Elliptic curve 2y^2=x^3+x over field size 8^91+5
<draft-brown-ec-2y2-x3-x-mod-8-to-91-plus-5-01.txt>

Abstract

  This document specifies a special elliptic curve with complex
  multiplication (by i) and a compact description (see title). This
  curve is recommended for cryptographic use in a strongest-link
  combination with dissimilar elliptic curves (e.g. NIST P-256,
  Curve25519, extension-field curves, etc.) as a defense in depth
  against an unlikely, unforeseen attack on otherwise preferred
  elliptic curves. The curve equation 2y^2=x^3+x is the Montgomery
  form of a curve y^2=x^3-x in a class of curves y^2=x^3-ax suggested
  by Miller in the first published paper elliptic curve cryptography,
  and an endomorphism usable for efficiency, an idea of Koblitz. The
  field size 8^91+5 is prime, and is relatively efficient and
  compactly described for its bit-size (273 bits). The document
  specifies some practical details such as: encoding a point (on the
  curve) into 34 bytes, public key validation, encoding a private key
  into 34 bytes, and encoding 34 bytes into a point. The document
  also provides pseudocode, motivation, and security considerations.

Status of This Memo

Copyright Notice

Table of Contents

[1]. Introduction

This document specifies some conventions for using the elliptic curve $2y^2=x^3+x$ over the field of size $8^{91}+5$ in cryptography.

This draft focuses on applications to Diffie--Hellman exchange.

[1.1](). **Background**

This document presumes that its reader already has familiarity with elliptic curve cryptography.

The symbol '^', as used in '2y^2=x^3+x' and '8^91+5' means exponentiation, also known as powering. In particular, it does not mean bit-wise exclusive-or (as in the C programming language operator). For example, y^3=yyy (or y*y*y, if * is used for multiplication.)

In particular, p=8^91+5 is a (positive) prime number. Its encoding into bytes, using little-endian ordering (least significant bytes first), requires 35 bytes, and has the form {5,0,0,...,2}, with the first byte equal to 5, the last 2, and the 33 intermediate bytes are each 0. A byte encoding of p is not needed for this document, and is only shown here for illustrative purposes. Its hexadecimal representation (i.e. big-endian, base 16), is 20...05, with 67 zeros between 2 and 5.

[1.2](). **Motivation**

The motivations for curve 2y^2=x^3+x over field 8^91+5 are discussed in [Appendix B]() (and in [[B1]()]).

In short, the main motivation is that the description of the curve is very short (for an elliptic curve), thereby reducing the room for a secretly embedded trapdoor, as in [[Teske]()].

[2](). **Requirements Language ([RFC 2119]())**

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]() [[BCP14]()].

[3](). **Encoding a point into 34 bytes**

Elliptic curve cryptography uses points for public keys and raw shared secrets. A point can be defined as either pair (x,y), where x and y are field elements, or a special point O located at infinity. Field elements for this curve are integers modulo 8^91+5.

Note: for practicality, an implementation will usually represent the x-coordinate as a ratio (X:Z) of field elements. This specification ignores that detail, assuming x has been normalized to (x:1).

To interoperably communicate, points must be encoded as byte
strings.

This draft specifies an encoding of finite points (x,y) as strings
of 34 bytes, as described in the following sections.

  Note: The 34-byte encoding is not injective. Each point is
  generally among a group of four points that share the same byte
  encoding.

  Note: The 34-byte encoding is not surjective. Approximately half
  of 34-byte strings do not encode a finite point (x,y).

  Note: In many typical ECC schemes, the 34-byte encoding works
  well, despite being neither injective nor surjective.

3.1. Encoding a point into bytes

In short: a finite point (x,y) by the little-endian byte
representation of x or -x, whichever fits into 34 bytes.

In detail: a point (x,y) is encoded into 34 bytes b[0], b[1], ...,
b[33], as follows.

First, ensure that x is fully reduced mod p=8^91+5, so that

 $0 <= x < 8^{91}+5$.

Second, further reduce x by a flipping its sign. Let

 $x' =: \min(x,p-x) \mod 2^{272}$.

Third, set the byte string b to be the little-endian encoding of the
reduced integer x', by finding the unique integers b[i] such that
0<=b[i]<256 and

 $(x' \mod 2^{272}) = b[0] + b[1]*256 + ... + b[33]*256^{33}$.

Pseudocode can be found in Appendix C.

3.2. Decoding bytes into a point

In short: the bytes are little-endian decoded into an integer which
becomes the x-coordinate. The y-coordinate is implicit (in
Diffie--Hellman).

```
+-----------------------------------------------------------+
|                                                           |
|        \  W  / /A\  |R) |N | I |N | /G   !                |
|        \/ \/ /   \ |^\ | \| | | \| \_7  0                 |
|                                                           |
|                                                           |
|   WARNING: Some byte strings b decode to an invalid       |
|   point (x,y) that does not belong to the curve           |
|   2y^2=x^3+x.  In some situations, such invalid b can     |
|   lead to a severe attack.  In these situations, the      |
|   decoded point (x,y) MUST be validated, as described     |
|   below in Section 4.                                     |
|                                                           |
+-----------------------------------------------------------+
```

(TO DO: if y is needed explicitly, then one of y matching x must be
solved; in that case, y-needing application, after a point (x,y) is
encoded to b, it should be replaced by (x',y'), where (x',y') is the
decoding of b. In the rare case that x and x' do not match, then
(x,y) should be re-generated or rejected.)

In greater detail: if the 34 bytes are b[0], b[1], ..., b[33], each
with an integer value between 0 and 255 inclusive, then

 x = b[0] + b[1]256 + ... + b[i]256^i + ... + b[33]256^33

4. Point validation

In elliptic curve cryptography, scalar multiplying an invalid public
key by a private key risks leaking information about the private
key.

