Elliptic curve 2y^2=x^3+x over field size 8^91+5
<draft-brown-ec-2y2-x3-x-mod-8-to-91-plus-5-08.txt>

Abstract

  Multi-curve elliptic curve cryptography with curve
  2y^2=x^3+x/GF(8^91+5) hedges a risk of new curve-specific attacks.
  This curve features: isomorphism to Miller's curve from 1985; low
  Kolmogorov complexity (little room for embedded weaknesses of
  Gordon, Young--Yung, or Teske); similarity to a Bitcoin curve;
  Montgomery form; complex multiplication by i
  (Gallant--Lambert--Vanstone); prime field; easy reduction,
  inversion, Legendre symbol, and square root; five 64-bit-word field
  arithmetic; string-as-point encoding; and 34-byte keys.

Status of This Memo

  This Internet-Draft is submitted in full conformance with the
  provisions of BCP 78 and BCP 79. Internet-Drafts are working
  documents of the Internet Engineering Task Force (IETF). Note that
  other groups may also distribute working documents as
  Internet-Drafts. The list of current Internet-Drafts is at
  http://datatracker.ietf.org/drafts/current.

  Internet-Drafts are draft documents valid for a maximum of six
  months and may be updated, replaced, or obsoleted by other documents
  at any time. It is inappropriate to use Internet-Drafts as
  reference material or to cite them other than as "work in progress."

Copyright Notice

Contents

[1](). Introduction

Elliptic curve cryptography (ECC) is now part of several IETF
protocols.

Multi-curve ECC can mitigate the risk of new curve-specific attacks
on ECC.

  Note: The risk of new curve-specific attacks is arguably smaller
  than the risk of quantum attacks breaking all current curves in
  ECC. The first risk can be addressed by combining ECC with
  post-quantum cryptography (PQC). Multi-curve ECC would then be
  added to an ECC+PQC combination. The extra cost of multi-curve
  over single-curve ECC would less noticeable in an ECC+PQC
  combination than it would be in an ECC-only implementation.

This document aims to contribute to multi-curve ECC by describing
how to use the curve

  2y^2=x^3+x / GF(8^91+5)

for elliptic curve Diffie--Hellman (ECDH).

[Appendix A]() expands on why and when 2y^2=x^3+x/GF(8^91+5) is useful
in multi-curve ECC.

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. Use ONLY in multi-curve ECC

  An implementation using curve $2y^2=x^3+x/GF(8^{91}+5)$ in elliptic
  curve cryptography SHOULD use it multi-curve ECC in a combination
  with more established, less special, curves, such as Curve25519 or
  NIST P-256, or even Brainpool curves.

  The curve $2y^2=x^3+x/GF(8^{91}+5)$ belongs to a very special class.
  Miller in 1985 suggested to exercise prudence around similarly
  special curves in ECC. This document continues to heed Miller's
  advice, by considering $2y^2=x^3+x/GF(8^{91}+5)$ riskier than less
  special curves, and hence advising to use it only a second layer of
  defense in ECC.

4. Representations

  Interoperable communication on most systems involves byte
  strings. Elliptic curve cryptography needs to use byte strings to
  work on these systems. Different curves use byte strings in
  different ways, due to differences in the curves such as the number
  of points on the curve.

  This section specifies a default compressed representation of points
  of $2y^2=x^3+x/GF(8^{91}+5)$ as 34-byte strings. The compressed
  representation SHOULD be used for communication whenever byte string
  representations are needed. The compressed representation MUST be
  used when computing an ECDH shared secret.

  An OPTIONAL uncompressed representation as 70-byte strings is also
  specified. The uncompressed representation MAY only be used for
  systems that optimize speed, and are willing to send extra an 35
  bytes. The uncompressed representation MUST NOT be used to
  represent an ECDH shared secret.

  An OPTIONAL representation of scalar multipliers, such as secret
  keys, as 34-byte strings is also provided. This representation is
  used for secrets, and is not for public communication. Different
  components of a single ECC system MAY use this representation to
  work together.

An OPTIONAL representation of 34-byte strings as points is also
described. This MAY be used in niche applications, such as in
deriving a point from the hash of a password, or in steganography,
to hide the presence of ECC, by choosing special points such that
represented in a way indistinguishable from uniformly random byte
strings. This uses the Elligator i method, a variant of the
Elligator 2 method.

4.1 Representation points in 34 bytes (compression)

Elliptic curve cryptography uses points on a curve for its public
keys and for it raw shared secret Diffie--Hellman keys.

Abstractly, points are mathematical objects. For curve $2y^2=x^3+x$,
a point is either a pair (x,y), where x and y are elements of
mathematical field, or a special point O (whose x and y coordinates
may be deemed as infinity).

  Note: The special point O should never be used as a key, in
  practice. In theory, point O is needed for the points to form a
  mathematical group.

For curve $2y^2=x^3+x/GF(8^{91}+5)$, the coordinates x and y of the
point (x,y) are integers modulo $8^{91}+5$, which can be represented as
integers in the interval $[0,8^{91}+4]$.

  Note: An implementation will often internally represent the
  x-coordinate as a ratio [X:Z] of field elements. Each field
  element has multiple such representations. The ratio [x:1] can
  viewed as the normalized representation of x. (Infinity can be
  then represented by [1:0].)

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 point (x,y).

  Note: In elliptic Diffie--Hellman (ECDH), the 34-byte encoding
  works well, despite being neither injective nor surjective.

[4.1.1](#). **Point encoding process**

  This section takes an abstract point (x,y), and outputs an encoding
  as a 34-byte string.

[4.1.1.1](#). **Summary**

  A point (x,y) is encoded by the little-endian byte representation of
  either x or -x, whichever fits into 34 bytes.

[4.1.1.2](#). **Details**

  A point (x,y) is encoded into 34 bytes, as follows.

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

    0 <= x < 8^91+5.

    Note: internal implementations might allow, for efficiency
    reasons, integer representations of x that are negative, or that
    are p or larger. Full reduction mod p is needed for interoperable
    encoding.

  Second, further reduce x by a flipping its sign, as explained next.
  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) = sum (0<=i<=33, b[i]*256^i).

  Pseudocode can be found in [Appendix C](#).

