CFRG
Internet-Draft
Intended status: Informational
Expires: August 1, 2015

A. Langley
Google
R. Salz
Akamai Technologies
S. Turner
IECA, Inc.

January 28, 2015

Elliptic Curves for Security
draft-irtf-cfrg-curves-01

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.

Status of This Memo

Copyright Notice

Table of Contents

1. 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, various '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 generating cryptographic elliptic curves over a given prime field. 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 input parameters that would permit 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 when p = 1 mod 4.

2. 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].

3. 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 5.

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].

4. 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.

5. 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. This document does not attempt to articulate all these considerations.

5.1. 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 3 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

5.2. 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 3 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. For 1 mod 4 primes, the cofactors are never equal. If the cofactor of the twist is larger than the cofactor of the curve, algorithms may be vulnerable to a small-subgroup attack if a point on the twist is incorrectly 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. Recommended Curves

For the ~128-bit security level, the prime 2^255-19 is recommended for performance on 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 37095705934669439343138083508754565189542113879843219016388785533085940283555

order 2^252 + 0x14def9dea2f79cd65812631a5cf5d3ed

cofactor 8

```
X(P)  15112221349535400772501151409588531511454012693041857206046113
   283949847762202

Y(P)  46316835694926478169428394003475163141307993866256225615783033
   603165251855960
```

The d value in 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 = 14781619447589544791020593568409986887264606134616475288964881837755586237401.

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].

7. The curve25519 function

The "curve25519" function performs scalar multiplication on the Montgomery form of the above curve. (This is used when implementing Diffie-Hellman.) The function takes a scalar and a u-coordinate as inputs and produces a u-coordinate as output. Although the function works internally with integers, the inputs and outputs are 32-byte strings and this specification defines their encoding.

U-coordinates are elements of the underlying field GF(2^255-19) and are encoded as an array of bytes, u, in little-endian order such that u[0] + 256 * u[1] + 256^2 * u[2] + ... + 256^n * u[n] is congruent to the value modulo p and u[n] is minimal. When receiving such an array, implementations MUST mask the most-significant bit 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.

For example, the following functions implement this in Python, although the Python code is not intended to be performant nor side-channel free:

```
def decodeLittleEndian(b):
    return sum([b[i] << 8*i for i in range(32)])

def decodeUCoordinate(u):
    u_list = [ord(b) for b in u]
    u_list[31] &= 0x7f
    return decodeLittleEndian(u_list)

def encodeUCoordinate(u):
    u = u % p
    return ''.join([chr((u >> 8*i) & 0xff) for i in range(32)])
```

(EDITORS 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.)

Scalars are assumed to be randomly generated bytes. In order to decode 32 bytes into an integer scalar, set the three least significant bits of the first byte and the most significant bit of the last to zero, set the second most significant bit of the last byte to 1 and, finally, decode as little-endian. This means that resulting integer is of the form 2^254 + 8 * {0, 1, ..., 2^(251) - 1}.

```
def decodeScalar(k):
    k_list = [ord(b) for b in k]
    k_list[0] &= 248
    k_list[31] &= 127
    k_list[31] |= 64
    return decodeLittleEndian(k_list)
```

To implement the "curve25519(k, u)" function (where "k" is the scalar and "u" is the u-coordinate) first decode "k" and "u" and then perform the following procedure, taken from [curve25519] and based on formulas from [montgomery]. All calculations are performed in GF(p), i.e., they are performed modulo p. The constant a24 is (486662 - 2) / 4 = 121665.

```
x_1 = u
x_2 = 1
z_2 = 0
x_3 = u
z_3 = 1
swap = 0

For t = 254 down to 0:
    k_t = (k >> t) & 1
    swap ^= k_t
    // Conditional swap; see text below.
    (x_2, x_3) = cswap(swap, x_2, x_3)
    (z_2, z_3) = cswap(swap, z_2, z_3)
    swap = k_t

    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(swap, x_2, x_3)
(z_2, z_3) = cswap(swap, z_2, z_3)
Return x_2 * (z_2^(p - 2))
```

(TODO: Note the difference in the formula from Montgomery's original paper. See https://www.ietf.org/mail-archive/web/cfrg/current/msg05872.html.)

Finally, encode the resulting value as 32 bytes in little-endian order.

