Implementing cryptographic pairings: *a magma tutorial*

Luis J Dominguez Perez*, Ezekiel J Kachisa**, and Michael Scott.**

School of Computing
Dublin City University
Ireland
ldominguez@computing.dcu.ie ekachisa@computing.dcu.ie
mike@computing.dcu.ie

Abstract. In this paper we show an efficient implementation of the Tate, ate and R-ate pairings in *magma*. This will be demonstrated by using the KSS curves with embedding degree $k = 18$.

1 Introduction

One of the first well known applications of cryptographic pairings is the transformation of an elliptic curve discrete logarithm problem (ECDLP) instance into an instance of discrete logarithm problem (DLP) in the finite field. This was done by Menezes, Okamoto and Vanstone [32] by using the Weil pairing while later Frey and Rück [18] also presented a similar technique using the Tate pairing. These applications were examples of a "negative" use of pairings in ECC. However, recently pairing implementations had been created for more constructive uses. Joux [26] showed how pairings can be used for a tripartite authentication protocol. This opened the flood gates for the positive applications of pairings. Shamir [41] for example in 1984 posed as a challenge the concept of an Identity-Based Encryption scheme. Boneh and Franklin [9] proposed a solution to this challenge based on a bilinear map using the Weil pairing. Many more state-of-the-art protocols using bilinear pairings were proposed such as short digital signatures [10].

In response, novel constructions of non-supersingular pairing-friendly elliptic curves have been proposed by different authors for pairing based protocols. With constructions such as MNT [35], Freeman [15], BN [6] one can produce ideal pairing-friendly elliptic curves. These are curves whose group bit length size is equal to the bit length size of the underlying field. The importance and effectiveness of such curves is mainly felt in the costs incurred in the computation of pairings. Some of these curves also support twists of higher order, for example sextic twists on the BN curves. Then there are pairing-friendly elliptic curves which are near-ideal. For instance curves as proposed by Barreto, Lynn

* This author acknowledge support from the Consejo Nacional de Ciencia y Tecnología
** These authors acknowledge support from the Science Foundation Ireland under Grant No. 06/MI/006

and Scott [4], and later as generalised by Brezing and Weng [12], and KSS curves as proposed in [27]. In these curves the bit size of the finite field is somewhat larger compared to the bit size of the prime order subgroup. As such there is always a need to look for optimal ways to implement pairings on such curves.

The computation of pairings basically involves two groups, G_1 and G_2. These two groups are finite cyclic additively-written groups and at least one of which is of prime order r. The pairing will take an element from each of the two groups and map them to the third group G_T, which is a finite cyclic multiplicatively-written group also of prime order r. A useful cryptographic pairing satisfies the following properties:

- *Bilinearity*:
  For all $P, P' \in G_1$ and all $Q, Q' \in G_2$, one has: $e(P + P', Q) = e(P, Q) \times e(P', Q)$ and $e(P, Q + Q') = e(P, Q) \times e(P, Q')$
- *Non-degeneracy*:
  For all $P \in G_1$ with $P \neq 0$, there is some $Q \in G_2$ such that $e(P, Q) \neq 1$.
  For all $Q \in G_2$ with $Q \neq 0$, there is some $P \in G_1$ such that $e(P, Q) \neq 1$.
- *Computable*:e can be easily evaluated.

The best known method for computing pairings is based on Miller's algorithm. This is a standard method and many researchers have been trying to improve its efficiency. The emphasis mainly has been to optimize the *Miller's loop* and the *final exponentiation* in the algorithm [[3], [1], [14], [31], [38],[5],[22]].

Our implementation uses the Magma software package which is a large, well-supported package designed to solve computationally hard problems in algebra, number theory, geometry and combinatorics [11]. Magma currently has internal functions to compute the Weil or Tate pairing. However one can also implement the computation of the pairings using ones own functions to eliminate unnecessary operations and to exploit specific properties of the chosen family of curves for an efficient implementation.

In this paper we will demonstrate using *magma* a step-by-step approach to the efficient implementation of pairings given a family of pairing friendly elliptic curves, and propose some specific optimisations for the chosen $k = 18$ family of such curves in the computation of the Tate, ate and R-ate pairings.

1.1 Organisation of the paper

The rest of the paper is organised as follows: the next section explains the KSS curves, in particular a family of curves of embedding degree, $k = 18$. In Section §3 we discuss the twists in elliptic curves and how to create sextic twist curves in magma. In §4 we discuss how to compute the Tate pairing in Magma for curves of degree 18, similarly in sections §5 and §6 for the ate pairing and R-ate pairing respectively. In §7 we compare the Miller loop length as experienced in sections §4, §5 and §6. Finally, in §8 and §9 we discuss an optimisation of the final exponentiation in the pairing computations. We conclude our discussions in §10. Appendix §A illustrates how to count points on the elliptic curve over an

extension field and over a twist of a curve. Appendix §B shows how to generate the T_i pairs for the R-ate paring.

2 Kachisa-Schaefer-Scott Curves

Kachisa et al. [27] proposed families of pairing-friendly elliptic curves of embedding degrees $k = 16, 18, 32, 36$, and 40. The main idea in the construction is to use minimal polynomials of the elements of the cyclotomic field other than the cyclotomic polynomial $\Phi_l(x)$ to define the cyclotomic field $\mathbb{Q}(\zeta_l)$. Interestingly, all these families of curves admit higher order twists. In particular, the family of curves for $k = 18$ is parameterised by the authors as follows:

$$t(x) = (x^4 + 16x + 7)/7$$

$$p(x) = (x^8 + 5x^7 + 7x^6 + 37x^5 + 188x^4 + 259x^3 + 343x^2 + 1763x + 2401)/21$$

$$r(x) = (x^6 + 37x^3 + 343)/343$$

where $t(x)$ is the trace of Frobenius, $p(x)$ represents the field size and $r(x)$ represents the pairing-friendly subgroup. As is well known the number of points on the elliptic curve $E(\mathbb{F}_p)$ is $\#E = p + 1 - t$. For this curve the ratio of the size of the field to the size of the group $\rho = \deg(p(x))/\deg(r(x)) = 4/3$. One is able to construct a curve using the above parameters when $x \equiv 14 \bmod 42$. In this class, $t(x)$ represent integers, and $p(x)$ and $r(x)$ can represent primes. For a randomly generated $x \equiv 14 \bmod 42$ as input, p and $r \in \mathbb{Z}$, are not necessary both primes. But by using a loop and iterating positive and negative values of x, it is not difficult to find a desired size of p and r that are both primes. Listing 1 shows the code in *magma* that can be used to find the primes p and r. The co-factor $c(x) = (p(x) + 1 - t(x))/r(x)$ is also calculated.

Listing 1. Code for finding p and r

```
// Input  <- any random integer number;
// Output -> p,r,t,x Integers, p and r primes.
KSSCurves:= function(x);
while (x-14) mod 42 ne 0 do
  x:=x+1;
end while;
while true do
  r:=(x^6 + 37*x^3 + 343) div 343;
  if IsPrime(r) then
    p:=(x^8 + 5*x^7 + 7*x^6 + 37*x^5 + 188*x^4 + 259*x^3 + 343*x^2
    + 1763*x + 2401) div 21;
    if IsPrime(p) then
      break;
    end if;
  end if;
```

```
  x:=x+42;
end while;
t:=(x^4+16*x+7) div 7;
c:=(p+1-t) div r;
return p,r,t,c,x;
```

Such a curve would suit, for example, an AES-192 security level protocol. AES-192 would require a 384-bit subgroup size [30]. Since the ρ-value is the rounded ratio of the size in bits of the modulus p to the size in bits of the group order r [38], it is not possible to get the exact number of bits expected. Therefore the authors decided to use a prime p of size 512-bits and a prime r of 379-bits for the purposes of this paper.

A suitable field over p and a prime group order r of these sizes, is found using as a seed $x := 0x1500000150000B7CE$, which was found to produce an $r(x)$ of low hamming weight. The actual values are given in Listing 2.

Listing 2. Field size, subgroup size, trace of Frobenius and co-factor

```
p:= 898399025747640384869366543419163653522011601817338789661118\
425362328282990603269356269942610720855598014085594837113711948 7\
303577983463490963245140511947;
r:= 834588325238359397017496169232723349635358800304932055763787\
073529864528483523055250452641305501342615931041985801;
t:=620506368937375113510572988169916990378876326499177450128973 1\
6876824887966225;
c:=10764576960635731125531981251036245532323;
```

Since this type of family of curves has a complex multiplication discriminant $D = -3$, then the equation of the curve will take the form of $y^2 = x^3 + b$, where $b \neq 0$. Using basically the same algorithm as in [6, Algorithm 1], but including the co-factor c, Listing 3 returns an elliptic curve, a finite field of size p, the b parameter in the elliptic curve formula, and a point G of order r.

