

SOFTWARE IMPLEMENTATION OF FINITE FIELDS OF CHARACTERISTIC THREE, FOR USE IN PAIRING-BASED CRYPTOSYSTEMS

K. HARRISON, D. PAGE AND N. P. SMART

Abstract

In this paper, the authors examine a number of ways of implementing characteristic three arithmetic for use in cryptosystems based on the Tate pairing. Three alternative representations of the field elements are examined, and the resulting algorithms for the field addition, multiplication and cubing are compared. Issues related to the arithmetic of supersingular elliptic curves over fields of characteristic three are also examined. Details of how to compute the Tate pairing itself are not covered, since these are well documented elsewhere.

1. *Introduction*

Much interest has recently been created by the use of the Weil/Tate pairings on supersingular elliptic curves to produce identity-based cryptographic schemes, such as the encryption scheme of Boneh and Franklin [4]. As was pointed out by Galbraith [7], in terms of bandwidth efficiency it is more efficient to use supersingular elliptic curves in characteristic three for systems based on the Weil/Tate pairing. This is in contradiction to the standard advice in elliptic curve cryptography, where one uses fields of either large prime characteristic or characteristic two. Hence, very little work has been conducted into implementation issues related to elliptic curves in characteristic three. In this paper we address this gap in the literature, paying particular attention to parameters usable in cryptographic systems based on the Weil pairing, such as the encryption algorithm of Boneh and Franklin mentioned above. We do not concentrate on the actual implementation of the Weil/Tate pairings, since that is covered fully in other papers such as [2] and [8].

Originally, the existence of the Tate and Weil pairings was thought to be a bad thing in cryptography. For example, in [11] it was shown that the discrete logarithm problem in supersingular curves was reducible to that in a finite field using the Weil pairing. This led supersingular elliptic curves to be dropped from cryptographic use. The situation changed with the work of Joux [9], who gave a simple tripartite Diffie–Hellman protocol based on the Weil pairing on supersingular curves. Since Joux’s paper, a number of other applications have arisen, including an identity-based encryption scheme [4] and a general signature algorithm [5]. In [13], an ID-based public key signature algorithm is given, which uses the Weil pairing. In addition, the extension to higher-genus curves has also recently been fully explored in [7]. This new work has resulted in a rekindling of cryptographic interest in supersingular elliptic curves. Although most of the literature discusses these schemes in terms of the Weil pairing, it turns out that it is far more efficient to use the Tate pairing.

Received 12 February 2002, revised 14 August 2002; published 25 November 2002.

2000 Mathematics Subject Classification 68R01, 11T71

© 2002, K. Harrison, D. Page and N. P. Smart

Table 1: Parameters in characteristic two and three

Field	Curve	ECDHP security	MOV security
\mathbb{F}_{397}	$y^2 = x^3 - x + 1$	151	922
$\mathbb{F}_{2^{241}}$	$y^2 + y = x^3 + x + 1$	241	964

We let \mathbb{G} denote a prime-order subgroup of an elliptic curve E over the field \mathbb{F}_q , which for the moment we assume is a general finite field of arbitrary characteristic. Let the order of \mathbb{G} be denoted by l , and define α to be the smallest integer such that

$$l|q^\alpha - 1.$$

In pairing-based cryptosystems we require α to be small (but not too small), and so we shall usually take E to be a supersingular curve over \mathbb{F}_q .

The security of pairing-based cryptosystems is based on two problems.

- *The computational elliptic curve Diffie–Hellman problem (ECDHP) in \mathbb{G} :*
Here, the relevant security parameter is $\log_2 l$, and the currently adopted practice is to choose $\log_2 l \approx 160$.
- *The finite field Diffie–Hellman problem in $\mathbb{F}_{q^\alpha}^*$ (usually called the MOV security):*
Here, the relevant security parameter is $\alpha \log_2 q$, and the currently adopted practice is to choose $\alpha \log_2 q \approx 1024$.

Note that the decision Diffie–Hellman problem on supersingular elliptic curves is easy, due to the existence of the Weil and Tate pairings, as was first pointed out by Joux [9].

