CFRG A. Langley
Internet-Draft Google
Intended status: Informational January 28, 2015
Expires: August 1, 2015

Elliptic Curves for Security
draft-irtf-cfrg-curves-00

Abstract

This memo describes an algorithm for deterministically generating parameters for elliptic curves over prime fields offering high practical security in cryptographic applications, including Transport Layer Security (TLS) and X.509 certificates. It also specifies a specific curve at the ~128-bit security level.

Table of Contents

1. Note on authorship

This document is a merging of "draft-black-rpgecc-01" (by Benjamin Black, Joppe W. Bos, Craig Costello, Patrick Longa and Michael Naehrig) and "draft-turner-thecurve25519function-01" (by Watson Ladd, Rich Salz and Sean Turner). They are the actual authors of the words and figures, but authorship also implies support and so are not listed as authors until they have confirmed that they support this document. None the less, they deserve any credit for the contents.

2. Introduction

Since the initial standardization of elliptic curve cryptography (ECC) in [SEC1] there has been significant progress related to both efficiency and security of curves and implementations. Notable examples are algorithms protected against certain side-channel attacks, different 'special' prime shapes which allow faster modular arithmetic, and a larger set of curve models from which to choose. There is also concern in the community regarding the generation and potential weaknesses of the curves defined in [NIST].

This memo describes a deterministic algorithm for generation of elliptic curves for cryptography. The constraints in the generation process produce curves that support constant-time, exception-free scalar multiplications that are resistant to a wide range of side-channel attacks including timing and cache attacks, thereby offering high practical security in cryptographic applications. The

deterministic algorithm operates without any hidden parameters, reliance on randomness or any other processes offering opportunities for manipulation of the resulting curves. The selection between curve models is determined by choosing the curve form that supports the fastest (currently known) complete formulas for each modularity option of the underlying field prime. Specifically, the Edwards curve x^2 + y^2 = 1 + dx^2y^2 is used with primes p with p = 3 mod 4, and the twisted Edwards curve -x^2 + y^2 = 1 + dx^2y^2 is used for primes p with p = 1 mod 4.

3. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

4. Security Requirements

For each curve at a specific security level:

1. The domain parameters SHALL be generated in a simple, deterministic manner, without any secret or random inputs. The derivation of the curve parameters is defined in Section 6.

2. The trace of Frobenius MUST NOT be in {0, 1} in order to rule out the attacks described in [Smart], [AS], and [S], as in [EBP].

3. MOV Degree: the embedding degree k MUST be greater than (r - 1) / 100, as in [EBP].

4. CM Discriminant: discriminant D MUST be greater than 2^100, as in [SC].

5. Notation

Throughout this document, the following notation is used:

p Denotes the prime number defining the underlying field.

GF(p) The finite field with p elements.

d An element in the finite field GF(p), not equal to -1 or zero.

Ed An Edwards curve: an elliptic curve over GF(p) with equation x^2 + y^2 = 1 + dx^2y^2.

tEd A twisted Edwards curve where a=-1: an elliptic curve over GF(p) with equation -x^2 + y^2 = 1 + dx^2y^2.

oddDivisor The largest odd divisor of the number of GF(p)-rational points on a (twisted) Edwards curve.

oddDivisor' The largest odd divisor of the number of GF(p)-rational points on the non-trivial quadratic twist of a (twisted) Edwards curve.

cofactor The cofactor of the subgroup of order oddDivisor in the group of GF(p)-rational points of a (twisted) Edwards curve.

cofactor' The cofactor of the subgroup of order oddDivisor in the group of GF(p)-rational points on the non-trivial quadratic twist of a (twisted) Edwards curve.

trace The trace of Frobenius of Ed or tEd such that #Ed(GF(p)) = p + 1 - trace or #tEd(GF(p)) = p + 1 - trace, respectively.

P A generator point defined over GF(p) of prime order oddDivisor on Ed or tEd.

X(P) The x-coordinate of the elliptic curve point P.

Y(P) The y-coordinate of the elliptic curve point P.

6. Parameter Generation

This section describes the generation of the curve parameter, namely d, of the elliptic curve. The input to this process is p, the prime that defines the underlying field. The size of p determines the amount of work needed to compute a discrete logarithm in the elliptic curve group and choosing a precise p depends on many implementation concerns. The performance of the curve will be dominated by operations in GF(p) and thus carefully choosing a value that allows for easy reductions on the intended architecture is critical for performance. This document does not attempt to articulate all these considerations.