Listing 3. Generating the Elliptic Curve

```
// Input  <- (p,r,c) defining the Elliptic Curve
// Output -> The EC, its Finite Field, b from the EC, and a point
CurveGen:= function(p,r,c)
  Fp:= FiniteField(p);
  b := Fp!0;
  repeat
    repeat
        b := b + 1;
    until IsSquare(b + 1);
    y := Fp!Root(b + 1, 2);
    E := EllipticCurve([Fp|0,b]);
    G := E![1,y];
  until IsZero(r*c*G);
return E,Fp,b,G;
```

Listing 3 begins searching for a curve from $b = 1$. When one gets to $b = 5$, that defines an elliptic curve $E : y^2 = x^3 + 21$ and a point $G(1, \sqrt{6})$ with the correct order.

Although the curve remains defined over $\mathbb{F}_p$, we can also consider points which are defined over the extension field $\mathbb{F}_{p^k}$. The Listing 4 shows the code in *magma* for creating an Elliptic Curve `ExtCurve` defined over the extension field `Fpk`.

Listing 4. Generating an Extension Field and its Elliptic Curve

```
k:=18;

Fpk:=FiniteField(p,k);
ExtCurve:=EllipticCurve([Fpk!0,b]);
```

3 Twists of the curves

One way to speed up pairings computation is through the use of twist curves. If E and E' are elliptic curves defined over $\mathbb{F}_p$, then E' is said to be a twist of degree d if there exists an isomorphism $\psi : E' \longrightarrow E$ defined over $\mathbb{F}_{p^e}$ with e minimal. It was shown by Naehrig et al [36] that such curves leads to compressed values of line functions which can be computed by a few field operations in $\mathbb{F}_{p^{k/d}}$ compared to the full field, $\mathbb{F}_{p^k}$. This optimisation was also used in [[38], [14], [36], [1]]. Unfortunately, there are limited possible values of d such that for $p \geq 5$ only *quadratic, cubic, quartic and sextic* twists are possible.

Since the curve under consideration, $k = 18$, is divisible by 6 and has a CM discriminant $D = 3$, then in this family of curves higher order twists, in particular *sextic twists*, exist. For the $k = 18$ family, to utilise the sextic twist we must define the point $P \in G_1$ defined over the curve $E(\mathbb{F}_p)[r]$ and $Q \in G_2$ defined over the twist curve $E'(\mathbb{F}_{p^3})$. To do this whenever $p \equiv 1 \pmod 6$, we choose $\chi \in \mathbb{F}_{p^3}$ such that $W^6 - \chi$ is irreducible over $\mathbb{F}_{p^3}[W]$. Furthermore, if $\delta \in \mathbb{F}_{p^{18}}$ is a root of $W^6 - \chi$, then there exists a homomorphism which maps points on the sextic twist to the points of the original curve as follows:

$$\psi : E'(F_{p^3}) \rightarrow E(\mathbb{F}_{p^{18}}) \text{ defined by: } (x', y') \rightarrow (x'\delta^{1/3}, y'\delta^{1/2}),$$

with an isomorphism given by:

$$[.] : \mu_6 \rightarrow Aut(E) : \delta \mapsto [\delta] \text{ with } [\delta](x, y) = (\delta^2 x, \delta^3 y) \text{ [24].}$$

The twist curve for a such a curve has the form of $E' : y'^2 = x'^3 + \frac{b}{\chi}$.

The following Listings demonstrate the code in *magma* for creating the same curve as in §2 using sextic twists. Now the second point Q is placed and manipulated on the twist curve $E'(\mathbb{F}_{p^3})$ and we can apply the twist map to bring the point Q to the full extension field when required by the pairing algorithm.

The `Fpd` in the Listings, here and thereafter, denotes the field constructed to obtain the extension field, $\mathbb{F}_{p^3}$. This is done by first finding an irreducible

polynomial of degree 3 over $\mathbb{F}_p$. For example, to find the irreducible polynomial $u^3 + 2$ for this extension field, one would use a magma function as follows: `IrreduciblePolynomial(Fp,3);`.

One can now use this polynomial to create the extension field $\mathbb{F}_{p^3}$. From this field we now look for an element χ such that $W^6 - \chi$ is irreducible over $\mathbb{F}_{p^3}[W]$ in order to find a suitable representation of $\mathbb{F}_{p^{18}}$ as a sextic extension of $\mathbb{F}_{p^3}$. It is necessary to pick the χ value that gives the correct order of the twist curve [6].

Firstly, we create a couple of general functions to help in the search for an appropriate χ. These functions will create the required extension fields and curves. However if χ is not an irreducible polynomial, the program will fail. This is especially bad when making long automated searches in magma. We need to use the try-or-catch mechanism to trap any error in the extension field and curve creation processes. This can be done as shown in listing 5.

Listing 5. Extension Fields and Curves

```
// Input  <- flag, finite field Fpd, polynomial X
// Output -> The extension field
GenerateFpk:= function(Fp, Fpd, X)
  try
    Fpk<v>:=ExtensionField<Fpd,v|v^6-X>;
    return Fpk;
  catch e
    return Fp;
  end try;
end function;

// Input  <- flag, finite field Fpd, b from the EC, polynomial X
// Output -> The EC in the twist
GenerateExtT:= function(E, Fpd, b, X)
  if IsEllipticCurve([Fpd|0,b/X]) then
    ExtT:=EllipticCurve([Fpd|0,b/X]);
    return ExtT;
  else
    return E;
  end if;
end function;
```

Not all the χ values will result in an Elliptic Curve of the right twist [6, Lemma 1]. When multiplying a random point on the curve by the co-factor c it **must** generate a point of order r.

If a point Q is mapped to a subgroup of order r, then $r \times Q$ will result in the point-at-infinity. The wrong twist of the elliptic curve will not generate a point Q in the correct subgroup of order r.

To create a point Q mapped to a subgroup of order r on the correctly selected twist of the elliptic curve, we use the homomorphism used in [22] and the addition chain proposed at [40], see Listing 6 for its implementation. With this program,

one can create a random point $Q \in E'(\mathbb{F}p^3)$ as: `Q:= HashG2(Random(ExtT), xx, deltas)`

(The δ's elements will be explained in more detail in §5.1.)

Listing 6. Fast Hashing in G_2 for KSS:$k = 18$ curves

```
// Input <- Point \in E'(\F_{p^e}), degree of function, \delta
// Output -> Point \in E'(\F_{p^e})
psi:=function(P,i,delta)
  E1:=Curve(P);
  Fp:=BaseField(BaseField(E1));
  P:=[Frobenius(P[1]*delta[1],Fp,i),Frobenius(P[2]*delta[2],Fp,i)];
  P:=E1![P[1]*delta[3],P[2]*delta[4]];
  return P;
end function;

// Input <- Point to Hash \in E'(\F_{p^e}), x, \delta
// Output -> Point \in E'(\F_{p^e})
HashG2:=function(Q,x,d);
  E1:=Curve(Q);
  x2:=x^2;
  Qx:=Q*x;
  Q_x:=Q*-x;
  Q_x2:=Q*-x2;
  Q_x3:=Q_x2*x;
  Q_x4:=Q_x2*x2;
  Q_x5:=Q_x4*x;
  Q_x6:=Q_x4*x2;
  Q_x7:=Q_x6*x;

  xA:=Q_x;      //x20
  xB:=-Q;       //-x21
  t1:=xA+xB;    //x20.x21
  t0:=psi(Qx,2,d)+xA;   //x16.x20
  t1:=2*t1;     //t1.t1
  t1:=t1+psi(Q_x,3,d);  //t1.x19
  t1:=2*t1;     //t1.t1
  t1:=t1+t0;    //t1.t0
  t0:=Q_x2+xB;  //x18.x21
  t0:=2*t0;     //t0.t0
  xB:=psi(Q_x2,3,d); //x17
  t0:=t0+xB;    //t0.x17
  t2:=t1+psi(Qx,1,d);   //t1.x14
  t1:=t1+psi(Qx,4,d);   //t1.x9
  t0:=t0+Q_x3;  //t0.x15
  t3:=psi(Q_x4,1,d)+psi(Q_x4,2,d)+Q_x3; //x5.x15
  t2:=t0+t2;    //t0.t2
  t4:=t2+psi(-Q,3,d);   //t2.x13
  t2:=t2+Q_x5+psi(Q_x4,3,d);    //t2.x7
  t4:=2*t4;     //t4.t4
  t4:=t4+psi(P,2,d)+psi(P,4,d); //t4.x11
  t6:=Q_x4+xB;//x12.x17
```

```
  t4:=t6+t4;    //t6.t4
  t6:=t6+t3;    //t6.t3
  t1:=t4+t1;    //t4.t1
  t4:=t4+t2;    //t4.t2
  xC:=psi(Q_x3,1,d)+psi(Q_x3,2,d); //x10
  t2:=xC+Q_x2;  //x10.x18
  t3:=psi(Q_x5,3,d)+xC; //x6.x10
  t1:=t2+t1;    //t2.t1
  t2:=t2+xB;    //t2.x17
  t7:=t1+psi(Q_x3,3,d); //t1.x8
  t4:=t7+t4;    //t7.t4
  t3:=t4+t3;    //t4.t3
  t4:=t4+psi(Q*x^2,4,d)+psi(Q_x2,1,d);  //t4.x4
  t3:=t3+t7;    //t3.t7
  t7:=t7+t2;    //t7.t2
  t1:=t3+t1;    //t3.t1
  t6:=t1+t6;    //t1.t6
  t6:=t6+t0;    //t6.t0
  t1:=t6+t1;    //t6.t1
  t6:=t6+t0;    //t6.t0
  t0:=t1+Q_x6;  //t1.x3
  t1:=t1+Q_x7+psi(Q_x7,1,d)+psi(Q_x7,2,d);  //t1.x0
  t0:=t0+t4;    //t0.t4
  t2:=t0+psi(Q_x6,1,d)+psi(Q_x6,2,d);   //t0.x2
  t0:=t0+t7;    //t0.t7
  t5:=t2+psi(Q_x6,3,d); //t2.x1
  t2:=t2+t6;    //t2.t6
  t2:=t2+t5;    //t2.t5
  t0:=2*t0;     //t0.t0
  t0:=t0+t5;    //t0.t5
  t1:=t2+t1;    //t2.t1
  t2:=t2+t4;    //t2.t4
  t0:=t1+t0;    //t1.t0
  t1:=t1+t3;    //t1.t3
  t0:=2*t0;     //t0.t0
  t0:=t0+t2;    //t0.t2
  t0:=2*t0;     //t0.t0
  t0:=t0+t1;    //t0.t1
  return t0;
end function;
```

At this point, the reader knows how to create a point Q, from an already selected twist of the elliptic curve. One can check if it is in fact mapped to a subgroup of order r with the *magma* function `IsZero(r*Q)`.