For supersingular elliptic curves, the value of α is bounded by four in characteristic two, by six in characteristic three, and by two for curves defined over large prime fields. As Galbraith [7] pointed out, the bandwidth performance of the schemes based on the Weil pairing usually grows with $\log_2 q$ rather than with $\alpha \cdot \log_2 q$, and hence it is better to try to minimize q . This leads us to wish to consider fields of characteristic three, since this aids us in minimizing the value of q .

If we wish to deploy a system with security roughly equivalent to 1024-bit RSA or 160-bit ECC, then we are led to consider the parameters in characteristic two and three shown in Table 1. We shall consider these parameters when describing our implementation of characteristic three arithmetic below, using the above field of characteristic two as a benchmark.

2. Implementation of arithmetic in \mathbb{F}_{3^n}

Let our finite field be given by

$$\mathbb{F}_{3^n} = \mathbb{F}_3[x]/(f(x)),$$

where $f(x)$ is a suitably chosen irreducible polynomial in $\mathbb{F}_3[x]$, of degree n .

To make reduction modulo $f(x)$ as simple as possible, we choose if possible an irreducible trinomial

$$f(x) = x^n + ax^k + b,$$

where $a, b \in \{1, 2\}$. The reduction algorithm works much like that used in implementations of characteristic two arithmetic, so we shall not discuss it further.

A simple search reveals that for most prime values of n , such a trinomial exists. Of particular interest for cryptography, given the possible supersingular elliptic curves in characteristic three with nearly prime group order, are the following trinomials:

$$\begin{aligned} x^{79} + x^{21} + 2, \\ x^{97} + x^{12} + 2, \\ x^{163} + x^{80} + 2, \\ x^{167} + x^{97} + 2, \\ x^{173} + x^{166} + 2, \\ x^{193} + x^{12} + 2, \\ x^{239} + x^{24} + 2. \end{aligned}$$

In the next two sections we look at two possible ways of implementing polynomial arithmetic modulo 3, but first we examine how inverses are computed in \mathbb{F}_{3^n} . The same algorithm was used in all of our implementations, namely a form of ternary extended Euclidean algorithm. This is a natural extension of the standard binary Euclidean algorithm [10], pseudo-code for which we present in [Appendix A](#).

3. Polynomial arithmetic modulo three: Type I

3.1. Addition

A trivial first observation is that if we hold our integers modulo three as two bits in a redundant way, by having two representatives for the zero residue class, testing whether a reduction modulo 3 needs to be carried out after an addition is very simple. (We use standard C syntax; in particular, the operator '>>' denotes a right shift, so 'c >> 2' evaluates to true (a non-zero value) if and only if c is greater than three.) Thus we denote:

```
c=a+b;
if (c>>2) { c-=3; }
```

as opposed to

```
c=a+b;
if (c>=3) { c-=3; }
```

as would be the case if we used a standard representation. This does not look like a great performance increase at first sight, since comparisons and bit shifts are usually single instructions on modern processors. Hence the increased complexity of having a redundant representation does not seem to have bought us any performance increase.

However, if we now 'pack' more than one element of \mathbb{F}_3 into a word, then we obtain the ability to add elements of \mathbb{F}_3 in parallel. Suppose that we use three bits to specify each integer, of which the highest-order bit is always set to zero. This means that we can pack ten elements of \mathbb{F}_3 into a single 32-bit word. Parallel addition of ten elements of \mathbb{F}_3 can then be performed via the following C-code.

```
c=a+b;
r=(c>>2)&0x9249249;
c=c-3*r;
```

So in five RISC machine instructions we can perform ten additions modulo three. Since we are interested in multiplication as well as addition, it will be convenient to represent each element of \mathbb{F}_3 in eight bits, and hence in practice we shall pack only four elements of \mathbb{F}_3 into each 32-bit word. The reasons for this choice are outlined in the next subsection.

