                      Elliptic Curves for Security

Abstract

   This memo specifies two elliptic curves over prime fields that offer
   a high level of practical security in cryptographic applications,
   including Transport Layer Security (TLS). These curves are intended
   to operate at the ~128-bit and ~224-bit security level, respectively,
   and are generated deterministically based on a list of required
   properties.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

   Since the initial standardization of Elliptic Curve Cryptography (ECC
   [RFC6090]) 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 that 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 by NIST [NIST].

   This memo specifies two elliptic curves ("curve25519" and "curve448")
   that lend themselves to constant-time implementation and an
   exception-free scalar multiplication that is resistant to a wide
   range of side-channel attacks, including timing and cache attacks.
   They are Montgomery curves (where $v^2 = u^3 + A*u^2 + u$) and thus
   have birationally equivalent Edwards versions. Edwards curves
   support the fastest (currently known) complete formulas for the
   elliptic-curve group operations, specifically the Edwards curve
   $x^2 + y^2 = 1 + d*x^2*y^2$ for primes p when p = 3 mod 4, and the
   twisted Edwards curve $-x^2 + y^2 = 1 + d*x^2*y^2$ when p = 1 mod 4.
   The maps to/from the Montgomery curves to their (twisted) Edwards
   equivalents are also given.

This memo also specifies how these curves can be used with the
Diffie-Hellman protocol for key agreement.

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. 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.

A   An element in the finite field GF(p), not equal to -2 or 2.

d   A non-zero element in the finite field GF(p), not equal to
        1, in the case of an Edwards curve, or not equal to -1, in
        the case of a twisted Edwards curve.

order   The order of the prime-order subgroup.

P   A generator point defined over GF(p) of prime order.

U(P)   The u-coordinate of the elliptic curve point P on a
        Montgomery curve.

V(P)   The v-coordinate of the elliptic curve point P on a
        Montgomery curve.

X(P)   The x-coordinate of the elliptic curve point P on a
        (twisted) Edwards curve.

Y(P)   The y-coordinate of the elliptic curve point P on a
        (twisted) Edwards curve.

u, v   Coordinates on a Montgomery curve.

x, y   Coordinates on a (twisted) Edwards curve.

4. Recommended Curves

4.1. Curve25519

   For the ˜128-bit security level, the prime 2^255 - 19 is recommended
   for performance on a wide range of architectures. Few primes of the
   form 2^c-s with s small exist between 2^250 and 2^521, and other
   choices of coefficient are not as competitive in performance. This
   prime is congruent to 1 mod 4, and the derivation procedure in
   Appendix A results in the following Montgomery curve
   v^2 = u^3 + A*u^2 + u, called "curve25519":

   p 2^255 - 19

   A 486662

   order 2^252 + 0x14def9dea2f79cd65812631a5cf5d3ed

   cofactor 8

   U(P) 9

   V(P) 14781619447589544791020593568409986887264606134616475288964881
      8 37755586237401

   The base point is u = 9, v = 14781619447589544791020593568409986887264
   6061346164752889648818 37755586237401.

   This curve is birationally equivalent to a twisted Edwards curve -x^2
   + y^2 = 1 + d*x^2*y^2, called "edwards25519", where:

   p 2^255 - 19

   d 37095705934669439343138083508754565189542113879843219016388785533
      0 85940283555

   order 2^252 + 0x14def9dea2f79cd65812631a5cf5d3ed

   cofactor 8

   X(P) 15112221349535400772501151409588531511454012693041857206046113
      2 83949847762202

   Y(P) 46316835694926478169428394003475163141307993866256225615783033
      6 03165251855960

The birational maps are:

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

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

4.2. Curve448

For the ~224-bit security level, the prime $2^{448} - 2^{224} - 1$ is recommended for performance on a wide range of architectures. This prime is congruent to 3 mod 4, and the derivation procedure in Appendix A results in the following Montgomery curve, called "curve448":

p $2^{448} - 2^{224} - 1$

A 156326

order 2^{446} -
   0x8335dc163bb124b65129c96fde933d8d723a70aadc873d6d54a7bb0d

cofactor 4

U(P) 5

V(P) 35529392678556817526412750206378333480897639938771427183188089 8435169088786967410002932673765864550910142774147268105838985595290 606362

