Pairing-Friendly Twisted Hessian Curves

No Author Given

No Institute Given

Abstract. This paper introduces concrete constructions of families of pairing-friendly twisted Hessian curves with embedding degree $k \equiv 3, 9, 15 \pmod{18}$ and $k \equiv 0 \pmod{6}$ where $18 \nmid k$. This paper also provides explicit formulas to compute the optimal ate pairing on these families of twisted Hessian curves. Curves generated by our constructions are guaranteed to have twists of degree 3. We also explain a method to eliminate denominators for odd embedding degrees. Our constructions provide alternative embedding degrees for a better adjustment of the base field and the embedding degree in view of the recent attacks on the discrete logarithm problem for elliptic curves over finite fields with composite embedding degree.

Keywords: twisted Hessian curves, pairing-friendly curves, ate pairing, odd embedding degrees, explicit formulas

1 Introduction

Pairings on elliptic curves have various applications in cryptography, ranging from very basic key exchange protocols, such as one round tripartite Diffie–Hellman [30] [31], to complicated protocols, such as identity-based encryption [9] [27] [24] [49]. Pairings also help to improve currently existing protocols, such as signature schemes, to have shortest possible signatures [10].

Curves that are suitable for pairings are called *pairing-friendly curves*, and these curves must satisfy specific properties. It is extremely rare that a randomly generated elliptic curve is pairing-friendly, so pairing-friendly curves have to be generated in a specific way. Examples of famous and commonly used pairing-friendly curves include Barreto-Naehrig curves [6] (BN curves), Barreto-Lynn-Scott curves [5] (BLS curves), and Kachisa-Schaefer-Scott curves [33] (KSS curves).

Performance of pairing-based cryptography relies on elliptic-curve-point arithmetic, computation of line functions and pairing algorithms. A pairing is a bilinear map from two elliptic curve groups $\mathbb{G}_1$ and $\mathbb{G}_2$ to a target group $\mathbb{G}_T$. Therefore, to achieve a good performance, as well as having an efficient pairing algorithm, it is also desirable to have a fast elliptic-curve-point arithmetic in both $\mathbb{G}_1$ and $\mathbb{G}_2$.

Security of pairings depends on the cost of solving discrete logarithm problem (DLP) in the three groups previously mentioned, namely, $\mathbb{G}_1$, $\mathbb{G}_2$, and $\mathbb{G}_T$. Since one can attack pairing-based protocols by attacking any of these three groups, the cost of solving DLP must be sufficiently high in all of these three groups.

1.1 Choice of curves and embedding degrees

One way to improve the performance of pairings is to improve the performance of the underlying point arithmetic. Since pairing computation arises naturally from a geometric representation of Weierstrass curves, most pairing formulas use curves written in Weierstrass form. However, the algorithms for performing point arithmetic on elliptic curves in Weierstrass form are known to be quite slow compared to elliptic curves with special properties allowing them to be written in, for example, Hessian form [54] [32] or Edwards form [19] [8].

Pairings based on Edwards curves, along with examples of pairing-friendly Edwards curves, were proposed by Arene, Lange, Naehrig and Ritzenthaler [1]. They found that the computation of line functions necessary to compute the pairing is much more complicated than if the curves were written in Weierstrass form. In other words, even though Edwards curves allow faster point arithmetic, this gain is outweighed by the slower computation of line functions.

Pairings based on Hessian curves have been considered by Gu, Gu and Xie [25]. They provided a geometric interpretation of the group law on Hessian curves along with an algorithm for computing Tate pairing on elliptic curves in Hessian form. However, no pairing-friendly curves in Hessian form were given.

Bos, Costello and Naehrig [11] investigated the possibility of first using a model of a curve (such as Edwards or Hessian) allowing for fast group operations, then mapping to Weierstrass form for the computation of the pairing. They found that for every elliptic curve E in the BN-12, BLS-12, and KSS-18 families of pairing-friendly curves, if E is isomorphic over $\mathbb{F}_q$ to a curve in Hessian or Edwards form, then it is not isomorphic over $\mathbb{F}_{q^k}$ to a curve in Hessian or Edwards form, where k is the embedding degree, so that the computations in either $\mathbb{G}_1$ or $\mathbb{G}_2$ have to use slower algorithms for group operations on curves in Weierstrass form. Moreover, this idea of using different curve models comes at a cost of at least one conversion between other curve models into the Weierstrass form.

Regardless of which model of elliptic curve was being studied, most of the previous articles on this topic were considering "even" embedding

degrees. One of the main advantages of even embedding degrees is the applicability of a denominator elimination technique in the pairing computation, which is unfortunately not trivially available for odd embedding degrees. Examples of pairing algorithms for curves in Weierstrass form with odd embedding degree include the work by Lin, Zhao, Zhang and Wang in [42], by Mrabet, Guillermin and Ionica in [44], and by Fouotsa, Mrabet and Pecha in [21]. In this paper, we concentrate on curves with embedding degrees divisible by 3, as pairing-friendly elliptic curves in (twisted) Hessian form are equipped with natural twists of degree 3.

Due to recent advances in number field sieve techniques for attacking the discrete logarithm problem for pairing-friendly elliptic curves over finite fields, it is necessary to increase the size of the finite fields for pairing-friendly curves. One way of doing this is by increasing the embedding degree. This means increasing the conventional even embedding degree 12 to 18 or to 24 for higher security levels (see below).

1.2 Attacks on solving DLP over finite fields

The series of advances in attacking the discrete logarithm problem for elliptic curves over finite fields, i.e., [35] [3] [4], have weakened the security of pairing-friendly elliptic curves by dramatically decreasing the security level on the finite field side. In order to retain the security level, either the size of the prime field or the embedding degree (or both) has to be increased. For example, BN-12 (BN curves with embedding degree 12) used to be very popular for 128-bit security level, because a 256-bit prime field maps to a 3072-bit finite field where the security on both the prime field side and the finite field side have a 128-bit security level. A recent article on "Updating key size estimations for parings" by Barbulescu and Duquesne [2] suggests that, for BN curves, the prime field has to be increased to over 400 bits in order to achieve 128-bit security level. Increasing the field size has a penalty of slowing down arithmetic operations which has a huge impact on performance.

1.3 Our contributions

In terms of the performance of pairing-based protocols, the paper [11] by Bos, Costello and Naehrig inspires the question whether it is possible to construct pairing-friendly curves that could be represented in the same curve models, preferably with fast point arithmetic and fast computation of line functions, for both $\mathbb{G}_1$ and $\mathbb{G}_2$,

Twisted Hessian curves with faster point arithmetic have recently been proposed by Bernstein, Chuengsatiansup, Kohel and Lange [7]. This leads to questions whether the newer point arithmetic would also lead to fast formulas for computing line functions, and whether these formulas would be applicable to computing pairings on twisted Hessian curves.

We are the first, to our knowledge, to present concrete constructions of a pairing on curves in twisted Hessian form with odd embedding degree. Note in particular that among the popular pairing-friendly curve families, BN-12 and KSS-18 have embedding degrees 12 and 18 respectively, which gives a large difference in field size. Our results provide an alternative of embedding degree 15, in an attempt to give a better balance between the size of the prime field and of the embedding degree. For higher security levels, our constructions allow, for example, embedding degrees 21 or 33.

The setup of this paper is as follows: in Section 2, we recall Miller's algorithm to compute the optimal ate pairing on elliptic curves in twisted Hessian form. In Section 3, we state constructions of families of pairing-friendly elliptic curves given in [22] that can be written in twisted Hessian form and show the conversion. In Section 4, we observe that the efficiency of computing line functions on curves in twisted Hessian form is similar to the Weierstrass case and explain how to compute them. In Section 5, we state explicit formulas for the computation of the optimal ate pairing on twisted Hessian curves based on the state-of-the-art point arithmetic formulas. In Section 6, we compare our results to previous works. In Appendix A, we provide full Magma codes for a toy example of pairing-friendly twisted Hessian curves with embedding degree $k = 21$.

2 Background on Pairings

Let E be an elliptic curve defined over a finite field $\mathbb{F}_q$ where q is a prime. Let r be the largest prime factor of $n = \#E(\mathbb{F}_q) = q + 1 - t$ where t is a trace of Frobenius. The *embedding degree* with respect to r is defined to be the smallest positive integer k such that $r|(q^k - 1)$. Let $\mu_r \subseteq \mathbb{F}_{q^k}^*$ be the group of r-th roots of unity. For $m \in \mathbb{Z}$ and $P \in E[r]$, let $f_{m,P}$ be a function with divisor $\mathrm{div}(f_{m,P}) = m(P) - ([m]P) - (m-1)(\mathcal{O})$, where $\mathcal{O}$ denotes the point at infinity on E. The reduced Tate pairing is defined as

$$\begin{aligned} \tau_r : E(\mathbb{F}_{q^k})[r] \times E(\mathbb{F}_{q^k})/[r]E(\mathbb{F}_{q^k}) &\longrightarrow \mu_r \\ (P, Q) &\mapsto f_{r,P}(Q)^{\frac{q^k-1}{r}}. \end{aligned}$$

We address the computation of the reduced Tate pairing restricted to $\mathbb{G}_1 \times \mathbb{G}_2$, where

$$\mathbb{G}_1 = E[r] \cap \ker(\phi_q - [1]) \text{ and } \mathbb{G}_2 = E[r] \cap \ker(\phi_q - [q]) \subseteq E(\mathbb{F}_{q^k}).$$

Here ϕ_q denotes the q-power Frobenius morphism on E. We denote the restriction of τ_r to $\mathbb{G}_1 \times \mathbb{G}_2$ by

$$e_r : \mathbb{G}_1 \times \mathbb{G}_2 \longrightarrow \mu_r.$$

Let $T = t-1$. We define the *ate pairing* a_T by restricting the Tate pairing to $\mathbb{G}_2 \times \mathbb{G}_1$ so that

$$\begin{aligned} a_T : \mathbb{G}_2 \times \mathbb{G}_1 &\longrightarrow \mu_r \\ (Q, P) &\mapsto f_{T,Q}(P)^{\frac{q^k-1}{r}}. \end{aligned}$$

Note that in addition to the arguments P and Q being switched, the subscript r (i.e. the number of loops) is also changed to T.