For curve $2y^2=x^3+x$ over $8^{91}+5$, the underlying attacks are a
little milder than the average a typical elliptic curve.

4.1. When a public key MAY, SHOULD or MUST be validated

Every public key MAY be validated, just as an extra precaution, or
defense in depth.

If an implementation cannot to afford validate every public key, but
also cannot follow the more complicated rules that follow, the
implementation can use the following simple rule:

```
+------------------------------------------------------------------+
|   STATIC                                                         |
|   SECRET                                                         |
|    KEY       ------\                        _  ___              |
|     +             )   PUBLIC |\/| | | (_`  |                    |
|   UNPROVEN   ------/    KEY   |  | \_/ ._)  |   BE VALIDATED.   |
|    PUBLIC                                                        |
|     KEY                                                         |
+------------------------------------------------------------------+
```

However, the more complicated rules described below aim to only
impose a requirement to validate when there is a known attack, when
a requirement is absolutely necessary.

Public key validation has a non-negligible cost, and is sometimes
not necessary for security. Here are some criteria under which
public key validation becomes a SHOULD or MUST

1) The public key P potentially originates from an potential
   adversary.

2) The public key P will be used in Diffie--Hellman key agreement to
   compute a value sP, where:

   a) s is a secret

   b) s will be or has been re-used to compute other values (other
      than just sP)

   c) proof of knowledge of sP has not been received (see Note)

   d) proof of knowledge of sP has been requested (see Note)

   e) the direct value of sP has been requested

   f) sP is computed by one of the following methods:

      I) first explicitly decompressing P to (x,y), but without
         checking (x,y) is on the true curve or that intermediate
         candidate square root are correct, second computing sP
         using formulas that are correct even if P lies on some
         other (false) curve.

        II) using a (1 or 2 dimensinoal) Montgomery ladder, or
            similar method, that ensures P is internally represented
            as point on the curve or its twist, regardless of the
            bytes used to deliver P,

  3) The public key P will be used in some other algorithm, such as
     Menezes--Qu--Vanstone key agreement, that combines P with a
     long-term (static) secret s and an ephemeral secret e.

  4) The public key P will be used in some algorithm, such as
     signature verification algorithm, that does not combine P with
     any secrets.

     g) The algorithm involving P is used primarily to prove some
        property of P is itself, such as proof-of-possession.

Note: proof of knowledge of sP can take many forms. For example,
deriving an message authetnication code key (HMAC) from sP and then
computig a tag of a knowable message. For a second example,
deriving a symmetric encryption key from sP, then encrypting a
message that is non-random in the sense it contains enough
redundancy that decryption proves knowledge of sP. Obviously,
direct exposure (e) of sP is a proof of knowledge of sP.

Public key validation MUST be done when the following sets of
criteria hold, because of the attacks summarized.

- {1,2,a,b,f,I}: The attacker pre-computes values P that
  decompress to a point (x,y) of a very low-order point P that is
  neither on the curve nor its twist, but on some other false curve.
  Finding such P may be hard. The adversary can prove knowledge of
  sP by guessing s mod ord(P), due to their very low order, though
  many proofs will fail. Using these points P finds the secret s
  quickly, by the Chinese remainder theorem. The number of failed
  interactions with the owner of s may be in the thousands.
  Fortunately, in this situtation public key validation is very
  fast, since it can be done by checking that $2y^2=x^3+x$.

- {1,2,a,b,c,f,I}: The attacker pre-computes values P that
  decompress to a point (x,y) of a very low-order point P that is
  neither on the curve nor its twist, but on some other false curve.
  Finding such P may be hard. The attacker guesses (s mod ord(P)).
  The attacker ascertains whether the guess is correct by conducting
  a reaction attack, seeing whether the owner of s acts as though is
  proper. Using these points P finds the secret s quickly, by the
  Chinese remainder theorem. Fortunately, in this situtation public
  key validation is very fast, since it can be done by checking that
  $2y^2=x^3+x$.

- {1,2,a,b,c,e,f,II}: The attacker, (easily) pre-computes moderately
  low-order points P on the twist, receives sP, and solves the
  discrete log (s mod ord(P)). The attack takes computation of
  about 2^65 group operations. Only esotertic protocols require sP
  to be directly exposed: usually sP is passed through a 1-way hash
  before any other use.

- {1,2,a,b,c,d,f,II}: The attacker (easily) pre-computes moderately
  low-order points P on the twists, receives proof-of-knowledge of
  sP, exhaustively searches values of (s mod ord(P)). The attack
  takes computation of at least 2^70 group operations.

If an implementation of the compute of sP from s and P can be used
 in one of the situtations above, then it MUST either validate P
before
computing sP, or it must have a clearly documented input flag to
indicate whether P can be trusted.

Public key validation SHOULD be done in the following situations,
because of the following attacks:

- {1,2,a,b,d,f,II}: The attacker (easily) generates a point P on the
  twist of order 1526119141 and makes approximately 1526119141/2
  guesses g such gP = sP, uses the guesses as proof of knowledge of
  sP towards the owner of the secret s. This involves the owner of
  s unwittingly or unstoppably participating in about half a billion
  failed crypto operations. The attacker then learns about 30 bits
  of the secret s, which could be used to speed up on discrete
  logarithm attack on s to cost of about 2^120 group operations.