This curve is birationally equivalent to the Edwards curve $x^2 + y^2 = 1 + d*x^2*y^2$ where:

p $2^{448} - 2^{224} - 1$

d 6119758507445291761604232209655533175432196968710166263289689364 15087860042636474891785599283666020414768678979989378147065462815545 017

order 2^{446} -
   0x8335dc163bb124b65129c96fde933d8d723a70aadc873d6d54a7bb0d

cofactor 4

```
X(P)   34539749303972951637400860415053741026665526007518329021640697
       02816456950736723444304817877593406332217083915834204178892412456
       7700732
```

```
Y(P)   36341936214780344527466190394400226717682068034365903014074509
       95903061640833653863431981918493382729650444422309218186805267490
       09182718
```

The birational maps are:

```
  (u, v) = ((y-1)/(y+1), sqrt(156324)*u/x)
  (x, y) = (sqrt(156324)*u/v, (1+u)/(1-u))
```

Both of those curves are also 4-isogenous to the following Edwards
curve x^2 + y^2 = 1 + d*x^2*y^2, called "edwards448", where:

p 2^448 - 2^224 - 1

d -39081

order 2^446 -
       0x8335dc163bb124b65129c96fde933d8d723a70aadc873d6d54a7bb0d

cofactor 4

```
X(P)   22458004029592430018760433409989603624678964163256413424612546
       16869504154674060329090291928693579532825780320751464461736746026
       35247710
```

```
Y(P)   29881921007848149267601793044393067343754404015408024209592824
       13723315061898358760035368786554187847339823032335034625005315450
       62832660
```

The 4-isogeny maps between the Montgomery curve and this Edwards
curve are:

```
  (u, v) = (y^2/x^2, (2 - x^2 - y^2)*y/x^3)
  (x, y) = (4*v*(u^2 - 1)/(u^4 - 2*u^2 + 4*v^2 + 1),
            -(u^5 - 2*u^3 - 4*u*v^2 + u)/
            (u^5 - 2*u^2*v^2 - 2*u^3 - 2*v^2 + u))
```

The curve edwards448 defined here is also called "Goldilocks" and is
equal to the one defined in [goldilocks].

5. The X25519 and X448 Functions

   The "X25519" and "X448" functions perform scalar multiplication on
   the Montgomery form of the above curves. (This is used when
   implementing Diffie-Hellman.) The functions take a scalar and a
   u-coordinate as inputs and produce a u-coordinate as output.
   Although the functions work internally with integers, the inputs and
   outputs are 32-byte strings (for X25519) or 56-byte strings (for
   X448) and this specification defines their encoding.

   The u-coordinates are elements of the underlying field $GF(2^{255} - 19)$
   or $GF(2^{448} - 2^{224} - 1)$ and are encoded as an array of bytes, u, in
   little-endian order such that $u[0] + 256*u[1] + 256^2*u[2] + ... +
   256^{(n-1)}*u[n-1]$ is congruent to the value modulo p and $u[n-1]$ is
   minimal. When receiving such an array, implementations of X25519
   (but not X448) MUST mask the most significant bit in the final byte.
   This is done to preserve compatibility with point formats that
   reserve the sign bit for use in other protocols and to increase
   resistance to implementation fingerprinting.

   Implementations MUST accept non-canonical values and process them as
   if they had been reduced modulo the field prime. The non-canonical
   values are $2^{255} - 19$ through $2^{255} - 1$ for X25519 and $2^{448} - 2^{224}
   - 1$ through $2^{448} - 1$ for X448.

