A New Family of Pairing-Friendly elliptic curves

Michael Scott

Cryptographic Researcher
MIRACL Labs
mike.scott@miracl.com

Abstract. There have been recent advances in solving the finite extension field discrete logarithm problem as it arises in the context of pairing-friendly elliptic curves. This has lead to the abandonment of approaches based on super-singular curves of small characteristic, and to the reconsideration of the field sizes required for implementation based on non-supersingular curves of large characteristic. This has resulted in a revision of recommendations for suitable curves, particularly at a higher level of security. Indeed for AES-256 levels of security the BLS48 curves have been suggested, and demonstrated to be superior to other candidates. These curves have an embedding degree of 48. The well known taxonomy of Freeman, Scott and Teske only considered curves with embedding degrees up to 50. Given some uncertainty around the constants that apply to the best discrete logarithm algorithm, it would seem to be prudent to push a little beyond 50. In this note we announce the discovery of a new family of pairing friendly elliptic curves with an embedding degree of 54.

1 Introduction

One of great break-throughs in pairing-based cryptography was the discovery of the BN curves [3]. A group size of 256-bits (to match the AES-128 bit level of security) can be supported by an elliptic curves over a field also of 256-bits, and since the embedding degree is 12, the size of the discrete logarithm (DL) problem over the extension field is 3072 bits. Which was a serendipitous direct hit on the size of DL problem believed to correspond to AES-128. The fit was perfect.

Recall that protocols based on bilinear pairings typically consist of operations on 3 groups, denoted as $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_T$, and the calculation of the pairing itself, usually denoted as $w = e(P, Q)$, where the pairing takes two elliptic curve point parameters $P \in \mathbb{G}_1$ and $Q \in \mathbb{G}_2$ respectively, and evaluates to an element in the finite extension field $w \in \mathbb{G}_T$. Here $\mathbb{G}_1$ is contained in the elliptic curve $E(\mathbb{F}_p)$, $\mathbb{G}_2$ is contained in $E'(\mathbb{F}_{p^{k/d}})$, and $\mathbb{G}_T$ is contained in the finite extension field $\mathbb{F}_{p^k}$, where k is the embedding degree associated with the pairing-friendly curve, and d is a divisor of k corresponding to a supported twisted curve E'. Note that $\mathbb{G}_2$ points can be manipulated on the smaller twisted curve, and transformed to a point on $E(\mathbb{F}_{p^k})$ only when needed.

The pairing calculation consists of two parts, a Miller loop followed by a final exponentiation. In real-world protocols much of the action takes place in the smallest group $\mathbb{G}_1$, although implementors have tended to concentrate more on the pairing itself. In more complex protocols products of pairings are required, and here a single final exponentiation can be applied to an amalgamation of Miller loops [11].

A BN curve is an example of a parameterised pairing-friendly curve, that is fixed polynomial formulae exist for the prime modulus p and the group order r in terms of an integer parameter u, which is chosen such that both p and r are prime. Such parameterised curves have become very popular for many reasons. First the ratio between the group and field size can be as low as one, and secondly multiple optimizations become possible. The most significant of these would be the development of the optimal Ate pairing [13], for which the number of iterations of the Miller loop is reduced from the number of bits in r (as required for the original Tate pairing) to the number of bits in u. Rather paradoxically this implies that the number of iterations required in the Miller loop actually tends to decrease as the security level increases. Also a simpler form of final exponentiation applies [12].

However almost immediately after BN curves were introduced, Schirokauer [10] in a paper introducing the Number Field Sieve (NFS), warned us that: "Without discussing the evident difficulty of implementing the NFS for degree 12 fields, we observe that the special form of p may reduce the difficulty of computing logarithms in $\mathbb{F}_{p^{12}}$". In the absence of any concrete evidence to support this concern, the BN curve was nonetheless widely adopted. However Schirokauer's warning has proven to be prescient and the field of pairing-based cryptography has been disrupted by the recent, but not entirely surprising, discovery of a faster algorithm for solving the discrete logarithm problem in the finite extension field that arises when using these types of pairings. See [1], [6].