Public key validation SHOULD be also done in the following
situtations, either because it is so efficient (in 2,f,I), or
because of potential attacks, in order of decreasing risk (as
estimated by me):

- {1,2,a,b,e}
- {1,2,a,b,c,d}
- {1,2,a,b,f,I}
- {1,2,a,b}
- {1,2,f,I,3}
- {1,2,f,I,4}
- {2,f,I}

Note that the twist has order:

2^2 * 5 * 1526119141 * 788069478421 * 182758084524062861993 *
34524649304516773003600525204032 8546941

OLD TEXT BELOW:

If a party Alice has a secret key a for the curve 2y^2=x^3+x over
8^91+5, which she will to establish two (hashed) Diffie--Hellman
keys, agreement with 2 or more public keys from other parties, say
Bob and Charles, then Alice SHOULD apply public-key validation to
the public key points of the other parties (Bob and Charlies).

MUST undergo validation if they are
combined with private keys as part of multiple Diffie--Hellman
computations:

Additionally, public keys SHOULD undergo validation if they are
received from an unauthenticated source, even if the scalar is
ephemeral or public.

ATTEMPT (TO BE CONFIRMED):

4.2. How to validate a point (given only x)

Upon decoding the 34 bytes into x, the next step is to compute
z=2(x^3+x). Then one checks if z has a nonzero square root. If z
has a nonzero square root, then the represented point is valid,
otherwise it is not valid.

Equivalently, one can check that x^3 + x has no square root (that
is, x^3+x is a quadratic non-residue).

To check z for a square root, one can compute the Legendre symbol
(z/p) and check that is 1. (Equivalently, one can check that
((x^3+x)/p)=-1.)

The Legendre symbol can be computed using Gauss' quadratic
reciprocity law, but this requires implementing modular integer
arithmetic for moduli smaller than 8^91+5.

More slowly, but perhaps more simply, one compute the Legendre
symbol using powering in the field: (z/p) = z^((p-1)/2) =
z^(2^272+2). This will have value 0,1 or p-1 (which is equivalent
to -1).

More generally, in signature applications, where the y-coordinate is
also needed, the computation of y, which involves computing a square
root will generally include a check that x is valid.

The curve 2y^2=x^3+x is not twist-secure. So, using the Montgomery
ladder for scalar multiplication is not enough to thwart invalid
public key attacks. In other words, public key validation MUST be
combined with the Montgomery ladder, unless the scalar multiplier
involved is public or a single-DH-use secret (i.e. computing kG and
kP, counts as a single DH use of k).

   Note: a given point need only be validated once, if the
   implementation can track validation state.

OPTIONAL: In some rare situations, it is also necessary to ensure
that the point has large order, not just that it is on the curve.

For points on this curve, each point has large order, unless it has
torsion by 12. In other words, if 12P != O, then the point P has
large order.

OPTIONAL: In even rarer situations, it may be necessary to ensure
that the point also has prime order. To be completed.

5. OPTIONAL encodings

The following two encodings are not usually required to obtain
interoperability in the typical ECC applications, but can sometimes
be useful.

5.1. Encoding scalar multipliers as 34 bytes

To be completed.

Basically, little-endian byte encoding of integers is recommended.

The main application is to signatures.

Another application is for test vectors (to be completed).

5.2. Encoding 34 bytes into a point (sketch)

In special applications, beyond mere Diffie--Hellman key exchange or
digital signatures, it may be desired to encode arbitrary bytes as
points.

Example reasons are anonymity, or hiding the presence of a key
exchange.

   Note: the point encoding described earlier does a different job.
   It encodes every point. The task here is to encode every byte
   string.

  This method is slower than the representations above, and yields
  biased elliptic curve points, but has the advantage that the
  byte-strings are unbiased.

  The idea is a minor variation of the Elligator 2 construction
  [Elligator]. Unfortunately, Elligator 2 itself fails for curves
  with j-invariant 1728, which includes 2y^2=x^3+x. In case of
  confusion, this map here can be called Elligator i, (see also [B1]).

  Fix a square root i of -1 in the field.

  Given any random field element r, compute

    x=i- 3i/(1-ir^2)

  If there is no y solving 2y^2=x^3+x for this x, then replace x by
  x+i and try to solve for y once again.

  If the first x fails, then the second x succeeds.

  So, now r determines a unique x. To determine y, solve it per the
  equation, getting two roots. Label the 2 roots y0 and y1 according
  to a deterministic rule. Then choose y0 if the first x works, else
  choose y2. This ensures that the map from r^2 to (x,y) is
  injective.

  Finally, to encode a byte string b, just let it represent a field
  element r. Note that -r will be require more than 34 bytes. So the
  map from b to (x,y) is now injective.

  This map is reversible.

  To be completed.

6. Cryptographic schemes

  To be completed, or even removed!

  List all possible cryptographic schemes in which this curve could be
  used is outside the scope of this short document. Only a few
  highlights are mentioned.

6.1. Diffie--Hellman key agreement

   To be completed.

   Question: should DH use cofactor multiplication? For now, let's say
   no.

   Non-cofactor multiplication risks leaking the private key mod 72, or
   at least mod 12, or perhaps even worse (if the field arithmetic has
   additional leaks).

   But cofactor multiplication reduces the private key size similarly.
   Also, if we start from a 34-byte private key scalar, then we achieve
   a similar effect to cofactor multiplication.

6.2. Signatures

   For signatures, such as ECDSA, the verifier must fully decompress
   the 34-byte representation. The verifier must do this twice, once
   with the signer's public key, and once with one component of the
   signature.

   To do this, the verifier can take, and make the most natural choice
   of the two possible y. The signer, anticipating the verifier, then
   must ensure that the signature will verify correctly under the
   verifier's choices for the y values. The signer incurs only a small
   extra cost for ensuring this.

   To be completed.

   Given that this curve is experimental and non-radically distinct
   from previous curves, signers and may opt to consider an
   experimental and non-radically distinct signature scheme with the
   curve $2y^2=x^3+x$.

   The RKHD ElGamal signature scheme [B2] is an example of such a
   signature scheme.