When 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 "k". It is also important that the arithmetic used not leak information about the integers modulo p (such as having b*c be distinguishable from c*c).

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

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

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

```
cswap(swap, x_2, x_3):
      dummy = mask(swap) 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(swap)" is the all-1 or all-0 word of the same length as x_2 and x_3, computed, e.g., as mask(swap) = 1 - swap. The latter version is often more efficient.

7.1. Test vectors

```
Input scalar:
  a546e36bf0527c9d3b16154b82465edd62144c0ac1fc5a18506a2244ba449ac4
Input scalar as a number (base 10):
  31029842492115040904895560451863089656472772604678260265531221036453811406496
Input U-coordinate:
  e6db6867583030db3594c1a424b15f7c726624ec26b3353b10a903a6d0ab1c4c
Input U-coordinate as a number:
  34426434033919594451155107781188821651316167215306631574996226621102155684838
Output U-coordinate:
  c3da55379de9c6908e94ea4df28d084f32eccf03491c71f754b4075577a28552

Input scalar:
  4b66e9d4d1b4673c5ad22691957d6af5c11b6421e0ea01d42ca4169e7918ba0d
Input scalar as a number (base 10):
  35156891815674817266734212754503633747128614016119564763269015315466259359304
Input U-coordinate:
  e5210f12786811d3f4b7959d0538ae2c31dbe7106fc03c3efc4cd549c715a493
Input U-coordinate as a number:
  8883857351183929894090759386610649319417338800022198945255395922347792736741
Output U-coordinate:
  95cbde9476e8907d7aade45cb4b873f88b595a68799fa152e6f8f7647aac7957
```

8. Diffie-Hellman

The "curve25519" function can be used in an ECDH protocol as follows:

Alice generates 32 random bytes in f[0] to f[31] and transmits K_A = curve25519(f, 9) to Bob, where 9 is the u-coordinate of the base point and is encoded as a byte with value 9, followed by 31 zero bytes.

Bob similarly generates 32 random bytes in g[0] to g[31] and computes K_B = curve25519(g, 9) and transmits it to Alice.

Alice computes curve25519(f, K_B); Bob computes curve25519(g, K_A) using their generated values and the received input.

Both now share K = curve25519(f, curve25519(g, 9)) = curve25519(g, curve25519(f, 9)) as a shared secret. Alice and Bob can then use a key-derivation function, such as hashing K, to compute a key.

Note that this Diffie-Hellman protocol is not contributory, e.g. if the u-coordinate is zero then the output will always be zero. A contributory Diffie-Hellman function would ensure that the output was unpredictable no matter what the peer's input. This is not a problem for the vast majority of cases but, if a contributory function is specifically required, then "curve25519" should not be used.

8.1. Test vectors

```
Alice's private key, f:
  77076d0a7318a57d3c16c17251b26645df4c2f87ebc0992ab177fba51db92c2a
Alice's public key, curve25519(f, 9):
  8520f0098930a754748b7ddcb43ef75a0dbf3a0d26381af4eba4a98eaa9b4e6a
Bob's private key, g:
  5dab087e624a8a4b79e17f8b83800ee66f3bb1292618b6fd1c2f8b27ff88e0eb
Bob's public key, curve25519(g, 9):
  de9edb7d7b7dc1b4d35b61c2ece435373f8343c85b78674dadfc7e146f882b4f
Their shared secret, K:
  4a5d9d5ba4ce2de1728e3bf480350f25e07e21c947d19e3376f09b3c1e161742
```

9. Acknowledgements

This document merges "draft-black-rpgecc-01" and "draft-turner-thecurve25519function-01". The following authors of those documents wrote much of the text and figures but are not listed as authors on this document: Benjamin Black, Joppe W. Bos, Craig Costello, Patrick Longa, Michael Naehrig and Watson Ladd.

The authors would also like to thank Tanja Lange and Rene Struik for their reviews.

10. References

10.1. Normative References

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

10.2. 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>.

Authors' Addresses

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

Email: agl@google.com

Rich Salz
Akamai Technologies
8 Cambridge Center
Cambridge, MA 02142
US

Email: rsalz@akamai.com

Sean Turner
IECA, Inc.
3057 Nutley Street
Suite 106
Fairfax, VA 22031
US

Email: turners@ieca.com