3.2. Parallel multiplication

Suppose that we have a redundant representation as above, where each element of \mathbb{F}_3 is held as an element from the set $\{0, 1, 2, 3\}$. When multiplying two such elements we shall produce a value at most nine, and so a method needs to be developed to produce a result back in our set of representatives. Luckily, Montgomery reduction [12] allows us to do this with very simple operations. Recall that usually for arithmetic modulo p one chooses a value of $R = 2^t$ which is a power of two and greater than p . Then Montgomery reduction, given a value $x < pR$, will produce the value of

$$x \cdot R^{-1} \pmod{p}$$

using the following operations:

```

u=(x*q)&(R-1);
x=x+p*u;
x=x>>t;
if (x>=p) { x=x-p; }

```

where $q = -1/p \pmod{R}$. When we specialise this for $p = 3$ and choose $R = 4$, then we find that $q = 1$, and as $R \equiv 1 \pmod{3}$, we find that Montgomery reduction is equal to modular reduction in this case. Combining our redundant representation with Montgomery reduction, we find that the following pseudo-code will multiply two elements of \mathbb{F}_3 held in two integer variables a and b .

```

c=a*b;
u=c&3;
c=c+3*u;
c=c>>2;

```

A simple evaluation of all the possible cases implies that we do not need to perform a final reduction as in the standard Montgomery algorithm, due to our redundant representation. However, the above is only a simple multiplier, which does not work well with our parallel addition technique given earlier. To see how to multiply in parallel, we need to consider that we are not implementing a naive modulo three multiplier, but require multiplication of polynomials over \mathbb{F}_3 .

Recall that we said that we would pack four elements of \mathbb{F}_3 into a word; we place each element on a byte boundary. We now treat this word as representing a polynomial of degree three:

$$a = a_3X^3 + a_2X^2 + a_1X + a_0,$$

where a_0 is the least significant byte and a_3 is the most significant byte. Given another such polynomial

$$b = b_3X^3 + b_2X^2 + b_1X + b_0,$$

where $a_i, b_i \in \{0, 1, 2, 3\}$, we find that (as polynomials)

$$\begin{aligned}
 a \cdot b &= a_3b_3X^6 + (a_3b_2 + a_2b_3)X^5 + (a_3b_1 + a_2b_2 + a_1b_3)X^4 \\
 &\quad + (a_3b_0 + a_2b_1 + a_1b_2 + a_0b_3)X^3 + (a_2b_0 + a_1b_1 + a_0b_2)X^2 \\
 &\quad + (a_1b_0 + a_0b_1)X + a_0b_0.
 \end{aligned}$$

Hence the polynomial multiplication of a and b , as integer polynomials, corresponds to the integer multiplication of the two words a and b as 32-bit integers, producing a 64-bit result. This is because the above equation will hold if we replace X by 2^8 , since each coefficient in the multiplication of the polynomials over the integers will be at most

$$4 \cdot 3 \cdot 3 = 36,$$

since each a_i or b_i is at most three. So whilst this allows the parallel implementation of multiplication of integer polynomials of degree three, we are still left with the problem of how to perform reduction modulo three on the resulting degree-six polynomial.

We adopt our Montgomery trick as above, but this time using a value of R equal to 16. This means that Montgomery reduction will take a value x less than $p \cdot R = 48$, which is larger than 36, and will produce the value $x \cdot R^{-1} = x \pmod{3}$. But as $R = 1 \pmod{3}$ this produces exactly the required modular reduction. The pseudo-code to do this is given below.

```
u = (5 * x) & 0x0F;
x = x + 3 * u;
x = x >> 4;
t = x >> 2;
x = x - 3 * t;
```

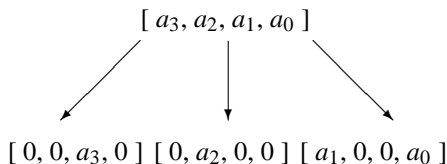
We see that the above code on input of a value of x less than 37 produces no intermediate value greater than 255. Hence the above code can be used in parallel on our 32-bit words resulting from the integer multiply instruction. This requires the following pseudo-code.

```
u = (5 * x) & (0x0F0F0F0F);
t = x + 3 * u;
x = t >> 4;
h = (x >> 2) & (0x01010101);
x = x - h * 3;
```

The above code for Montgomery reduction of a 32-bit word holding four possible values takes around nine machine instructions, and we need to perform two of these, plus one integer multiply instruction to produce the initial degree-six polynomial. Hence we require around 19 machine instructions to perform the multiplication of two polynomials of degree three over \mathbb{F}_3 .

Multiplication of arbitrary-degree polynomials can now be built up from this atomic operation, using school-book arithmetic, although it is often more efficient to use Karatsuba multiplication in a standard way.

Cubing in characteristic three is particularly efficient. Just as with squaring in characteristic two [3], one can ‘thin out’ the coefficients when cubing. The four coefficients held in one word are expanded into three words in the following manner.