To find a twist of an elliptic curve, we began a linear search for a small χ in Listing 7. However as stated earlier not all the irreducible polynomials define an extension field and elliptic curves of the right twist. Hence, not all the irreducible polynomials can be used. The reader is also encouraged to add more powers in χ if preferred.

If one follows the strategy suggested in [6, Lemma 1], one would get a bigger χ than in Listing 7 and still have to test the order of a random point. It can take some time to create a random point on a curve like `ExtT` from Listing 6.

Listing 7. Searching for χ

```
for i1:= 0 to 100 do
  for i0:= 1 to 100 do
    X:=i1*u+i0;
    X:=Fpd!X;
    Fpk:=Fp;
    Fpk:=GenerateFpk(Fp, Fpd, X);
    if Fpk ne Fp then
      ExtT:=E;
      ExtT:=GenerateExtT(E, Fpd, b, X);
      if ExtT ne E then
        V:=Name(Fpk,1);
        deltas:=[];
        Append(~deltas,V^2);
        Append(~deltas,V^3);
        Append(~deltas,1/deltas[1]);
        Append(~deltas,1/deltas[2]);
        Q:=HashG2(Random(ExtT),xx,deltas);
        if IsZero(r*Q) then
          print "OK",i1,"*u+",i0;
        end if;
      end if;
    end if;
  end for;
end for;
```

Once the irreducible polynomials are known, the reader may prefer some simpler code for constructing a twist curve and picking up points P and Q of order r as shown below:

Listing 8. Twisted elliptic curve definition and map of points

```
Fp:=FiniteField(p);
E:=EllipticCurve([Fp! 0,b]);

Fpd<u>:=ExtensionField<Fp,u|u^3+2>;
X:=u+5;
ExtT:=EllipticCurve([Fpd|0,b/X]);

Fpk<v>:=ExtensionField<Fpd,v|v^6-X>;
ExtCurve:=EllipticCurve([Fpk|0,b]);
deltas:=[];
V:=Name(Fpk,1);
Append(~deltas,V^2);
Append(~deltas,V^3);
```

```
Append(~deltas,1/deltas[1]);
Append(~deltas,1/deltas[2]);
P:=c*Random(E);
Q:=HashG2(Random(ExtT),ExtT,ExtCurve,xx,Fp);
```

4 The Tate Pairing

Originally there were two cryptographic pairings, the Weil and Tate-Lichtenbaum. The Weil pairing requires two *Miller loops* to generate the m^{th} roots of unity [42, III.§8], while the Tate pairing requires only one application of the *Miller loop*, making it more efficient than the Weil pairing.

The Tate pairing was introduced by Tate as a rather general pairing on Abelian varieties over local fields. Lichtenbaum gave an application of this pairing to the Jacobians of curves over local fields[13]. The Tate-Lichtenbaum pairing is hereafter referred to as the Tate pairing.

Recalling some notation from §2 or [27], Let k be the *embedding degree*, which sometimes is referred to as *security multiplier*, of an elliptic curve E defined over a finite field $\mathbb{F}_p$, and let r be the large prime number that divides $\#E$ such that r divides (p^k-1) with p a large prime number. The definition of the Tate pairing is as follows [8]:

Let $P \in E(\mathbb{F}_p)[r]$ and $Q \in E(\mathbb{F}_{p^k})$, consider the divisor $D = (Q+S)-(S)$ with S a random point in $E(\mathbb{F}_{p^k})$. Let $f_{a,P}$ be a function with a divisor $(f_{a,P}) = a(P)-(aP)-(a-1)(0)$ for $a \in \mathbb{Z}$. A non degenerate bilinear Tate pairing can be defined as a map:

- $e_r : E(\mathbb{F}_p)[r] \times E(\mathbb{F}_{p^k})/rE(\mathbb{F}_{p^k}) \rightarrow \mathbb{F}^*_{p^k}/(\mathbb{F}^*_{p^k})^r$
  $(P,Q) \mapsto \langle P,Q\rangle_r = f_{r,P}(D)$

This value of the pairing is in an equivalence class, $\mathbb{F}^*_{p^k}/(\mathbb{F}^*_{p^k})^r$. For practical purposes it is preferred to raise the value of the pairing to the power of $(p^k-1)/r$ to obtain a unique representative of the class, i.e $e_r(P,Q) = \langle P,Q\rangle^{(p^k-1)/r}$. This exponentiation is known as the *final exponentiation*, and the pairing is referred to as the *Reduced Tate Pairing*.

4.1 The Miller Loop for the Tate Pairing

Implementing the Tate Pairing is almost as easy as implementing the *Miller loop*. An easy way to implement the Miller formula is explained in [3, Theorem 2] and [38].

The inputs to the pairing are the points $P \in E(\mathbb{F}_p)[r]$ and $Q \in E(\mathbb{F}_{p^k})$. It is necessary to apply the double-and-add, line-and-tangent algorithm until the point P, on being multiplied by its order r, finally reaches the point-at-infinity. It will arrive there after $\lg(r-1)$ iterations of the Miller loop.

In this case Q may not be of order r. We get a random point $\in E'(\mathbb{F}_{p^3})$ and apply the ψ map. See Listing 9

Listing 9. Generating point $Q \in E(\mathbb{F}_{p^k})$

```
Q:=Random(ExtT);
Q:=ExtCurve![Q[1]*deltas[1],Q[2]*deltas[2]];
```

The code in Listing 10 is used to compute line functions $l_{A,B}(Q)$ when evaluating the contribution to the pairing value of the elliptic curve point addition, $A + B$. In essence these are distances calculated between the fixed point Q and the lines that arise when adding points A and B. In this code there are three cases to be considered.

Listing 10. Line function

```
//Input <- A,B in G_1, Q in E(\F_{p^k})
//Output -> return in G_T
L:= function(A,B,Q)
    if A eq -B then
        return Q[1]-A[1];
    end if;
    if A eq B then
        l:=(3*A[1]^2) / (2*A[2]);
    else // A ne B
        l:=(B[2]-A[2])/(B[1]-A[1]);
    end if;
    return (l*(Q[1]-A[1]) + A[2]-Q[2]);
end function;
```

The first case is where $A = B$, and the line in this case is the tangent to the curve at point A, the second case is when $A \neq B$. These two cases use the formula `(l*(Q[1]-A[1]) + A[2]-Q[2])`, where `l` is the slope of the line, to compute the values of the functions. The last case is when the point A is a negative of point B. In this case we have a vertical line and use the formula `Q[1]-A[1]` to compute the line functions. Refer to [42] for more details.

Finally, the full *Tate Pairing* algorithm is presented in Listing 11. In this code the double-and-add stages can be identified by $2 * T$ (double) and $T + P$ (add), where the add is required when a 1 is present in the binary expansion of r. In addition, there is the *"vanilla"* final exponentiation step. This step obtains the unique value of the cosets of $\mathbb{F}^*_{p^k}/(\mathbb{F}^*_{p^k})^r$. This will be discussed in §8 in more detail where we will also look at some optimisations that apply to this step.

Listing 11. Basic Tate pairing with single Miller Loop

```
// Input <- P \in G_1, Q \in E(\F_{p^k})
// Output -> f \in G_T
pairing:= function(P,Q)
f:=1;
T:=P;
i:=Floor(Log(2,r))-1;
si:=Intseq(r,2);
while i ge 0 do
    f:=f^2*L(T,T,Q);
```

```
    T:=2*T;
    if si[i+1] eq 1 then
        f:=f*L(T,P,Q);
        T:=T+P;
    end if;
    i:=i-1;
end while;
f:=f^(((p^k)-1) div r);
return f;
end function;
```

5 The ate Pairing

The ate pairing[24] is a variant of Tate pairing and it is a generalisation of the Eta pairing to ordinary pairing-friendly elliptic curves. The ate pairing is particularly suitable for pairing-friendly curves with small values of the trace of Frobenius.