    Note: The loss of information that happens upon replacing x by -x
    corresponds to applying complex multiplication by i on the curve,
    because i(x,y) = (-x,iy) is also a point on the curve. (To see
    this: note 2(iy)^2 = -(2y^2) = -(x^3+x) = (-x)^3+(-x).) In many
    applications, particularly Diffie--Hellman key agreement, this
    loss of information is carried through to the final shared secret,
    which means that Alice and Bob can agree on the same secret 34
    bytes.

  In ECC systems where the original x-coordinate and the decoded
  x-coordinate need to match exactly, the 34-byte encoding is probably
  not usable unless the following pre-encoding procedure is practical:

Given a point x, where x is larger than min(x,p-x), first replace
x by x'=p-x, on the encoder's side, using the new value x'
(instead of x) for any further step in the algorithm. In other
words, replace the point (x,y) by the point (x',y')=(-x,iy). Most
algorithms will also require a discrete logarithm d of (x,y),
meaning (x,y) = [d] G for some point G. Since (x',y') = [i](x,y),
we can replace by d' such that [d']=[i][d]. Usually, [i] can be
represented by an integer, say j, and we can compute d' = jd (mod
ord(G)).

4.1.2. Point decoding process

4.1.2.1. Summary

To decode a 34-byte string, interpret the bytes as little-endian
represented integer, which becomes the x-coordinate.

When needed, public-key validation is done, to ensure that x is
valid for a point on the curve.

If needed, the y-coordinate is recovered from x.

4.1.2.2. Detail

If byte i is b[i], with an integer value between 0 and 255
inclusive, then

 x = sum(0<=i<=33, b[i]*256^i)

  Note: a value of -x (mod p) will also be suitable, and results in
  a point (-x,y') which might be different from the originally
  encoded point. However, it will be one of the points [i](x,y) or
  -[i](x,y) where [i] means complex multiplication by [i].

In many cases, such as Diffie--Hellman key agreement using the
Montgomery ladder, neither the original value of coordinate x (among
x and -x) nor coordinate y of the point is needed. In these cases,
the decoding steps can be considered completed.

```
     +------------------------------------------------------+
     |                                                      |
     |      \  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.  Some applications would suffer from a  |
     |   severe  attack if they allow use of (x,y) not on   |
     |   the curve.  Such vulnerable applications MUST      |
     |   validate that the decoded point (x,y) is on the    |
     |   curve.  Validation is described in Section 5.      |
     |                                                      |
     +------------------------------------------------------+
```

In cases where a value for at least one of y, -y, iy, or -iy is
needed (such as in Diffie--Hellman key agreement using Edwards
coordinates), a candidate value for y can be obtained by computing a
square root:

  $y = ((x^3+x)/2)^{(1/2)}$.

  Note: Recovery of y can be combined with validation of x.

In some applications (but not Diffie--Hellman), it is important for
the decoded value of x to match the original value of x exactly. In
that case, the encoder should use the procedure that replaces x by
p-x, and adjusts the discrete logarithm appropriately. These steps
can be done by the encoder, with the decoder doing nothing.

4.2. OPTIONAL: uncompressed point representation

A speed-optimized version of elliptic curve Diffie--Hellman over
curve 2y^2=x^3+x/GF(8^91+5) might be fastest if the peer's public
key is supplied as an uncompressed point (x,y), because of methods
like twisted Edwards coordinates.

In an arbitrary point, each coordinate uses at most 274 bits, in the
standard bit representation. These 274 bits can be fit into 35
bytes. Little-endian ordering of bytes is used.

Both coordinates together fit into a 70-byte string, with the 35
bytes of x appearing first in the string.

Each point (except the identity point O, point-at-infinity) has
exactly one 70-byte uncompressed representation. Each 70-byte
string represents at most one point.

Converting back and forth between uncompressed and compressed
representations is straightforward in principle: just decode to a
point, and then re-encode with the other format. Each compressed
byte string typically represents four different point ((x,y),
(-x,iy), (x,-y), (-x,iy)), which have four different uncompressed
representations.

Given a compressed representation, there exists a point whose
uncompressed version shares the first 34 bytes with the given
compressed representation, because the decoding of a compressed
representation gives an x < 2^272, meaning x can be encoded in 34
bytes. The next byte (which is the 35th byte, which is also byte 34
in the 0-indexed array) of the uncompressed representation is set to
zero. The next 35 bytes are computed from a y recovered from x.
The most expensive step of computing is the square root operation,
since y = ((x^3+x)/2)^(1/2).

Given an uncompressed representation (b), if the 35th byte (b[34])
is zero, then the compressed representation is just the first 34
bytes of the uncompressed representation. Otherwise, use the fist
35 bytes to compute x, then put p-x mod p into little-endian to form
the compressed representation.

Point validation for uncompressed 70-byte string is more efficient
than for compressed points. To check that the point is on the
curve, just check that the curve equation 2y^2=x^3+x holds, which is
more efficient than computing a Legendre symbol (the most expensive
step of the checking that the point is on the curve given only x).

[4.3](). OPTIONAL: Representation of scalar multipliers

Scalars (integer point multipliers) sometimes need to be encoded as
byte strings. Typical examples are the following applications.

 - Digital signature in ECC generally require scalar encodings.
   This draft does not specify signature algorithms in detail, only
   providing some general suggestions.

  - An ECC implementation needs to store scalars, over at least a
    short time period, because often scalars are used at least twice,
    and must be stored between these two uses. For example, in
    elliptic curve Diffie--Hellman (ECDH), Alice has scalar a, and
    she sends Bob point aG. Alice keeps scalar a until she receives
    point B back from Bob. Alice then applies a to B, computing aB.
    If a is ephemeral, she then deletes a. While waiting for B,
    Alice must store a. An implementation is free to use any
    encoding to store scalars. Implementation are often constructed
    in modular pieces. Any pieces handling the same scalar need to
    store the scalar in a consistent, interoperable way.

 Elliptic curve Diffie--Hellman with curve 2y^2=x^3+x/GF(8^91+5)
 would typically use a base point G of large prime order q. The size
 of q is approximately p/72. In this case, the value of a scalar s
 usually only matters mod q. So, an implementation can reduce s, by
 replacing s by s mod q. This ensures that s<q. Since q < 2^267 <
 256^34, the scalar s can be represented in 34 bytes.