This is done for all words representing the field element; a reduction operation is then performed modulo the polynomial $f(x)$.

4. Polynomial arithmetic modulo three: Type II

As a second representation of polynomial arithmetic modulo three, we consider the following idea. Each set of 32 polynomial coefficients is held in two 32-bit words, which we shall denote by w_1 and w_2 . A given bit in w_1 is set if the corresponding coefficient of the polynomial is equal to one, whilst if the given bit in w_2 is set, then the coefficient of the polynomial is equal to two. If both bits are clear, then the coefficient is zero, whilst the case of both bits set is considered invalid.

4.1. Addition

Addition of polynomials is done on a word-by-word basis, using the following seven machine instructions, which add the degree-31 polynomial represented by the words (a_1, a_2) to the degree-31 polynomial represented by the words (b_1, b_2) , to produce the result (c_1, c_2) .

$$\begin{aligned} t &= (a_1 \mid b_2) \wedge (a_2 \mid b_1); \\ c_1 &= (a_2 \mid b_2) \wedge t; \\ c_2 &= (a_1 \mid b_1) \wedge t; \end{aligned}$$

One should note, however, that the order in which the seven instructions are used produces a big impact on efficiency, especially for processors such as the Pentium, which have a small number of internal registers. Notice that negation in this representation is particularly easy, since

$$-(a_1, a_2) = (a_2, a_1).$$

4.2. Multiplication

The natural way to multiply elements in this representation is in a bit-serial manner. In this method, we take two operands and perform a multiply by repeatedly shifting the multiplier down by one bit position and shifting the multiplicand up by one bit position. The multiplicand is then added or subtracted from the output value, on each iteration, depending on whether the least significant bit of the first or second word of the multiplier is set to one.

There are then three ways of building up multiplication for \mathbb{F}_{3^p} . In the first two, we use a bit-serial multiplier to multiply two degree-31 polynomials together, and then use Karatsuba or school-book multiplication to perform the full multiply. After a full multiplication has been computed, we can then reduce the result modulo $f(x)$. In the third way of producing a full multiplier, we apply the bit-serial method to all p bits at once, taking care to reduce the multiplicand by $f(x)$ on each iteration.

The advantage of this full bit-serial technique is that it requires less intermediate storage, and is more suited to a hardware implementation, a topic that we shall return to in a later paper. However, a major disadvantage of the full bit-serial multiplier is that an analogous cubing operation is only as fast as a general multiply. With the other two methods, which reduce to 32-bit bit-serial multiplications, one can produce a more efficient cubing operation than a general multiply. If memory usage is not a concern, then the use of look-up tables to implement the cubing operation is the most efficient method.

5. Implementation of arithmetic in \mathbb{F}_{36p}

In performing the calculations in cryptosystems based on the Tate pairing for supersingular elliptic curves in characteristic three, we are not only required to perform some

Table 2: Timings in micro-seconds for field operations

Field	Addition	Multiplication	Cubing	Squaring	Inversion
$\mathbb{F}_{3^{97}}\text{-N}$	2.70	830	37	419	1200
$\mathbb{F}_{3^{97}}\text{-K}$	2.70	415	37	309	1200
$\mathbb{F}_{3^{97}}\text{-I-N}$	1.10	84	4	49	680
$\mathbb{F}_{3^{97}}\text{-I-K}$	1.10	45	4	44	680
$\mathbb{F}_{3^{97}}\text{-II-N}$	0.20	23	10 or 1.5	16	240
$\mathbb{F}_{3^{97}}\text{-II-K}$	0.20	15	10 or 1.5	16	240
$\mathbb{F}_{3^{97}}\text{-II-BS}$	0.20	27	29	32	300
$\mathbb{F}_{2^{241}}$	0.01	13	14	0.4	≈ 0

operations in \mathbb{F}_{3^p} , but we also need to compute in the extension $\mathbb{F}_{3^{6p}}$. Since, in applications, p is a prime greater than 5, we can use the following representation of the finite field $\mathbb{F}_{3^{6p}}$:

$$\mathbb{F}_{3^{6p}} = \mathbb{F}_{3^p}[\theta]/(\theta^6 + \theta + 2).$$

This provides an efficient reduction operation for the multiplication operations. The multiplication operation itself is performed using a variant of the Karatsuba method, where we first divide the degree-five polynomial into two, as in the standard Karatsuba method. Then the products of the degree-two polynomials are computed, using a variant of Karatsuba where one divides the polynomials into three. This is exactly the same as the method described in [1] for optimal extension fields of degree six.