Let π_p be the Frobenius endomorphism, $\pi_p : E \longmapsto E : (x, y) \longmapsto (x^p, y^p)$. We denote $G_1 = E[r] \cap Ker(\pi_p - [1]), G_2 = E[r] \cap Ker(\pi_p - [p])$. Let $T = t - 1$. Let $N = gcd(T^k - 1, p^k - 1), T^k - 1 = LN$. For $Q \in G_2$ and $P \in G_1$ ate pairing is defined as [24]:

$$a_T : (Q, P) \longmapsto f_{T,Q}(P)^{c_T(p^k-1)/N},$$

where $c_T = \sum_{i=0}^{k-1-i} p^i \equiv kp^{k-1} \mod r$. The ate pairing is a bilinear non-degenerate pairing if $r \nmid L$. What should be noted here also is the change in arguments. The first parameter P is defined over the extension field and Q is defined over the base field.

5.1 The Miller Loop for the ate Pairing

In the ate pairing the number of iterations in the Miller loop depends on the size of the trace of the Frobenius t rather than on the size of the subgroup r. Thus, as noted in [38] if $\omega = \log r / \log |t|$ is greater than one for a particular family then it is possible to compute the ate pairing faster for those type of curves. The larger the ω the faster the ate computation compared to the Tate pairing computation. For $k = 18$ curves we have $\omega = 3/2$. Therefore this curve is suitable for an implementation of the ate pairing.

The code in magma for computing the ate pairing is given in Listing 12. Where $G_1 \in E(\mathbb{F}_p)[r])$ and $G_2 \in E'(\mathbb{F}_{p^3})$. Note that here we define G_2 on the sextic twist.

Listing 12. ate pairing with single Miller Loop

```
//Input <- P \in G_2, Q \in G_1$
//Output -> f \in G_T
pairing:= function(P,Q)
```

```
f:=1;
T:=P;
s:= t-1;
i:=Floor(Log(2,s))-1;
si:=Intseq(s,2);
while i ge 0 do
  f:=f^2*L(T,T,Q);
  T:=2*T;
  if si[i+1] eq 1 then
    f:=f*L(T,P,Q);
    T:=T+P;
  end if;
  i:=i-1;
end while;
f:=f^(((p^k)-1) div r);
return f;
end function;
```

However, in the Miller loop, some operations on the curve over the extension field are required. Since one is using a point $Q \in E'(\mathbb{F}_{p^{k/d}})$ instead of $E(\mathbb{F}_{p^k})$, it is therefore required to apply the map ψ, discussed in [§3], in the line function. Each of the x and y coordinate of a point $Q \in E'(\mathbb{F}_{p^3})$ has three components defined over the base field $\mathbb{F}_p$. Hence instead of applying the map function ψ, one can also place the 3 components to their corresponding positions in the x and y-coordinates of Q.
Unfortunately, this cannot be done in magma. Instead we apply the "full" ψ map $(x.\delta^2, y.\delta^3)$. However, since δ is a constant for the generated curve, we can precompute these values and the cost of the mapping will be just one multiplication by a coordinate component (each time the L-function is called.)

Listing 13 show the modified code.

Listing 13. Line Function with ψ map

```
//Input <- A,B \in G_2, Q \in G_1
//Output -> return in G_T
L:= function(A,B,Q)
  Ax:=A[1]*deltas[1];
  Ay:=A[2]*deltas[2];
  Bx:=B[1]*deltas[1];
  By:=B[2]*deltas[2];
  if (Ax eq Bx) then
    if (Ay eq -By) then
      return Q[1]-Ax;
    else
      if (Ax eq Bx) and (Ay eq By) then
        l:=(3*Ax^2) / (2*Ay);
      end if;
    end if;
  else // A ne B
      l:=(By-Ay)/(Bx-Ax);
  end if;
```

```
  return (l*(Q[1]-Ax) + Ay-Q[2]);
end function;
```

6 The R-ate Pairing

The new R-ate pairing introduced by Lee, Lee and Park [29] is a generalisation of the ate[24] and ate$_i$[45] pairing improving its computation efficiency. It takes three short *Miller loops* to calculate the pairing, that together requires a shorter loop than a single typical application of the ate pairing. The *R-ate* pairing can be regarded as a ratio of any two pairings, hence the name.

The definition of the *R-ate* pairing with $A = aB + b$ where $A, B, a, b, \in \mathbb{Z}$ is as follows:

$$e_{A,B}(P,Q) = f_{a,BP}(Q) \times f_{b,P}(Q) \times G_{aBP,bP}(Q)$$

Generally this definition does not always give a bilinear and non-degenerate pairing. However, with a careful choice of pairs A and B one can succeed. For efficiency, we look for a working and non-trivial combination of A and B that would give the shortest *Miller loop*.

In [29, Algorithm 2] there are three *Miller loops* to compute and the *final exponentiation* is calculated at the end of the computation. These computations won't produce the m^{th} roots of unity since they are not full-*Miller loops*. They give a partial result that is used as part of the pairing T, as in Listing 11.

The code for the *Miller loop* denoted as `M` in Listing 14 is used to compute the *R-ate pairing*. It makes function calls to Listing 10.

Listing 14. Miller loop as to be used by the *R-ate* pairing

```
// Input <- P \in G_2, Q \in G_1, l \in \Z
// Output -> f \in G_T, T \in \Z
M:= function (P,Q,l)
T:=P;
f:=1;
i:=Floor(Log(2,l))-1;
li:=Intseq(l,2);
while i ge 0 do
  f:=f^2*L(T,T,Q);
  T:=2*T;
  if li[i+1] eq 1 then
    f:=f*L(T,P,Q);
    T:=T+P;
  end if;
  i:=i-1;
end while;
return f,T;
end function;
```

There are a few differences between the Listing 11 and Listing 14. In particular the absence of the final exponentiation, the introduction of the parameter `l` that prescribes the number of loops, instead of `r` or `t-1` and the use of `T` as an extra output in Listing 14.

The *R-ate* pairing, like in the ate$_i$ [45], requires the calculation of $T_i \equiv p^i \bmod r$, with $0 \leq i < k$, where k is the *embedding degree*. This pairing is constructed from the parameters (p, r), which are also used to define the ate and Tate pairing, and a combinations of T_i.

Listing 16 shows an implementation of the *R-ate* pairing function. As noted in [29, Algorithm 2], f_a, f_b, aQ and bQ with $\{a, b\} = \{\texttt{m1}, \texttt{m2}\}$ is used, in addition, parameters $\texttt{m1} = max\{a, b\}$ and $\texttt{m2} = min\{a, b\}$ play an important role in the algorithm.

The *R-ate* pairing, also called $R_{A,B}(P, Q)$, is relative to parameters A and B. For the KSS, $k = 18$, the authors founded the linear combination, $T_{13} = \frac{3}{7}xT_6 + \frac{2}{7}x$, that is $a = \frac{3}{7}x, j = 6$ and $b = \frac{2}{7}x$ as one of the working non-trivial combinations for the *R-ate* pairing computation.

This is a good choice since `A` and `B` have almost the same number of bits with both values less than $\texttt{x} \in \mathbb{Z}$, as such it provides a short *Miller length* of which one can use the `f3` and `T` value from the Listing 14 and return them to get the final values. Substituting the last four lines in Listing 14 with those in Listing 15 one obtains an optimised Miller's loop for *R-ate* pairing.

Listing 15. Modifications to the Miller loop for the R-ate pairing

```
    if i eq 1 then
      f2:=f; T2:=T;
    end if;
    i:=i-1;
end while;
return f,T,f2,T2;
end function;
```

We know from [29, Algorithm 2] that $c \leftarrow [\frac{m_1}{m_2}]$ and $d \leftarrow m1 - c \times m_2$. For the $a = \frac{3}{7}x$, $b = \frac{2}{7}x$ values proposed, we got $d = \frac{a}{2}$. With these values, the first and third Miller function call can be integrated into one as shown in Listing 15.

We also get $c = 1$ from these combination. Then it is not necessary to call Listing 14 for the second loop.

Furthermore, one can omit unnecessary use of memory or computations since by definition a, b, j, `m1`, `m2`, `c`, `d`, `fcm2` and `cm2Q` are known.

Listing 16. R-ate pairing function

```
// Input <- P \in G_1, Q \in G_2, l \in \Z
// Output -> f \in G_T, T \in \Z
pairing:= function(P,Q)
dd:=(xx) div 7;
m1:=(3*dd);
m2:=(2*dd);
jj:=6;
```

```
//Compute
fm2,m2Q,fd,dQ:=M(Q,P,m2);

f1:=fm2*fd;
fm1:=f1*L(m2Q,dQ,P);
m1Q:=m2Q+dQ;

//Exponentiation
f2:=Frobenius(fm1,Fp,jj)*fm2;
Q1:=[Frobenius(m1Q[1]*V2,Fp,jj),Frobenius(m1Q[2]*V3,Fp,jj)];
Q1:=ExtT![Q1[1]*V1_2,Q1[2]*V1_3];
f3:=f2*L(Q1,m2Q,P);

//Final Exponentiation
f3:=f3^(((p^k)-1) div r);

return f3;
end function;
```

The code, `fm2,m2Q,fd,dQ:= M(Q,P,m2);`, in the *Compute* section of Listing 16 returns f_{m_2} as `f` and `m2Q` as `T` in Listings 14 or 15. It is easy to note that `Q` and `m2` are the parameters of the `M` function, where `m2` is the *Miller length*. Hence `m2Q` is the actual state of `T`.