Miller's algorithm for computing pairings works as follows. Let m be an integer expressed in a binary format as $(m_{\ell-1}, \ldots, m_1, m_0)_2$. First, a rational function $f_{m,P}$ associated to a point P and to m is constructed. Then, this function is evaluated at a point Q. Miller's algorithm computes $f_{m,P}$ in the double-and-add manner and uses the following relation

$$f_{i+j,P} = f_{i,P} f_{j,P} \frac{l}{v},$$

where l is a line through points iP and jP, and v is a line through points $(i+j)P$ and $-(i+j)P$.

Constructing the line l can be done by substituting x and y coordinates of points iP and jP into the equation $y = mx + b$ and solving for m and b. For Weierstrass curves, the line v is simply a vertical line through $(i+j)P$. Therefore, the line v is of the form $x = c$ where c is the x-coordinate of the point $(i+j)P$. Note that the points $(i+j)P$ and $-(i+j)P$ have the same x-coordinate. For (twisted) Hessian curves, however, the line v is not a vertical line and thus has to be constructed in a similar way as the line l, i.e., solving for m and b of $y = mx + b$ using points $(i+j)P$ and $-(i+j)P$.

Algorithm 1 shows Miller's algorithm. We denote $\ell_{2R} = l_{2R}/v_{2R}$ where l_{2R} is the tangent line at point R while v_{2R} is the line passing through points $2R$ and its negation $-2R$. We denote $\ell_{R,P} = l_{R,P}/v_{R,P}$ where $l_{R,P}$ is the line passing through points R and P while $v_{R,P}$ is the line passing through points $R + P$ and its negation $-(R+P)$.

Algorithm 1 Miller's algorithm

Require: $m = (m_{\ell-1}, \ldots, m_1, m_0)_2$ and $P, Q \in E[r]$ with $P \neq Q$

1: Initialize $R = P$ and $f = 1$
2: **for** $i := \ell - 2$ **down to** 0 **do**
3: $\quad f \leftarrow f^2 \cdot \ell_{2R}(Q)$
4: $\quad R \leftarrow 2R$
5: $\quad$ **if** $m_i = 1$ **then**
6: $\quad\quad f \leftarrow f \cdot \ell_{R,P}(Q)$
7: $\quad\quad R \leftarrow R + P$
8: $f \leftarrow f^{(q^k-1)/r}$

3 Curve constructions

Even though every elliptic curve can be written in a Weierstrass form, only those that contain points of order 3 can be written in a (twisted) Hessian form. Almost all methods to generate pairing-friendly curves are for generating pairing-friendly Weierstrass curves. Therefore, we will generate pairing-friendly (twisted) Hessian curves by searching through constructions of pairing-friendly Weierstrass curves for curves that have points of order 3, then convert those curves into the (twisted) Hessian form.

The succeeding subsections explain how to obtain pairing-friendly twisted Hessian curves. We begin by explaining the twists of curves, specifically the twists of degree 3 as all of our constructions allow this type of twist. Then, we give concrete constructions to generate pairing-friendly Weierstrass curves that are guaranteed to have points of order 3 so that these curves can be converted into the (twisted) Hessian form. Finally, we show an explicit transformation from Weierstrass into twisted Hessian form by explicitly stating the formulas.

3.1 Twists of degree 3

Let E and E' be elliptic curves over $\mathbb{F}_q$. We call E' a *twist* of E if E and E' are isomorphic over some field extension of $\mathbb{F}_q$. More precisely, E' is a *degree-d twist* of E if they are isomorphic over a degree d extension and not over any smaller field.

Twists of curves speed up pairing computations by allowing arithmetic to take place in a subfield instead of the full extension field. That is, instead of working over the full extension field $\mathbb{F}_{q^k}$, a degree-d twist makes it possible to work over a subfield $\mathbb{F}_{q^{k/d}}$ (given that d divides k). The larger the degree of twist, the smaller the subfield can be. Thus, it is desirable

to use the highest degree of twist available. However, the only possible degrees of twist are $d \in \{2, 3, 4, 6\}$ (see, e.g., [53]).

To use a degree-d twist of a curve over $\mathbb{F}_{q^k}$, the degree d must divide k. For the case $k \equiv 3, 9, 15 \pmod{18}$, the only possible twist degree is $d = 3$. For the case $k \equiv 0 \pmod 6$ where $18 \nmid k$, even though a twist of degree 6 is also possible, we are only interested in degree 3 twists as our curves must have a point of order 3.

Recall that we focus only on curves for which both the curves and their twists contain points of order 3. To express twists of curves in the Hessian form, those twists must also contain points of order 3. To check whether the twisted curve E' of E contains points of order 3 or not, we use the formulas in [26] which state the number of points on twisted curves. The formulas for calculating the number of points on twisted curves always come in pairs. For example, the formulas for $d = 3$ stated in [26] are as follows:

$$\begin{aligned} \#E'(\mathbb{F}_q) &= q + 1 - (3f - t)/2 \qquad &with \quad t^2 - 4q = -3f^2, \\ \#E'(\mathbb{F}_q) &= q + 1 - (-3f - t)/2 \qquad &with \quad t^2 - 4q = -3f^2. \end{aligned}$$

To determine the right twist, we use the fact that $\#E'(\mathbb{F}_q)$ must also be divisible by r (recall that r was the largest prime factor of $\#E(\mathbb{F}_q)$); exactly one of the two possible twists satisfies this condition.

3.2 Generating curves

Recall that E is an elliptic curve defined over a finite field $\mathbb{F}_q$ where q is prime, and r is the largest prime factor of $\#E(\mathbb{F}_q)$. The embedding degree k is the smallest integer k such that $r|q^k - 1$. To construct parametric families of pairing-friendly curves for given polynomials $q(x)$ and $r(x)$, one needs to search for some integer x_0 such that $q(x_0)$ is prime. Then, there is an elliptic curve E defined over $\mathbb{F}_{q(x_0)}$ with a subgroup of order $r(x_0)$ and embedding degree k with respect to $r(x_0)$.

Cyclotomic families are families of curves where the underlying field K is a cyclotomic field, the size r of the largest prime-order subgroup of the group of $\mathbb{F}_q$-points is a cyclotomic polynomial, and the field K contains $\sqrt{-D}$ for some small discriminant D. We searched through [22] and found three cyclotomic-family constructions that satisfy our point-of-order-3 condition. These constructions are based on a cyclotomic field containing a cube root of unity, i.e., fields contain $\sqrt{-3}$. Therefore, we choose the discriminant $D = 3$.

The following constructions are for generating pairing-friendly Weierstrass curves which can be converted into twisted Hessian curves (see [7] Section 5). Note that twists of these curves (see Section 3.1) are also expressible in the twisted Hessian form. We categorized these constructions by embedding degrees. Note that $q(x)$ is a prime field, $r(x)$ is a subgroup of $E(\mathbb{F}_{q(x)})$ which may not have prime order, and $t(x)$ is a trace of Frobenius defined as $t(x) = q(x) + 1 - \#E(\mathbb{F}_{q(x)})$ where $\#E(\mathbb{F}_{q(x)})$ denotes the number of points on $E(\mathbb{F}_{q(x)})$. We denote $\Phi_n(x)$ the cyclotomic polynomial of degree n.

Construction 1: $k \equiv 3 \pmod{18}$. This construction follows Construction 6.6 in [22] under the first subcase where $k \equiv 3 \pmod 6$. Pairing-friendly curves with embedding degree $k \equiv 3 \pmod{18}$ can be constructed using the following polynomials:

$$\begin{aligned} r(x) &= \Phi_{2k}(x), \\ t(x) &= x^{k/3+1} + 1, \\ q(x) &= \frac{1}{3}(x^2 - x + 1)(x^{2k/3} - x^{k/3} + 1) + x^{k/3+1}. \end{aligned}$$

Note that for this construction, the resulting curves and their twists all have points of order 3. However, there is no such x_0 for which both $q(x_0)$ and $r(x_0)$ are prime. This means that $r(x_0)$ factors, and the largest prime-order subgroup of $E(\mathbb{F}_q)$ actually has less than $r(x_0)$ elements. Recall that the discriminant $D = 3$. This implies in particular that the curves are defined by an equation of the form $y^2 = x^3 + b$. This means that the possible twists are cubic ($d = 3$) and sextic ($d = 6$) twists. It is obvious that this construction allows only twist of degree $d = 3$ because $6 \nmid k$.

Construction 2: $k \equiv 9, 15 \pmod{18}$. This construction follows Construction 6.6 in [22] under the second subcase where $k \equiv 3 \pmod 6$. Pairing-friendly curves having embedding degree $k \equiv 9, 15 \pmod{18}$ can be constructed using the following polynomials:

$$\begin{aligned} r(x) &= \Phi_{2k}(x), \\ t(x) &= -x^{k/3+1} + x + 1, \\ q(x) &= \frac{1}{3}(x + 1)^2(x^{2k/3} - x^{k/3} + 1) - x^{2k/3+1}. \end{aligned}$$

Note that similar remarks also apply to this construction. The resulting curves have points of order 3 and also on their twists. There is no such

x_0 for this construction to achieve both $q(x_0)$ and $r(x_0)$ being prime. The parameter $r(x_0)$ factors, and the largest prime-factor subgroup is smaller than $r(x_0)$. Having discriminant $D = 3$ means that the possible twists are cubic ($d = 3$) and sextic ($d = 6$) twists. However, this construction allows only twist of degree $d = 3$ since $6 \nmid k$.