   In short, fix a base point G. The signing key is d, the verifying
   key is Q=dG. A pair (R,s), R is a point, and s is an integer, is a
   (valid) signature of message with integer hash h, if

        sG = rR + hQ

where r is obtained from R by re-interpreting its byte as an
integer.

To sign a message with hash h, the signer computes a
message-unique secret k, computes R=kG, computers r as above, and
computes

$$s = rk + hd \bmod n$$

where n is the order of G.

The signer may compute k as the hash of s and h, or through some
other method which ensures that k depends (pseudorandomly) on h.

The signer MUST choose k such that no linear relation between the k
for different h can be discovered by the adversary. The signer
SHOULD use some kind of pseudorandom function to achieve this.

  Note: this ElGamal signature variant corresponds to type 4 ElGamal
  signature in the Handbook of Applied Cryptography.

6.3 Menezes--Qu--Vanstone key agreement

To be completed.

7. IANA Considerations

This document requires no actions by IANA, yet.

8. Security considerations

No cryptographic algorithms is without risks. Consequently, risks
are comparative. This section will not fully list the risks of all
other forms of elliptic curve cryptography. Instead it will list
the most plausible risks of this curve, and only to a limited degree
contrast these to a few other standardized curves.

8.1. Field choice

The field $8^{91}+5$ has the following risks.

- $8^{91}+5$ is a special prime. As such, it is perhaps vulnerable to
  some kind of attack. For example, for some curve shapes, the
  supersingularity depends on the prime, and the curve size is
  related in a simple way to the field size, causing a potential
  correlation between the field size and the effectiveness of an
  attack, such as the Pohlig--Hellman attack.

Many other standard curves, such as the NIST P-256 and
Curve25519, also use special prime field sizes, so have a similar
risk. Yet other standard curves, such as the Brainpool, use
pseudorandom field sizes, so have less risk to this threat.

- 8^91+5, while implementable in five 64-bit words, has some risk of
  overflowing, or of not fully reducing properly. Perhaps a smaller
  field, such as that used in Curve25519, has a simpler reduction
  and overflow-avoidance properties.

- 8^91+5, by virtue of being well-above 256 bits in size, risks its
  user doing extra, and perhaps unnecessary, computation to protect
  their 128-bit keys, whereas smaller curves might be faster (as
  expected) yet still provide enough security. In other words, the
  extra cost is wasteful, and partially a form of denial of service.

- 8^91+5, is smaller than 8^95-9, yet uses no fewer symbols. Since
  larger field sizes lead to strong Pollard rho resistance, it can
  be argued that this field size does not optimize security against
  (specification) simplicity. (The main reason this document
  prefers 8^91+5 over 8^95-9 is its simpler field inversion.)
  Similarly, 8^91+5 is smaller than the six-symbol primes 9^99+4 and
  9^87+4, but these are not close to powers of two, which means that
  modular multiplication and reduction for them is not likely to be
  as efficient as for 8^91+5.

- 8^91+5, is smaller than 2^283 (used by sect283k1 in Zigbee), and
  many other five-symbol and four-symbol powers of primes (such as
  9^97). So, it less to provide less resistance to Pollard rho.
  Recent progress in the elliptic curve discrete logarithm problem,
  [HPST] and [Nagao], is the main reason to prefer prime fields
  instead of power of prime fields. A second reason to prefer prime
  field 8^91+5 (and other large characteristic fields) over small
  characteristic fields, is the generally better software speed of
  large characteristic fields: which arises because most software is
  implemented on a general purpose hardware processor that has fast
  multiplication circuits. (This speed advantage probably does not
  apply for hardware.)

See [B1] for further discussion.

8.2. Curve choice

A first risk of using 2y^2=x^3+x is the fact that it is a special
curve, with complex multiplication leading to an efficient
endomorphism. Many other standard curves, NIST P-256, Curve25519,
Brainpool, do not have any efficient endomorphisms. Yet some
standard curves do, NIST K-283 and secp256k1 (in BitCoin).
Furthermore, it is not implausible [KKM] that special curves,
including those efficient endomorphisms, may survive an attack on
random curves.

A second risk of 2y^2=x^3+x over 8^91+5 is the fact that it is not
twist-secure. What may happen is that an implementer may use the
Montgomery ladder in Diffie--Hellman and re-use private keys. They
may think, despite the (ample?) warnings in this document, that
public key validation in unnecessary, modeling their implementation
after Curve25519 or some other twist-secure curve. This implementer
is at risk of an invalid public key attack. Moreover, the
implementer has an incentive to skip public-key validation, for
better performance. Finally, even if the implementer uses
public-key validation, then the cost of public-key validation is
non-negligible.

A third risk is a biased ephemeral private key generation in a
digital signature scheme. Most standard curve lack this risk
because the field is close to a power of two, and the cofactor is a
power of two.

A fourth risk is a Cheon-type attack. Few standard curves address
this risk.

A fifth risk is a small-subgroup confinement attack, which can also
leak a few bits of the private key.

8.3. Encoding choices

To be completed.

8.4. General subversion concerns

Although the main motivation of curve 2y^2=x^3+x over 8^91+5 is to
minimize the risk of subversion via a backdoor, such as the one
described by [Teske], it is only fair to point out that its
appearance in this very document can be viewed with suspicion as an
possible effort at subversion (via a front-door). (See [BCCHLV] for
some further discussion.)

Any other standardized curve can be view with a similar suspicion
(except, perhaps, by the honest authors of those standards for whom
such suspicion seems absurd and unfair). A skeptic can then examine
both (a) the reputation of the (alleged) author of the standard,
making an ad hominem argument, and (b) the curve's intrinsic merits.