 Alternatively, an implementation might instead generate and store
 scalar s as a uniformly random 34-byte integer, so that 0 <= s <
 256^34. Because of the approximation 256^34 ~ (36*q) (1 - 2^(-130)
 ), scalar s chosen and represented this way has negligible bias when
 reduced mod q. Furthermore, elliptic curve Diffie--Hellman likely
 tolerates much higher than a digital signature would.

 Little-endian byte encoding of scalars is probably simplest, due to
 the consistency with the little-endian byte encoding of field
 elements.

4.4. OPTIONAL: Representing strings as points using Elligator i

 In niche applications, it may be useful to encode an arbitrary
 string as a point on a curve.

 Example reasons to encode arbitrary 34-byte strings include:

   - Encoding passwords (or their hashes) as points in a
     password-authenticated key exchange (PAKE).

   - Hiding the fact that ECC is being used, by making sure that ECC
     public keys have the same probability distribution as uniformly
     random 34-byte strings.

 To this end, this section sketches a method to reversibly encode
 an arbitrary 34-byte string as a point.

Note: To encode variable-length strings as points, one can first
compute a 34-byte hash of the variable-length string, and then
encode the hash. Such encoding of variable-length strings cannot
be reversible.

Note: The point decoding scheme of Section 4.2 does not suffice to
encode strings, because only about half of all 34-byte strings are
decodable.

Note: The string-as-point encoding has the the property that only
about half of all points are decodable as 34-bytes strings.
Encoding a uniformly distributed 34-byte string as a point yields
non-uniformly distributed points. In other words, the byte string
is arbitrary, but the resulting points are not arbitrary.

The encoding is called Elligator i.

Note: The Elligator i encoding is a minor variation of the
Elligator 2 construction [Elligator], introduced in [B1]. A minor
variation is necessary, because Elligator 2 fails for curves with
j-invariant 1728. Curve 2y^2=x^3+x has j-invariant 1728. The fix
is just a minor tweak to Elligator 2, hence the bulk of the name
is retained.

Fix a square root i of -1 in the field in GF(8^91+5). For example,
2^(8^89+1) mod 8^91+5.

To encode a 34-byte string b,

  1. Let b represent a field element r, using little-endian base
     256.

  2. Compute x = i-3i/(1-ir^2). Let j=1.

  3. If 2y^2=x^3+x has no solution y, then replace x by x+i and j by
     j+1.

  4. Find two solutions y[1] and y[2] to 2y^2=x^3+x, such that
     y[1]<y[2].

  5. Compute y=y[j].

Now (x,y) is a point on the curve 2y^2=x^3+x.

The Elligator i encoding is reversible, because it has the decoding
sketched below.

The input to decoding is a point (x,y). The point (x,y) has encoded
a byte string b. The job of decoding is to recover b from (x,y).
For a point (x,y) that does not encode any b, the decoding reports
an error.

If y>p-y, replace x by x-i. Compute s = -i - 3/(i-x). Let r =
sqrt(s). (If s has no square root, then report an error.) If r >
p-r, replace r by p-r. Write r in little-endian base 256 to get a
34-byte string b.

> Note: Just to illustrate a contrast between Elligator i encoding
> and the normal point encoding, consider the useless example of
> applying both encodings. Start with 34-byte string b. Apply
> Elligator i encoding to get a point (x,y). Apply the point
> encoding to (x,y) to get a 34-byte string b'. In summary,
> b'=encode(encode(b)). The byte string b' has no significant
> relation to b. The map b->b' from 34-byte strings to themselves
> is lossy (non-injective) with ratio approximately 4:1, and the
> image set is about one quarter of all 34-byte strings.

[5]. Point validation

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

To avoid leaking information about a user private key, the user can
validate that the peer's public key is a point on the curve, or even
that it is point in the correct subgroup.

[5.1]. Schedules for point validation

This section specifies three schedules (mandatory, simplified, and
minimal) for deciding when to validate whether a given point (x,y)
is on the curve $2y^2=x^3+x/GF(8^{91}+5)$.

[5.1.1]. Mandatory schedule for point validation

As a precaution, an implementation MAY opt to apply a mandatory
schedule for point validation, meaning every public key (and point)
is validated, before subsequent use.

[5.1.2]. Simplified schedule for point validation

A small, general-purpose, implementation aiming for high speed might
not be able to afford the cost of the mandatory schedule for
validation from [Section 5.1.1], because each validation of a 34-byte
encoded point costs about 10% of a scalar multiplication.

As a practical middle ground, an implementation MAY opt to apply
simplified validation, which is the rule is that a not-yet-trusted
peer's public key is validated before being scalar multiplied by a
static secret key.

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

Note: The simplified schedule for validation implies that, when
the secret key is ephemeral (for example, used in one
Diffie--Hellman transaction), the peer's public key need not be
validated.

Note: The simplified schedule validation implies that when the
point being scalar multiplied is a known valid fixed point, or a
previously validated public key (including a public key from a
certificate in which a trusted certification authority has a
policy to validate public keys), then validation is not needed.

5.1.3. Minimal schedule for point validation

An implementation MAY opt to use a minimal schedule for point
validation, meaning doing as little point validation as possible,
just enough to resist known attack against the implementation,
reducing the risk to a negligible level.

The curve 2y^2=x^3+x is not twist-secure: using the Montgomery
ladder for scalar multiplication is not enough to thwart invalid
public key attacks.

However, consider a static hashed-ECDH implementation implemented
with a Montgomery ladder, such that the static secret key is used in
at most ten million times hashed-ECDH transactions. Even if exposed
to invalid points on the twist, the security risk is nearly
negligible. Therefore, the minimal validation would not validate
the peer's public keys.

5.2. Point validation process

  Upon decoding a 34-byte string into x, the next step is to compute
  z=2(x^3+x). Then one checks if z has a nonzero square root (in the
  field of size 8^91+5). If z has a nonzero square root, then the x
  represents a valid point, otherwise x is invalid.

  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 integral moduli smaller than 8^91+5.

  Instead, one can compute the Legendre symbol using powering in the
  field: (z/p) = z^((p-1)/2) = z^(2^272+2). This is much slower than
  using quadratic reciprocity.

  More generally, in signature applications (such as [B2]), where the
  y-coordinate is also needed, the computation of y, which involves
  computing a square root will generally include an implicit check
  that x is valid.