Once again, cubing can be performed efficiently; this time, we not only need to thin out the coefficients, but we also need to cube the thinned-out coefficients in \mathbb{F}_{3^p} .

6. Timing of field operations

We obtained the timings (in micro-seconds) given in Table 2 on a Sparc Ultra 10 computer. We also give the timings in characteristic two for fields that would result in Tate-pairing-based systems with a similar security. In the table, the fields are denoted as follows.

- $\mathbb{F}_{3^{97}}\text{-N}$ corresponds to an implementation using the standard technique of representing each element in 3^{97} as an array of 97 integers, where arithmetic is performed using a naive multiplication algorithm; $\mathbb{F}_{3^{97}}\text{-K}$ corresponds to a similar implementation, where arithmetic is performed using a Karatsuba multiplication algorithm.
- $\mathbb{F}_{3^{97}}\text{-I-N}$ corresponds to an implementation using Type I arithmetic and a naive multiplication algorithm, and $\mathbb{F}_{3^{97}}\text{-I-K}$ corresponds to an implementation using Type I arithmetic and a Karatsuba multiplication algorithm.
- $\mathbb{F}_{3^{97}}\text{-II-N}$, $\mathbb{F}_{3^{97}}\text{-II-K}$ and $\mathbb{F}_{3^{97}}\text{-II-BS}$ refer to our Type II arithmetic. The notation $\mathbb{F}_{3^{97}}\text{-II-N}$ refers to a word-based implementation with school-book multiplication, whilst $\mathbb{F}_{3^{97}}\text{-II-BS}$ refers to a full bit-serial multiplier. The notation $\mathbb{F}_{3^{97}}\text{-II-K}$ refers to a word-based implementation with Karatsuba multiplication. Where we give two timings for cubing, these are based on an implementation of cubing using either reduction to a 32-bit bit serial cubing operation, or reduction to lookup tables.

One can see that if memory is a concern and lookup tables cannot be used, then the Type I arithmetic has a much faster cubing operation than the Type II arithmetic. This is important for cryptosystems based on the Weil and Tate pairings in characteristic three, which make use of a large number of cubing operations. If memory is not a concern, then clearly the Type II arithmetic is more efficient.

We turn now to the timings of the extension field arithmetic. The timings for our target field, and the corresponding field of characteristic two of similar security parameters, are given in Table 3. The timings, in micro-seconds, for the Type II field arithmetic assume the use of lookup tables for the cubing operation.

Table 3: Timings in micro-seconds for extension field arithmetic

Field	Base field	Addition	Multiplication	Cubing	Squaring	Inversion
$\mathbb{F}_{3^{6\cdot 97}}$	Type K	17	7772	281	5674	117000
$\mathbb{F}_{3^{6\cdot 97}}$	Type I-K	7	904	49	868	18000
$\mathbb{F}_{3^{6\cdot 97}}$	Type II-K	1	319	24	327	6000
$\mathbb{F}_{2^{4\cdot 241}}$	–	0.3	126	130	2	1000

7. Implementation of the curve operations

Due to the high relative cost of inversion compared to multiplication in fields of characteristic three, we implemented a projective addition law on the elliptic curve; however, we shall see that using an affine representation is also efficient. Hence we shall compare and describe both representations.

Since cubing an element in \mathbb{F}_{3^n} is more efficient than squaring or multiplying two elements, we expressed the group law in such a way as to maximise the number of cubing operations. This also led us to examine the equations for tripling (as well as doubling and adding) a point, since that is also an efficient operation in characteristic three for supersingular curves.

Recall that in this paper we have restricted ourselves to considering only supersingular elliptic curves, in which case it turns out that the most efficient projective coordinates are given by $(x, y) = (X/Z^2, Y/Z^3)$. Hence the projective representation that we consider is of the form

$$Y^2 = X^3 - XZ^4 \pm Z^6,$$

which is the supersingular elliptic curve with the most interesting properties (in terms of the MOV parameter) in characteristic three.