By the very definition of this document, the user is encouraged to
take an especially skeptical viewpoint of curve 2y^2=x^3+x over
8^91+5. So, it is expected that skeptical users of the curve will
either

- use the curve for its other merits (other than its backdoor
  mitigations), such as efficient endomorphism, field inversion,
  high Pollard rho resistance within five 64-bit words, meanwhile
  holding to the evidence-supported belief ECC that is now so mature
  that worries about subverted curves are just far-fetched nonsense,
  or

- as an additional of layer of security in addition to other
  algorithms (ECC or otherwise), as an extra cost to address the
  non-zero probability of other curves being subverted.

To paraphrase, consider users seriously worried about subverted
curves (or other cryptographic algorithms), either because they
estimate as high either the probability of subversion or the value
of the data needing protection. These users have good reason to
like 2y^2=x^3+x over 8^91+5 for its compact description.
Nevertheless, the best way to resist subversion of cryptographic
algorithms seems to be combine multiple dissimilar cryptographic
algorithms, in a strongest-link manner. Diversity hedges against
subversion, and should the first defense against it.

8.5. Concerns about 'aegis'

The exact curve 2y^2=x^3+x over 8^91+5 was (seemingly) first
described to the public in 2017 [AB]. So, it has a very low age.

Furthermore, it has not been submitted for a publication with peer
review to any cryptographic forum such as the IACR conferences like
Crypto and Eurocrypt. So, it has been review by very few eyes, most
of which had little incentive to study it seriously.

Under the metric of aegis, as in age * eyes, it scores low.
Counting myself (but not quantifying incentive) it gets an aegis
score of 0.1 (using a rating 0.1 of my eyes factor in the aegis
score: I have not discovered any major ECC attacks of my own.) This
is far smaller than some more well-studied curves.

However, in its defense, the curve 2y^2=x^3+x over 8^91+5 has
similarities to some of the better-studied curves with much higher
aegis:

- Curve25519: has field size 8^85-19, which a little similar to
  8^91+5; has equation of the form by^2=x^3+ax+x, with b and a
  small, which is similar to 2y^2=x^3+x. Curve25519 has been around
  for over 10 years, has (presumably) many eyes looking at it, and
  has been deployed thereby creating an incentive to study. An
  estimated aegis score is 10000.

- P-256: has a special field size, and maybe an estimated aegis
  score of 200000. (It is a high-incentive target. Also, it has
  received much criticism, showing some intent of cryptanalysis.
  Indeed, there has been incremental progress in finding minor
  weakness (implementation security flaws), suggestive of actual
  cryptanalytic effort.) The similarity to 2y^2=x^3+x over 8^91+5
  is very minor, so very little of the P-256 aegis would be relevant
  to this document.

- secp256k1: has a special field size, though not quite as special
  as 8^91+5, and has special field equation with an efficient
  endomorphism by a low-norm complex algebraic integer, quite
  similar to 2y^2=x^3+x. It is about 17 years old, and though not
  studied much in academic work, its deployment in Bitcoin has at
  least created an incentive to attack it. An estimated aegis score
  is 10000.

- Miller's curve: Miller's 1985 paper introducing ECC suggested,
  among other choices, a curve equation y^2=x^3-ax, where a is a
  quadratic non-residue. Curve 2y^2=x^3+x is isomorphic to
  y^2=x^3-x, which is essentially one of Miller's curves, except
  that a=1 is a quadratic residue. Miller's curve has not been
  studied directly, but probably much more so than this than the
  curve in this document. Miller also hinted that it was not
  prudent to use a special curve y^2=x^3-ax: such a comment may have
  encourage some cryptanalysts, but discouraged cryptographers,
  perhaps balancing out the effect on the eyes factor the aegis
  score. An estimate aegis score is 300.

Obvious cautions to the reader:

- Small changes in a cryptographic algorithm sometimes cause large
  differences in security. So security arguments based on
  similarity in cryptographic schemes should be given low priority.

- Security flaws have sometimes remained undiscovered for years,
  despite both incentives and peer reviews (and lack of hard
  evidence of conspiracy). So, the eyes-part of the aegis score is
  very subjective, and perhaps vulnerable false positives by a herd
  effect. Despite this caveat, it is not recommended to ignore the
  eyes factor in the aegis score: don't just flip through old books
  (of say, fiction), looking for cryptographic algorithms that might
  never have been studied.

9. References

9.1. Normative References

[BCP14] Bradner, S., "Key words for use in RFCs to Indicate
        Requirement Levels", BCP 14, RFC 2119, March 1997,
        <http://www.rfc-editor.org/info/bcp14>.

9.2. Informative References

To be completed.

[AB] A. Allen and D. Brown. ECC mod 8^91+5, presentation to CFRG,
     2017.
     <https://datatracker.ietf.org/doc/slides-99-cfrg-ecc-mod-8915/>

[B1] D. Brown. ECC mod 8^91+5, IACR ePrint, 2018.
      <http://ia.cr/2018/121>

[B2] D. Brown. RKHD ElGamal signing and 1-way sums, IACR ePrint,
     2018. <http://ia.cr/2018/186>

[KKM] A. Koblitz, N. Koblitz and A. Menezes. Elliptic Curve
      Cryptography: The Serpentine Course of a Paradigm Shift, IACR
      ePrint, 2008. <http://ia.cr/2008/390>

[BCCHLV] D. Bernstein, T. Chou, C. Chuengsatiansup, A. Hulsing,
         T. Lange, R. Niederhagen and C. van Vredendaal. How to
         manipulate curve standards: a white paper for the black
         hat, IACR ePrint, 2014. <http://ia.cr/2014/571>

[Elligator] To do: fill in this reference.

[NIST-P-256] To do: NIST recommended 15 elliptic curves for
cryptography, the most popular of which is P-256.