  OPTIONAL: In some 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 [12]P != O, then the point P has
  large order. Most applications of elliptic curve Diffie--Hellman do
  not require this check.

  OPTIONAL: In some situations, it may be necessary to ensure that a
  point P also has a prime order q = ord(G). One method to check this
  is to compute that [q]P, checking that [q]P = O. An alternative
  method is to try to solve for R in the equation [12]R=P, which
  involves methods such as division polynomials. The alternative has
  potential advantage of being faster than computing [q]P, but the
  clear disadvantage of requiring extra algorithms (beyond the scalar
  multiplication already necessary for Diffie--Hellman). Most
  applications of elliptic curve Diffie--Hellman do not need this
  check.

6. IANA considerations

  This document requires no actions by IANA, yet.

[7](#). **Security considerations**

No cryptographic algorithm is without risk.

Potential security risks of 2y^2=x^3+x/GF(8^91+5) are listed in this
section. The listed risks are worth considering before deciding to
use 2y^2=x^3+x/GF(8^91+5). Independent risk analysis beyond that
listed here is also important to consider.

This section aims to thoroughly describe the risks, perhaps erring
on the side over-stating many of the risks.

Absolute risk is difficult to quantify, especially against unknown,
potential attacks. Relative risk is slightly easier to qualify, if
a comparable cryptographic system is available as a benchmark.

Most of the security risks of 2y^2=x^3+x/GF(8^91+5) listed in the
section are compared to the risks of a typical generic (nearly
prime-sized) curve in ECC, or to the risks of specific
well-established curves in ECC (such as NIST P-256 and Curve25519).

  Note: Because 2y^2=x^3+x/GF(8^91+5) MUST be used only in
  multi-curve ECC, comparison to other curves is mainly for
  selection from a pool of curves of a set of curves in a
  multi-curve ECC implementation.

  Note: For possible security benefits of 2y^2=x^3+x/GF(8^91+5), see
  [Appendix A](#). This section and [Appendix A](#) are not entirely
  independent, since they discuss two sides of the same coin.

[7.1](#). **Field choice**

The field size 8^91+5 has the following risks.

See [[B1](#)] for further discussion about the relative merits of 8^91+5.

[7.1.1](#). **A special prime**

8^91+5 is a special prime that might lead a mathematical attack
speeding up the elliptic curve discrete logarithm problem (ECDLP).
Known correlations between ECC field size and ECDLP attacks are
evidence of this risk.

For some curve equations, the supersingularity, and thus security
due to vulnerability to MOV attack, depends on the prime of the
field size. For some curve equations, 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.

The curve 2y^2=x^3+x/GF(8^91+5) already resists the well-known
field-size-dependent ECDLP attacks, but there exists a risk is that
yet more unpublished field-size-dependent ECDLP attacks might exist.

Some other standard curves similarly use special field sizes
(different from 8^91+5 but still not pseudorandom). The standard
curves NIST P-256 and Curve25519 use special prime field sizes,
which have faster modular reduction (than random primes of similar
size). These other standard curves have a risk of suffering from
unpublished field-size-dependent ECDLP attacks similar to the risk
that 2y^2=x^3+x/GF(8^91+5) has.

By contrast, yet other standard curves, such as the Brainpool
curves, somewhat mitigate this risk of unpublished
field-size-dependent ECDLP attacks by using field sizes derived by
applying a pseudorandom function to a nothing-up-my-sleeve number.
The pseudorandom function helps reduce the risk of
field-size-dependent attacks which affect only a rare subset of
field sizes. (The risk of unpublished field-size-dependent attacks
that works against most field sizes is not mitigatable.)

Arguably, all the special primes and the Brainpool primes have some
qualities of being nothing-up-my-sleeve numbers, which helps
mitigate the risk against intentional secret (subversive)
field-size-dependent attacks. In this regard, 8^91+5 aims to nearly
optimize this mitigation.

7.1.2. Risk of 64-bit integer overflow

Modular arithmetic can be difficult to implement correctly,
because of the problem of integer overflow.

For example, adding two 64-bit numbers could result in a 65-bit
number, which cannot be stored in 64 bits, resulting in an integer
overflow. A standard solution is capture the extra bit as a carry
bit. Some computer languages, such as C, allow 64-bit integers, but
do not give access to the carry bit even if present in the computer
hardware.

Another common solution is to avoiding integer overflow is to ensure
that all additions (or subtractions) will not cause an integer
overflow. For example, ensuring that the two numbers added have 63
bits or fewer. Arranging this to always happen takes care, but can
result in fast arithmetic. One method is to partially reduce the
bit sizes of words after a modular multiplication.

Implementing arithmetic modulo 8^91+5 by using with five 64-bit
words has some risk of integer overflow, if not done carefully. In
this form, each 64-bit integer stores a value of at least 273/5 ~ 55
bits, well below the 63 bits needed to avoid overflow. But during
the course of complicated operations of elliptic curve, the 64-bit
integer words can take large values.

By contrast, a smaller field such as 2^255-19, as used in
Curve25519, can be implemented with five 64-bit integers, each
storing values of at least 51 bits, considerably smaller than 55
bits. This may be more tolerant to the accumulation of word sizes
during complication of elliptic curve arithmetic. The partial
reduction needed may be less, and thus may be faster.

7.1.3. Excessive size for 128-bit security

The prime 8^91+5 is well-above 256 bits in size.

A user aiming to resist 2^128 step attacks, could use a smaller, and
presumably faster, curve, and still have Pollard rho attacks taking
2^128 elliptic curve operations.

A user of ECC field size 8^91+5 therefore arguably uses both more
computation and more communication than would be minimally necessary
to protect their 128-bit keys.

In other words, the extra cost for exceeding 256 bits might be
wasteful. The extra cost might have prevented the user to spend on
other security protections.

Taking this excessive security reasoning further suggests that even
a 256-bit ECC key is excessive to protect a 128-bit symmetric key,
because 2^128 ECC operations of an ECDLP attack could be much more
expensive than a 2^128 symmetric-key operations in an exhaustive
key-search attack. If it were 256 times more expensive, then it
would seem that 2^120-ECC-operation attack would be tolerable, so
that a 240-bit ECC key could be secure enough to protect a 128-bit
symmetric key. One example of such a curve is sect239k1, but
otherwise such smaller curves are not popular, with 256-bit curves
pre-dominating in many standards. This trend suggests that the
reasoning about excessive security in ECC is not being given much
weight.