6.1. Edwards Curves

For p = 3 mod 4, the elliptic curve Ed in Edwards form is determined by the non-square element d from GF(p) (not equal to -1 or zero) with smallest absolute value such that #Ed(GF(p)) = cofactor * oddDivisor, #Ed'(GF(p)) = cofactor' * oddDivisor', cofactor = cofactor' = 4, and both subgroup orders oddDivisor and oddDivisor' are prime. In addition, care must be taken to ensure the MOV degree and CM discriminant requirements from Section 4 are met.

These cofactors are chosen because they are minimal.

```
Input: a prime p, with p = 3 mod 4
Output: the parameter d defining the curve Ed
1. Set d = 0
2. repeat
     repeat
       if (d > 0) then
         d = -d
       else
         d = -d + 1
       end if
     until d is not a square in GF(p)

     Compute oddDivisor, oddDivisor', cofactor and cofactor' where #Ed(GF(p)) =
     cofactor * oddDivisor, #Ed'(GF(p)) = cofactor' * oddDivisor', cofactor and
     cofactor' are powers of 2 and oddDivisor, oddDivisor' are odd.
   until ((cofactor = cofactor' = 4), oddDivisor is prime and oddDivisor' is
prime)
3. Output d
```

GenerateCurveEdwards

6.2. Twisted Edwards Curves

For a prime p = 1 mod 4, the elliptic curve tEd in twisted Edwards form is determined by the non-square element d from GF(p) (not equal to -1 or zero) with smallest absolute value such that #tEd(GF(p)) = cofactor * oddDivisor, #tEd'(GF(p)) = cofactor' * oddDivisor', cofactor = 8, cofactor' = 4 and both subgroup orders oddDivisor and oddDivisor' are prime. In addition, care must be taken to ensure the MOV degree and CM discriminant requirements from Section 4 are met.

These cofactors are chosen so that they are minimal such that the cofactor of the main curve is greater than the cofactor of the twist. It's not possible in this case for the cofactors to be equal, but it is possible for the twist cofactor to be larger. The latter is considered dangerous because algorithms that depend on the cofactor of the curve may be vulnerable if a point on the twist is accepted.

```
Input: a prime p, with p = 1 mod 4
Output: the parameter d defining the curve tEd
1. Set d = 0
2. repeat
     repeat
       if (d > 0) then
         d = -d
       else
         d = -d + 1
       end if
     until d is not a square in GF(p)

     Compute oddDivisor, oddDivisor', cofactor, cofactor' where #tEd(GF(p)) =
     cofactor * oddDivisor, #tEd'(GF(p)) = cofactor' * oddDivisor', cofactor
     and cofactor' are powers of 2 and oddDivisor, oddDivisor' are odd.
   until (cofactor = 8 and cofactor' = 4 and rd is prime and rd' is prime)
3. Output d
```

GenerateCurveTEdwards

6.3. Generators

Any point with the correct order will serve as a generator for the group. The following algorithm computes a possible generator by taking the smallest positive value x in GF(p) (when represented as an integer) such that (x, y) is on the curve and such that (X(P),Y(P)) = 8 * (x, y) has large prime order oddDivisor.

```
Input: a prime p and curve parameters non-square d and
       a = -1 for twisted Edwards (p = 1 mod 4) or
       a = 1 for Edwards (p = 3 mod 4)
Output: a generator point P = (X(P), Y(P)) of order oddDivisor
1. Set x = 0 and found_gen = false
2. while (not found_gen) do
    x = x + 1
    while ((1 - a * x^2) * (1 - d * x^2) is not a quadratic
           residue mod p) do
      x = x + 1
    end while
    Compute an integer s, 0 < s < p, such that
           s^2 * (1 - d * x^2) = 1 - a * x^2 mod p
    Set y = min(s, p - s)

    (X(P), Y(P)) = 8 * (x, y)

    if ((X(P), Y(P)) has order oddDivisor on Ed or tEd, respectively) then
      found_gen = true
    end if
  end while
3. Output (X(P),Y(P))
```

GenerateGen

7. Recommended Curves

For the ~128-bit security level, the prime 2^255-19 is recommended for performance over a wide-range of architectures. This prime is congruent to 1 mod 4 and the above procedure results in the following twisted Edwards curve, called "intermediate25519":

p 2^255-19

d 121665

order 2^252 + 0x14def9dea2f79cd65812631a5cf5d3ed

cofactor 8

In order to be compatible with widespread existing practice, the recommended curve is an isogeny of this curve. An isogeny is a "renaming" of the points on the curve and thus cannot affect the security of the curve:

p 2^255-19

d 370957059346694393431380835087545651895421138798432190163887855330
85940283555

order 2^252 + 0x14def9dea2f79cd65812631a5cf5d3ed

cofactor 8

X(P) 151122213495354007725011514095885315114540126930418572060461132
83949847762202

Y(P) 463168356949264781694283940034751631413079938662562256157830336
03165251855960

The d value in the this curve is much larger than the generated curve and this might slow down some implementations. If this is a problem then implementations are free to calculate on the original curve, with small d as the isogeny map can be merged into the affine transform without any performance impact.