New in Listing 16 is the use of the `Frobenius` function in exponentiation not only at the computation of `Q1`, but also for `f2`. This will be covered in more detail later at §8 as one of the optimisations for the computation of pairings.

Additionally, since there is only one `M` function call, one would be able to insert the *Miller loop* inside a modified version of the Listing 16.

Finally, $Q1 \in E(\mathbb{F}_{p^{18}})$ and the `L`-function requires it to be $\in E'(\mathbb{F}_{p^3})$, hence, we apply the inverse ψ map from Section §3 before the line function. Optionally, one may use a modified version of this function.

7 Comparison of the Miller-loop length

In this section we compare the *Miller length* of the pairings implemented in sections §4, §5 and §6.

The Tate pairing was presented in section §4. From Listing 11 one can see how the number of *Miller loops* iterations is related to r. One can refer to the Tate pairing as $e_r(P,Q)$ and its length is $\lfloor log_2(r) \rfloor - 1$.

In section §5 the ate pairing presented in Listing 12 shows the number of *Miller loops* iterations is related to t. Similarly to the Tate pairing, the ate pairing can be referred to as $e_t(P,Q)$, and its length is $\lfloor log_2(t-1) \rfloor - 1$.

However, in §6 there are three *Miller loops* required with lengths depending on the A and B parameters. One can use $e_{A,B}(P,Q)$ as a way of referring to the R-ate pairing. Its length is $\lfloor log_2(m_2) \rfloor + \lfloor log_2(c) \rfloor + \lfloor log_2(d) \rfloor - 3$, where $m_1 = max\{a,b\}$, $m_2 = min\{a,b\}$, $c \leftarrow [\frac{m_1}{m_2}]$ and $d \leftarrow m1 - c \cdot m_2$. Simplifying for the chosen parameters in §6, the length is $\lfloor log_2(m_2) \rfloor - 1$.

Using the parameters in Listing 3 and 6, the *Miller loop length* for these 3 pairings is shown in Table 1. Clearly, the *Miller loop length* of the *R-ate* pairing is $\frac{1}{6}\lfloor log_2(r) - 1 \rfloor$.

Miller-length in iterations		
Tate	$e_r(P,Q)$	376
ate	$e_t(P,Q)$	253
R-ate	$e_{A,B}(P,Q)$	61

Table 1. Comparison of Miller-loop length

8 Final exponentiation

One of the most expensive operation in the pairing computation is the final exponentiation. In this section we discuss how the costs of final exponentiation can be reduced for our $k = 18$ curves.

In the Tate, ate and R-ate pairings one is required to perform an exponentiation by $(p^k - 1)/r$. This process eliminates the r-th powers and facilitates obtaining the r-th roots of unity. There have been many suggestions as to how one can speed up the computation in the *final exponentiation*.

Devegili et al. [14] observed that if the exponent $(p^k - 1)$ is appropriately factored, then one can get easy exponentiations using the Frobenius. In the $k = 18$ curves we apply the same idea here by factoring $(p^{18} - 1)$ into $(p^9 - 1)$ and $(p^9 + 1)$. Furthermore, we can factor $(p^9 + 1)$ into $(p^3 + 1)$ and $\Phi_{18}(p)$. Since $r|\Phi_{18}(p)$, one can apply the Frobenius map on $(p^9 - 1)$ and $(p^3 + 1)$ and be left with $\Phi_{18}(p)/r$, where $\Phi_{18}(p) = (p^6 - p^3 + 1)$. Listing 17 gives the generic *magma* code which can replace the final exponentiation in the *Tate, ate* and *R-ate* pairing algorithms to utilise the cheap computation of the Frobenius map.

Listing 17. The Final Exponentiation

```
f:=Frobenius(f,Fp,9)*f^(-1);

f:=Frobenius(f,Fp,3)*f;

f:=f^((p^6-p^3+1) div r);
```

Now if we let $\lambda = (p^6 - p^3 + 1)/r$, the "hard part" of the final exponent, one can find a way to simplify λ. For example, the exponent λ can be represented to the base p as $\lambda_0 + \lambda_1 p + \lambda_2 p^2 + \cdots + \lambda_5 p^5$. Such that f^λ can be written as:

$$f^\lambda = f^{\lambda_0 + \lambda_1 p + \lambda_2 p^2 + \cdots + \lambda_5 p^5} = (f)^{\lambda_0} \cdot (f^p)^{\lambda_1} \cdot (f^{p^2})^{\lambda_2} \cdots (f^{p^5})^{\lambda_5}$$

This decomposition can be obtained using the Listing 18 where the λ_i are calculated as polynomials of x. These exponents are not as simple as in [14], which are for the BN Curves. However, one can calculate them off-line.

Listing 18. Base p representation of the λ exponent

```
getlambdas:= function(xx)
  Zx<x>:=PolynomialRing(RationalField());
  px:=(x^8 + 5*x^7 + 7*x^6 + 37*x^5 + 188*x^4 + 259*x^3 + 343*x^2 +
     1763*x + 2401) / 21;
  rx:=(x^6 + 37*x^3 + 343) div 343;
  lambda:= [];
  Append(~lambda,(px^6-px^3+1) div rx);
  for i:=1 to 5 do
    Append(~lambda,lambda[i] mod px^(6-i));
  end for;
  for i:=1 to 6 do
    lambda[i]:= Evaluate(lambda[i] div px^(6-i),xx);
  end for;

  return lambda;
end function;
```

As before, one can use the Frobenius exponentiation for the f^{p^i} elements combined with any multi-exponentiation technique. However, we can use the same approach as in [39, 7.2] to simplify the computation of the hard part of the final exponentiation.

Listing 19. Code for the hard part of the final exponentiation

```
HardExpo:= function(f3,x)
f3x1:=f3^(x);
f3x2:=f3x1^x;
f3x3:=f3x2^x;
f3x4:=f3x3^x;
f3x5:=f3x4^x;
f3x6:=f3x5^x;
f3x7:=f3x6^x;

xB:= Frobenius((f3x1)^(-1),Fp,2);//x27
xA:= Frobenius((f3)^(-1),Fp,2);//x28
t4:= xB*xA;//x27.x28
t0:= t4*Frobenius((f3x2)^(-1),Fp,2);//x24
t4:= t4^2;
t4:= t4*xA;//x28
xA:= Frobenius(f3x1,Fp,1);//x26
t4:= t4*xA;//x26
t3:= Frobenius((f3x5)^(-1),Fp,2)*Frobenius(f3x4,Fp,4)*Frobenius(
    f3x2,Fp,5)*xA;//x8.x26
t4:= t4^2;
t4:= t4*t0;
xA:= Frobenius(f3x2,Fp,1);//x25
t0:= Frobenius((f3x4)^(-1),Fp,2)*Frobenius(f3x1,Fp,5)*xA;//x15.x25
t6:= xA^2;//x25
t6:= t6*Frobenius(f3x2,Fp,4);//x21
t2:= t4*Frobenius(f3x1,Fp,4);//x23
```

```
t4:= t4*xB;//x27
t2:= t2^2;
xA:= Frobenius((f3x2)^(-1),Fp,3);//x22
t6:= t6*xA;//x22
t1:= t6*(f3x2)^(-1);//x19
t6:= t6*t4;
xB:= Frobenius((f3x3)^(-1),Fp,3);//x20
t1:= t1*xB;//x20
t2:= t1*t2;
t4:= t2*Frobenius((f3x3)^(-1),Fp,2)*Frobenius(f3,Fp,5);//x18
t5:= t2*t3;
t5:= t5*t4;
xC:= (f3x3)^(-1);//x17
t4:= t4*xC;//x17
t3:= Frobenius((f3x6)^(-1),Fp,3)*xC;//x6.x17
t4:= t4^2;
t4:= t4*t6;
xC:= Frobenius(f3x3,Fp,1);//x16
t0:= t0*xC;//x16
t2:= xC*xA;//x16.x22
t7:= t0*t4;
t4:= t0*Frobenius(f3x4,Fp,1);//x13
t0:= t7*Frobenius(f3x3,Fp,4);//x14
t6:= t7*Frobenius((f3x5)^(-1),Fp,3)*Frobenius(f3x5,Fp,4);//x7
t4:= t0*t4;
t0:= t0*Frobenius(f3x6,Fp,1);//x3
t4:= t4^2;
t4:= t4*t5;
xA:= Frobenius((f3x4)^(-1),Fp,3);//x12
t5:= (f3x4)^(-1)*xA;//x10.x12
t3:= t3*xA;//x12
xA:= Frobenius(f3x5,Fp,1);//x11
t7:= xA^2;//x11
t7:= t7*t6;
t6:= t5*Frobenius(f3,Fp,3);//x9
t4:= t6*t4;
t6:= t6*Frobenius((f3x7)^(-1),Fp,3);//x1
t1:= t4*t1;
t4:= t4*Frobenius(f3x6,Fp,4);//x2
t1:= t1^2;
t1:= t1*(f3x5)^(-1);//x5
t2:= t7*t2;
t4:= t4*t7;
t2:= t2^2;
t2:= t2*t0;
t1:= t1*t3;
t0:= t1*xB;//x20
t1:= t1*t6;
t0:= t0^2;
t0:= t0*t5;
xA:= (f3x6)^(-1);//x4
```

```
t2:= t2*xA;//x4
t2:= t2^2;
t2:= t2*t4;
t0:= t0^2;
t0:= t0*t3;
t1:= t2*t1;
t0:= t1*t0;
t1:= t1*xA;//x4
t0:= t0^2;
t0:= t0*t2;
xA:= f3*(f3x7)^(-1);//x0
t0:= t0*xA;//x0
t1:= t1*xA;//x0
t0:= t0^2;
t0:= t0*t1;

return t0;
end function;
```

During the computations some intermediate results are required and one can still gain some advantages optimizing the code to use less memory.