7.1.4. Non-maximality among six-symbol (DEC) primes

Among six-symbol primes (decimal exponential complexity 6), 8^91+5
is not the largest.

Prime 8^91+5 is smaller than 8^95-9, 9^99+4 and 9^87+4. Curves
defined over these larger field sizes would have better Pollard rho
security (against ECDLP attacks). A reason to to prefer 8^91+5 over
these others, is that it has better security against other important
types of attacks.

The primes 9^99+4 and 9^87+4 are not close to a power of two. This
makes implementing them in software almost twice as slow as 8^91+5.
A more generic form of modular reduction is needed. Generic modular
reduction can be more prone to implementation attacks such as side
channel attacks.

The prime 8^95-9 is quite similar to 8^91+5 in many respects.

Being larger than p, the prime 8^95-9 has a slightly greater risk of
an arithmetic overflow implementation fault than 8^91+5 has.

Field size 8^95-9 has more complicated powering algorithms for
computing field inverses, Legendre symbols, and square roots,
because the binary expansion of numbers like (8^95-9)-2 have many
more 1-bits in their binary expansion. This discrepancy arises
because 8^95-9 is just a little below of power of two, while
p=8^91+5 is just a little above a power of two.

The more complicated powering algorithms needed for 8^95-9 might
make an implementation a little slower. They require more
complicated code, increasing the development cost (such as
debugging), and perhaps increasing the chance of implementation
error.

7.1.5. Five-symbol and four-symbol field sizes

Non-prime field sizes can be use in ECC, and some of these field
sizes can be described in few than the six symbols (of decimal
exponential complexity).

The field size 2^283 in ECC is standardized in curve sect283k1
[SEC2], [Zigbee]. This field size is larger than 8^91+5 (which is
approximately 2^273), so Pollard rho attacks against ECC over field
GF(2^283) are about 32=2^5 times faster than Pollard rho attacks
against ECC over GF(8^91+5).

Indeed, several four-symbol field sizes (consider, 7^99, 8^97, 9^97)
are also larger than 8^91+5.

That all said, composite field sizes in ECC have two security risks:

- Recent progress in ECDLP attacks over composite fields [HPST] and
  [Nagao] is a major red flag against ECC over composite fields,
  suggesting that ECC over composite fields is riskier, and much
  closer to being broken.

- Software speed for large-characteristic field arithmetic is
  typically much higher than small-characteristic field arithmetic.
  Actually, the speed advantage is for software that runs on
  general-purpose hardware with dedicated 64-bit integer
  multiplication circuits. The composite field sizes larger than
  8^91+5 have small characteristic. Software for
  small-characteristic field sizes is not only slower, but more
  complicated, and therefore more prone to implementation faults.

7.1.6. Vulnerability to faulty hardware

Efficient software implementation of 8^91+5, and other
large-characteristic fields, has security that relies of hardware
for 64-bit integer arithmetic.

Corrupted hardware might produce incorrect results for a single pair
of inputs. Detecting such a hardware error by exhaustive search
would be infeasible, because there are 2^128 possible pairs of
inputs. An attacker might be able to find this hardware error by
examining the hardware directly. Yet, a typical user might not be
able to detect this hardware error, or to prevent it.

In this case, the attacker might be able to induce the secret
hardware error by supplying the user with a maliciously crafted the
public key. In this case, the public key would cause the provide
the faulty input values into the hardware. The resulting error
might cause the user to leak their secret key to the attacker, as in
[bug attacks] and [IT].

7.1.7. Other measures of complexity

Decimal exponential complexity means the minimum number of the
symbols from the set

  0 1 2 3 4 5 6 7 8 9 + - * ^ ( )

to specify a number, with the usual meanings of the symbols. (The
symbols * and ^ are ASCII adaptations of traditional mathematical
notations, and their meaning has become fairly standard in
mathematical programming.)

Decimal exponential complexity is only one way to measure Kolmogorov
complexity of a non-negative integer. Other measures of complexity
measures allow larger primes to be expressed in six or even fewer
symbols.

The factorial operation is often notated with an exclamation mark!
Primes larger than 8^91+5 are then expressible in five symbols:
94!-1 is a 485-bit prime number, expressible in five symbols. Such
numbers -- so far as I know -- are not close to a power of two.
Therefore, they are likely have similar inefficiency and
implementability defects to the primes 9^99+4 and 9^87+4, which are
also far from a powers of two.

At some point, increasing the size of the symbol set should be
regarded as an increase in the Kolmogorov complexity. Arguably, the
exclamation mark notation favors the factorial operation over
exponentiation. For example, 7! uses two symbols and 2^7 uses three
symbols, but 5040=7! seems no simpler than 128=2^7.

It is generally understood that there can be no absolutely objective
and best measure of Kolmogorov complexity.

However, in [B3], an alternative measure of Kolmogorov complexity is
considered, that aims to be somewhat objective. It follows Godel's
ideas for the most fundamental definition computable functions,
which Turing later proved to be equivalent in computing power to his
tape machines.

Unlike decimal exponential complexity, the system in [B3] has no
built-in preference for decimal, or even for standard arithmetic
operations like addition, multiplication, and exponentiation.
Instead, all computations must be built up from the successor
function and primitive recursion (or minimization recursion, but
that should be avoided).

The absolute complexity measure in [B3] is difficult to determine
for a given prime. Upper bounds are easy to find, by exhibiting
descriptions, but it is difficult to determine the shortest
description. So far, it seems that the prime 2^521-1 has lower
complexity than 8^91+5, because, among the descriptions found so
far, 2^521-1 currently has shorter description than the shortest yet
found for 8^91+5.

The main downside of 2^521-1 compared to 8^91+5 is its larger size
(almost twice the bit length), and also more complicated powering
algorithms for inversion, Legendre symbols and square roots.

7.2. Curve choice

There are several security risks that might be associated with
specific curve 2y^2=x^3+x/GF(8^91+5).

7.2.1. Special curve equation

The curve equation 2y^2=x^3+x is special among elliptic curve
equations. To see how special it is, we list some of its features.

- Its elliptic curve discriminant is low, at 4, whereas random
  curves have discriminants that average around p/2.