The latter curve is isomorphic to a Montgomery curve defined by v^2 = u^3 + 486662u^2 + u where the maps are:

```
(u, v) = ((1+y)/(1-y), sqrt(-1)*sqrt(486664)*u/x)
(x, y) = (sqrt(-1)*sqrt(486664)*u/v, (u-1)/(u+1)
```

The base point maps onto the Montgomery curve such that u = 9, v = 14
78161944758954479102059356840998688726460613461647528896488183775558
6237401.

The Montgomery curve defined here is equal to the one defined in [curve25519] and the isomorphic twisted Edwards curve is equal to the one defined in [ed25519].

8. Wire-format of field elements

When transmitting field elements in the Diffie-Hellman protocol below, they MUST be encoded as an array of bytes, x, in little-endian order such that x[0] + 256 * x[1] + 256^2 * x[2] + ... + 256^n * x[n] is congruent to the value modulo p and x[n] is minimal. On receiving such an array, implementations MUST mask the (8-log2(p)%8)%8 most-significant bits in the final byte. This is done to preserve compatibility with point formats which reserve the sign bit for use in other protocols and to increase resistance to implementation fingerprinting.

(NOTE: draft-turner-thecurve25519function also says "Implementations MUST reject numbers in the range [2^255-19, 2^255-1], inclusive." but I'm not aware of any implementations that do so.)

9. Elliptic Curve Diffie-Hellman

This section describes how to perform Diffie-Hellman using curves generated by the above procedure. For safety reasons, Diffie-Hellman is performed on the Montgomery isomorphism of the curve and the public values transmitted are u coordinates.

Let U denote the projection map from a point (u,v) on E, to u, extended so that U of the point at infinity is zero. U is surjective onto GF(p) if the v coordinate takes on values in GF(p) and in a quadratic extension of GF(p).

Then DH(s, U(Q)) = U(sQ) is a function defined for all integers s and elements U(Q) of GF(p). Proper implementations use a restricted set of integers for s and only u-coordinates of points Q defined over GF(p). The remainder of this section describes how to compute this function quickly and securely, and use it in a Diffie- Hellman scheme.

Let s be a 255 bits long integer, where s = sum s_i * 2^i with s_i in {0, 1}.

Computing DH(s, u) is done by the following procedure, taken from [curve25519] based on formulas from [montgomery]. All calculations are performed in GF(p), i.e., they are performed modulo p. The parameter a24 is a24 = (486662 - 2) / 4 = 121665.

```
x_1 = u
x_2 = 0
z_2 = 1
x_3 = u
z_3 = 1
For t = 254 down to 0:
    // Conditional swap; see text below.
    (x_2, x_3) = cswap (s_t, x_2, x_3)
    (z_2, z_3) = cswap (s_t, z_2, z_3)
    A = x_2 + z_2
    AA = A^2
    B = x_2 - z_2
    BB = B^2
    E = AA - BB
    C = x_3 + z_3
    D = x_3 - z_3
    DA = D * A
    CB = C * B
    x_3 = (DA + CB)^2
    z_3 = x_1 * (DA - CB)^2
    x_2 = AA * BB
    z_2 = E * (AA + a24 * E)
    // Conditional swap; see text below.
    (x_2, x_3) = cswap (s_t, x_2, x_3)
    (z_2, z_3) = cswap (s_t, z_2, z_3)
Return x_2 * (z_2^(p - 1))
```

In implementing this procedure, due to the existence of side-channels in commodity hardware, it is important that the pattern of memory accesses and jumps not depend on the values of any of the bits of s. It is also important that the arithmetic used not leak information about the integers modulo p (such as having b * c distinguishable from c * c).

The cswap instruction SHOULD be implemented in constant time (independent of s_t) as follows:

```
cswap(s_t, x_2, x_3)
      dummy = s_t * (x_2 - x_3)
      x_2 = x_2 - dummy
      x_3 = x_3 + dummy
Return (x_2, x_3)
```

where s_t is 1 or 0. Alternatively, an implementation MAY use the following:

```
cswap(s_t, x_2, x_3)
      dummy = mask(s_t) AND (x_2 XOR x_3)
      x_2 = x_2 XOR dummy
      x_3 = x_3 XOR dummy
Return (x_2, x_3)
```

where mask(s_t) is the all-1 or all-0 word of the same length as x_2 and x_3, computed, e.g., as mask(s_t) = 1 - s_t. The latter version is often more efficient.