A pairing-friendly curve can be characterised by the defining triplet $\{\rho, k, d\}$, where ρ is the ratio between the number of bits in p and the number of bits in r. Given a group size of g bits, the field size of $\mathbb{G}_1$ is $f_1 = \rho g$, the extension field size of $\mathbb{G}_T$ is $f_T = \rho k g$, and the field size of $\mathbb{G}_2$ is $f_2 = f_T/d$, where d is from the set of possible twists $\{1, 2, 3, 4, 6\}$, and is usually the maximum from this set that divides k.

When choosing a suitable curve, the starting point is the security level in AES-equivalent bits, typically 128, 192 or 256. The group size should ideally be exactly twice this, and the other field sizes are then immediately fixed as shown above by the defining triplet.

For example for AES-128 security for the BN curve, the defining triplet is $1, 12, 6$, and given $g = 256$, then $f_1 = 256$, $f_T = 3072$, and $f_2 = 512$. For a BLS12 curve (by which we mean a BLS curve with embedding degree of 12, see below), the triplet is $3/2, 12, 6$, and given $g = 256$, then $f_1 = 384$, $f_T = 4608$, and $f_2 = 768$.

The main problem is to satisfy the security requirement for $\mathbb{G}_T$, so that it matches that for $\mathbb{G}_1$. See Table 1. Note that these numbers are rather imprecise

as exact analysis is difficult. We have mainly followed the analysis of Barbulescu and Duquesne [1], extrapolating in places, rather than the less conservation estimates of Menezes, Sarkar and Singh [8]. However the estimates they provide depend on certain constants, in which one can have diminishing confidence as the security level increases.

DL Algorithm	AES-128	AES-192	AES-256
NFS	3072	7680	15360
exTNFS	3618	9241	18480
SexTNFS	5004	12871	27410

Table 1. Recommended extension field sizes

Basically, according to current knowledge, the NFS estimates now apply only to prime order fields of prime extension. The extNFS estimates apply to composite order extensions, and the SexNFS estimates to parameterised prime, composite order extensions, like the BN curves. It is now clear that BN curves are not quite as perfect as originally thought. As Barbulescu and Duquense put it: "Variants of NFS where p is parametrized are considered to be the dream situation for an attacker", although they do go on to offer some reassurance that they do not expect any further improvements in the SexTNFS algorithm.

We should say a word about $\mathbb{G}_2$. Since this is of a size an integer multiple of $\mathbb{G}_1$, we can be confident that if $\mathbb{G}_1$ is secure then so is $\mathbb{G}_2$. However in the optimal Ate pairing [13], each iteration of the Miller loop typically involves at least a point doubling in $\mathbb{G}_2$. Therefore we would like $\mathbb{G}_2$ to be as small as possible, and therefore the twist d to be as large as possible. The maximum possible on an elliptic curve is $d = 6$, and therefore we would like to attain this if possible. A necessary condition for this to be possible is that $6|k$. Note however that as the embedding degree k increases, so must $\mathbb{G}_2$. Ideally we do not want $\mathbb{G}_2$ growing too large, as elliptic curve cryptography over large extension fields will be very slow (and probably best implemented using affine coordinates).

From an implementation point of view the ideal solution is one that keeps f_1 as small as possible, while meeting all of the security constraints. This assumes that the value of ρ is small, that the embedding degree is such that we serendipitously hit the appropriate target in Table 1, and that a sextic twist applies and so the embedding degree is a multiple of 6.

An alternative response might be to revert to the Cocks-Pinch construction [9], avoiding parameterized curves, while continuing to use composite order extensions. It is not difficult to generate such curves for $k = 0 \bmod 6$ such that sextic twists can be supported, although only for $\rho > 2$. The idea would be to revert to the original Tate pairing and accept the lower extTNFS estimates. However this is unlikely to prove competitive in practise.