Construction 3: $k \equiv 0 \pmod 6$ and $18 \nmid k$. This construction follows the last case of Construction 6.6 in [22]. Pairing-friendly curves with embedding degree $k \equiv 0 \pmod 6$ where $18 \nmid k$ can be constructed using the following polynomials:

$$\begin{aligned} r(x) &= \Phi_k(x), \\ t(x) &= x + 1, \\ q(x) &= \frac{1}{3}(x-1)^2(x^{k/3} - x^{k/6} + 1) + x. \end{aligned}$$

Note that for this construction, the resulting curves and their twists all have points of order 3. There also exists x_0 such that both $q(x_0)$ and $r(x_0)$ are prime. Recall that this construction also has discriminant $D = 3$. This means that the possible twists are cubic ($d = 3$) and sextic ($d = 6$) twists. Even though this construction allows curves with embedding degree 6 (because $6 \mid k$), only curves with embedding degree 3 can be expressed in the twisted Hessian form for both the original curves and their twists.

Sage scripts. We provide Sage scripts (Figure 1) for constructing pairing-friendly Weierstrass curves with embedding degree $k = 21$ (see Construction 1 above) which can be converted to twisted Hessian curves. Note that the scripts also work for different embedding degrees $k \equiv 3 \pmod{18}$ by changing `k = 21` to the desired embedding degrees. For other constructions, in addition to changing the embedding degree, the polynomials `rx, r, t, q` also need to be replaced by the corresponding ones. For auxiliary functions in `util.sage`, please refer to Appendix B.

Figure 2 shows the output of pairing-friendly Weierstrass curves generated by the Sage scripts in Figure 1. Note that the purpose of these scripts is to provide a concrete method to generate pairing-friendly Weierstrass curves that can be converted into the twisted Hessian form, and not to propose a secure curve at any particular security levels. We will later on use the parameters in the output from the scripts for our toy example. Therefore, we use rather small prime field size for the purpose of checking correctness of the algorithm.

```
load("util.sage")

k = 21
x = 2^10
D = 3

num = 1
while num > 0 :

  # k = 3 mod 18 #
  rx = cyclotomic_polynomial(2*k)
  r = rx(x)
  t = x^(k/3 + 1) + 1
  q = (1/3) * (x^2 - x + 1) * (x^(2*k/3) - x^(k/3) + 1) + x^(k/3 + 1)

  if (q.denominator() == 1):
    if (ZZ(q).is_prime() and ZZ(r).is_prime()):

      deg = k/3
      tk = tn(t,deg,q)
      f2 = (4*q^deg - tk^2)/3
      if f2.is_square():
        f = sqrt(f2)
        t1 = (q^deg+1-(+3*f-tk)/2)
        t2 = (q^deg+1-(-3*f-tk)/2)
      else:
        t1 = 1
        t2 = 1

      if (q+1-t)%3 != 0 or (t1%3 != 0 and t2%3 != 0):
        x += 1
        continue

      fq = GF(q)

      y = sqrt((4*q - t^2) / D)
      a = 0
      b = find_b(q,t/2,y/2)

      fq_a = fq(a)
      fq_b = fq(b)

      E = EllipticCurve(fq,[fq_a,fq_b])
      nE = E.cardinality()

      if nE != (q+1-t) :
        x += 1
        continue

      print "x =",x
      print "t =",t
      print "q =",q
      print "r =",r
      print "a,b =",a,b

      num -= 1
  x += 1
```

Fig. 1. Sage scripts to generate pairing-friendly Weierstrass curves of embedding degree $k = 21$ that can be converted into twisted Hessian form and allow twist of degree 3.

```
x = 5054
t = 42567868144026523521756069913 7
q = 60388831224640627688578323697279079263669799534119323634669
r = 27778498887314511245242191684643503527185407 1
a,b = 0 144
```

Fig. 2. The output of the Sage scripts in Figure 1 to generate pairing-friendly Weierstrass curves where 'x' denotes an integer that results in $q(x) = \frac{1}{3}(x^2 - x + 1)(x^{2k/3} - x^{k/3} + 1) + x^{k/3+1}$ being prime; 't' denotes a trace of Frobenius; 'q' denotes a prime field; 'r' denotes a subgroup of the resulting curve; and 'a,b' denotes the parameters of the Weierstrass curve $y^2 = x^3 + ax + b$.

3.3 Converting Weierstrass to twisted Hessian curves

To convert pairing-friendly Weierstrass curves that are guaranteed to have points of order 3 into the twisted Hessian form, we follow the explanation in [7]. Specifically, the proofs of Theorem 5.2 and Theorem 5.3 in [7] describe the isomorphism from Weierstrass curves to twisted Hessian curves via triangular curves using a series of substitutions.

To be more precise, the proofs start by giving a point of order 3 $P_3 = (u_3, v_3)$ and writing a curve (defined over $\mathbb{F}_q$) in a long Weierstrass form

$$v^2 + e_1uv + e_3v = u^3 + e_2u^2 + e_4u + e_6.$$

Note however that in our case, we consider curves of the form $y^2 = x^3 + ax + b$. Thus, the curve parameters $e_1 = e_3 = e_2 = e_4 = 0$, and $e_6 = b$. Therefore, we start with the equation

$$v^2 = u^3 + b.$$

Next step is to substitute $u = x + u_3$ and $v = t + v_3$ to obtain the curve of the form

$$t^2 + c_1xt + c_3t = x^3 + c_2x^2 + c_4x + c_6.$$

In our case, we have

$$t^2 + (2v_3)t = x^3 + (3u_3)x^2 + (3u_3^2)x + (b - v_3^2).$$

This means that

$$c_3 = 2v_3, \qquad c_2 = 3u_3, \qquad c_4 = 3u_3^2, \qquad c_6 = b - v_3^2.$$

Then, another substitution $t = y + \lambda x$ for $\lambda \in \mathbb{F}_q$ is performed, which leads to a curve of the form

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6,$$

where $a_2 = a_4 = a_6 = 0$ and hence in triangular form $y^2 + dxy + ay = x^3$. In our case, we obtain

$$y^2 + (2\lambda)xy + c_3 y = x^3 + (c_2 - \lambda^2)x^2 + (c_4 - c_3\lambda)x + c_6.$$

This means that

$$\begin{gathered} a_1 = 2\lambda, \qquad a_3 = c_3, \\ a_2 = c_2 - \lambda^2 = 0, \qquad a_4 = c_4 - c_3\lambda = 0, \qquad a_6 = c_6 = 0, \end{gathered}$$

which implies $\lambda = c_4/c_3$.

We rename the parameters to $d = a_1$ and $a = a_3$. The curve parameters d' and a' for twisted Hessian curve

$$H : a'X^3 + Y^3 + Z^3 = d'XYZ$$

are defined as

$$a' = d^3 - 27a, \qquad d' = 3d.$$

By making the substitution $x = U/W$ and $y = V/W$, we homogenize the above form of the triangular curve $y^2 + dxy + ay = x^3$ to

$$VW(V + dU + aW) = U^3.$$

Once we obtain the parameters a' and d' for twisted Hessian curve, it remains to map the coordinates $(U : V : W)$ of the triangular curve to coordinates $(X : Y : Z)$ of the twisted Hessian curve. This map $\phi(U : V : W) = (X : Y : Z)$ is defined as:

$$\begin{aligned} X &= U, \\ Y &= \omega(V + dU + AW) - \omega^2 V - aW, \\ Z &= \omega^2(V + dU + aW) - \omega V - aW, \end{aligned}$$

where $\omega \in \mathbb{F}_q$, $\omega^3 = 1$ and $\omega \neq 1$.

To summarize, to compute related parameters (before the map ϕ) we need to perform the following computation:

$$\begin{array}{lll} c_3 = 2v_3, & c_4 = 3u_3^2, & \lambda = c_3/c_4. \\ d = 2\lambda, & a = c_3. & \\ d' = 3d, & a' = d^3 - 27a. & \end{array}$$

Next, given a point $(U : V : W)$, a point of order 3 $(u_3, v_3, 1)$, and related parameters a, d, λ, ω, we compute:

$$U = U - u_3, \qquad V = V - v_3 - \lambda(U - u_3), \qquad W = W,$$
$$A = \omega(V + dU + aW), \qquad B = \omega V, \qquad C = aW.$$

Then, the map ϕ is computed as:

$$X = U,$$
$$Y = \omega(V + dU + aW) - \omega^2 V - aW = A - \omega B - C,$$
$$Z = \omega^2(V + dU + aW) - \omega V - aW = \omega A - B - C.$$

Note that in our case, the parameter d' is zero. To convert the coordinates, we also need to compute $\omega \in \mathbb{F}_q$ where $\omega^3 = 1$ and $\omega \neq 1$. Let $\mathbf{m}, \mathbf{s}$ and $\mathbf{m}_c$ denote field multiplication, field squaring and field multiplication by a small constant respectively. Therefore, to compute related parameters (before the map ϕ) this step costs $\mathbf{m}+2\mathbf{s}+5\mathbf{m}_c$ plus one inversion for $\lambda = c_4/c_3$ and one cube root computation for computing ω. The cost for computing the map ϕ is $8\mathbf{m}$. Thus, the total cost for the whole conversion is $9\mathbf{m} + 2\mathbf{s} + 5\mathbf{m}_c$ plus one inversion and one cube root computation.

Sage scripts. We provide Sage scripts (Figure 3) for converting Weierstrass curves into twisted Hessian form. The function `w2h` in the scripts takes as inputs the prime field `fq` and Weierstrass curve parameters `a` and `b`. It computes the necessary parameters for converting the Weierstrass curve with the given input to the corresponding twisted Hessian curve. For auxiliary functions in `util.sage`, please refer to Appendix B.