It will be necessary to change the last line of §17 to `f:=HardExpo(f,xx);` in order to call this function. This will result in a significant gain in speed.

9 Multiplication in G_2

Gallant, Lambert and Vanstone [19] introduced a method to speed up general point multiplication $nP \in E(\mathbb{F}_p)[r]$ when there is an efficient computable endomorphism ψ on E defined over $\mathbb{F}_p$ such that $\psi(P) = \lambda P$. In the case of this paper, since KSS $k = 18$ curves [27] have the form $y^2 = x^3 + b$ as in [19, Example 4], the GLV method applies and it would require only one multiplication in $\mathbb{F}_p$ to apply the endomorphism to point in G_1.

The idea is to compute nP efficiently by writing $n \equiv n_0 + n_1\lambda \pmod{r}$ with $|n_i| < \sqrt{r}$ and performing a double exponentiation $n_0P + n_1\psi(P)$ [19]. In this case, the number of bits in n_0 and n_1 are half the bitlength of n. One can save a significant number of point doublings at the expense of a few point additions and the application of a map. If the map ψ is also cheaper than a point addition then, it is possible to get a few extra computational savings.

Gallant et al [19] pointed that their method can be generalised for using higher powers of the endomorphism. Galbraith and Scott [22] recently showed a technique on how to do it for the groups G_2 and G_T, as follows.

To get an m-dimensional expansion $n \equiv n_0 + n_1\lambda + \ldots + n_{m-1}\lambda^{m-1} \pmod{r}$ of nP, one must compose n with powers of λ sufficiently different modulo r. This can be done by solving a closest vector problem in a lattice as done in the Babai's rounding method [2]. However, an LLL-reduced lattice basis must be precomputed[22].

For KSS $k = 18$ curves [27], it will be possible to get a "natural" 6-dimensional expansion.

The modular lattice basis is defined as follows:

$$L = \left\{ x \in \mathbb{Z}^m : \sum_{i=0}^{m-1} x_i \lambda^i \equiv 0 \pmod{r} \right\}$$

where $\lambda = T = t - 1$ as in [22, Example 5].This 6-dimensional modular lattice L will be used to construct a 6×6 matrix. Then, one can fill the matrix with any combination of λ that gives $L_{i,j} \equiv 0 \pmod{r}$. One can use a predefined sequence of values for the L-matrix to be LLL-reduced, however it is possible to use random values. See Listing 20 for a random and an ordered definition of the matrix L. Also see the function `LLL()` for the LLL-reduction at the end of the program.

Listing 20. Initial Lattice and LLL-reduction

```
x:=25672080793808285942;
m:=6;
r:=(x^6 + 37*x^3 + 343) div 343;
t:=(x^4+16*x)/7+1;
lambda:=t-1;

// Option 1, randomly
L:= Matrix(Rationals(), m, [
  lambda,-1,0,0,lambda,-1,
  0,0,lambda,-1,0,0,
  0,0,0,lambda,-1,0,
  0,lambda,-1,0,lambda,-1,
  lambda,-1,0,lambda,-1,0,
  r,lambda,-1,lambda,-1,0
]);

// Option 2, ordered
L:= Matrix(Rationals(), m, [
  r,0,0,0,0,0,
  lambda,-1,0,0,0,0,
  lambda^2,0,-1,0,0,0,
  lambda^3,0,0,-1,0,0,
  lambda^4,0,0,0,-1,0,
  lambda^5,0,0,0,0,-1
]);

B:=LLL(L);
```

This way, `B` is the LLL-reduced matrix, similarly to [22, Example 5]. Note that one can automate the `L` creation in the second matrix in Listing 20.

One can verify the lattice consistency mod r with the code in Listing 21

Listing 21. Validate matrix B

```
checklattice:=procedure(L)
  sum:=0;
```

```
  for i:=1 to m do
    for j:= 1 to m do
      sum:=sum+L[i][j]*lambda^(j-1);
    end for;
    sum:= Integers()!sum mod r;
    if sum eq 0 then print "OK"; else print "BAD",i; end if;
    sum:=0;
  end for;
end procedure;
```

At this point, there is a valid LLL-reduced B matrix.

$$B = \begin{pmatrix} -\frac{2x}{7} & -1 & 0 & -\frac{x}{7} & 0 & 0 \\ 0 & -\frac{2x}{7} & -1 & 0 & -\frac{x}{7} & 0 \\ 0 & 0 & \frac{2x}{7} & 1 & 0 & \frac{x}{7} \\ \frac{x}{7} & 0 & 0 & -\frac{3x}{7} & -1 & 0 \\ 0 & \frac{x}{7} & 0 & 0 & -\frac{3x}{7} & -1 \\ -1 & 0 & -\frac{x}{7} & 1 & 0 & \frac{3x}{7} \end{pmatrix}$$

Choose a random exponent n for decomposing a vector (n,0,0,0,0,0) with respect to the basis formed by B. However, and with the λ value generating B from Listing 20, one obtains a vector $v \approx wB^{-1}$ as follows:

$$v \approx (-\tfrac{3x^5+56x^2}{343}, \tfrac{8x^4+147x}{343}, \tfrac{19x^3+343}{343}, \tfrac{x^5+3x^2}{343}, -\tfrac{5x^4+98x}{343}, -\tfrac{18x^3+343}{343}).\tfrac{n}{r}$$

Since all the elements of v are divided by r and $\notin \mathbb{Z}$, one just needs to *round* each element [2],[20] of v before getting the vector $u = w - vB$. See Listing 22.

Listing 22. Rounding and generating vector u

```
n:= Random(r);
w:= Matrix(RationalField(), 1, m, [n,0,0,0,0,0]);

// Rounding
v0:= w*B^-1;
v:= Matrix(RationalField(), 1, m, [0,0,0,0,0,0]);
for i:= 1 to m do
  v[1][i]:= Round(v0[1][i]);
end for;

// Main vector
u:= w-v*B;
```

In the code of Listing 22 the vector u contains the coefficients n_i with the decomposition of n mentioned at the beginning of this section. Finally, one can verify $n \equiv n_0 + n_1\lambda + \ldots + n_{m-1}\lambda^{m-1} \pmod{r}$ for this vector in Listing 23

Listing 23. Verification of the n_i components

```
sum:=0;
for i:= 1 to m do
```

```
  sum:= sum + u[1][i]*lambda^(i-1);
end for;
if (Integers()!sum mod r) eq n then print "OK" else print "BAD";
```

10 Conclusion

In this paper we have taken the reader through the processes of implementation of pairings on a pairing-friendly elliptic curves using the magma software. This has been demonstrated using a family of pairing friendly elliptic curves of embedding degree 18. Some optimizations such as the use of a twist of a curve and methods to speed up the final exponentiation have been discussed in relation to $k = 18$ curves. These optimisations can be extended to other embedding degrees with similar structure. The code in this paper can be used as a basic guide in the implementation of cryptographic pairings.

11 Acknowledgements

The authors would like to thank Professor Gary McGuire UCD(CSI) for his useful clarifications on the counting of points on the elliptic curves.

References

1. Antonio, C.A., Tanaka, S. and Nakamula, K., (2007) *Implementing Cryptographic Pairings over Curves of Embedding Degrees 8 and 10*. Cryptology ePrint Archive Report 2007/426, http://eprint.iacr.org/2007/426.
2. Babai, Li. (1986) *On Lovasz lattice reduction and the nearest lattice point problem*. Combinatronica 1986, 6(1), pp. 1–13, Springer-Verlag.
3. Barreto P.S.L.M., Lynn, B., Kim H. and Scott, M. (2002) *Efficient Algorithms for Pairing-Based Cryptosystems*. Advances in Cryptology – Crypto'2002, Lecture Notes in Computer Science 2442, pp. 354-368, Springer-Verlag
4. Barreto P.S.L.M., Lynn, B. and Scott, M., (2002) *Constructing elliptic curves with prescribed embedding degree*. Security in Communication Networks -SCN 2002, Lecture Notes in Computer Science 2576, pp. 263–273, Springer-Verlag
5. Barreto P.S.L.M., Lynn, B. and Scott, M., (2003) *On the Selection of Pairing-Friendly Groups*. Symposium on Applied Computing -SAC 2003, Lecture Notes in Computer Science 3006, pp. 17–25, Springer-Verlag
6. Barreto, P.S.L.M., and Naehrig M., (2006) *Pairing-friendly elliptic curves of prime order*. Selected Areas in Cryptography – SAC'2005, Lecture Notes in Computer Science 3897, pp 319–331 Springer-Verlag
7. Blake, I., Seroussi, G., and Smart, N., (1994) *Elliptic Curves in Cryptography*. London Mathematical Society, Cambridge: Cambridge University Press.
8. Blake, I., Seroussi, G., and Smart, N. (2005). *Advances in Elliptic Curve Cryptography*. Cambridge University Press.
9. Boneh, D., Franklin, M. (2001) *Identity-based encryption from the Weil pairing*. Lecture Notes in Computer Science 2139, pp. 213229, Springer-Verlag.