2 BLS and KSS curves

BLS curves are the original small discriminant parameterised family of families of pairing-friendly elliptic curves [2]. For any positive embedding degree $k = 0 \bmod 6$ (except if $18|k$), they provide simple formulae from which can be derived pairing friendly curves which support the maximal twist of $d = 6$, and have a relatively small ρ value given by $\rho = (2 + k/3)/\varphi(k)$ [4]. Observe that the value of ρ decreases with increasing values of k. Having a range of embedding degrees to choose from makes it easier to hit the optimal values for any security level.

For example Barbulescu and Duquesne [1] have demonstrated that the BLS12 curve is a good fit for the AES-128 level of security, and the BLS24 curve is the best choice for AES-192. In another recent paper Kiyomura et al. [7], reacting to the new understanding, demonstrated that a BLS48 curve is also the best choice of pairing-friendly curve to meet the new estimates for the 256-bit level of security. In this case the security requirement could be met with a group size of 512-bits, a modulus of 576 bits (as $\rho = 9/8$), and a finite field extension size of $48.576 = 27648$.

However the BLS curves do not exist for $18|k$, as in these cases the polynomial formula for p is not irreducible, and therefore cannot generate primes [4]. Serendipitously for the cases of $k = 18$ and $k = 36$ there do exist the alternative KSS curves [5], which, as luck would have it, provide curves with the same ρ values as determined by the above formula for the missing BLS curves. However since the taxonomy of Freeman, Scott and Teske does not explore beyond $k = 50$, the situation for $k = 54$ is currently unknown. But if a BLS curve did exist for $k = 54$, then from the formula given above, it would have a ρ value of $10/9$.

3 The Method

We used the KSS method as described in [5]. However the new curve found with embedding degree 54 is not of the form of a typical KSS curve, where integer solutions exist only in a restricted set of residue classes. Recall that the KSS method also rediscovers the BN curves [3]. It would appear possible that the new family of curves is, like the BN curves, a "sporadic", and not related to any existing family. On the other hand it has a certain symmetry, which is suggestive that it might be a member of an as-yet undiscovered family of families.

4 The new curve family

We find $-\zeta_{54} - \zeta_{54}^{10}$ as a suitable element $\in \mathbb{Q}(\zeta_{54})$, and following the KSS method [5] from there we obtain the solution

$$
\begin{aligned}
p &= 1 + 3u + 3u^2 \quad + \quad 3^5u^9 + 3^5u^{10} + 3^6u^{10} + 3^6u^{11} \quad + \quad 3^9u^{18} + 3^{10}u^{19} + 3^{10}u^{20} \\
r &= 1 + 3^5u^9 + 3^9u^{18} \\
t &= 1 + 3^5u^{10} \\
c &= 1 + 3u + 3u^2
\end{aligned}
\tag{1}
$$

where p is the prime modulus, r is the prime order of the pairing-friendly group, t is the trace of the Frobenius, and c is a cofactor. It can be verified that the Complex Multiplication (CM) discriminant is $D = 3$ because $4p - t^2 = 3f^2$, for some polynomial f. This implies that the curve has twists of degree 6 which, as with the BN and BLS curves, facilitates an important optimization. Observe that the prime p can be any of 1, 3, 5 or 7 mod 8 depending on the choice of u. The total number of points on the curve will be $\#E = cr$.

Recall that the embedding degree is the smallest value of k such that $r|(p^k - 1)$ [9]. In this case it is easily confirmed that $k = 54$. The value of $\rho = \deg(p)/\deg(r) = 10/9$, which is close to the ideal value of 1.

5 An example construction

An actual curve can be generated using the seed value $u = C404042_{16}$, which has a low Hamming weight of 6. Then the curve

$$y^2 = x^3 + 12$$

is a pairing-friendly elliptic curve with a group order r of 512-bits, and a modulus p of 569-bits. Given the embedding degree of 54, the finite extension field is of size 30726 bits, comfortably above the size recommended for an overall security equivalent to AES-256. The embedding degree $k = 54$ is obviously of the desirable form $k = 2^i3^j$, which simplifies implementation [7].