[Zigbee] To do: Zigbee allows the use of a small-characteristic
special curve, which was also recommended by NIST, called K-283, and
also known as sect283k1. These types of curves were introduced by
Koblitz. These types of curves were not recommended by NSA in Suite
B.

[Brainpool] To do: the Brainpool consortium (???) recommended some
elliptic curves in which both the field size and the curve equation
were derived pseudorandomly from a nothing-up-my-sleeve number.

[SEC2] Standards for Efficient Cryptography. SEC 2: Recommended
      Elliptic Curve Domain Parameters, version 2.0, 2010.
        <http://www.secg.org/sec2-v2.pdf>

[IT] T. Izu and T. Takagi. Exceptional procedure attack on elliptic
     curve cryptosystems, Public key cryptography -- PKC 2003,
     Lecture Notes in Computer Science, Springer, pp. 224--239,
     2003.

[PSM] To do: Projective coordinates leak. Pointcheval, Smart,
Malone-Lee?

[BitCoin] To do: BitCoin uses curve secp256k1, which has an
efficient endomorphism.

[Bleichenbacher] To do: Bleichenbacher showed how to attack DSA
using a bias in the per-message secrets.

[Gordon] To do: Gordon showed how to embed a trapdoor in DSA
parameters.

[HPST] Y. Huang, C. Petit, N. Shinohara and T. Takagi. On
      Generalized First Fall Degree Assumptions, IACR ePrint 2015.
        <http://ia.cr/2015/358>

[Nagao] K. Nagao. Equations System coming from Weil descent and
      subexponential attack for algebraic curve cryptosystem, IACR
      ePrint, 2015. <http://ia.cr/2013/549>

[Teske] E. Teske. An Elliptic Curve Trapdoor System, IACR ePrint,
      2003. <http://ia.cr/2003/058>

[YY] To do: Yung and Young, generalized Gordon's ideas [Gordon] into
Secretly-embedded trapdoor ... also known as a backdoor.

[Appendix A](#). **Test vectors**

   To be completed.

[Appendix B](#). **Motivation: minimizing the room for backdoors**

   To be completed.

   See [[AB](#)] and [[B1](#)] for some details.

   The field and curve are described with very few symbols, while
   retaining many basic security and speed features.

   A prime field was chosen due to recent asymptotic advances on
   discrete logarithms in low-characteristic fields [[HPST](#)] and
   [[Nagao](#)]. According to [[Teske](#)], some characteristic-two elliptic
   curves could be equipped with a secretly embedded backdoor.

      Note: this curve is isomorphic to the non-Montgomery curve
      y^2=x^3-x, which requires just 9 symbols in its description, 1
      fewer than required by 2y^2=x^3+x.

[Appendix C](#). **Pseudocode**

   This section uses a C-like pseudocode to describe some of the
   algorithms useful for implementing this curve.

   Real-world implementations adapting this pseudocode had better
   harden this pseudocode against real-world implementation issues.
   Better yet, real-world code could start from scratch, using the
   pseudocode only for comparison.

      Note: the pseudocode relies on some C idioms (hacks?), which might
      make the pseudocode unclear to those unfamiliar with these idioms.

      Note: this pseudocode was adapted from a few different
      experimental prototypes of the author, (which might not be
      consistent). The pseudocode has not yet received any independent
      review.

      Note: this pseudocode uses a terse non-conventional coding style,
      partly as an exercise in arbitrary source code compression (code
      golf), but also in the mathematics tradition of using many
      single-letter variable names, which enables seeing an entire
      formula in a single view and emphasizes the essential mathematical
      operations rather than the variable's purpose.

Note: the pseudocode does not use the C operator ^ for bitwise XOR
of integers, which (luckily) avoid possible confusion with the use
of ^ as exponentiation operator in the rest of this document.

C.1. Byte encoding

  Pseudocode for byte representation encoding process is

```
bite(c b,f x) {
 i j=34,k=5; f t;
 mal(t,-1,x);
 mal(x,cmp(t,x),x);
 fix(x);
 for(;j--;) b[j]=x[j/7]>>((8*j)%55);
 for(;--k;) b[7*k-1]+=x[k]<<(8-k);
 }
```

  The input variable is x and the output variable is b. The declared
  types and functions are as follows:

  - type c: curve representative, length-34 array of non-negative
    8-bit integers ("characters"),

  - type f: field element, a length-5 array of 64-bit integers
    (negatives allowed), representing a field element as an integer in
    base 2^{55},

  - type i: 64-bit integers (e.g. entries of f),

  - function mal: multiply a field element by a small integer (result
    stored in 1st argument),

  - function fix: fully reduce an integer modulo $8^{91}+5$,

  - function cmp: compare two field element (after fixing), returning
    -1, 0 or 1.

    Note: The two for-loops in the pseudocode are just radix
    conversion, from base 2^{55} to base 2^8. Because both bases are
    powers of two, this amount to moving bits around. The entries of
    array b are compute modulo 256. The second loop copies the bits
    that the first loop misses (the bottom bits of each entry of f).

    Note: Encoding is lossy, several different (x,y) may encode to the
    same byte string b. Usually, if (x,y) generated as a part of
    Diffie--Hellman key exchange, this lossiness has no effect.

   Note: Encoding should not be confused with encryption. Encoding
   is merely a conversion or representation process, whose inverse is
   called decoding.

[C.2](#). Byte decoding

  Pseudocode for decoding is:

```
feed(f x,c b) {
 i j=34;
 mal(x,0,x);
 for(;j--;) x[j/7]+=((i)b[j])<<((8*j)%55);
 fix(x);
 }
```

  with similar conventions as used in the pseudocode function bite
  (defined in the section on encoding), and some extra conventions:

  - the expression (i)b[j] means that 8-bit integer b[j] is converted
    to a 64-bit integer (so is no longer treated modulo 256). (In C,
    this is operation is called casting.)