10. Boneh, D., Lynn, B., and Shacham, H. (2001). *Short Signatures from the Weil Pairing*. Lecture Notes in Computer Science 2248, pp. 514-532. Springer, Verlag.
11. Bosma, W., Cannon, J., and Playoust, C. (1997). *The Magma algebra system. I. The user language*. J. Symbolic Comput., 24(3-4):235-265, 1997
12. Brezing F. and Weng A., (2005) *Elliptic curves suitable for pairing based cryptography.*, Designs Codes and Cryptography, Vol. 37, No. 1, pp. 133–141
13. Cohen, H., Frey, G. (2006). *Hanbook of Elliptic and Hyperelliptic Curve Cryptography* Chapman & Hall/CRC
14. Devegili, A. J., Scott, M. and Dahab R., (2007) *Implementing Cryptographic Pairings over Barreto-Naehrig Curves*. Pairing 2007, Lecture Notes in Computer Science 4575, pp. 197–207, Springer-Verlag
15. Freeman, D., (2006) *Constructing pairing-friendly elliptic curves with embedding Degree 10*. In Algorithmic Number Theory Symposium ANTS-VII, Lecture Notes in Computer Science 4096, pp. 452–465. Springer-Verlag
16. Freeman, D., Scott, M. and Teske, E., (2006) *A Taxonomy of pairing-friendly elliptic curves*. Cryptography ePrint Archive, Report 2006/372, http:// eprint.iacr.org/2006/372
17. Frey, G. (2001). *Applications of Arithmetical Geometry to Cryptographic Constructions* Finite Field 5, 128–161, 2001.
18. Frey, G., and Rück, H-G., (1994). *A Remark concerning m-Divisibility and the Discrete Logarithm in the Divisor Class Group of Curves*. Mathematics of Computation, Volume 62, Number 206, pp. 865–874. American Mathematical Society.
19. Gallant Robert P., Lambert, Robert J., and Vanstone, Scott A. *Faster Point Multiplication on Elliptic Curves with Efficient Endomorphisms*. Crypto 2001, Lecture Notes in Computer Science 2139, pp 190–200. Springer-Verlag.
20. Galbraith, S.D. Personal communication.
21. Galbraith, S.D, McKee, J. and Valenca, P., (2004) *Ordinary Abelian varieties having small embedding degree*. Cryptography ePrint Archive, Report 2004/365, http://eprint.iacr.org/2004/365.
22. Galbraith, S. D., Scott, M. (2008). *Exponentiation in pairing-friendly groups using homomorphisms*. Pairing 2008, Lecture Notes in Computer Science 5209, pp 211–224 Springer-Verlag
23. Hankerson, D., Menezes, A. and Vanstone, S., (2004) *Guide to elliptic curve cryptography*. New York: Springer-Verlag.
24. Hess, F., Smart, N. and Vercauteren F., (2006) *The Eta Pairing revisited*. IEEE Trans. Information Theory, Vol. 52, pp. 4595–4602
25. *IEEE Standard Specification 1363-2000. Annex A.* IEEE, 2000.
26. Joux, A, (2000). *A One Round Protocol for Tripartite DiffieHellman*. Lecture Notes in Computer Science, 1838, pp. 385-394. Springer, Verlag.
27. Kachisa, E., Schaeffer, E.F. and Scott M (2007) *Constructing Brezing-Weng pairing friendly elliptic curves using elements in the cyclotomic field*, Pairing 2008, Lecture Notes in Computer Science 5209, pp 126–135 Springer-Verlag
28. Koblitz, N., Menezes, A. *Pairing-Based Cryptography at High Security Levels* Lecture Notes in Computer Science 3796, pp 13–36, 2005.
29. Lee, E., Lee, H.-S., Park,C.-M. *Efficient and Generalized Pairing Computation on Abelian Varieties* Cryptology ePrint Archive, Report 2008/040, 2008.
30. Lenstra, A. K. *Unbelievable Security Matching AES security using public key systems*, http://www.win.tue.nl/ klenstra/aes_match.pdf
31. Matsuda, S., Kanayama, N., Hess, F. and Okamoto, E., (2007) *Optimised versions of the ate and Twisted ate Pairings*. Cryptology ePrint Archive, Report 2007/013, http://eprint.iacr.org/2007/013

32. Menezes, A., Okamoto, T., and Vanstone, S., (1991) *Reducing Elliptic Curve Logarithms to Logarithms in a Finite Field*. Proceedings of the twenty-third annual ACM symposium on Theory of computing, pp. 80-89. Association for Computing Machinery.
33. Miller, V. S. *Short Programs for functions on Curves*. IBM Thomas J. Watson Research Center (available at http://crypto.stanford.edu/Miller/Miller.ps), 1986.
34. Mitsunari, A., Sakai, R., and Kasahara, M. (2002). *A New Traitor Tracing*. Transactions on Fundamentals of Electronics, Communications and Computer Science, vol. E85-A, no. 2, pp. 481-484.
35. Miyaji, A., Nakabayashi, M. and Takano, S., (2001) *New explicit conditions of elliptic curve traces for FR-reduction*. IEICE Trans. Fundamentals, E84 pp. 1234 - 1243.
36. Naehrig, M. and Barreto, P.S.L.M., (2007) *On compressible pairings and their computation*. Progress in Cryptology AFRICACRYPT 2008. Lecture Notes in Computer Science, 5023, pp 371-388. Springer-Verlag
37. Sakai, R., Ohgishi, K., and Kasahara, M. (2000). *Cryptosystems based on pairing*. Symposium on Cryptography and Information Security (SCIS2000), Okinawa, Japan, Jan. 2628, 2000.
38. Scott, M. (2007). *Implementing Cryptographic Pairings*. Lecture Notes in Computer Science, 4575, pp. 177–196. Springer, Verlag.
39. Scott, M. Benger, N. Charlemagne, M. Dominguez P., L. J., Kachiza, E. (2008). *On the final exponentiation for calculating pairings on ordinary elliptic curves*. Cryptology ePrint Archive, Report 2008/490, http://eprint.iacr.org/2008/490
40. Scott, M. Benger, N. Charlemagne, M. Dominguez P., L. J., Kachiza, E. (2008). *Fast hashing to G2 on pairing friendly curves*. Cryptology ePrint Archive, Report 2008/530, http://eprint.iacr.org/2008/530
41. Shamir, A. (1984). *Identity-Based Cryptosystems and Signature Schemes*. Lecture Notes in Computer Science, 196, pp. 47-53. Springer, Verlag.
42. Silverman, J. H. (1986). *The Arithmetic of Elliptic Curves*. Springer-Verlag.
43. Solinas, J. A. (2003). *ID-Based Digital Signature Algorithms*. ECC 2003. http://www.cacr.math.uwaterloo.ca/conferences/2003/ecc2003/solinas.pdf
44. Tanaka, S. and Nakamula, K., (2007) *More constructing pairing-friendly elliptic curves for cryptography*. Mathematics arXiv Archive, Report 0711.1942, http://arxiv.org/abs/0711.1942
45. Zhao, C-A., and Zhang, F. and Huang, J., (2007) *A Note on the ate Pairing*. Cryptology ePrint Archive, Report 2007/247, http://eprint.iacr.org/2007/247

Appendix

A Counting rational points on the curve

The number of points on an elliptic curve is related to the size of the field as follows $\#E(\mathbb{F}_p) = p + 1 - t$, where t is the trace of the Frobenious of Endormorphism. If α is a root to the characteristic polynomial $X^2 - tX + p$, then the number of points on the curve defined over the extension field is defined as follows, $\#E(\mathbb{F}_{p^k}) = p^k + 1 - \alpha^k - \bar{\alpha}^k$. In *magma* one compute the number of points on $E(\mathbb{F}_{p^k})$ using the following code:

Listing 24. Counting points in the curve over the extension field

```
// Input <- (p,k,t) defining the Elliptic Curve
// Output -> The number of points in the curve
GetnpFpk:= function(p,k,t)
  tr:=[];
  Append(~tr,2);
  Append(~tr,t);
  for i:=2 to k do
    Append(~tr,t*tr[i]-p*tr[i-1]);
  end for;
return (p^k)+1-tr[k+1];
end function;
```

However, when constructing a twist curve the number of rational points varies. For $k = 18$, for example, one can define $q = p^3$ and $T = t^3 - 3pt$ for a sextic twist of these curves [24]. Then, $\#E'(\mathbb{F}_q) = q + 1 - (3F + T)/2$, with $T^2 - 4q = -3F^2$. Solving for F, one can easily calculate the number of points in the twisted curve as in Listing 25.