   The following functions implement this in Python, although the Python
   code is not intended to be performant nor side-channel free. Here,
   the "bits" parameter should be set to 255 for X25519 and 448 for
   X448:

```
<CODE BEGINS>
def decodeLittleEndian(b, bits):
    return sum([b[i] << 8*i for i in range((bits+7)/8)])

def decodeUCoordinate(u, bits):
    u_list = [ord(b) for b in u]
    # Ignore any unused bits.
    if bits % 8:
        u_list[-1] &= (1<<(bits%8))-1
    return decodeLittleEndian(u_list, bits)

def encodeUCoordinate(u, bits):
    u = u % p
    return ''.join([chr((u >> 8*i) & 0xff)
                    for i in range((bits+7)/8)])
<CODE ENDS>
```

Scalars are assumed to be randomly generated bytes. For X25519, in
order to decode 32 random bytes as 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 the
resulting integer is of the form 2^254 plus eight times a value
between 0 and 2^251 - 1 (inclusive). Likewise, for X448, set the two
least significant bits of the first byte to 0, and the most
significant bit of the last byte to 1. This means that the resulting
integer is of the form 2^447 plus four times a value between 0 and
2^445 - 1 (inclusive).

```
<CODE BEGINS>
def decodeScalar25519(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, 255)

def decodeScalar448(k):
    k_list = [ord(b) for b in k]
    k_list[0] &= 252
    k_list[55] |= 128
    return decodeLittleEndian(k_list, 448)
<CODE ENDS>
```

To implement the X25519(k, u) and X448(k, u) functions (where k is
the scalar and u is the u-coordinate), first decode k and u and then
perform the following procedure, which is 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 for curve25519/X25519 and (156326 - 2) / 4
= 39081 for curve448/X448.

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

   For t = bits-1 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))
```

   (Note that these formulas are slightly different from Montgomery's
   original paper. Implementations are free to use any correct
   formulas.)

   Finally, encode the resulting value as 32 or 56 bytes in little-
   endian order. For X25519, the unused, most significant bit MUST be
   zero.

The cswap function SHOULD be implemented in constant time (i.e.,
independent of the swap argument). For example, this can be done as
follows:

```
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) = 0 - swap.

5.1. Side-Channel Considerations

   X25519 and X448 are designed so that fast, constant-time
   implementations are easier to produce. The procedure above ensures
   that the same sequence of field operations is performed for all
   values of the secret key, thus eliminating a common source of side-
   channel leakage. However, this alone does not prevent all side-
   channels by itself. 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, for example by having b*c be
   distinguishable from c*c. On some architectures, even primitive
   machine instructions, such as single-word division, can have variable
   timing based on their inputs.

   Side-channel attacks are an active research area that still sees
   significant, new results. Implementors are advised to follow this
   research closely.

5.2. Test Vectors

   Two types of tests are provided. The first is a pair of test vectors
   for each function that consist of expected outputs for the given
   inputs. The inputs are generally given as 64 or 112 hexadecimal
   digits that need to be decoded as 32 or 56 binary bytes before
   processing.

   X25519:

   Input scalar:
     a546e36bf0527c9d3b16154b82465edd62144c0ac1fc5a18506a2244ba449ac4
   Input scalar as a number (base 10):
     31029842492115040904895560451863089656
     47277260467826026553122103645381140649 6
   Input u-coordinate:
     e6db6867583030db3594c1a424b15f7c726624ec26b3353b10a903a6d0ab1c4c
   Input u-coordinate as a number (base 10):
     34426434033919594451155107781188821651
     31616721530663157499622662110215568483 8
   Output u-coordinate:
     c3da55379de9c6908e94ea4df28d084f32eccf03491c71f754b4075577a28552

   Input scalar:
     4b66e9d4d1b4673c5ad22691957d6af5c11b6421e0ea01d42ca4169e7918ba0d
   Input scalar as a number (base 10):
     35156891815674817266734212754503633747
     12861401611956476326901531546625935930 4
   Input u-coordinate:
     e5210f12786811d3f4b7959d0538ae2c31dbe7106fc03c3efc4cd549c715a493
   Input u-coordinate as a number (base 10):
     88838573511839298940907593866106493194
     17338800022198945255395922347792736741
   Output u-coordinate:
     95cbde9476e8907d7aade45cb4b873f88b595a68799fa152e6f8f7647aac7957

