elliptic Curves in Cryptography*. London Mathematical Society Lecture Note Series. Cambridge University Press, 1999.
4. D. V. Chudnovsky and G. V. Chudnovsky. "Sequences of numbers generated by addition in formal groups and new primality and factorization tests." In *Adv. in Appl. Math.* 7 (1986) 385–434.
5. Darrel Hankerson, Julio Lopez Hernandez and Alfred Menezes. "Software Implementation of Elliptic Curve Cryptography over Binary Fields". In *CHES 2000*. p. 1-24.
6. M. A. Hasan, "Look-up Table Based Large Finite Field Multiplication in Memory Constrained Cryptosystems," In *Proceeding of the 7th IMA Conference on Cryptography and Coding*, Cirencester, UK, December 1999, LNCS, Springer Verlag, pp. 213-221.
7. M.A. Hasan. "Efficient computation of Multiplicative Inverses for Cryptographic Applications". In *CORR 2001-03* <http://cacr.math.uwaterloo.ca>.
8. T. Kobayashi, H. Morita, K. Kobayashi, and F. Hoshino. "Fast elliptic curve algorithm combining Frobenius map and table reference to adapt to higher characteristic", *Advances in Cryptology - Eurocrypt '99*, Springer-Verlag, 1992, pp 176-189.
9. Neal Koblitz, *Elliptic curve cryptosystems*, Mathematics of Computation, Vol. 48, No. 177, 1987, 203-209.
10. Neal Koblitz, *Algebraic Aspects of Cryptography, Algorithms and Computation in Mathematics*, Vol. 3, Springer-Verlag, New York, 1998.
11. Neal Koblitz, "CM-curves with good cryptographic properties", In *Advances in Cryptology - Crypto '91*, Springer-Verlag, 1992, 279-287.
12. Kenji Koyama and Yukio Tsuruoka. "Speeding up elliptic curve cryptosystems by using a signed binary window method", *Advances in Cryptology-Crypto'92*, Springer-Verlag, 1992, pp 345-357.
13. Erik Woodward Knudsen. "Elliptic Scalar Multiplication Using Point Halving". In *Advances in Cryptology - ASIACRYPT '99*. LNCS Vol. 1716, Springer, 1999, p. 135-149
14. Julio Lopez and Ricardo Dahab. "Improved Algorithms for Elliptic Curve Arithmetic in $GF(2^n)$ ". In *Selected Areas in Cryptography '98, SAC'98*. LNCS 1556, Springer, 1999, p. 201-212
15. J. Lopez and R. Dahab. "Performance of elliptic curve cryptosystems". In *Technical report, IC-00-08*. May 2000. Available at <http://www.dcc.unicamp.br/ic-main/publications-e.html>
16. Alfred Menezes, *Elliptic Curve Public Key Cryptosystems*, Kluwer Academic Publishers, 1993.
17. Victor S. Miller, "Use of Elliptic Curves in Cryptography", In *Advances in Cryptology CRYPTO 1985*, Springer-Verlag, New York, 1985, pp 417-42
18. Volker Muller. "Efficient Algorithms for Multiplication on Elliptic Curves", Proceedings *GI-Arbeitskonferenz Chipkarten 1998*, TU Munchen, 1998.
19. Volker Muller. "Fast Multiplication on Elliptic curves over small fields of characteristic two", *Journal of Cryptology*, 11(4), 219-234, 1998.
20. NIST, *Recommended elliptic curves for federal use*, <http://www.nist.gov>
21. Richard Schroepel, Hilarie Orman, Sean W. O'Malley, Oliver Spatscheck: "Fast Key Exchange with Elliptic Curve Systems", In *Advances in Cryptology - CRYPTO '95*, Lecture Notes in Computer Science, Vol. 963, Springer, 1995, pp 43-56.
22. Rich Schroepel. "Elliptic Curves: Twice as Fast!". In *Rump session of CRYPTO 2000*.

23. Jerome A. Solinas, "An Improved Algorithm for Arithmetic on a Family of Elliptic Curves" In *Advances in Cryptology - CRYPTO 1997*, 357-371 1997, pp 357-371.
24. H. Wu and M. A. Hasan. "Closed-Form Expression for the Average Weight of Signed-Digit Representation". In *IEEE Trans. Computers*. vol. 48, August 1999, p. 848-851.
25. *IEEE P1363 Appendix A*. <http://www.grouper.org/groups/1363>
26. *WTLS Specification*, <http://www.wapforum.org>

8 Appendix

Using the strategies discussed concerning precomputing and saving lookup tables, one has great flexibility in deriving the "add" and "double" formula. In the following, we provide an example of a pipelined "add" and a pipelined "double". We utilize strategy 1, and require precomputed lookup tables for EC parameter a and EC parameter b .