6 Discussion

As speculated above, this family is indeed a member of a larger "family of families". For $n = 3^j$ and $m = \varphi(n)$, then a pairing-friendly curve with embedding degree of $k = 2n$ if j is odd, and $k = n$ if j is even, and with a ρ value of $(m + 2)/m$, can be found as

$$
\begin{aligned}
r &= 1 + 3^{(m+2)/4}u^{m/2} + 3^{m/2}u^m \\
t &= 1 + 3^{(m+2)/4}u^{1+m/2} \\
c &= 1 + 3u + 3u^2 \\
p &= c.r + t - 1
\end{aligned}
\tag{2}
$$

However this is not particularly useful for cases other than $k = 54$. For the embedding degrees that arise from these formulae which are less than 54 (6 and 9), there already exist curves with the same ρ value. The higher values of embedding degree (81,486) are probably not useful in practise.

7 Conclusion

We present a new family of pairing friendly curves with an embedding degree of $k = 54$, which fills a gap that might be useful in the event of a deeper understanding emerging of the true difficulty of the discrete logarithm problem as it applies to high-security pairing-based cryptography. We also strived to find a solution for the next "missing" case of $k = 72$, but failed despite an extensive computer search. Clearly the KSS method is a powerful tool for finding families of pairing-friendly curves. In the Appendix we provide our SAGE script for finding such curves.

References

1. R. Barbulescu and S. Duquesne. Updating key size estimations for pairings, 2017. `http://eprint.iacr.org/2017/334`.
2. P. S. L. M. Barreto, B. Lynn, and M. Scott. Constructing elliptic curves with prescribed embedding degrees. In *Security in Communication Networks – SCN'2002*, volume 2576 of *Lecture Notes in Computer Science*, pages 263–273. Springer-Verlag, 2002.
3. P.S.L.M. Barreto and M. Naehrig. Pairing-friendly elliptic curves of prime order. In *Selected Areas in Cryptography – SAC'2005*, volume 3897 of *Lecture Notes in Computer Science*, pages 319–331. Springer-Verlag, 2006.
4. D. Freeman, M. Scott, and E. Teske. A taxonomy of pairing-friendly elliptic curves. Cryptology ePrint Archive, Report 2006/372, 2006. `http://eprint.iacr.org/2006/372`.
5. E. Kachisa, E.F. Schaefer, and M. Scott. Constructing Brezing-Weng pairing friendly elliptic curves using elements in the cyclotomic field. In *Pairing 2008*, volume 5209 of *Lecture Notes in Computer Science*, pages 126–135. Springer-Verlag, 2008.
6. T. Kim and R. Barbulescu. The extended tower number field sieve: A new complexity for the medium prime case. In *Crypto 2016*, volume 9814 of *Lecture Notes in Computer Science*, pages 543–571. Springer-Verlag, 2016.
7. Y. Kiyomura, A. Inoue, Y. Kawahara, M. Yasuda, T. Takagi, and T. Kobayashi. Secure and efficient pairing at 256-bit security level. In *ACNS 2015*, volume 10355 of *Lecture Notes in Computer Science*, pages 59–79. Springer-Verlag, 2017.
8. A. Menezes, P. Sarkar, and S. Singh. Challenges with assessing the impact of NFS advances on the security of pairing-based cryptography. In *Mycrypt 2016*, volume 10311 of *Lecture Notes in Computer Science*, pages 83–108. Springer-Verlag, 2016.
9. N. El Mrabet and M. Joye, editors. *Guide to Pairing-Based Cryptography*. Chapman and Hall/CRC, 2016.
10. O. Schirokauer. The number field sieve for integers of low weight. Cryptography ePrint Archive, Report 2006/107, 2006. `http://eprint.iacr.org/2006/107`.