     X448:

     Input scalar:
       3d262fddf9ec8e88495266fea19a34d28882acef045104d0d1aae121
       700a779c984c24f8cdd78fbff44943eba368f54b29259a4f1c600ad3
     Input scalar as a number (base 10):
       599189175373889640278375601614521325615723085 6
       085026129926891459468622403380588640249457727
       683869421921443004045221642549886377526240828
     Input u-coordinate:
       06fce640fa3487bfda5f6cf2d5263f8aad88334cbd07437f020f08f9
       814dc031ddbdc38c19c6da2583fa5429db94ada18aa7a7fb4ef8a086
     Input u-coordinate as a number (base 10):
       382239910814107330116229961234899377031416365
       240571325148346555922438025162094455820962429
       142971339584360034337310079791515452463053830
     Output u-coordinate:
       ce3e4ff95a60dc6697da1db1d85e6afbdf79b50a2412d7546d5f239f
       e14fbaadeb445fc66a01b0779d98223961111e21766282f73dd96b6f

     Input scalar:
       203d494428b8399352665ddca42f9de8fef600908e0d461cb021f8c5
       38345dd77c3e4806e25f46d3315c44e0a5b4371282dd2c8d5be3095f
     Input scalar as a number (base 10):
       633254335906970592779259481534862372382525155
       252028961056404001332122152890562527156973881
       968934311400345568203929409663925541994577184
     Input u-coordinate:
       0fbcc2f993cd56d3305b0b7d9e55d4c1a8fb5dbb52f8e9a1e9b6201b
       165d015894e56c4d3570bee52fe205e28a78b91cdfbde71ce8d157db
     Input u-coordinate as a number (base 10):
       622761797758325444462922068431234180649590390
       024811299761625153767228042600197997696167956
       134770744996690267634159427999832340166786063
     Output u-coordinate:
       884a02576239ff7a2f2f63b2db6a9ff37047ac13568e1e30fe63c4a7
       ad1b3ee3a5700df34321d62077e63633c575c1c954514e99da7c179d

   The second type of test vector consists of the result of calling the
   function in question a specified number of times. Initially, set k
   and u to be the following values:

   For X25519:
     0900000000000000000000000000000000000000000000000000000000000000
   For X448:
     05000000000000000000000000000000000000000000000000000000000000
     00000000000000000000000000000000000000000000000000000000000000

   For each iteration, set k to be the result of calling the function
   and u to be the old value of k. The final result is the value left
   in k.

   X25519:

   After one iteration:
       422c8e7a6227d7bca1350b3e2bb7279f7897b87bb6854b783c60e80311ae3079
   After 1,000 iterations:
       684cf59ba83309552800ef566f2f4d3c1c3887c49360e3875f2eb94d99532c51
   After 1,000,000 iterations:
       7c3911e0ab2586fd864497297e575e6f3bc601c0883c30df5f4dd2d24f665424

   X448:

   After one iteration:
       3f482c8a9f19b01e6c46ee9711d9dc14fd4bf67af30765c2ae2b846a
       4d23a8cd0db897086239492caf350b51f833868b9bc2b3bca9cf4113
   After 1,000 iterations:
       aa3b4749d55b9daf1e5b00288826c467274ce3ebbdd5c17b975e09d4
       af6c67cf10d087202db88286e2b79fceea3ec353ef54faa26e219f38
   After 1,000,000 iterations:
       077f453681caca3693198420bbe515cae0002472519b3e67661a7e89
       cab94695c8f4bcd66e61b9b9c946da8d524de3d69bd9d9d66b997e37

6. Diffie-Hellman

6.1. Curve25519

   The X25519 function can be used in an Elliptic Curve Diffie-Hellman
   (ECDH) protocol as follows:

   Alice generates 32 random bytes in a[0] to a[31] and transmits K_A =
   X25519(a, 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 b[0] to b[31], computes
   K_B = X25519(b, 9), and transmits it to Alice.