7.1. Point addition

The affine addition of two points (x_1, y_1) and (x_2, y_2) on the curve can then be expressed in the following manner, where we keep track of the number of multiplications, cubings

and inversions in the field:

$$\begin{aligned}
 \lambda_1 &= x_2 - x_1 \\
 \lambda_2 &= y_2 - y_1 \\
 \lambda_3 &= \lambda_2^{-1} && \text{II} \\
 \lambda_4 &= \lambda_1 \cdot \lambda_3 && \text{1M} \\
 \lambda_5 &= \lambda_4^2 && \text{1M} \\
 x_3 &= \lambda_5 - x_1 - x_2 \\
 y_3 &= \lambda_4 \cdot (x_1 - x_3) - y_1 && \text{1M.}
 \end{aligned}$$

Hence we require three multiplications and one inversion to implement a point addition in affine coordinates.

The projective addition of two points, (x_1, y_1, z_1) and (x_2, y_2, z_2) , on the curve can then be expressed in the following manner, where we now keep track of the number of multiplications and cubings in the field:

$$\begin{aligned}
 \lambda_1 &= x_1 \cdot z_2^2 && \text{2M} \\
 \lambda_2 &= x_2 \cdot z_1^2 && \text{2M} \\
 \lambda_3 &= \lambda_1 - \lambda_2 \\
 \lambda_4 &= y_1 \cdot z_2^3 && \text{1M 1C} \\
 \lambda_5 &= y_2 \cdot z_1^3 && \text{1M 1C} \\
 \lambda_6 &= \lambda_4 - \lambda_5 \\
 \lambda_7 &= \lambda_1 + \lambda_2 \\
 \lambda_8 &= \lambda_4 + \lambda_5 \\
 z_3 &= z_1 \cdot z_2 \cdot \lambda_3 && \text{2M} \\
 x_3 &= \lambda_6^2 - \lambda_7 \cdot \lambda_3^2 && \text{3M} \\
 y_3 &= (\lambda_8 \cdot \lambda_3^3 - \lambda_6^3) && \text{1M 2C.}
 \end{aligned}$$

Hence we require twelve multiplications and four cubings to implement a point addition, as opposed to the sixteen multiplications usually required for fields of characteristic greater than two, or the fifteen multiplications and five squarings required in fields of characteristic two [3].

7.2. Point doubling

Doubling of a point (x_1, y_1) in affine coordinates is implemented using the following formulae:

$$\begin{aligned}
 \lambda_1 &= y_1^{-1} && \text{II} \\
 \lambda_2 &= \lambda_1^2 && \text{1M} \\
 x_3 &= \lambda_2 + x_1 \\
 y_3 &= \lambda_4 \cdot (x_1 - x_3) - y_1 && \text{1M.}
 \end{aligned}$$

Hence we require two multiplications, one inversion and no cubings to double a point in affine coordinates.

Doubling of a point (x_1, y_1, z_1) in projective coordinates is implemented using the following formulae:

$$\begin{aligned} \lambda_1 &= -z_1^4 && 1\text{M} \quad 1\text{C} \\ z_3 &= -y_1 \cdot z_1 && 1\text{M} \\ \lambda_2 &= x_1 \cdot y_1^2 && 2\text{M} \\ x_3 &= \lambda_1^2 + \lambda_2 && 1\text{M} \\ \lambda_3 &= -y_1^4 && 1\text{M} \quad 1\text{C} \\ y_3 &= \lambda_1 \cdot (\lambda_2 - x_3) - \lambda_3 && 1\text{M}. \end{aligned}$$

Hence this requires seven multiplications and two cubings. One should compare this to the ten multiplications usually required in a field of characteristic greater than three, and the five multiplications and five squarings required in fields of characteristic two.

7.3. Point tripling

As we pointed out earlier, tripling a point can also be performed efficiently in characteristic three for supersingular elliptic curves. In affine coordinates we obtain

$$\begin{aligned} \lambda_1 &= y_1^3 && 1\text{C} \\ \lambda_2 &= x_1^3 && 1\text{C} \\ \lambda_3 &= \lambda_2^3 && 1\text{C} \\ y_3 &= -\lambda_1^3 && 1\text{C} \\ x_3 &= \lambda_3 \mp 1. \end{aligned}$$

Hence we can triple a point, using no inversions and four cubings in affine coordinates.