Listing 25. Counting points in the curve over the twisted curve

```
// Input  <- (p,k,t) defining the Elliptic Curve
// Output -> The number of rational points on the twisted curve
    over Fp^3
GetnpFp3:= function(p,k,t)
  q:=p^3;
  T:=t^3-3*p*t;
  F:=Isqrt((T^2-4*q) div -3);
  np:=q+1-(3*F+T) div 2;
return np;
end function;
```

B T_i selection

The authors used a previously calculated T_i-T_j combination of $\frac{3}{7}x.T_6 + \frac{2}{7}x$. In this section is presented a way to get all the valid combinations.

It has been stated that $T_i \equiv p^i \bmod r$ and $A = a.B + b$. Then, $T_i = a.T_j + b$, similary $T_j \equiv p^j \bmod r$. However, it is also possible to use $(t-1)^i \bmod r$.

One can calculate all the T_i-T_j combinations and compare later. First, it is necessary to generate all the T_i's. There are up to k possible polynomials. One way to obtain them is shown as Listing 26.

Listing 26. Calculating the Ti's

```
Zx<x>:=PolynomialRing(RationalField());
k:=18;
px:=(x^8 + 5*x^7 + 7*x^6 + 37*x^5 + 188*x^4 + 259*x^3 + 343*x^2 +
    1763*x + 2401) / 21;
```

```
rx:=(x^6 + 37*x^3 + 343) div 343;

Poly:=[];
for i:=0 to k-1 do
    Append(~Poly,(px^i) mod rx);
end for;
```

In Listing 26 `RationalField` is being used for defining the $p(x)$ and $r(x)$ polynomials from [27]. One can modify these polynomials, with its respective `k` to calculate to corresponding T_i's polynomials for a different elliptic curve family.

The polynomials for [27] generated from Listing 26 are shown in Listing 27:

Listing 27. T_i polynomials for KSS, k=18

```
> Poly;
[
    1,
    1/7*x^4 + 16/7*x,
    -5/49*x^5 - 87/49*x^2,
    -x^3 - 18,
    3/7*x^4 + 55/7*x,
    -8/49*x^5 - 149/49*x^2,
    -x^3 - 19,
    2/7*x^4 + 39/7*x,
    -3/49*x^5 - 62/49*x^2,
    -1,
    -1/7*x^4 - 16/7*x,
    5/49*x^5 + 87/49*x^2,
    x^3 + 18,
    -3/7*x^4 - 55/7*x,
    8/49*x^5 + 149/49*x^2,
    x^3 + 19,
    -2/7*x^4 - 39/7*x,
    3/49*x^5 + 62/49*x^2
]
```

It is appreciated that the first and 10th polynomials are 1 and -1 respectively. These polynomials are not useful for the computation of the pairing but will be discarded later.

Now the T_i's polynomials are available, one can use the code in Listing 28 to test all the posible rational combinations for *R-ate* pairing. This is continuation of Listing 26.

In this program, trivial combinations will be avoided. Also, negative coefficients (a or b) are being discarded as they will lead to an inefficient pairing. Using negative coefficients mod r will give a *Miller length* greater than $t-1$. The ate pairing uses $t-1$ as the number of *Miller loops* iterations in the ate pairing, see §5. The *Miller loops* iterations in the R-ate pairing, all together, should be shorter than that.

Listing 28. Chosing Ti-Tj valid candidates

```
A:=[]; B:=[];
Ti:=[];Tj:=[]; Tl:=[];
tot:=0;
for i:=1 to k do
    // "No non-polynomials"
    if Poly[i] eq  1 then continue; end if;
    if Poly[i] eq -1 then continue; end if;
    for j:=i+1 to k do
        // "No non-polynomials"
        if Poly[j] eq  1 then continue; end if;
        if Poly[j] eq -1 then continue; end if;

        //Have gt be div by smaller
        // caution w/ negative coeff and same degree
        if Poly[i] ge Poly[j] then
            ii:=i; jj:=j;
            Coefii:=LeadingCoefficient(Poly[ii]);
            if Coefii lt 0 then
                if Degree(Poly[ii]) eq Degree(Poly[jj]) then
                    ii:=j; jj:=i;
                end if;
            end if;
        else
            ii:=j; jj:=i;
        end if;

        // Negative A,B components will give no efficient R-ate
        Coefii:=LeadingCoefficient(Poly[ii]);
        Coefjj:=LeadingCoefficient(Poly[jj]);
        if Coefii * Coefjj lt 0 then continue; end if;
        Bprev:=Poly[ii] mod Poly[jj];
        if LeadingCoefficient(Bprev) lt 0 then continue; end if;

        // No trivial combo.
        Aprev:=Poly[ii] div Poly[jj];
        if Aprev le 1 and Bprev le 1 then continue; end if;

        // if A and B are in Zz is not verified,
        // since it may vary with respect to x.

        Append(~A,Poly[ii] div Poly[jj]);
        Append(~B,Bprev);
        Append(~Ti,ii);
        Append(~Tj,jj);
        tot:=tot+1;
    end for;
end for;
```

This program (Listing 28) generates 4 indexed lists of information: `A`, `B`, `Ti` and `Tj`. They are ordered, and represent the elements of the polynomial

$A = a.B + b$ in the following way: $T_i = A.T_j + B$ (after substracting 1 from `Ti` and `Tj`). These combinations are the only candidates that are non-negative, non-trivial, and depending on its coefficients and the value of x can be $\in \mathbb{Z}$.

However, once again, not all T_i-T_j combinations will result in a bilinear and non-degeneracy pairing [29]. So, in Listing 29 is shown the code used for testing all the possible valid combinations. This is continuation of Listing 28. However, one needs a generic Listing 16 where all *Miller loops* will be called. Note the extra parameters of the `pairing` function.

The reader may notice in Listing 16 the lack of conditions compared with [29, Algorithm 2], since one already knows the actual value of A, B and j. In this case, one is evaluating all the T_i-T_j combinations.

Listing 29. Verifying Bilinearity and non-degeneracy

```
//tot is the number of Ti-Tj candidates. Obtained previously
PrintFile("rateloop",Sprintf("K=%o\tp=%o-bits\tr=%o-bits",k,Round(
    Log(2,p)+0.5),Round(Log(2,r)+0.5)));
ss:=Random(2,r-1);
ssP:=ss*P; ssQ:=ss*Q;
tot2:=0;
for i:=1 to tot do
    print "i=",i;
    Ax:=Evaluate(A[i],xx);
    Bx:=Evaluate(B[i],xx);
    if IsIntegral(Ax) then Ax:= Integers()!Ax; else continue; end
    if;
    if IsIntegral(Bx) then Bx:= Integers()!Bx; else continue; end
    if;
    f:= pairing(ssP,Q,Ax,Bx,Tj[i]-1);
    bilinearity:= f eq pairing(P,ssQ,Ax,Bx,Tj[i]-1);
    if bilinearity eq true then
      print"Bilinear";
      if f ne 1 then
        print"Non-degeneracy";
        print "#",i,",Tj=",Tj[i]-1;
        PrintFile("rateloop",Sprintf("A=%o,␣B=%o,␣Tj=%o",A[i],B[i],
    Tj[i]-1));
      end if;
    end if;
    tot2:=tot2+1;
end for;
PrintFile("rateloop","-.-.-.-.-.-.-.-.-.-.-.-.-.-");
```

New at this point is the inclusion of the `PrintFile` command that stores the many posibilities for the *R-ate* pairing in the file `rateloop` for latter review.

Another new inclusion is the `Evaluate` command. In Listing 26, the polynomials as shown in Listing 27 are $\in \mathbb{Q}$, but their actual value with respect to p and r as used in Listing 3 is needed. This command makes the `A` and `B` be a function with respect to `x` (Listing 26) and assigns the `xx` value (Listing 3).

Since they are used as execution times, one needs to verify if these evaluates $\in \mathbb{N}$. Otherwise, such combination should be discarded.

For the evaluation of the bilinearity, one creates `ss` a `Random` number and creates `ssP` and `ssQ` as `ss*P` and `ss*Q` with P and Q respectively as from Listing 3. And they are used for verifying the bilinearity: $e(P, ssQ) = e(ssP, Q)$. Also the non-degeneracy property is evaluated: $e(P, Q) \neq 1$.

The reader must load additionally to the Listing 3 and 14, the modified Listing 16 for computing the pairing. Listing 8 for the points definition. And Listing 26, 28 and, 29 for the generation of the T_i-T_j combinations.

Once the T_i-T_j combinations are generated. The reader must choose the shortest one from the generated `rateloop` file from Listing 29. Some readers may prefer another combination, and make its own optimizations. For example the 11th combination from the `rateloop` file: $\frac{2}{7}xT_{12} + \frac{3}{7}x$ combination is quite similar, but is left to the reader as an exercise.