Specifically, the function first computes a point of order 3 $P_3 = (u_3, v_3)$. This is done by randomly choosing a point on the curve, checking that it is not the neutral element, and multiplying by a correct cofactor. Next, it computes parameters for converting curves in Weierstrass form to triangular form, namely, `c3`, `c4`, `lambda`, `a1`, `a3`, `td` and `ta` which correspond to the above mentioned $c_3, c_4, \lambda, d, a, d'$ and a' respectively. Then, it computes parameters for converting curves in triangular form to twisted Hessian form, namely, `ha` and `hd` which correspond to the above mentioned a' and d'. Finally, it computes `omega` which is a cube root element in $\mathbb{F}_q$. The function then outputs: the coordinates of a point of order 3 $P_3 = (u_3, v_3)$; parameters d and a for the conversion; twisted Hessian curve parameters a' and d'; and a cube root element ω.

Figure 4 shows an example of the output of the function `w2h` of Sage scripts shown in Figure 3 which uses the output of the previous Sage

```
load("util.sage")

def w2h(fq,a,b):

  E = EllipticCurve(fq,[a,b])
  nE = E.cardinality()

  h = nE
  while (h%3 == 0):
    h /= 3

  p3 = E([0,1,0])
  while (p3.is_zero()):
    p3 = E.random_element()
    p3 = ZZ(h)*p3
    while not(3*p3).is_zero():
      p3 = 3*p3

  u3 = p3[0]
  v3 = p3[1]

  ################################

  # Weierstrass -> Triangular
  # W: v^2 = u^3 + e6
  # T: y^2 + dxy + ay = x^3

  c3 = 2*v3
  c4 = 3*u3^2

  lampda = c4/c3

  a1 = 2*lampda
  a3 = c3

  td = fq(a1)
  ta = fq(a3)

  ################################

  # Triangular -> Twisted Hessian
  # H: a'X^3 + Y^3 + Z^3 = d'XYZ

  ha = (td^3 - 27*ta)
  hd = 3*td

  omega = findCubeRoot(q)

  print "u3 =",u3
  print "v3 =",v3
  print "d  =",td
  print "a  =",ta
  print "a' =",ha
  print "d' =",hd
  print "w  =",omega

#####################################################

r = 27778498887314511245242191684643503527185407 1
q = 60388831224640627688578323697279079263669799534119323634669
a = 0
b = 144

fq = GF(q)
w2h(fq,fq(a),fq(b))
```

Fig. 3. Sage scripts to convert Weierstrass curves $E/\mathbb{F}_q : y^2 = x^3 + ax + b$ into twisted Hessian form $a'X^3 + Y^3 + Z^3 = d'XYZ$.

scripts in Figure 1 for input r, q, a, b, i.e., a pairing-friendly Weierstrass curve $E/\mathbb{F}_q : y^2 = x^3 + ax + b$ with a subgroup of prime order r. Note that we use `p3 = E.random_element()` in our scripts. This means that it is possible that the scripts give different output when run multiple times.

```
u3 = 0
v3 = 12
d  = 0
a  = 24
a' = 603888312246406276885783236972790792636697995341193236340 21
d' = 0
w  = 17923080803972475283541924324100117212007204172538782666
```

Fig. 4. The output of the Sage scripts in Figure 3 to convert Weierstrass curve into twisted Hessian form where 'u3,v3' denotes point-of-order-3 $P_3 = (u_3, v_3)$; 'd' and 'a' denote the parameters d and a which are used for the conversion; 'a′' and 'd′' denote parameters for twisted Hessian curve $a'X^3 + Y^3 + Z^3 = d'XYZ$; and 'w' denotes a cube root element ω.

4 Computation of line functions

The efficiency of pairings substantially relies on the computation of the line functions (as denoted by $\ell_{2R} = l_{2R}/v_{2R}$ and $\ell_{R,P} = l_{R,P}/v_{R,P}$ in Algorithm 1). Recall that an equation of a line is simply $y = mx + b$. Therefore, constructing the line l passing through points P and R is done by substituting the coordinates x and y of points P and R then solving for m and b. Once the line is constructed, it is then evaluated at another point Q

Specifically, to compute the line $l_{R,P}$ passing through points $R = (x_R, y_R)$ and $P = (x_P, y_P)$ where $R \neq P$ and evaluated at a point $Q = (x_Q, y_Q)$, we perform the following computations:

$$\begin{aligned} m_{l_{R,P}} &= (y_P - y_R)/(x_P - x_R), \\ b_{l_{R,P}} &= y_P - m_{l_{R,P}} \cdot x_P, \\ l_{R,P} &= y_Q - (m_{l_{R,P}} \cdot x_Q + b_{l_{R,P}}). \end{aligned}$$

Note that the line passing through points P and R is also passing through point $-(P + R)$. Using any two out of these three points would result in the same line l.

To compute the line l_{2R} tangent at the point R, we use the fact that this line also intersects the curve at the point $-(2R)$. Therefore, we construct a line passing through the points R and $-(2R)$. Recall that for

curves in twisted Hessian forms, a negation of a point $2R = (x_{2R}, y_{2R})$ is $-2R = (x_{2R}/y_{2R}, 1/y_{2R})$. Let $R = (x_R, y_R)$ and $-2R = (x_{2R}/y_{2R}, 1/y_{2R})$. The line l tangent at the point R evaluated at the point Q is computed as follows:

$$\begin{aligned} m_{l_{2R}} &= \left(\frac{1}{y_{2R}} - x_R\right) / \left(\frac{x_{2R}}{y_{2R}} - x_R\right), \\ b_{l_{2R}} &= y_R - m_{l_{2R}} \cdot x_R, \\ l_{2R} &= y_Q - (m_{l_{2R}} \cdot x_Q + b_{l_{2R}}). \end{aligned}$$

To construct the line v passing through points $(P+R)$ and its negation $-(P+R)$, recall that for Weierstrass curves, this line v is simply a vertical line and thus of the form $y = c$. However, for the (twisted) Hessian curves, the line v is not a vertical line and thus has to be constructed in a similar way as the line l, i.e., solving for the m and b of $y = mx + b$ using points $P + R$ and $-(P + R)$. Note also that the line passing through the point and its negation also passes through the point $(0, -1)$, the neutral point of twisted Hessian forms.

Let $(P+R) = (x_{RP}, y_{RP})$. By using the neutral point $(0, -1)$, the line $v_{R,P}$ is computed as follows:

$$\begin{aligned} m_{v_{R,P}} &= (y_{RP} + 1)/x_{RP}, \\ b_{v_{R,P}} &= -1, \\ v_{R,P} &= y_Q - (m_{v_{R,P}} \cdot x_Q + b_{v_{R,P}}). \end{aligned}$$

The advantage of using the neutral point is that it saves some computations and $b_{v_{R,P}}$ becomes -1 due to a multiplication by zero. For the line v_{2R}, the calculation is similar to the line $v_{R,P}$, namely, simply replace $(P + R)$ by $(2R) = (x_{2R}, y_{2R})$ and compute as above.

Recall that the lines $v_{R,P}$ and v_{2R} appear as denominators, namely, we have to compute l_{2R}/v_{2R} and $l_{R,P}/v_{R,P}$. Note that for even embedding degrees, this line v lies in a subfield of $\mathbb{F}_{q^k}$ which becomes 1 after computing the final exponentiation (line 8 in Algorithm 1 in Section 2). Therefore, for even embedding degrees, the line v can be omitted. However, for odd embedding degrees, we cannot neglect the denominator v. Because computing the inversion is expensive, we first apply the "denominator elimination" technique to optimize this inversion.

We follow a similar technique as described in [42] and [44] by rewriting an inversion into a fraction for which the denominator lies in a subfield. Define a field extension $\mathbb{F}_{q^k}$ of a finite field $\mathbb{F}_q$ and suppose that $x, y \in \mathbb{F}_{q^k}$.

Observe that

$$\frac{1}{x-y} = \frac{x^2+xy+y^2}{x^3-y^3}.$$

As x^3 and y^3 lie in a subfield of $\mathbb{F}_{q^k}$, we have that

$$(x^3-y^3)^{\frac{q^k-1}{r}} = 1.$$

Hence, after the final exponentiation in the computation of the pairing, this factor does not appear. This means that we can ignore the computation of the denominator x^3-y^3. Therefore, instead of computing $\frac{1}{x-y}$, we compute x^2+xy+y^2. The cost of division by $x-y$ then becomes the cost of multiplication by x^2+xy+y^2.

Note that in the previously presented formulas and explanations, we used points represented in affine coordinates, e.g., $P=(x,y)$. However, to avoid inversion, we use points represented in projective coordinates, e.g., $P=(X:Y:Z)$ where $x=X/Z$ and $y=Y/Z$. The formulas given in the following section use projective coordinates.

5 Explicit formulas

Recall the twisted Hessian curve equation in projective coordinates

$$\mathcal{H}: a'X^3+Y^3+Z^3 = d'XYZ.$$

This section shows formulas for computing point doubling, point addition, and line functions associated to these operations. These formulas work under an assumption that the curve parameter $d'=0$.

We denote operations in the base field, namely, field multiplication, field squaring and field multiplication by the curve parameter a' by $\mathbf{m}, \mathbf{s}, \mathbf{m}_a$ respectively. We denote the embedding degree by k and the twist degree by $d=3$.

The following formulas work for both Tate and ate pairings. We denote a point $P=(X_1,Y_1,1)$ and a point $R=(X_2,Y_2,Z_2)$ on $E/\mathbb{F}_q$ and a point $Q=(X_Q,Y_Q,1)$ on $E/\mathbb{F}_{q^k}$ where $Y_Q \in \mathbb{F}_{q^{k/d}}$. Note the points P and Q do not change throughout the pairing computation and thus can be preprocessed, namely, we assume that the $X_1=Z_Q=1$.