In projective coordinates, we obtain

$$\begin{aligned} \lambda_1 &= y_1^3 && 1\text{C} \\ \lambda_2 &= z_1^3 && 1\text{C} \\ \lambda_3 &= x_1^3 && 1\text{C} \\ z_3 &= \lambda_2^3 && 1\text{C} \\ y_3 &= -\lambda_1^3 && 1\text{C} \\ \lambda_4 &= z_3^2 && 1\text{M} \\ \lambda_5 &= \lambda_3^3 && 1\text{C} \\ x_3 &= \lambda_5 \mp \lambda_4. \end{aligned}$$

Hence, tripling requires one multiplication and six cubings.

7.4. Timings

Given the doubling and tripling formulae, we can give binary, ternary and nonary multiplication algorithms. The advantage of the ternary and nonary algorithms is that not only do they use the point-tripling formulae, which is more efficient than the point-doubling formulae, but the ternary and nonary expansions of the multiplicand are shorter than the binary expansion of the same multiplicand.

We then obtain the timings shown in Table 4, where we compare with a curve of similar security in characteristic two, all given in milli-seconds.

Table 4: Timings in milli-seconds for curve operations

Operation	$\mathbb{F}_{3^{97}}\text{-K}$	$\mathbb{F}_{3^{97}}\text{-I-K}$	$\mathbb{F}_{3^{97}}\text{-II-K}$	$\mathbb{F}_{2^{241}}$
Affine coordinates:				
Point addition	2.04	0.91	0.29	0.23
Point doubling	1.60	0.84	0.27	0.23
Point tripling	0.17	0.06	0.01	0.46
Point multiplication (binary)	395	197	62	65
Point multiplication (ternary)	145	65	19	100
Point multiplication (nonary)	116	51	15	80
Projective coordinates:				
Point addition	4.63	0.64	0.21	0.18
Point doubling	2.74	0.37	0.12	0.06
Point tripling	0.52	0.13	0.03	0.24
Point multiplication (binary)	749	104	33	23
Point multiplication (ternary)	343	53	16	55
Point multiplication (nonary)	277	44	12	48

Table 5: Timings in milli-seconds for the Tate pairing

Operation	$\mathbb{F}_{3^{97}}\text{-II-K}$	$\mathbb{F}_{2^{241}}$
Tate pairing	63	61

The projective multiplications also use a mixed coordinate system, as is standard practice [6]. Signed window analogues, see [3], of both the ternary and nonary methods can be implemented to give an improvement in performance, although not as great an improvement as for the binary method. It is interesting to note that for the naive implementation of $\mathbb{F}_{3^{97}}$ arithmetic we see that affine coordinates are more efficient for general curve arithmetic, but for the other implementations projective coordinates are more efficient. It should be noted that cryptosystems based on Tate pairings also require general curve operations, as well as pairing computations, and hence general point-multiplication times are important.

We do not discuss how to implement the Tate pairing, since this is explained fully in [2] and [8]. We do, however, obtain the timings (in milli-seconds) shown in Table 5, where we give a comparable timing for a system based on characteristic two arithmetic. We give a timing only for the field implementation that we have denoted $\mathbb{F}_{3^{97}}\text{-II-K}$.

8. Conclusion

We have shown how the novel use of 32-bit integer arithmetic can result in an implementation of characteristic three arithmetic suitable for use in cryptosystems based on the Tate pairing. The use of characteristic three with the Tate pairing is preferred, due to the improved bandwidth considerations implied by the security parameters. We have seen that

one of our implementation techniques provides a fast cubing operation without the use of lookup tables, whilst the other provides an efficient multiplication routine but a less efficient cubing operation unless one is prepared to use lookup tables. Both implementation techniques offer a considerable improvement over the standard techniques based on using an array of integers to hold the coefficients of the field elements.