- Its j-invariant is 1728, whereas random curves have j-invariants
  that average around p/2.

- Its automorphism group has size 4, whereas random curves have
  automorphism group of size 2, with overwhelming probability.

- Its curve equation has three monomial terms, whereas random curves
  seems have curve equations with at least 4 terms.

- It has complex multiplication by i, with [i] being a linear
  isogeny.

The specialness of 2y^2=x^3+x is well understood in elliptic curve
theory.

Miller, in 1985, already suggested exercising prudence when
considering such special curves.

More precisely, Miller suggested using curve equation y^2=x^3-ax,
with a is a quadratic non-residue. Such a curve has p+1 points.
This saves the expensive one-time cost of point-counting.

But Miller predicted -- correctly! -- that this convenience is not
free, it may be correlated with a risk. Miller's prediction was
correct, because of the an attack was subsequently found.

Menezes, Okamoto and Vanstone (MOV) found an attack on elliptic
curves of low embedding degree [MOV]. The Miller curve with
coefficient a quadratic non-residue is supersingular, which gives it
 an embedding degree of one. This means that the ECDLP could be
reduced
to the DLP in GF(p), which is much faster.

The curve 2y^2=x^3+x/GF(8^91+5) is not vulnerable to the MOV attack,
because it has high embedding degree.

  Note: To repeat, because the MOV attack was several years later
  than Miller's origin caution from 1985, Miller's prudence seems
  prescient. Perhaps he was also prescient about yet other
  potential attacks (still unpublished), and these attacks might
  affect 2y^2=x^3+x/GF(8^91+5). Perhaps, we ought to continue to
  heed Miller's prudence.

Gallant, Lambert and Vanstone found ways to slightly speed up
Pollard rho attacks against ECDLP, when the curve has an efficient
endomorphism. This attack makes a modest speed-up for binary
Koblitz curves (2^7 times speed-up), but for 2y^2=x^3+x/GF(8^91+5)
makes only a minor speed-up in the ECDLP attack (2 times speed-up),
not much more than the speed-up from from efficient arithmetic.

Many other standard curves, including NIST P-256 [NIST-P-256],
Curve25519, Brainpool [Brainpool], do not have complex
multiplication endomorphism by i, or any other low-degree
endomorphism. Essentially, these curves adhere more fully to
Miller's 1985 advice to be prudent about special curves.

Yet other (fairly) standard curves do have efficient endomorphisms.
Examples includes NIST K-283 (used in [Zigbee]) and secp256k1 (see
[SEC2] and [BitCoin]).

Finally, it is not implausible [KKM] that special curves, including
those efficient endomorphisms, may survive an attack on random
curves.

[7.2.2](#). **Twist insecurity**

The curve 2y^2=x^3+x over 8^91+5 is not twist-secure, here meaning
that an x-only Montgomery ladder implementation that does not
validate x is vulnerable to invalid x that could leak bits of the
scalar multiplier (which could be a long-term secret).

An implementer might use the Montgomery ladder in Diffie--Hellman
and might also re-use private keys. An implementer might imitate an
implementation of a twist-secure curve like Curve25519. The
twist-secure implementation would use a Montgomery ladder, and skip
public-key validation.

Skipping public-key validation with a re-used secret scalar in a
Montgomery ladder implementation might leak information about the
re-used secret scalar, and severely weaken its security.

This document provides ample warnings, but an implementer might be
too busy to read the document, or might only given a small excerpt
of this document, excluding the warnings about public-key
validation. An implementer might accidentally implement public-key
validation with a bug that does public-key validation incorrectly.

Several other standardized curves, such as NIST P-256, are also not
twist-secure. The risk from being twist-insecure for these curves
to quite similar to the risk for 2y^2=x^3+x/GF(8^91+5).

Curves like NIST P-256 might have less risk from being
twist-insecure, because they not Montgomery curves (and not even
isomorphic to Montgomery curves). An implementer has less incentive
to use a Montgomery ladder for these curves, because there is less
efficiency gain compared to more conventional scalar multiplication
algorithms. Conventional scalar multiplication algorithms are
already well-known to require public-key validation.

Actually, the Montgomery ladder might not be the fastest scalar
multiplication algorithm for 2y^2=x^3+x/GF(8^91+5), either, because
of the efficient endomorphism, for example.

So, it is unclear how the risk of twist-insecurity really compares
between it and NIST P-256.

[7.2.3](#). **Point order not near a power of two**

The base point G has prime order q which is not close to a power of
two.

   Note: Its leading hexadecimal expansion 71C71C... and half of
   leading digits (or hex-nibbles) match that of floor((8^90)/9), and
   q/2^266 is approximately 1.7778. See A.4.4.1. for the exact value
   of q.

 Digital signature schemes such as ECDSA have a known security
 vulnerability, first discovered by Bleichenbacher in the case of
 DSA, when q is not close to a power of two. A per-message secret k,
 essentially an ephemeral secret scalar, is computed when generating
 a signature. If the probability distribution k is not uniform in
 [0,q-1], in the sense that some large sub-intervals are about twice
 as likely as others, then the signatures leak information about the
 static signing key d. A leak in k means a leak in d. With enough
 signatures with a leaky k, eventually the leaks in d are enough to
 reveal the whole of d. Once d is revealed, the attacker can forge
 arbitrary signatures.

 A typical implementer generates secret scalar k by generating a
 uniformly random byte string b, with b will be uniformly distributed
 in an interval [0,2^m-1], and then setting k = b mod q. (Sometimes,
 the computation b mod q is delayed until the signing equation, but
 in all cases b mod q is the effective value for k.) If 2^m is
 chosen as the power of two nearest to q, then the effective k will
 have a bias, unless q is very close to 2^m. For
 2y^2=x^3+x/GF(8^91+5), the number q is not very close to 2^m.

 Signature standards mitigate this attack by several methods.
 Choosing longer byte string b, with max value 2^m > 2^64*q, reduces
 the bias of k to a negligible amount. For shorter byte strings, it
 is possible to re-generate b until it belongs to an an interval
 [0,uq-1] for some small integer u.

 By contrast, many standard curves, including NIST P-256 and
 Curve25519, already have q close to a power of two. The reason for
 this is that the field size is close to a power of two, and the
 cofactor is also a power of two (usually 1,2,4, or 8).