    Note: the decode function 'feed' only has 1 for-loop, which is the
    approximate inverse of the first of the 2 for-loops in the encode
    function 'bite'. The reason the 'bite' needs the 2nd for-loop is
    due to the lossy conversion from integers to bytes, whereas in the
    other direction the conversion is not lossy. The second loop
    recovers the lost information.

[C.3](#). Fermat inversion

  Projective coordinates help avoid costly inversion steps during
  scalar multiplication.

  Projective coordinates are not suitable as the final representation
  of an elliptic curve point, for two reasons.

  - Projective coordinates for a point are generally not unique: each
    point can be represented in projective coordinates in multiple
    different ways. So, projective coordinates are unsuitable for
    finalizing a shared secret, because the two parties computing the
    shared secret point may end up with different projective
    coordinates.

   - Projective coordinates have been shown to leak information about
     the scalar multiplier [PSM], which could be the private
     key. It would be unacceptable for a public key to leak
     information about the private key. In digital signatures, even a
     few leaked bits can be fatal, over a few signatures
     [Bleichenbacher].

  Therefore, the final computation of an elliptic curve point, after
  scalar multiplication, should translate the point to a unique
  representation, such as the affine coordinates described in this
  report.

  For example, when using a Montgomery ladder, scalar multiplication
  yields a representation (X:Z) of the point in projective
  coordinates. Its x-coordinate is then x=X/Z, which can be computed
  by computing the 1/Z and then multiplying by X.

  The safest, most prudent way to compute 1/Z is to use a side-channel
  resistant method, in particular at least, a constant-time method.
  This reduces the risk of leaking information about Z, which might in
  turn leak information about X or the scalar multiplier. Fermat
  inversion, computation of Z^(p-2) mod p, is one method to compute
  the inverse in constant time (if the inverse exists).

  Pseudocode for Fermat inversion is:

```
i inv(f y,f x) {
  i j=272;f z;
  squ(z,x);
  mul(y,x,z);
  for(;j--;) squ(z,z);
  mul(y,z,y);
  return !!cmp(y,(f){});
}
```

  Other inversion techniques, such as the binary extended GCD, may be
  faster, but generally run in variable-time.

  When field elements are sometimes secret keys, using a variable-time
  algorithm risk leaking these secrets, and defeating security.

C.4. Branchless Legendre symbol computation

  Pseudocode for branchlessly computing if a field element x has a
  square root:

```
i has_root(f x) {
  i j=270;f y,z;
  squ(y,x);squ(z,y);
  for(;j--;)squ(z,z);
  mul(y,y,z);
  return 0==cmp(y,(f){1});
}
```


  Note: Legendre symbol is usually most appropriately applied to
  public keys, which mostly obviates the need for side-channel
  resistance. In this case, the implementer can use quadratic
  reciprocity for greater speed.

C.5. Field multiplication and squaring

  To be completed.

  Note (on security): Field multiplication can be achieved most
  quickly by using hardware integer multiplication circuits. It is
  critical that those circuits have no bugs or backdoors.
  Furthermore, those circuits typically can only multiple integers
  smaller than the field elements. Larger inputs to the circuits
  will cause overflows. It is critical to avoid these overflows,
  not just to avoid interoperability failures, but also to avoid
  attacks where the attackers supplies inputs likely induce
  overflows [bug attacks], [IT]. The following pseudocode
  should therefore be considered only for illustrative purposes.
  The implementer is responsible for ensuring that inputs cannot
  cause overflows or bugs.

The pseudocode below for multiplying and squaring: uses unrolled
loops for efficiency, uses refactoring for source code compression,
relies on a compiler optimizer to detect common sub-expressions (in
squaring).

```
#define TRI(m,_)\
  zz[0]=m(0,0)_(1,4)_(2,3)_(3,2)_(4,1);\
  zz[1]=m(0,1)_(1,0)_(2,4)_(3,3)_(4,2);\
  zz[2]=m(0,2)_(1,1)_(2,0)_(3,4)_(4,3);\
  zz[3]=m(0,3)_(1,2)_(2,1)_(3,0)_(4,4);\
  zz[4]=m(0,4)_(1,3)_(2,2)_(3,1)_(4,0);
#define CYC(M) ff zz; TRI(+M,-20*M); mod(z,zz);
#define MUL(j,k) x[j]*(ii)y[k]
#define SQR(j,k) x[j]*(ii)x[k]
#define SQU(j,k) SQR(j>k?j:k,j<k?j:k)
mul(f z,f x,f y) {CYC(MUL);}
squ(f a,f x) {CYC{SQU};}
```

This pseudocode makes uses of some extra C-like pseudocode features:

- #define is used to create macros, which expand within the source
  code (as in C pre-processing).

- type ii is 128-bit integer

- multiplying a type i by a type ii variable yields a type ii
  variable. If both inputs can fit into a type i variable, then
  the result has no overflow or reduction: it is exact as a product
  of integers.

- type ff is array of five type ii values. It is used to represent
  a field in a radix expansion, except the limbs (digits) can be
  128-bits instead of 64-bits. The variable zz has type ff and is
  used to intermediately store the product of two field element
  variables x and y (of type f).

- function mod takes an ff variable and produce f variable
  representing the same field element. A pseudocode example may be
  defined further below.

TO DO: Add some notes (answer these questions):

- How small the limbs of the inputs to function mul and squ must be
  to ensure no overflow occurs?

- How small are the limbs of the output of functions mul and squ?

C.6. Field element partial reduction

   To be completed.