5.1 Doubling

First, we show formulas for point doubling. These formulas work under the condition that the curve parameter $d'=0$, which is the case for

pairing-friendly twisted Hessian generated by our constructions. Let $R = (X_1, Y_1, Z_1)$. The following formulas compute the doubling of a point $R = 2R = (X_3, Y_3, Z_3)$.

$$T = Y_1^2; \qquad A = Y_1 \cdot T; \qquad S = Z_1^2; \qquad B = Z_1 \cdot S;$$
$$X_3 = X_1 \cdot (A - B); \quad Y_3 = -Z_1 \cdot (2A + B); \quad Z_3 = Y_1 \cdot (A + 2B).$$

The above formulas for computing point doubling cost $5\mathbf{m} + 2\mathbf{s}$.

Let $Q = (X_Q, Y_Q, 1)$ be the point at which we evaluate. Let R and $2R$ be as defined above for point doubling. The associated line function for doubling to compute $l_{2R}(Q)$ is as follows

$$l_1 = aX_1^2X_Q + T \cdot Y_Q + S; \qquad l_a = X_3 \cdot Y_Q + X_3;$$
$$l_b = X_Q \cdot (Y_3 + Z_3); \qquad l_c = l_a^2 + l_b \cdot (l_a + l_b); \qquad l = l_1 \cdot l_c.$$

To compute the line function for doubling, it costs $k(\frac{14}{3}\mathbf{m} + \frac{1}{3}\mathbf{s}) + \mathbf{s} + \mathbf{m}_a$. Therefore, the total number of operations for the "doubling step" (line 3 in Algorithm 1) in the pairing computation is $k(\frac{14}{3}\mathbf{m} + \frac{1}{3}\mathbf{s}) + 5\mathbf{m} + 3\mathbf{s} + \mathbf{m}_a$.

5.2 Addition

Let $P = (X_1, Y_1, 1)$ and $R = (X_2, Y_2, Z_2)$. The following formulas compute the addition of points $P + R = (X_1, Y_1, 1) + (X_2, Y_2, Z_2) = (X_3, Y_3, Z_3)$.

$$A = X_1 \cdot Z_2; \qquad C = Y_1 \cdot X_2; \qquad D = Y_1 \cdot Y_2; \qquad F = aX_1 \cdot X_2;$$
$$G = (D + Z_2) \cdot (A - C); \quad H = (D - Z_2) \cdot (A + C);$$
$$J = (D + F) \cdot (A - Y_2); \quad K = (D - F) \cdot (A + Y_2);$$
$$X_3 = G - H; \qquad Y_3 = K - J;$$
$$Z_3 = J + K - G - H - 2(Z_2 - F) \cdot (C + Y_2).$$

The above formulas for computing point addition cost $9\mathbf{m} + \mathbf{m}_a$.

Let $Q = (X_Q, Y_Q, 1)$ be the point at which we evaluate. Let $P, R, (P + R)$ be as defined above for point addition. The associated line function for addition to compute $l_{R,P}$ is as follows

$$l_1 = (Y_1 \cdot Z_2 - Y_2) \cdot (X_1 - X_Q) + (Y_Q - Y_1) \cdot (X_1 \cdot Z_2 - X_2);$$
$$l_a = Y_Q \cdot X_3 + X_3; \quad l_b = X_Q \cdot (Y_3 + Z_3); \quad l_c = l_a^2 + l_a \cdot (l_b + l_b);$$
$$l = l_1 \cdot l_c.$$

To compute the line function for addition, it costs $k(\frac{14}{3}\mathbf{m} + \frac{1}{3}\mathbf{s}) + 2\mathbf{m}$. Therefore, the total number of operations for the "addition step" (line 6 in Algorithm 1) in the pairing computation is $k(\frac{14}{3}\mathbf{m} + \frac{1}{3}\mathbf{s}) + 11\mathbf{m} + \mathbf{m}_a$.

6 Comparison

We would like to emphasize that:

- Most of the previous works concerned Weierstrass curves.
- Most of the previous works concerned even embedding degrees.
- There were some previous works on Hessian curves but with even embedding degrees.
- There were also some previous works with odd embedding degrees but using Weierstrass curves.

Therefore, it is difficult to have an appropriate comparison between the constructions in this paper and previous works, as we are the first, to our knowledge, to consider the twisted Hessian curves with odd embedding degrees. Note that even embedding degrees have the advantage of being able to completely eliminate denominators in the computation of the pairing. However, with odd embedding degrees, we need to compute the denominator as it is non-trivial.

Table 1 shows the costs of the computation of pairings using our constructions in comparison with previous works. The first column gives the parity of the embedding degree k. The second column gives the curve models, where $\mathcal{J}$, $\mathcal{P}$, $\mathcal{E}$, $\mathcal{H}$ denote the Weierstrass model with Jacobian coordinates, the Weierstrass model with projective coordinates, the Edwards model and the Hessian model respectively. Note that this work considers the generalization of Hessian curves to "twisted" Hessian curves, and that the twists are of degree 3. The third and fourth columns show the cost of doubling (DBL) and mixed addition (mADD) including the computation of the line functions for pairing computations. By 'mixed addition' we mean that the z coordinate is set to be 1. We denote the cost of field multiplication and field squaring over the base field $\mathbb{F}_q$ by $\mathbf{m}$ and $\mathbf{s}$ respectively. The cost of multiplication by curve parameters is omitted from the table.

Let $\mathbf{M}$ and $\mathbf{S}$ denote the cost of field multiplication and field squaring over the extension field. In the doubling step, we have to compute $f^2 \cdot \ell_{2R}$. Therefore, the extra cost of $1\mathbf{M}+1\mathbf{S}$ is required. Similarly, in the addition step, we have to compute $f \cdot \ell_{R,P}$. Therefore, the extra cost of $1\mathbf{M}$ is required. These extra costs are the same for all curve models regardless of the embedding degree. Hence, we omit these costs from the table.

Note that there are various advantages of our constructions that cannot be justified by the cost shown in Table 1. First of all, the costs shown in Table 1 do not reflect the total cost for the pairing-based protocols. This

Table 1. Comparison of the cost for computing pairings on different curve models and embedding degrees

k	Curve models	DBL	mADD
even	$\mathcal{J}$, [28] [1]	$k\mathbf{m}+1\mathbf{m}+11\mathbf{s}$	$k\mathbf{m}+6\mathbf{m}+6\mathbf{s}$
	$\mathcal{J}, a=-3$, [1]	$k\mathbf{m}+6\mathbf{m}+5\mathbf{s}$	$k\mathbf{m}+6\mathbf{m}+6\mathbf{s}$
	$\mathcal{J}, a=0$, [1]	$k\mathbf{m}+3\mathbf{m}+8\mathbf{s}$	$k\mathbf{m}+6\mathbf{m}+6\mathbf{s}$
	$\mathcal{P}, a=0, b=c^2$, [16]	$k\mathbf{m}+3\mathbf{m}+5\mathbf{s}$	$k\mathbf{m}+10\mathbf{m}+2\mathbf{s}$
	$\mathcal{E}$, [1]	$k\mathbf{m}+6\mathbf{m}+5\mathbf{s}$	$k\mathbf{m}+12\mathbf{m}$
	$\mathcal{H}$, [25]	$k\mathbf{m}+3\mathbf{m}+6\mathbf{s}$	$k\mathbf{m}+10\mathbf{m}$
odd	$\mathcal{P}$, [17]	$k\mathbf{m}+6\mathbf{m}+7\mathbf{s}$	$k\mathbf{m}+13\mathbf{m}+3\mathbf{s}$
	$\mathcal{H}, d=3$, this paper	$k(\frac{14}{3}\mathbf{m}+\frac{1}{3}\mathbf{s})+5\mathbf{m}+3\mathbf{s}$	$k(\frac{14}{3}\mathbf{m}+\frac{1}{3}\mathbf{s})+11\mathbf{m}$

is because generically there are many group operations performed prior to relatively few pairing computations, and group operations on twisted Hessian curves allow faster point arithmetic operations than Weierstrass curves. Secondly, having curves represented in the same models for $\mathbb{G}_1$ and $\mathbb{G}_2$ does not induce the extra cost of conversion between curve models. (Recall that for BN, BLS, and KSS, this conversion is always necessary if one wants to take advantage of the fast point-arithmetic on Hessian or Edwards curves, as proven in [11].)

7 Concluding remarks

This paper presents concrete methods to generate pairing-friendly twisted Hessian curves. Curves generated by these methods are guaranteed to have twists of degree 3 and have embedding degree $k \equiv 3, 9, 15 \pmod{18}$ or $k \equiv 0 \pmod 6$ where $18 \nmid k$. We describe techniques to eliminate intermediate denominators for odd embedding degrees and to compute the optimal ate pairing on these curves. We also provide explicit formulas to compute line functions in pairing computations.

Although pairings on Hessian curves have already been considered, for example, by Gu, Gu, and Xie in [25] and by Li and Zhang in [41], this work is the first, to our knowledge, to concretely explain methods to generate pairing-friendly "twisted" Hessian curves with odd embedding degree. Twisted Hessian curves are a generalization of Hessian curves; using twists allows our formulas to be applied to more curves. We also take advantage of the state-of-the-art fast point arithmetic formulas available [7] on curves in twisted Hessian form.

The pairing-friendly families of BN, BLS, and KSS curves use primarily embedding degrees 12 or 18. Due to recent advances in the discrete logarithm problem, it has become necessary to increase the size of the finite field, which can be done either by increasing the size of the prime or by increasing the embedding degree. Our constructions address this problem by generating pairing-friendly twisted Hessian curves with embedding degrees 15 and 21, which allows for an increase in security without having to jump from embedding degree 12 to 18, or from 18 to 24.

We also would like to emphasize that embedding degrees such as 12, 18, 24 can split into many subfields. Recall that recent advances in number field sieve attacks on solving DLP [51] [50] [35] [36] target on finite field $\mathbb{F}_{q^n}$ where q is prime and n is composite. This implies that the attacks apply to pairings which use composite extension degrees.