   Using their generated values and the received input, Alice computes
   X25519(a, K_B) and Bob computes X25519(b, K_A).

   Both now share K = X25519(a, X25519(b, 9)) = X25519(b, X25519(a, 9))
   as a shared secret. Both MAY check, without leaking extra
   information about the value of K, whether K is the all-zero value and
   abort if so (see below). Alice and Bob can then use a key-derivation
   function that includes K, K_A, and K_B to derive a symmetric key.

   The check for the all-zero value results from the fact that the
   X25519 function produces that value if it operates on an input
   corresponding to a point with small order, where the order divides
   the cofactor of the curve (see Section 7). The check may be
   performed by ORing all the bytes together and checking whether the
   result is zero, as this eliminates standard side-channels in software
   implementations.

   Test vector:

   Alice's private key, a:
     77076d0a7318a57d3c16c17251b26645df4c2f87ebc0992ab177fba51db92c2a
   Alice's public key, X25519(a, 9):
     8520f0098930a754748b7ddcb43ef75a0dbf3a0d26381af4eba4a98eaa9b4e6a
   Bob's private key, b:
     5dab087e624a8a4b79e17f8b83800ee66f3bb1292618b6fd1c2f8b27ff88e0eb
   Bob's public key, X25519(b, 9):
     de9edb7d7b7dc1b4d35b61c2ece435373f8343c85b78674dadfc7e146f882b4f
   Their shared secret, K:
     4a5d9d5ba4ce2de1728e3bf480350f25e07e21c947d19e3376f09b3c1e161742

6.2. Curve448

   The X448 function can be used in an ECDH protocol very much like the
   X25519 function.

   If X448 is to be used, the only differences are that Alice and Bob
   generate 56 random bytes (not 32) and calculate K_A = X448(a, 5) or
   K_B = X448(b, 5), where 5 is the u-coordinate of the base point and
   is encoded as a byte with value 5, followed by 55 zero bytes.

   As with X25519, both sides MAY check, without leaking extra
   information about the value of K, whether the resulting shared K is
   the all-zero value and abort if so.

   Test vector:

   Alice's private key, a:
     9a8f4925d1519f5775cf46b04b5800d4ee9ee8bae8bc5565d498c28d
     d9c9baf574a9419744897391006382a6f127ab1d9ac2d8c0a598726b
   Alice's public key, X448(a, 5):
     9b08f7cc31b7e3e67d22d5aea121074a273bd2b83de09c63faa73d2c
     22c5d9bbc836647241d953d40c5b12da88120d53177f80e532c41fa0
   Bob's private key, b:
     1c306a7ac2a0e2e0990b294470cba339e6453772b075811d8fad0d1d
     6927c120bb5ee8972b0d3e21374c9c921b09d1b0366f10b65173992d
   Bob's public key, X448(b, 5):
     3eb7a829b0cd20f5bcfc0b599b6feccf6da4627107bdb0d4f345b430
     27d8b972fc3e34fb4232a13ca706dcb57aec3dae07bdc1c67bf33609
   Their shared secret, K:
     07fff4181ac6cc95ec1c16a94a0f74d12da232ce40a77552281d282b
     b60c0b56fd2464c335543936521c24403085d59a449a5037514a879d

7. Security Considerations

   The security level (i.e., the number of "operations" needed for a
   brute-force attack on a primitive) of curve25519 is slightly under
   the standard 128-bit level. This is acceptable because the standard
   security levels are primarily driven by much simpler, symmetric
   primitives where the security level naturally falls on a power of
   two. For asymmetric primitives, rigidly adhering to a power-of-two
   security level would require compromises in other parts of the
   design, which we reject. Additionally, comparing security levels
   between types of primitives can be misleading under common threat
   models where multiple targets can be attacked concurrently
   [bruteforce].