11. M. Scott. On the efficient implementation of pairing-based protocols. In *IMACC 2011*, volume 7089 of *Lecture Notes in Computer Science*, pages 296–308. Springer-Verlag, 2011.
12. M. Scott, N. Benger, M. Charlemagne, and L. Dominguez Perez. On the final exponentiation for calculating pairings on ordinary elliptic curves. In *Pairing 2009*, volume 5671 of *Lecture Notes in Computer Science*, pages 78–88. Springer-Verlag, 2009.
13. F. Vercauteren. Optimal pairings. *IEEE Transactions of Information Theory*, 56:455–461, 2009.

SAGE script for finding KSS curves

```
#
# Sage Code to search for twist friendly KSS curves
# M. Scott February 2018
#
# set K value here, MUST be a multiple of 4 or 6

K=54
nb=euler_phi(K)

# Set search parameters. We will be searching through polynomials of degree nb
# ... with nz non-zero coefficients of absolute size less than or equal to lim

nz=2  # must be > 1
lim=1 # must be positive

# Observe search progress

progress=True

# some successful searches
# K=8,  nz=2, lim=1
# K=12, nz=3, lim=1
# K=16, nz=2, lim=2
# K=18, nz=2, lim=2
# K=32, nz=2, lim=3
# K=36, nz=2, lim=2
# K=40, nz=2, lim=2
# K=54, nz=2, lim=1

def igcd(x,y) :
# integer GCD, returns GCD of x and y
        if y==0 :
                return x
        while True :
                r=x%y
                if r==0 :
                        break
                x=y
                y=r
        return y

def mylcm(a,b) :
        return (a*b)/gcd(a,b)

def flat(p) :
        lcm=1
        c=p.coefficients()
        #print c
        #print p.degree()
```

```
        for i in range(len(c)) :
                d=c[i].denom()
                lcm=mylcm(lcm,d)
        return lcm

def content(p) :
        con=1
        c=p.coefficients()
        con=c[0][0]
        #print c
        for i in range(1,len(c)) :
                d=c[i][0]
                con=gcd(con,d)
        return Integer(con)

def common(p,d) :
        c=p.coefficients()
        #print c
        for i in range(len(c)) :
                de=pow(d,c[i][1])
                if Integer(c[i][0])%de!=0 :
                        return False
        return True

def iter_bits(x,n) :
        gotone=False
        for i in range(0,n-1) :
                if x[i]==1 and x[i+1]==0 :
                        gotone=True
                        x[i+1]=1
                        x[i]=0
                        if x[0]==1 :
                                break
                        k=1
                        while True :
                                if x[k] != 0 :
                                        break
                                k=k+1
                        for j in range(0,i-k) :
                                x[j]=x[j+k]
                                x[j+k]=0
                        break
        return gotone

def iter_nums(v,m,lim) :
        for k in range(0,m) :
                if v[k]==-1 :
                        v[k]=1
                        break
                if v[k]<lim :
                        v[k]=v[k]+1
                        break
                v[k]=-lim
        for k in range(0,m) :
                if v[k]!=lim :
                        return True
        return False

u=[]
q=[]
s=[]
v=[]
rhobestn=3
rhobestd=2

D=0
if K%4 == 0 :
        D=1
```

```
if K%6 == 0 :
        D=3
if D==0 :
        sys.exit(0)     # only looking for twist-friendly curves

C.<z> = CyclotomicField(K)

for i in range(0,nb) :
        q.append(0)
        u.append(0)

for i in range(0,nz) :
        s.append(0)
        v.append(0)

more_bits=True
k=0
while True :
        if k==0 :
                for j in range(nz) :
                        u[j]=1
        else :
                more_bits=iter_bits(u,nb)
        j=0
        for i in range(nb) :
                if u[i]!=0 :
                        s[j]=i
                        j=j+1
        more_nums=True
        n=0
        while True :
                if n==0 :
                        for j in range(nz) :
                                v[j]=-lim
                else:
                        more_nums=iter_nums(v,nz,lim)

                for j in range(nb) :
                        q[j]=0
                for j in range(nz) :