7.2.4. Vulnerability to Cheon-type attacks

 The Brown--Gallant--Cheon attack can find d in q^(1/3) computations
 using q^(1/3) queries to an oracle that computes [d]P from any given
 input point P.

 The attack is not very serious for two reasons.

- A large number of queries it requires. Variations of the attack
  can work with fewer queries, but the variation then use more
  computation. As the number of queries approaches 0, then the
  computation approaches $q^{(1/2)}$, matching the cost of Pollard rho.

- Only a very specialized applications of ECC provide an oracle to
  compute [d]P. Static elliptic curve Diffie--Hellman usually
  hashes [d]P, so does not provide such an oracle (unless the hash
  fails to be one-way).

The details of attack include making u queries to the oracle, where
u is a factor of q-1 or q+1. The computation phase then takes about
$(q/u)^{(1/2)}$ steps. Putting u ~ $q^{(1/3)}$, if possible, approximately
minimizes the sum of the number of queries and off-line computation
steps.

A general mitigation to the attack is to choose q such q-1 and q+1
have no factors u in a range that makes the attack impactful. Such
a curve could be called Cheon-resistant. A Cheon-resistant curve
can be used in protocols that provide the static Diffie--Hellman
oracle, without worrying about degrading security from a large
number of queries.

Most standardized curves, including NIST P-256, Curve25519 and the
Brainpool curves, are not Cheon-resistant. Curve
$2y^2=x^3+x/GF(8^{91}+5)$ is not Cheon-resistant either.

7.2.5. Small-subgroup confinement attacks

A fifth risk is a small-subgroup confinement attack, which can also
leak a few bits of the private key.   Curve $2y^2=x^3+x$ over $8^{91}+5$
has 72 elements whose order divides 12.

7.3. Encoding choices

As in all ECC, 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 [NSS], 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 disastrous [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
specification.

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 $Z^{(p-2)} \bmod p$, is one method to
compute the inverse in constant time (if the inverse exists).

7.4. General subversion concerns

The main motivation of curve $2y^2=x^3+x$ over $8^{91}+5$ is to minimize
the risk of curve subversion via a backdoor ([Gordon], [YY],
[Teske]).

A security skeptic ought to consider any security technology's
appearance in a standards development organization document with
suspicion, as an possible effort at subversion. (See [BCCHLV] for
some detailed analysis.) This document, despite its intended
experimental status, can be seen as possible subversion. Other
standards for curves could be also seen as even more successful
efforts a subversion.

  Note: A skeptic needs to be careful not to become paranoid.
  Interoperable ECC has a strong need for standardized curves. If
  there is a non-subverted secure curve, standardizing it is good
  for security.

A skeptic needing to choose between standardized curves can try
objectively examine the curve's intrinsic merits. If that proves
inconclusive, a skeptic can try to objectively examine the
reputation of the proponents of the standard.

This document asks all users, including skeptics, to prefer
mainstream curves like NIST P-256 or Curve25519, to this document's
curve 2y^2=x^3+x/GF(8^91+5). This document invites users to use two
or more curves in ECC, and to use 2y^2=x^3+x/GF(8^91+5) only as a
second line of defense.

The reader is encouraged to take a skeptical viewpoint of curve
2y^2=x^3+x/GF(8^91+5), and of other curves. Skeptical users of the
curve are expected to make a rational as possible decision to add
2y^2=x^3+x/GF(8^91+5) to the list of curves used in multi-curve
ECC.

In other words, 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.

7.5. Risk of low age and eyes (aegis)

The exact curve 2y^2=x^3+x/GF(8^91+5) was (seemingly) first
described to the public in 2017 [AB]. So, it has a very low age, at
least compared to more established curves, like NIST P-256 (from
1999) and Curve25519 (from 2005).

The curve 2y^2=x^3+x/GF(8^91+5) has not been submitted for a
publication to any formally peer-reviewed academic cryptographer
forum, such as the IACR conferences like Crypto and Eurocrypt.

Arguably, other reviewers have little incentive to study it
critically, for a couple reasons:

- The looming threat of a quantum computer has diverted many
  researchers towards studying post-quantum cryptography, such as
  supersingular isogeny Diffie--Hellman.

- Past CFRG disputes over NIST P-256 and Curve25519 (and other ECC
  alternatives) have perhaps tired some reviewers, many of whom
  would prefer to concentrate on deployment of ECC.

The simplistic metric of aegis, meaning in age times eyes (times
incentive), 2y^2=x^3+x/GF(8^91+5), 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 my estimates (see below) some more well-studied curves.

Nonetheless, the curve 2y^2=x^3+x/GF(8^91+5) has some 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 for Curve25519 is 10000.

- NIST P-256: has a special field size, and maybe an estimated aegis
  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 for
  secp256k1 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, essentially one of Miller's curves, except that a=1 is
  a quadratic residue. Miller's curve may not have been studied
  intensely as other curves, but its age matches that ECC itself.
  Miller also hinted that it was not prudent to use a special curve
  y^2=x^3-ax: such a comment may have encouraged some cryptanalysts,
  but discouraged cryptographers, perhaps balancing out the effect
  on the eyes factor the aegis. An estimated aegis for Miller's
  curves is 300.

  Some of the direct estimates of aegis for the similar curves could
  be counted towards to 2y^2=x^3+x/GF(8^91+5), as an indirect form of
  aegis. Some obvious cautions to the reader:

- Small changes in a cryptographic algorithm can cause large
  differences in security. 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). The eyes factor of the aegis score is
     very subjective. It is vulnerable to false positives via a herd
     effect. Despite this caveat, it is not sensible to completely
     ignore the eyes factor in the aegis score. Otherwise, one might
     deploy for cryptographic schemes that might never have been
     studied, which might include many insecure schemes.

7.6. Risk of ECC and multi-curve ECC

   This document argues for continued use of ECC, and also for
   multi-curve ECC.

   There is large consensus to use ECC for the time-being, but not yet
   much consensus for multi-curve ECC.

7.6.1. Risk of ECC, overall

   Some critics might not consider ECC safe to use at all, including
   even the CFRG curves like Curve25519.

   Two criticisms are discussed below.

7.6.1.1. Risk of hidden pre-quantum attacks on ECC

   Extreme skeptics might consider all (or most) ECC to be insecure,
   vulnerable to some unpublished attack, runnable today using
   conventional pre-quantum computers.