   The function mod used by pseudocode function mul and squ above is
   defined below.

```
#define QUO(x)(x>>55)
#define MOD(x)(x&((((i)1)<<5)-1))
#define Q(j) QUO(QUO(zz[j]))
#define P(j) MOD(QUO(zz[j]))
#define R(j)     MOD(zz[j])
mod(f z,ff zz){
 z[0]=R(0)-P(4)*20-Q(3)*20;
 z[1]=R(1)-P(0)-Q(4)*20;
 z[2]=R(2)-P(1)-Q(0);
 z[3]=R(3)-P(2)-Q(1);
 z[4]=R(4)-P(3)-Q(2);
 z[1]+=QUO(z[0]);
 z[0]=MOD(z[0]);
 }
```

   TO DO: add notes answering these questions:

   - How small must be the input limbs to avoid overflow?

   - How small are the output limbs (to know how to safely use of
     output in further calculations).

C.7. Field element final reduction

   To be completed.

   The partial reduction technique is sometimes known as lazy
   reduction. It is an optimization technique. It aims to do only
   enough calculation to avoid overflow errors.

   For interoperability, field elements need to be fully reduced,
   because partial reduction means the elements still have multiple
   different representations.

Pseudocode that aims for final reduction is the following:

```
#define FIX(j,r,k) {q=x[j]>>r;\
 x[j]-=q<<r; x[(j+1)%5]+=q*k;}
fix(f x) {
 i j,q,t=2;
 for(;t--;) for(j=0;j<5;j++) FIX(j,(j<4?55:53),(j<4?1:-5));
 q=x[0]<0;
 x[0]+=q*5; x[4]+=q>>53;
 }
```

C.8. Scalar point multiplication

Work in progress.

A recommended method of scalar point multiplication is the Montgomery ladder. However, the curve 2y^2=x^3+x has an efficient endomorphism. So, this can be used to speed-up scalar point multiplication, as suggested by Gallant, Lambert and Vanstone.

Combining both GLV and Montgomery is also possible, such as suggested as by Bernstein.

  Note: The following pseudocode is not entirely consistent with previous pseudocode examples.

  Note and Warning: The following pseudocode uses secret indices to access (small) arrays. This has a risk of cache-timing attacks.

```
  typedef f p[2];
  typedef struct rung {i x0; i x1; i y; i z;} k[137];
  monty_2d (f ps,k sk,f px) {
    i j,h; f z; p w[3],x[3],y[2]={{{},{1}}},z[2];
    fix(px);mal(y[0][0],1,px);
    endomorphism_1_plus_i(z[0],px);
    endo_i(y[1],y[0]); endo_i(z[1],z[0]);
    copy(x[1],y[0]); copy(x[2],z[0]);
    double_xz(x[0],y[0]);
    for(j=0;j<137;j+=){
     double_xz(w[0],     x[sk[j].x0 /* cache attack here? */ ]);
     diff_add (w[1],x[1],x[sk[j].x1],y[sk[j].y]);
     diff_add (2[2],x[2],x[0],       z[sk[j].z]);
     for(h=0;h<3;h++) {copy(x[h],w[h]);}
    }
    inv(ps,x[1][1]);
    mul(ps,x[1][0],ps);
    fix(ps);
  }
```

   Note: The pseudocode uses some other functions not defined here,
   but whose meaning can be inferred by ECC experts.

   Note: The pseudocode uses a specialized format for the scalar.
   Normal scalars would have to be re-coded into this format, and
   re-coding has non-negligible run-time. Perhaps in
   Diffie--Hellman, re-coding is not necessary if one can ensure that
   uniformly selection of coded scalars is not a security risk.

  TO DO:
  - Define the functions used by monty_2d.
  - Prove that these function avoid overflow.
  - Define functions to re-code scalars for monty_2d.

C.9. Diffie--Hellman pseudocode

  To be completed.

  This pseudocode would show how to use to scalar multiplication,
  combined with point validation, and so on.

C.10. Elligator i

  To be completed.

This pseudocode would show how to implement to the Elligator i map
from byte strings to points.

Pseudocode (to be verified):

```
typedef f xy[2] ;
#define X p[0]
#define Y p[1]
lift(xy p, f r) {
  f t ; i b ;
  fix(r);
  squ(t,r);         // r^2
  mul(t,I,t);       // ir^2
  sub(t,(f){1},t); // 1-ir^2
  inv(t,t);         // 1/(1-ir^2)
  mal(t,3,t);       // 3/(1-ir^2)
  mul(t,I,t);       // 3i/(1-ir^2)
  sub(X,I,t);       // i-3i/(1-ir^2)
  b = get_y(t,X);
  mal(t,1-b,I);     // (1-b)i
  add(X,X,t);       // EITHER  x  OR  x + i
  get_y(Y,X);
  mal(Y,2*b-1,Y);   // (-1)^(1-b)""
  fix(X);  fix(Y);
}

drop(f r, xy p)
{
  f t ; i b,h ;
  fix(X); fix(Y);
  get_y(t,X);
  b=eq(t,Y);
  mal(t,1-b,I);
  sub(t,X,t);    // EITHER x or x-i
  sub(t,I,t);    // i-x
  inv(t,t);      // 1/(i-x)
  mal(t,3,t);    // 3/(i-x)
  add(t,I,t);    // i+ 3/(i-x)
  mal(t,-1,t);   // -i-3/(i-x)) = (1-3i/(i-x))/i
  b = root(r,t) ;
  fix(r);
  h = (r[4]<(1LL<<52)) ;
  mal(r,2*h-1,r);
  fix(r);
}
```

```
   elligator(xy p,c b) {f r; feed(r,b); lift(p,r);}

   crocodile(c b,xy p) {f r; drop(r,p); bite(b,r);}
```

Acknowledgments

Author's Address

   Dan Brown
   4701 Tahoe Blvd.
   BlackBerry, 5th Floor
   Mississauga, ON
   Canada
   danibrown@blackberry.com