                        q[s[j]]=v[j]

                if s[0]!=1 :  # ??
                        n=n+1
                        if not more_nums :
                                break
                        continue

# all above here manages the search loop


                pb=0    # create next polyomial in QQ
                for j in range(nb) :
                        pb=pb+q[j]*z^j

                if progress :
                        print q   # q or pb

                r=pb.minpoly()
                M.<w> = NumberField(r)
                rz=M.roots_of_unity()
                nrz=len(rz)

                # get CM discriminant D as a polynomial

                sd=0
                if K%4 == 0 :
                        if nrz < K/4:
```

```
                n=n+1
                if not more_nums :
                        break
                continue;
        sd=rz[K/4-1]
if K%6 == 0 :
        if nrz < K/3:
                n=n+1
                if not more_nums :
                        break
                continue;
        sd=(2*rz[K/3-1]+1)/3

for i in range(nrz) :

        # search though K-th roots of unity

        if igcd(i+1,K) != 1 :
                continue;
        pru=rz[i]
        ft=pru+1
        fy=sd*(pru-1)

        t=ft.polynomial()

        y=fy.polynomial()
        p=(t*t+D*y*y)/4

        rhon=p.degree()
        rhod=r.degree()
        ig=igcd(rhon,rhod)
        rhon/=ig
        rhod/=ig

        if rhon<rhod :
                continue

        if rhobestd*rhon>rhobestn*rhod :
                continue  # rho is not interesting

        if not p.is_irreducible() :
                continue

        # solution looks interesting...
        # convert polynomials over QQ to ZZ (with one common integer divisor m)

        plcm=flat(p)
        tlcm=flat(t)

        m=mylcm(plcm,tlcm)
        p=p*m
        t=t*m

        b=0
        tries1=0
        tries2=0
        fail=False;

        while True :

                # try to find any residue class that works

                tries1=tries1+1
                if tries1>200000 :  # give up..
                        fail=True
                        break
                if p(b)%m != 0 :
                        b=b+1
```

```
                continue
        if t(b)%m !=0 :
                b=b+1
                continue

        tries1=0
        sp=p(x=m*x+b).expand()/m

        # try 100 times to find p that doesn't have an integer factor

        if content(sp)!=1 :
                tries2=tries2+1
                if tries2>100 :
                        fail=True
                        break
                b=b+1
                continue
        tries2=0
        if not sp.polynomial(ZZ).is_irreducible() :
                b=b+1
                continue

        if fail :
                break

        st=t(x=m*x+b).expand()/m
        sr=r(x=m*x+b).expand()
        c=content(sr)
        sr=sr/c
        if not sr.polynomial(ZZ).is_irreducible() :
                b=b+1
                continue;
        break

if fail :
        continue

# try simplifying formulae

for j in range(2,23) :
        while common(sp,Integer(j)) and common(st,Integer(j))
                                        and common(sr,Integer(j)) :
                sp=sp(x=x/j).expand()
                st=st(x=x/j).expand()
                sr=sr(x=x/j).expand()

ct=gcd(content(t),m)
mt=m/ct
t=t/ct

sp=sp.polynomial(ZZ)
st=st.polynomial(ZZ)
sr=sr.polynomial(ZZ)

np=sp+1-st

if np%sr == 0 :

        cf=np/sr

        print "Solution found, rho= ",rhon,"/",rhod
        print "p= (",p,")/",m
        print "t= (",t,")/",mt
        print "r= ",r
        print "For sample residue class ",m,"*x +",b
        print "p= ",sp
        print "t= ",st
        print "r= ",sr
```

```
                                print "c= ",cf
                                print

                                rhobestn=rhon
                                rhobestd=rhod


# all below here manages the search loop

                        if not more_nums :
                                break
                        n=n+1
                if not more_bits :
                        break
                k=k+1
```