The ~224-bit security level of curve448 is a trade-off between
performance and paranoia. Large quantum computers, if ever created,
will break both curve25519 and curve448, and reasonable projections
of the abilities of classical computers conclude that curve25519 is
perfectly safe. However, some designs have relaxed performance
requirements and wish to hedge against some amount of analytical
advance against elliptic curves and thus curve448 is also provided.

Protocol designers using Diffie-Hellman over the curves defined in
this document must not assume "contributory behaviour". Specially,
contributory behaviour means that both parties' private keys
contribute to the resulting shared key. Since curve25519 and
curve448 have cofactors of 8 and 4 (respectively), an input point of
small order will eliminate any contribution from the other party's
private key. This situation can be detected by checking for the all-
zero output, which implementations MAY do, as specified in Section 6.
However, a large number of existing implementations do not do this.

Designers using these curves should be aware that for each public
key, there are several publicly computable public keys that are
equivalent to it, i.e., they produce the same shared secrets. Thus
using a public key as an identifier and knowledge of a shared secret
as proof of ownership (without including the public keys in the key
derivation) might lead to subtle vulnerabilities.

Designers should also be aware that implementations of these curves
might not use the Montgomery ladder as specified in this document,
but could use generic, elliptic-curve libraries instead. These
implementations could reject points on the twist and could reject
non-minimal field elements. While not recommended, such
implementations will interoperate with the Montgomery ladder
specified here but may be trivially distinguishable from it. For
example, sending a non-canonical value or a point on the twist may
cause such implementations to produce an observable error while an
implementation that follows the design in this text would
successfully produce a shared key.

8. References

8.1. Normative References

   [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate
              Requirement Levels", BCP 14, RFC 2119,
              DOI 10.17487/RFC2119, March 1997,
              <http://www.rfc-editor.org/info/rfc2119>.

8.2. Informative References

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

   [bruteforce]
              Bernstein, D., "Understanding brute force", April 2005,
              <http://cr.yp.to/snuffle/bruteforce-20050425.pdf>.

   [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://link.springer.com/
              chapter/10.1007/978-3-642-23951-9_9>.

   [goldilocks]
              Hamburg, M., "Ed448-Goldilocks, a new elliptic curve",
              2015, <http://eprint.iacr.org/2015/625.pdf>.

   [montgomery]
              Montgomery, P., "Speeding the Pollard and Elliptic Curve
              Methods of Factorization", January 1987,
              <http://www.ams.org/journals/mcom/1987-48-177/
              S0025-5718-1987-0866113-7/S0025-5718-1987-0866113-7.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>.

   [reducing] Menezes, A., Okamoto, T., and S. Vanstone, "Reducing
              elliptic curve logarithms to logarithms in a finite
              field", DOI 10.1109/18.259647, 1993,
              <http://ieeexplore.ieee.org/xpl/
              articleDetails.jsp?arnumber=259647>.

   [RFC6090] McGrew, D., Igoe, K., and M. Salter, "Fundamental Elliptic
              Curve Cryptography Algorithms", RFC 6090,
              DOI 10.17487/RFC6090, February 2011,
              <http://www.rfc-editor.org/info/rfc6090>.

   [safecurves]
             Bernstein, D. and T. Lange, "SafeCurves: choosing safe
             curves for elliptic-curve cryptography", Oct 2013,
             <http://safecurves.cr.yp.to/>.

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

   [SEC1]   Certicom Research, "SEC 1: Elliptic Curve Cryptography",
             September 2000, <http://www.secg.org/sec1-v2.pdf>.

   [semaev]   Semaev, I., "Evaluation of discrete logarithms on some
             elliptic curves", 1998, <http://www.ams.org/journals/
             mcom/1998-67-221/S0025-5718-98-00887-4/
             S0025-5718-98-00887-4.pdf>.

   [smart]   Smart, N., "The Discrete Logarithm Problem on Elliptic
             Curves of Trace One", 1999,
             <http://www.hpl.hp.com/techreports/97/HPL-97-128.pdf>.