9.1. Diffie-Hellman protocol

The DH function can be used in an ECDH protocol with the recommended curve as follows:

Alice generates 32 random bytes in f[0] to f[31]. She masks the three rightmost bits of f[0] and the leftmost bit of f[31] to zero and sets the second leftmost bit of f[31] to 1. This means that f is of the form 2^254 + 8 * {0, 1, ..., 2^(251) - 1} as a little-endian integer.

Alice then transmits K_A = DH(f, 9) to Bob, where 9 is the number 9.

Bob similarly generates 32 random bytes in g[0] to g[31], applies the same masks, computes K_B = DH(g, 9) and transmits it to Alice.

Alice computes DH(f, DH(g, 9)); Bob computes DH(g, DH(f, 9)) using their generated values and the received input.

Both of them now share K = DH(f, DH(g, 9)) = DH(g, DH(f, 9)) as a shared secret. Alice and Bob can then use a key-derivation function, such as hashing K, to compute a key.

10. Test vectors

The following test vectors are taken from [nacl]. All numbers are shown as little-endian hexadecimal byte strings:

Alice's private key, f:

```
77 07 6d 0a 73 18 a5 7d 3c 16 c1 72 51 b2 66 45
df 4c 2f 87 eb c0 99 2a b1 77 fb a5 1d b9 2c 2a
```

Alice's public key, DH(f, 9):

```
85 20 f0 09 89 30 a7 54 74 8b 7d dc b4 3e f7 5a
0d bf 3a 0d 26 38 1a f4 eb a4 a9 8e aa 9b 4e 6a
```

Bob's private key, g:

```
5d ab 08 7e 62 4a 8a 4b 79 e1 7f 8b 83 80 0e e6
6f 3b b1 29 26 18 b6 fd 1c 2f 8b 27 ff 88 e0 eb
```

Bob's public key, DH(g, 9):

```
de 9e db 7d 7b 7d c1 b4 d3 5b 61 c2 ec e4 35 37
3f 83 43 c8 5b 78 67 4d ad fc 7e 14 6f 88 2b 4f
```

Their shared secret, K:

```
4a 5d 9d 5b a4 ce 2d e1 72 8e 3b f4 80 35 0f 25
e0 7e 21 c9 47 d1 9e 33 76 f0 9b 3c 1e 16 17 42
```

11. References

11.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

11.2. Informative References

[AS] Satoh, T. and K. Araki, "Fermat quotients and the polynomial time discrete log algorithm for anomalous elliptic curves", 1998.

[EBP] ECC Brainpool, "ECC Brainpool Standard Curves and Curve Generation", October 2005, <http://www.ecc-brainpool.org/download/Domain-parameters.pdf>.

[NIST] National Institute of Standards, "Recommended Elliptic Curves for Federal Government Use", July 1999, <http://csrc.nist.gov/groups/ST/toolkit/documents/dss/NISTReCur.pdf>.

[S] Semaev, I., "Evaluation of discrete logarithms on some elliptic curves", 1998.

[SC] Bernstein, D. and T. Lange, "SafeCurves: choosing safe curves for elliptic-curve cryptography", June 2014, <http://safecurves.cr.yp.to/>.

[SEC1] Certicom Research, "SEC 1: Elliptic Curve Cryptography", September 2000, <http://www.secg.org/collateral/sec1_final.pdf>.

[Smart] Smart, N., "The discrete logarithm problem on elliptic curves of trace one", 1999.

[curve25519] Bernstein, D., "Curve25519 -- new Diffie-Hellman speed records", 2006, <http://www.iacr.org/cryptodb/archive/2006/PKC/3351/3351.pdf>.

[ed25519] Bernstein, D., Duif, N., Lange, T., Schwabe, P., and B. Yang, "High-speed high-security signatures", 2011, <http://ed25519.cr.yp.to/ed25519-20110926.pdf>.

[montgomery] Montgomery, P., "Speeding the Pollard and elliptic curve methods of factorization", 1983, <http://www.ams.org/journals/mcom/1987-48-177/S0025-5718-1987-0866113-7/S0025-5718-1987-0866113-7.pdf>.

[nacl] Bernstein, D., "Cryptography in NaCl", 2009, <http://cr.yp.to/highspeed/naclcrypto-20090310.pdf>.

Author's Address

Adam Langley
Google
345 Spear St
San Francisco, CA 94105
US

Email: agl@google.com