Appendix A. *Ternary Euclidean algorithm*

The following analogue of the binary extended Euclidean algorithm will find the inverse D of the polynomial b with respect to the defining polynomial f ; that is,

$$D = b^{-1} \pmod{f},$$

where $D, b, f \in \mathbb{F}_3[x]$.

```

a=f;
B=0;
D=1;
while (a!=0)
  { while (tc(a)==0)
    { if (tc(B)!=0)
      { if (tc(B)==tc(f)) { B=B-f; }
        else               { B=B+f; }
      }
      a=a/x;  B=B/x;
    }
    while (tc(b)==0)
      { if (tc(D)!=0)
        { if (tc(D)==tc(f)) { D=D-f; }
          else               { D=D+f; }
        }
        b=b/x;  D=D/x;
      }
    if (deg(a)>=deg(b))
      { if (tc(a)==tc(b))
        { a=a-b;  B=B-D; }
        else
        { a=a+b;  B=B+D; }
      }
    else
      { if (tc(a)==tc(b))
        { b=b-a;  D=D-B; }
        else
        { b=b+a;  D=D+B; }
      }
  }
if (b!=1) { D=-D; }

```

In the above pseudo-code we assume that $tc(f)$ refers to the trailing coefficient of f , namely the coefficient of x^0 .

References

1. D. BAILEY and C. PAAR, 'Efficient arithmetic in finite field extensions with application in elliptic curve cryptography', *J. Cryptology* 14 (2001) 153–176. [187](#)
2. P. S. L. M. BARRETO, H. Y. KIM and M. SCOTT, 'Efficient algorithms for pairing-based cryptosystems', *Advances in Cryptology – CRYPTO 2002*, Lecture Notes in Comput. Sci. 2442 (Springer, 2002) 354–368. [181](#), [191](#)
3. I. F. BLAKE, G. SEROUSSI and N. P. SMART, *Elliptic curves in cryptography* (Cambridge University Press, 1999). [185](#), [189](#), [191](#)
4. D. BONEH and M. FRANKLIN, 'Identity-based encryption from the Weil pairing', *Advances in Cryptology – CRYPTO 2001* Lecture Notes in Comput. Sci. 2139 (Springer, 2001) 213–229. [181](#), [181](#)
5. D. BONEH, B. LYNN and H. SHACHAM, 'Short signatures from the Weil pairing', *Advances in Cryptology – ASIACRYPT 2001*, Lecture Notes in Comput. Sci. 2248 (Springer, 2001) 514–532. [181](#)
6. H. COHEN, A. MIYAJI and T. ONO, 'Efficient elliptic curve exponentiation using mixed coordinates', *Advances in Cryptology – ASIACRYPT '98*, Lecture Notes in Comput. Sci. 1514 (Springer, 1998) 51–65. [191](#)
7. S. D. GALBRAITH, 'Supersingular curves in cryptography', *Advances in Cryptology – ASIACRYPT 2001*, Lecture Notes in Comput. Sci. 2248 (Springer, 2001) 495–513. [181](#), [181](#), [182](#)
8. S. D. GALBRAITH, K. HARRISON and D. SOLDERA, 'Implementing the Tate pairing', *Algorithmic number theory – ANTS-V*, Lecture Notes in Comput. Sci. 2369 (Springer, 2002) 324–337. [181](#), [191](#)
9. A. JOUX, 'A one round protocol for tripartite Diffie–Hellman', *Algorithmic number theory – ANTS-IV*, Lecture Notes in Comput. Sci. 1838 (Springer, 2000) 385–394. [181](#), [182](#)
10. D. E. KNUTH, *The art of computer programming. 2 – Semi-numerical algorithms*, 2nd edn (Addison-Wesley, 1981). [183](#)
11. A. J. MENEZES, T. OKAMOTO and S. VANSTONE, 'Reducing elliptic curve logarithms to logarithms in a finite field', *IEEE Trans. Inform. Th.* 39 (1993) 1639–1646. [181](#)
12. P. L. MONTGOMERY, 'Modular multiplication without trial division', *Math. Comp.* 44 (1985) 519–521. [184](#)
13. R. SAKAI, K. OHGISHI and M. KASAHARA, 'Cryptosystems based on pairing', *Symposium on Cryptography and Information Security – SCIS 2000*, Okinawa, Japan. [181](#)

K. Harrison keith_harrison@hp.com

Hewlett-Packard Laboratories
Filton Road, Stoke Gifford
Bristol, BS34 8QZ

D. Page page@cs.bris.ac.uk
N. P. Smart nigel@cs.bris.ac.uk

Dept. Computer Science
University of Bristol
Merchant Venturers Building, Woodland Road
Bristol, BS8 1UB