To be more precise, these attacks rely on splitting n into $n = \eta\kappa$ for non-trivial factorization. Since, for example, 12 can split as $(\eta, \kappa) = (2, 6), (3, 4), (4, 3), (6, 2)$, it allows many possibles choices of subfields to perform the attacks. Similar remarks also apply to 18 which can be split as $(2, 9), (3, 6), (6, 3), (9, 2)$ and $k = 24$ which can be split as $(2, 12)$, $(3, 8)$, $(4, 6)$, $(6, 4)$, $(8, 3)$, $(12, 2)$.

On the other hand, there are sufficiently fewer subfields in our case. For example, 15 splits into either $(3, 5)$ or $(5, 3)$ while 21 splits into either $(3, 7)$ or $(7, 3)$. This means that we reduce the possible choices of subfields that the attacks can target.

In this paper, we do not propose any concrete parameters for our pairing-friendly twisted Hessian curves. This is due to the fact that the complexity of the new attacks have neither been completely understood nor reached the stable stage yet. Moreover, there is no analysis of how those attacks would apply to embedding degree 15 or 21. We plan to study how the attacks apply to our constructions in order to be able to evaluate the security level and propose concrete parameters. We also consider the optimized implementation as future work.

References

1. Christophe Arene, Tanja Lange, Michael Naehrig, and Christophe Ritzenthaler. Faster Computation of the Tate Pairing. *IACR Cryptology ePrint Archive*, 2009:155, 2009. http://eprint.iacr.org/2009/155.
2. Razvan Barbulescu and Sylvain Duquesne. Updating key size estimations for pairings. *IACR Cryptology ePrint Archive*, page 334, 2017. http://eprint.iacr.org/2017/334.

3. Razvan Barbulescu, Pierrick Gaudry, Aurore Guillevic, and François Morain. Improving NFS for the discrete logarithm problem in non-prime finite fields. In *Eurocrypt 2015 [46]*, pages 129–155, 2015.
4. Razvan Barbulescu, Pierrick Gaudry, and Thorsten Kleinjung. The tower number field sieve. In *Asiacrypt 2015 [29]*, pages 31–55, 2015.
5. Paulo S. L. M. Barreto, Ben Lynn, and Michael Scott. On the Selection of Pairing-Friendly Groups. In *SAC 2003 [43]*, pages 17–25, 2003.
6. Paulo S.L.M. Barreto and Michael Naehrig. Pairing-Friendly Elliptic Curves of Prime Order. In *SAC 2005 [47]*, pages 319–331, 2006. http://cryptosith.org/papers/pfcpo.pdf.
7. Daniel J. Bernstein, Chitchanok Chuengsatiansup, David Kohel, and Tanja Lange. Twisted Hessian Curves. In *LATINCRYPT 2015 [40]*, pages 269–294, 2015. http://cr.yp.to/papers.html#hessian.
8. Daniel J. Bernstein and Tanja Lange. Faster addition and doubling on elliptic curves. In *Asiacrypt 2007 [38]*, pages 29–50, 2007. http://cr.yp.to/newelliptic/newelliptic-20070906.pdf.
9. Dan Boneh and Matthew K. Franklin. Identity-Based Encryption from the Weil Pairing. In *CRYPTO 2001 [34]*, pages 213–229, 2001. http://www.iacr.org/archive/crypto2001/21390212.pdf.
10. Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the Weil pairing. *Journal of Cryptology*, 17(4):297–319, 2004. http://crypto.stanford.edu/~dabo/pubs/papers/weilsigs.ps.
11. Joppe W. Bos, Craig Costello, and Michael Naehrig. Exponentiating in Pairing Groups. In *SAC 2013 [39]*, 2013. http://cryptosith.org/papers/#exppair.
12. Wieb Bosma, editor. *Algorithmic Number Theory, 4th International Symposium, ANTS-IV, Leiden, The Netherlands, July 2–7, 2000, proceedings*, volume 1838 of *Lecture Notes in Computer Science*. Springer, 2000.
13. Çetin Kaya Koç, David Naccache, and Christof Paar, editors. *Cryptographic hardware and embedded systems — CHES 2001, third international workshop, Paris, France, May 14–16, 2001, proceedings*, volume 2162 of *Lecture Notes in Computer Science*. Springer, 2001.
14. Jung Hee Cheon and Tsuyoshi Takagi, editors. *Advances in Cryptology — ASIACRYPT 2016, 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4–8, 2016, Proceedings, Part I*, volume 10031 of *Lecture Notes in Computer Science*. Springer, 2016.
15. Dipanwita Roy Chowdhury, Vincent Rijmen, and Abhijit Das, editors. *Progress in Cryptology — INDOCRYPT 2008, 9th International Conference on Cryptology in India, Kharagpur, India, December 14–17, 2008. Proceedings*, volume 5365 of *Lecture Notes in Computer Science*. Springer, 2008.
16. Craig Costello, Hüseyin Hisil, Colin Boyd, Juan Manuel González Nieto, and Kenneth Koon-Ho Wong. Faster pairings on special weierstrass curves. In *Pairing 2009 [52]*, pages 89–101, 2009.
17. Craig Costello, Tanja Lange, and Michael Naehrig. Faster pairing computations on curves with high-degree twists. In *PKC 2010 [45]*, pages 224–242, 2010.
18. Ronald Cramer, editor. *Advances in Cryptology — EUROCRYPT 2005, 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Aarhus, Denmark, May 22–26, 2005, Proceedings*, volume 3494 of *Lecture Notes in Computer Science*. Springer, 2005.

19. Harold M. Edwards. A normal form for elliptic curves. *Bulletin of the American Mathematical Society*, 44:393–422, 2007. http://www.ams.org/bull/2007-44-03/S0273-0979-07-01153-6/home.html.
20. Serge Fehr, editor. *Public-Key Cryptography — PKC 2017, 20th IACR International Conference on Practice and Theory in Public-Key Cryptography, Amsterdam, The Netherlands, March 28–31, 2017, Proceedings, Part I*, volume 10174 of *Lecture Notes in Computer Science*. Springer, 2017.
21. Emmanuel Fouotsa, Nadia El Mrabet, and Aminatou Pecha. Optimal ate pairing on elliptic curves with embedding degree 9, 15 and 27. *IACR Cryptology ePrint Archive*, 2016:1187, 2016. http://eprint.iacr.org/2016/1187.
22. David Freeman, Michael Scott, and Edlyn Teske. A Taxonomy of Pairing-Friendly Elliptic Curves. *Journal of Cryptology*, 23(2):224–280, 2010. http://eprint.iacr.org/2006/372/.
23. Steven D. Galbraith and Kenneth G. Paterson, editors. *Pairing-Based Cryptography — Pairing 2008, Second International Conference, Egham, UK, September 1–3, 2008, Proceedings*, volume 5209 of *Lecture Notes in Computer Science*. Springer, 2008.
24. Craig Gentry and Alice Silverberg. Hierarchical ID-Based Cryptography. In *Asiacrypt 2002 [55]*, pages 548–566, 2002. http://www.cs.ucdavis.edu/~franklin/ecs228/pubs/extra_pubs/hibe.pdf.
25. Haihua Gu, Dawu Gu, and WenLu Xie. Efficient pairing computation on elliptic curves in hessian form. In *ICISC 2010*, pages 169–176, 2010.
26. Florian Hess, Nigel P. Smart, and Frederik Vercauteren. The Eta Pairing Revisited. *IEEE Transactions on Information Theory*, 52(10):4595–4602, 2006. http://eprint.iacr.org/2006/110.
27. Jeremy Horwitz and Ben Lynn. Toward Hierarchical Identity-Based Encryption. In *Eurocrypt 2002 [37]*, pages 466–481, 2002. http://theory.stanford.edu/~horwitz/pubs/hibe.pdf.
28. Sorina Ionica and Antoine Joux. Another approach to pairing computation in edwards coordinates. In *INDOCRYPT 2008 [15]*, pages 400–413, 2008.
29. Tetsu Iwata and Jung Hee Cheon, editors. *Advances in Cryptology — ASIACRYPT 2015, 21st International Conference on the Theory and Application of Cryptology and Information Security, Auckland, New Zealand, November 29 – December 3, 2015, Proceedings, Part II*, volume 9453 of *Lecture Notes in Computer Science*. Springer, 2015.
30. Antoine Joux. A One Round Protocol for Tripartite Diffie-Hellman. In *ANTS-IV [12]*, pages 385–393, 2000. http://cgi.di.uoa.gr/~aggelos/crypto/page4/assets/joux-tripartite.pdf.
31. Antoine Joux. A One Round Protocol for Tripartite Diffie-Hellman. *Journal of Cryptology*, 17(4):263–276, 2004.
32. Marc Joye and Jean-Jacques Quisquater. Hessian elliptic curves and side-channel attacks. In *CHES 2001 [13]*, pages 402–410, 2001. http://joye.site88.net/.
33. Ezekiel J. Kachisa, Edward F. Schaefer, and Michael Scott. Constructing Brezing-Weng Pairing-Friendly Elliptic Curves Using Elements in the Cyclotomic Field. In *Pairing 2008 [23]*, pages 126–135, 2008.
34. Joe Kilian, editor. *Advances in Cryptology — CRYPTO 2001, 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19–23, 2001, Proceedings*, volume 2139 of *Lecture Notes in Computer Science*. Springer, 2001.