Random curve: Lookup tables – add

z_1x_2	$z_1^2y_2$	$z_1(x_1 + z_1x_2) \cdot (y_1 + z_1^2y_2)$
$z_1(x_1 + z_1x_2)$		$z_1(x_1 + z_1x_2) \cdot (x_1 + z_1x_2)^2$
z_3x_2	$[z_1(x_1 + z_1x_2) \cdot (y_1 + z_1^2y_2)](z_3x_2 + x_3)$	
$z_3(y_2 + a)$		
$z_3 \cdot (x_3 + z_3y_2)$		
z_3a		

Random curve: Lookup tables – double

z_1x_1	z_1^4b	$(y_1 + x_1^2)x_3$
$(z_1x_1) \cdot (bz_1^4)$		
$(z_1x_1) \cdot (z_1x_1(bz_1^4 + (y_1 + x_1^2)x_3)$		

Example of a pipelined add for a Random curve

(1) $D \leftarrow z_1^2$	(12) $z_3 \leftarrow z_3^2$
(2) make LT(z_1)	(13) $x_3 \leftarrow az_3$ NO LT
(3) $A \leftarrow z_1x_2$	(14) $x_3 \leftarrow x_3 + D$
(4) $z_3 \leftarrow z_1(A + x_1)$	(15) $x_3 \leftarrow x_3 + A + B$
(5) $B \leftarrow a \cdot D$ NO LT	(16) make LT(z_3)
(6) $B \leftarrow B + y_1$	(17) $x_3 \leftarrow z_3x_2$
(7) $A \leftarrow A^2$	(18) $B \leftarrow z_3y_2$
(8) make LT(z_3)	(19) $B \leftarrow S(B + x_3)$
(9) $D \leftarrow z_3B$	(20) $y_3 \leftarrow x_3 + y_3$
(10) $A \leftarrow z_3A$	(21) $y_3 \leftarrow y_3 \cdot D$
(11) $B \leftarrow B^2$	(22) $y_3 \leftarrow y_3 + B$

Example of a pipelined double for a random curve

(1) $B \leftarrow z_1^2$	(8) $C \leftarrow y_1^2$
(2) $A \leftarrow x_1^2$	(9) $C \leftarrow C + B$
(3) $z_3 \leftarrow A \cdot B$	(10) $y_3 \leftarrow az_3$ NO LT
(4) $B \leftarrow B^2$	(11) $y_3 \leftarrow y_3 + C$
(5) $B \leftarrow bB$ NO LT	(12) $y_3 \leftarrow y_3 \cdot x_3 B$
(6) $x_3 \leftarrow A^2$	(13) $C \leftarrow z_3 \cdot B$
(7) $x_3 \leftarrow x_3 + b$	(14) $y_3 \leftarrow y_3 + C$

As stated earlier each multiplication requires a lookup table. For those cases where we share lookup table we have indicated the creation of a lookup table (by $\text{Make LT}()$). Otherwise, the creation of the lookup table is generated within the multiplication algorithm. However, in line (5) we refer to NO LT. The implication is that no lookup table needs to be generated here. That is, b is a coefficient of the Weierstrass equation. One can create the lookup table for b at the start of the computation of kP , and save it.

8.1 How to Generate Alternate Projective Point Representations

We continue with the notation described in section 3.

Often there are more than one set of equations which describe $+$. Recall that $(x', y', 1) \sim (x, y, z)$ implies $x' = \frac{x}{z^\alpha}$ and $y' = \frac{y}{z^\beta}$. Fix positive integers α and β . Let $P = (x'_1, y'_1)$ and $Q = (x'_2, y'_2)$. Step one, replace x_i and y_i found in Equations (1) by the ratios $\frac{x_i}{z_i^\alpha}$ and $\frac{y_i}{z_i^\beta}$, respectively for $i=1,2$ (these equation determine x'_3). Step two, in Equations (2) replace x_i and y_i by the ratios $\frac{x_i}{z_i^\alpha}$ and $\frac{y_i}{z_i^\beta}$, respectively ($i=1,2$) and replace x_3 by the correct case of the two equations described in step one. The result will be, after simplification, four rational equations in the variables $x_1, y_1, z_1, x_2, y_2, z_2$, one set of equations for x'_3 (the case where $P \neq Q$ and the other case for $P = Q$) and another set of equations for y'_3 (one case for $P \neq Q$ and the other case for $P = Q$). To provide an example, choose z_3 to be the least common multiple of all four denominators (which is a polynomial in $x_1, y_1, z_1, x_2, y_2, z_2$). Set $x_3 = z_3^\alpha x'_3$, as α is a positive integer, the denominator will cancel out (in both cases $P \neq Q$ and $P = Q$), and that x_3 is equal to a polynomial in $x_1, y_1, z_1, x_2, y_2, z_2$. Set $y_3 = z_3^\beta y'_3$ and for a similar reason y_3 is equal to a polynomial in $x_1, y_1, z_1, x_2, y_2, z_2$. The result is four equations, two which describe x_3 and two which describe y_3 . The pair of equations that describe z_3 are identical (the least common multiple). For two projective points (x_1, y_1, z_1) and (x_2, y_2, z_2) , both not equal to the identity, we describe the equations that define $+$ as: $(x_1, y_1, z_1) + (x_2, y_2, z_2) = (x_3, y_3, z_3)$, where z_3 was chosen to be least common multiple of all four denominators, and where $x_3 = z_3^\alpha x'_3$ and $y_3 = z_3^\beta y'_3$. If one of the points was the origin, then the sum would be the other point. The goal was to describe the equations so that the inverse did not need to be employed. The result $+$ is a binary operation on the $\text{Image}(\phi(E))$ such that $\psi(\phi(P) + \phi(Q)) = P + Q$. This argument illustrates how to define $+$ on $\phi(E)$. Most important, we are able to achieve the addition without the use of a field inverse.