Appendix A. Deterministic Generation

   This section specifies the procedure that was used to generate the
   above curves; specifically, it defines how to generate the parameter
   A of the Montgomery curve $y^2 = x^3 + A*x^2 + x$. This procedure is
   intended to be as objective as can reasonably be achieved so that
   it's clear that no untoward considerations influenced the choice of
   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), so
   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.

   The value (A-2)/4 is used in several of the elliptic curve point
   arithmetic formulas. For simplicity and performance reasons, it is
   beneficial to make this constant small, i.e., to choose A so that
   (A-2) is a small integer that is divisible by four.

   For each curve at a specific security level:

   1. The trace of Frobenius MUST NOT be in {0, 1} in order to rule out
       the attacks described in [smart], [satoh], and [semaev], as in
       [brainpool] and [safecurves].

   2. MOV Degree [reducing]: the embedding degree MUST be greater than
       (order - 1) / 100, as in [brainpool] and [safecurves].

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

A.1. p = 1 mod 4

   For primes congruent to 1 mod 4, the minimal cofactors of the curve
   and its twist are either {4, 8} or {8, 4}. We choose a curve with
   the latter cofactors so that any algorithms that take the cofactor
   into account don't have to worry about checking for points on the
   twist, because the twist cofactor will be the smaller of the two.

   To generate the Montgomery curve, we find the minimal, positive A
   value such that A > 2 and (A-2) is divisible by four and where the
   cofactors are as desired. The find1Mod4 function in the following
   Sage script returns this value given p:

   <CODE BEGINS>

```
def findCurve(prime, curveCofactor, twistCofactor):
    F = GF(prime)

    for A in xrange(3, int(1e9)):
        if (A-2) % 4 != 0:
          continue

        try:
          E = EllipticCurve(F, [0, A, 0, 1, 0])
        except:
          continue

        groupOrder = E.order()
        twistOrder = 2*(prime+1)-groupOrder

        if (groupOrder % curveCofactor == 0 and
            is_prime(groupOrder // curveCofactor) and
            twistOrder % twistCofactor == 0 and
            is_prime(twistOrder // twistCofactor)):
            return A

def find1Mod4(prime):
    assert((prime % 4) == 1)
    return findCurve(prime, 8, 4)
```

   <CODE ENDS>

                  Generating a curve where p = 1 mod 4

A.2. p = 3 mod 4

   For a prime congruent to 3 mod 4, both the curve and twist cofactors
   can be 4, and this is minimal. Thus, we choose the curve with these
   cofactors and minimal, positive A such that A > 2 and (A-2) is
   divisible by four. The find3Mod4 function in the following Sage
   script returns this value given p:

```
<CODE BEGINS>
def find3Mod4(prime):
    assert((prime % 4) == 3)
    return findCurve(prime, 4, 4)
<CODE ENDS>
```

                 Generating a curve where p = 3 mod 4

A.3. Base Points

   The base point for a curve is the point with minimal, positive u
   value that is in the correct subgroup. The findBasepoint function in
   the following Sage script returns this value given p and A:

```
<CODE BEGINS>
def findBasepoint(prime, A):
    F = GF(prime)
    E = EllipticCurve(F, [0, A, 0, 1, 0])

    for uInt in range(1, 1e3):
      u = F(uInt)
      v2 = u^3 + A*u^2 + u
      if not v2.is_square():
        continue
      v = v2.sqrt()

      point = E(u, v)
      pointOrder = point.order()
      if pointOrder > 8 and pointOrder.is_prime():
        return point
<CODE ENDS>
```

                        Generating the base point

Authors' Addresses

   Adam Langley
   Google
   345 Spear Street
   San Francisco, CA 94105
   United States

   Email: agl@google.com

   Mike Hamburg
   Rambus Cryptography Research
   425 Market Street, 11th Floor
   San Francisco, CA 94105
   United States

   Email: mike@shiftleft.org

   Sean Turner
   sn3rd

   Email: sean@sn3rd.com