35. Taechan Kim and Razvan Barbulescu. Extended tower number field sieve: A new complexity for the medium prime case. In *CRYPTO 2016 [48]*, pages 543–571, 2016.
36. Taechan Kim and Jinhyuck Jeong. Extended tower number field sieve with application to finite fields of arbitrary composite extension degree. In *PKC 2017, [20]*, pages 388–408, 2017.
37. Lars R. Knudsen, editor. *Advances in Cryptology — EUROCRYPT 2002, International Conference on the Theory and Applications of Cryptographic Techniques, Amsterdam, The Netherlands, April 28 – May 2, 2002, proceedings*, volume 2332 of *Lecture Notes in Computer Science*. Springer, 2002.
38. Kaoru Kurosawa, editor. *Advances in Cryptology — ASIACRYPT 2007, 13th International Conference on the Theory and Application of Cryptology and Information Security, Kuching, Malaysia, December 2–6, 2007, Proceedings*, volume 4833 of *Lecture Notes in Computer Science*. Springer, 2007.
39. Tanja Lange, Kristin Lauter, and Petr Lisonek, editors. *Selected areas in cryptography, 20th international conference, SAC 2013, Burnaby, BC, Canada, August 14–16, 2013, Revised Selected Papers*, volume 8282 of *Lecture Notes in Computer Science*. Springer, 2014.
40. Kristin E. Lauter and Francisco Rodríguez-Henríquez, editors. *Progress in Cryptology — LATINCRYPT 2015, 4th International Conference on Cryptology and Information Security in Latin America, Guadalajara, Mexico, August 23–26, 2015, Proceedings*, volume 9230 of *Lecture Notes in Computer Science*. Springer, 2015.
41. Liangze Li and Fan Zhang. Tate pairing computation on generalized hessian curves. In *WISA 2012*, pages 111–123, 2012.
42. Xibin Lin, Changan Zhao, Fangguo Zhang, and Yanming Wang. Computing the Ate Pairing on Elliptic Curves with Embedding Degree $k = 9$. *IEICE Transactions*, 91-A(9):2387–2393, 2008.
43. Mitsuru Matsui and Robert J. Zuccherato, editors. *Selected Areas in Cryptography, 10th Annual International Workshop, SAC 2003, Ottawa, Canada, August 14–15, 2003, Revised Papers*, volume 3006 of *Lecture Notes in Computer Science*. Springer, 2004.
44. Nadia El Mrabet, Nicolas Guillermin, and Sorina Ionica. A study of pairing computation for elliptic curves with embedding degree 15. *IACR Cryptology ePrint Archive*, 2009:370, 2009. http://eprint.iacr.org/2009/370.
45. Phong Q. Nguyen and David Pointcheval, editors. *Public Key Cryptography — PKC 2010, 13th International Conference on Practice and Theory in Public Key Cryptography, Paris, France, May 26–28, 2010. Proceedings*, volume 6056 of *Lecture Notes in Computer Science*. Springer, 2010.
46. Elisabeth Oswald and Marc Fischlin, editors. *Advances in Cryptology — EUROCRYPT 2015, 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26–30, 2015, Proceedings, Part I*, volume 9056 of *Lecture Notes in Computer Science*. Springer, 2015.
47. Bart Preneel and Stafford E. Tavares, editors. *Selected Areas in Cryptography, 12th International Conference, SAC 2005, Kingston, ON, Canada, August 11–12, 2005, Revised Selected Papers*, volume 3897 of *Lecture Notes in Computer Science*. Springer, 2006.
48. Matthew Robshaw and Jonathan Katz, editors. *Advances in Cryptology — CRYPTO 2016, 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14–18, 2016, Proceedings, Part I*, volume 9814 of *Lecture Notes in Computer Science*. Springer, 2016.

49. Amit Sahai and Brent Waters. Fuzzy Identity-Based Encryption. In *Eurocrypt 2005 [18]*, pages 457–473, 2005. http://eprint.iacr.org/2004/086/.
50. Palash Sarkar and Shashank Singh. A general polynomial selection method and new asymptotic complexities for the tower number field sieve algorithm. In *Asiacrypt 2016 [14]*, pages 37–62, 2016.
51. Palash Sarkar and Shashank Singh. A generalisation of the conjugation method for polynomial selection for the extended tower number field sieve algorithm. *IACR Cryptology ePrint Archive*, page 537, 2016. http://eprint.iacr.org/2016/537.
52. Hovav Shacham and Brent Waters, editors. *Pairing-Based Cryptography — Pairing 2009, Third International Conference, Palo Alto, California, USA, August 12–14, 2009, proceedings*, volume 5671 of *Lecture Notes in Computer Science*. Springer, 2009.
53. Joseph H. Silverman. *The Arithmetic of Elliptic Curves*, volume 106 of *Graduate Texts in Mathematics*. Springer, 2009.
54. Nigel P. Smart. The Hessian form of an elliptic curve. In *CHES 2001 [13]*, pages 118–125, 2001.
55. Yuliang Zheng, editor. *Advances in Cryptology — ASIACRYPT 2002, 8th International Conference on the Theory and Application of Cryptology and Information Security, Queenstown, New Zealand, December 1–5, 2002, Proceedings*, volume 2501 of *Lecture Notes in Computer Science*. Springer, 2002.

Supplementary Material

A Magma codes for pairing on twisted Hessian curves

The following is the full Magma codes for computing Tate and ate pairing on a pairing-friendly twisted Hessian curve having embedding degree $k = 21$.

```
// x := 5054
k := 21;
a := 0;
b := 144;
q := 60388831224640627688578323697279079263669799534119323634669;
r := 27778498887314511245242191684643503527185 4071;
t := 42567868144026523521756069913 7;

Fq := GF(q);
R<z> := PolynomialRing(Fq);
Fq7<u> := ExtensionField<Fq,z|z^7+z+3>;
R2<zz> := PolynomialRing(Fq7);

//foundW := false;
//w6 := Random(Fq7);
//while not foundW do
//  w6 := Random(Fq7);
//  if (not IsEmpty(AllRoots(w6,2))) and (IsEmpty(AllRoots(w6,3))) \
//  and (IsIrreducible(zz^3-w6)) then
//    E := EllipticCurve([Fq|a,b]);
//  Et := EllipticCurve([Fq7|0,b*w6]);
//    print "...";
//    nE := #E;
//    nEt := #Et;
//    if (nEt mod r) eq 0 then
//      foundW := true;
//    end if;
//    print "---";
//  end if;
//end while;

w6 := \
5732789654586086993956942170806419639721739871460735460 4704*u^6 + \
5098604061087527935625257068619355371689520479809855477160 4*u^5 + \
4387105082668244487545170800242127259675594328694519349658 4*u^4 + \
2276601704177502630679950992444758888963510304223206641605 5*u^3 + \
5176080573360927903986124338666821205485442361006543558643 0*u^2 + \
4690724760515227756159945663835675755075373694738478453676*u + \
1450211146574326492565047591291836750969530493405668365259 3;

Fq21<v> := ExtensionField<Fq7,zz|zz^3-w6>;

E := EllipticCurve([Fq|a,b]);
Et := EllipticCurve([Fq7|0,b*w6]);
E21 := EllipticCurve([Fq21|a,b]);

nE := \
60388831224640627688578323696853400582229534298901762935533;

nEt := \
29288448743846579171046459943425857899139686514258819477624479870198098443711615\
14858088350678666011478128779229213681785672849471870467193040443795216964269264\
83723203667917570364108098665239247869405326780135877864148241048758896075195589\
87506573653005034418234778765416515251915855991750885449439774412257674502275405\
64419069187805563811115651467208948533612200755004604765692711853548172572362114\
173800034881;

nE21 := \
```

```
25124018813403743264709406900748618635355336999124354076058797248134552366238855\
46500233836328890808048433747391376396655653539325116180125573543075069424987990\
46085168514769377764179342661664446783417252384175533420157270434611013184571806\
52079731692843055001454033044767406333117303304397564039231422780674238658329551\
73869257448798859003394290924529857658134195975978149775338907342075251807168437\
92740469523736701690016936282060596804063888652627544202343924021143130894851966\
26665615997921018362766422941488482571945412297022883036038972828654036458749148\
42518689129736947338379837741622366048789105140575266100523432087297954541476952\
33078233038867303826187849133017873867644945530014587207829950248815202129720148\
44547634663383373176809162060934458631093797081497959799942974176590536410637075\
45367650831299985961236095750353214887264496625489052060326737746308694482422365\
05210757340068135353812853686234464049595684081248084352854497220677142137436983\
76529515119902739623915410933440940614984162170496708473221755803233450245838462\
18686661470467670143473133865586043065212122589178755560918750181538296199608734\
04738908185161783546423955426391691617452080833059753552526796723329059352823384\
98541636424288823658118445428231244;

//------------------------------------------------------------//

function findCubeRoot(p);
  a := PrimitiveElement(Fq);
  return a^((p-1) div 3);
end function;

function w2h_param(a,b,Fq,E,nE,e);
  h := nE;
  while (h mod 3) eq 0 do
    h := h div 3;
  end while;

  P3 := E![0,Integers()!(Sqrt(b)),1];
  P3 := h*P3;

  u3 := P3[1];
  v3 := P3[2];

  c3 := Fq!(2*v3);
  c4 := Fq!(3*u3^2);

  lambda := Fq!(c4 / c3);

  a1 := Fq!(2*lambda);
  a3 := Fq!(c3);

  td := Fq!(a1);
  ta := Fq!(a3);

  ha := Fq!(td^3 - 27*ta);
  hd := Fq!(3*td);

  omega := findCubeRoot(q^e);

  return ta,td,ha,omega,lambda,u3,v3;
end function;

function w2h(ta,td,omega,lambda,u3,v3,W,Fq);
  u := W[1];
  v := W[2];
  w := W[3];

  T := u - u3;

  U := Fq!(T);
  V := Fq!(v - v3 - lambda*T);
  W := Fq!(w);

  A := omega*(V + td*U + ta*W);
  B := omega*V;
  C := ta*W;

  X := Fq!(U);
```

```
  Y := Fq!(A - omega*B - C);
  Z := Fq!(omega*A - B - C);

  return [X,Y,Z];
end function;

//----------------------------------------//

function Tr (E,k,P);
  Q := E![P[1],P[2]];
  for i := 1 to k - 1 do
    Q +:= E![P[1]^(q^i),P[2]^(q^i)];
  end for;
  return Q;
end function;

function aTr(E,k,P);
  kP := k*P;
  trP := Tr(E,k,P);
  N := kP - trP;
  return N;
end function;

function point_hesADD(P,Q,a);
  A := P[1] * Q[3];
  B := P[3] * Q[3];
  C := P[2] * Q[1];
  D := P[2] * Q[2];
  E := P[3] * Q[2];
  F := P[1] * Q[1] * a;
  G := (D + B) * (A - C);
  H := (D - B) * (A + C);
  J := (D + F) * (A - E);
  K := (D - F) * (A + E);
  X3 := G - H;
  Y3 := K - J;
  Z3 := J + K - G - H - 2*(B - F) * (C + E);
  return [X3,Y3,Z3];
end function;

function point_hesDBL(P,a);
  T := P[2]^2;
  A := P[2] * T;
  S := P[3]^2;
  B := P[3] * S;
  X3 := P[1] * (A - B);
  Y3 := -P[3] * (2*A + B);
  Z3 := P[2] * (A + 2*B);
  return [X3,Y3,Z3];
end function;

function hes_smult(n,P,a);
  R := P;
  seq := Reverse(IntegerToSequence(n,2));
  for i := 2 to #seq do
    R := point_hesDBL(R,a);
    if seq[i] eq 1 then
      R := point_hesADD(R,P,a);
    end if;
  end for;
  return R;
end function;

//------------------------------------------------//
// combined point operations and line functions //
//------------------------------------------------//

function hesDBL(R,Q,a);
  T := R[2]^2;
  A := R[2] * T;
  S := R[3]^2;
```

```
  B := R[3] * S;

  X3 := R[1] * (A - B);
  Y3 := -R[3] * (2*A + B);
  Z3 := R[2] * (A + 2*B);

  R2 := [X3,Y3,Z3];

  l1 := a * R[1]^2 * Q[1] + T * Q[2] + S;
  la := X3 * Q[2] + X3;
  lb := Q[1] * (Y3 + Z3);
  lc := la^2 + lb * (la + lb);
  l := l1 * lc;

  return l,R2;
end function;

function hesADD(R,P,Q,a);
  A := P[1] * R[3];
  C := P[2] * R[1];
  D := P[2] * R[2];
  F := P[1] * R[1] * a;
  G := (D + R[3]) * (A - C);
  H := (D - R[3]) * (A + C);
  J := (D + F) * (A - R[2]);
  K := (D - F) * (A + R[2]);
  X3 := G - H;
  Y3 := K - J;
  Z3 := J + K - G - H - 2*(R[3] - F) * (C + R[2]);

  RP := [X3,Y3,Z3];

  l1 := (P[2] * R[3] - R[2]) * (P[1] - Q[1]) + \
    (Q[2] - P[2]) * (P[1] * R[3] - R[1]);
  la := Q[2] * X3 + X3;
  lb := Q[1] * (Y3 + Z3);
  lc := la^2 + lb * (la + lb);
  l := l1 * lc;

  return l,RP;
end function;

//---------------//
// Hessian: Tate //
//---------------//

function hes_Tate(P,Q,s,ha);
  R := P;
  f := 1;
  for i:= 2 to #s-1 do
    l,R := hesDBL(R,Q,ha);
    f := f^2 * l;
    if s[i] eq 1 then
      l,R := hesADD(R,P,Q,ha);
      f := f * l;
    end if;
  end for;

  l,R := hesDBL(R,Q,ha);
  f := f^2 * l;

  l := Q[2]*P[1] - Q[1]*(P[2] + 1) + P[1];
  f := f*l;

  f := f^(Integers()!((q^k-1)/r));

  return f;
end function;

//---------------//
// Hessian: Ate //
```

```
//---------------//

function hes_Ate(P,Q,s,ha);
  R := Q;
  f := 1;
  for i:= 2 to #s do
    l,R := hesDBL(R,P,ha);
    f := f^2 * l;
    if s[i] eq 1 then
      l,R := hesADD(R,Q,P,ha);
      f := f * l;
    end if;
  end for;

  f := f^(Integers()!((q^k-1)/r));

  return f;
end function;

//-------------------------------------------------------------//

ta ,td ,ha ,omega ,lambda ,u3 ,v3  := w2h_param(a,b,Fq,E,nE,1);
ta3,td3,ha3,omega3,lambda3,u33,v33 := w2h_param(a,b,Fq21,E21,nE21,1);

P := PointsAtInfinity(E)[1];
while P eq PointsAtInfinity(E)[1] do
  P := Random(E);
  P := (Order(P) div r)*P;
end while;

Qt := PointsAtInfinity(Et)[1];
while Qt eq PointsAtInfinity(Et)[1] do
  Qt := Random(Et);
  Qt := (nE21 div r^2) * Qt;
end while;
w2list := AllRoots(Fq21!(w6),3);
w3list := AllRoots(w6,2);
Q := E21![Qt[1]/(w2list[2]),Qt[2]/(w3list[1])];
Q := aTr(E21,k,Q);

s:=Reverse(IntegerToSequence(r,2));
s2:=Reverse(IntegerToSequence(t-1,2));

//----------//

H := w2h(ta,td,omega,lambda,u3,v3,P,Fq);
H := [H[1]/H[3],H[2]/H[3],1];

K := w2h(ta3,td3,omega3,lambda3,u33,v33,Q,Fq21);
K := [K[1]/K[3],K[2]/K[3],1];

c1 := Random(2,5);
c2 := Random(6,10);

H_c1 := hes_smult(c1,H,ha);
H_c1 := [H_c1[1]/H_c1[3], H_c1[2]/H_c1[3], 1];

H_c2 := hes_smult(c2,H,ha);
H_c2 := [H_c2[1]/H_c2[3], H_c2[2]/H_c2[3], 1];

K_c1 := hes_smult(c1,K,ha3);
K_c1 := [K_c1[1]/K_c1[3], K_c1[2]/K_c1[3], 1];

K_c2 := hes_smult(c2,K,ha3);
K_c2 := [K_c2[1]/K_c2[3], K_c2[2]/K_c2[3], 1];

// Hessian Tate //
ansH11 := hes_Tate(H_c1,K_c2,s,ha);
ansH12 := hes_Tate(H_c1,K,s,ha);
ansH12 := ansH12^c2;
ansH13 := hes_Tate(H,K_c2,s,ha);
```

```
ansH13 := ansH13^c1;
ansH14 := hes_Tate(H_c2,K_c1,s,ha);
print ansH11 eq ansH12, ansH12 eq ansH13, ansH13 eq ansH14, ansH14 ne 1;
print ansH11^r eq 1, ansH12^r eq 1, ansH13^r eq 1, ansH14^r eq 1;

// Hessian Ate //
ansH21 := hes_Ate(H_c1,K_c2,s2,ha3);
ansH22 := hes_Ate(H_c1,K,s2,ha3);
ansH22 := ansH22^c2;
ansH23 := hes_Ate(H,K_c2,s2,ha3);
ansH23 := ansH23^c1;
ansH24 := hes_Ate(H_c2,K_c1,s2,ha3);
print ansH21 eq ansH22, ansH22 eq ansH23, ansH23 eq ansH24, ansH24 ne 1;
print ansH21^r eq 1, ansH22^r eq 1, ansH23^r eq 1, ansH24^r eq 1;
```

B Auxiliary functions for Sage script

The following scripts show the auxiliary functions for the Sage scripts described in Section 3.

```
def tn(t,n,q):
  if n == 0: return 2
  elif n == 1: return t
  else: return t*tn(t,n-1,q) - q*tn(t,n-2,q)

def tHesADD(P,Q,a):
  X1,Y1,Z1 = P[0],P[1],P[2]
  X2,Y2,Z2 = Q[0],Q[1],Q[2]
  A = X1 * Z2
  B = Z1 * Z2
  C = Y1 * X2
  D = Y1 * Y2
  E = Z1 * Y2
  F = X1 * X2 * a
  X3 = A*B - C*D
  Y3 = D*E - F*A
  Z3 = F*C - B*E
  return [X3,Y3,Z3]

def tHesDBL(P,a):
  X1,Y1,Z1 = P[0],P[1],P[2]
  D = X1^3
  E = Y1^3
  F = Z1^3
  G = a*D
  X3 = X1 * (E - F)
  Y3 = Z1 * (G - E)
  Z3 = Y1 * (F - G)
  return [X3,Y3,Z3]

def expmod_loop(b,e,m):
        p = b
        t = 1
        while e > 0 :
                if e&1 : t = (t*p) % m
                e = e >> 1
                p = (p*p) % m
        return t

def find_b(p,U,V):
# suppose d = 3 for y^2 = x^3 + b
        exp = ZZ((p-1)/6)

  if ((2*V)%3 == 0 and (2*U)%3 == 2):
    return 16
```

```
  if ((2*V)%3 == 0 and (2*U)%3 == 1):
    b = 1
    found = false
    while not found:
      chk = expmod_loop(b,exp,p)
      if chk == GF(p)(-1):
        found = true
        return 16*b
      else:
        b += 1
  if ((2*V)%3 != 0 and (2*U)%3 == 2):
    if ((2*V)%3 != 1):
      V = -V
    if ((2*U)%3 == 2):
      b = 1
      found = false
      while not found:
        chk = expmod_loop(b,exp,p)
        rem = ((2*U)/(3*V-U))%p
        if chk == GF(p)(rem):
          found = true
          return 16*b
        else:
          b += 1
  else:
    b = 1
    found = false
    while not found:
      chk = expmod_loop(b,exp,p)
      rem = ((2*U)/(3*V+U))%p
      if chk == GF(p)(rem):
        found = true
        return 16*b
      else:
        b += 1

def findCubeRoot(p):

  exp = ZZ((p-1)/3)
  a = 2
  found = false
  while not found:
    w = expmod_loop(a,exp,p)
    if (w^3) == GF(p)(1) and not(w == 1):
      found = true
    else:
      a += 1
  return w
```