   These skeptics are outliers, given how much ECC is used today.

   Nonetheless, they have arguments with some merit. They could argue
   that ECC is too sophisticated, in that only a few elite can claim to
   have truly contributed to its security analysis. Indeed, ECC is so
   sophisticated, that nobody can truly claim to fully understand it,
   especially its security.

   These skeptics might prefer RSA or finite-field Diffie--Hellman.

   These skeptics ought to be willing to use a hybrid of ECC in a
   combination with some other form of public-key cryptography.

7.6.1.2. Risk of quantum computer attacks on forward secrecy

   Some users believe that the two probabilities below are so high,
   that ECC can no longer provide much security (in terms of secrecy).

- The probability that an attacker records their ECC-protected
  ciphertext.

- The probability that an attacker has, or will soon have, a quantum
  computer capable of breaking ECC.

These users are optimistic about quantum computers. These
quantum-optimists might think that we should stop bother to ECC,
maybe even right now, and take our chances with some post-quantum
cryptography.

The general consensus seems to be address this quantum threat by
using ECC combined with a post-quantum cryptography.

7.6.2. Multi-curve ECC might be wasteful

Milder skeptics might argue that multi-curve ECC does not add enough
benefit over single-curve ECC to justify its cost.

These skeptics might infer the risk of curve-specific attacks is
already low enough. Major current curves, such as NIST P-256,
Curve25519 and Brainpool, already include mitigations against
subversive curve-specific attacks. The existing mitigation might be
enough for milder skeptics. After all, the latest curve-specific
ECDLP attacks are from 2000, at least for prime-field curves, which
might convince milder skeptics that curve-specific ECDLP attacks
have been exhausted for prime-field curves. The skeptics might also
feel that curves like Curve25519 have resolved the risk of
implementation attacks to the lowest possible for ECC.

8. References

8.1. 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>.

8.2. Informative References

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

[AMPS] M. Albrecht, J. Massimo, K. Paterson, and J. Somorovsky.
   Prime and Prejudice: Primality Testing Under Adversarial
   Conditions, IACR ePrint, 2018. <https://ia.cr/2018/749>

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

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

[B3] D. Brown. Rolling up sleeves when subversion's in the field?
    IACR eprint, 2020. <https://ia.cr/2020/074>

[B4] D. Brown. GLV+HWCD for 2y^2=x^3+x/GF(8^91+5). IACR ePrint,
    2021. <http://ia.cr./2021/383>

[GG] S. Galbraith and P. Gaudry. Recent progress on the elliptic
   curve discrete logarithm problem. IACR ePrint,
   2015. <https://ia.cr/2015/1022>

[KKM] A. Koblitz, N. Koblitz and A. Menezes. Elliptic Curve
    Cryptography: The Serpentine Course of a Paradigm Shift, IACR
    ePrint, 2008. <https://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. <https://ia.cr/2014/571>

[Elligator] D. Bernstein, M. Hamburg, A. Krasnova and T. Lange.
   Elligator: Elliptic-curve points indistinguishable from uniform
   random strings, IACR ePrint, 2013. <https://ia.cr/2013/325>

[NIST-P-256] NIST, FIPS PUB 186-4, Digital Signature Standard (DSS).
    <https://csrc.nist.gov/publications/detail/fips/186/4/final>.

[Zigbee] Zigbee Smart Energy Profile Specification, 2019.
    <https://zigbeealliance.org/wp-content/uploads/2019/11/docs-07-
    5356-19-0zse-zigbee-smart-energy-profile-specification-v1.0.pdf>

[Brainpool] M. Lochter and J. Merkle. RFC 5639. Elliptic Curve
   Cryptography (ECC) Brainpool Standard Curves and Curve Generation

[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.

   [NSS] D. Naccache, N. Smart and J. Stern. Projective coordinates
      leak, IACR ePrint, 2003. <https://ia.cr/2003/191>

   [BitCoin] BitCoin core. <https://bitcoincore.org/en/about/>
      <https://github.com/bitcoin-core/secp256k1>

   [Bleichenbacher] D. Bleichenbacher. On The Generation of One-Time
      Keys in DL Signature Schemes. Presentation at IEEE P1363 Working
      Group meeting, November 2000

   [Gordon] D. Gordon, Designing and detecting trapdoors for discrete
      log cryptosystems, Advances in Cryptology - Crypto '92, 1992.

   [HPST] Y. Huang, C. Petit, N. Shinohara and T. Takagi. On
      Generalized First Fall Degree Assumptions, IACR ePrint 2015.
      <https://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] A. Young and M. Yung. The Dark Side of Black-Box Cryptography,
      or: Should we trust Capstone? Advances in Cryptology -- Crypto
      '96, 1996.

Appendix A. Why 2y^2=x^3+x/GF(8^91+5)?

   This section says why curve 2y^2=x^3+x/GF(8^91+5) can improve ECC,
   if used properly in multi-curve ECC.

      Note: Other sections (especially 4, 5, 6, B, C, and D) cover
      some relatively routine ECC details about how to use
      2y^2=x^3+x/GF(8^91+5). Section 8 covers some reasons why one
      might not want to use 2y^2=x^3+x/GF(8^91+5).

A.1. Not for single-curve ECC

   Curve 2y^2=x^3+x/GF(8^91+5) is riskier than other IETF-approved
   curves, such as NIST P-256 and Curve25519, for at least the
   following reasons:

   - it is newer, so riskier, all else equal, and

   - it is special, with complex multiplication by i: consensus
     continues to agree with Miller's original 1985 opinion that
     using (such) special curves is not "prudent".

  Koblitz, Koblitz and Menezes [KKM] somewhat dissent from the
  consensus against special curves. They list several plausible cases
  of special curves -- including some with complex multiplication --
  that they argue might well be safer than random curves.

  Despite this dissent, this report adheres to the consensus, which is
  to prefer other curves for single-curve ECC.

  The relative newness of 2y^2=x^3+x/GF(8^91+5) is not entire. The
  curve equation is isomorphic to one proposed by Miller in 1985,
  making it older than the isomorphism class of curve equations in
  NIST P-256 or Curve25519. The field size, the prime 8^91+5=2^273+5,
  is a prime likely to have been considered before the field size
  primes NIST P-256 or Curve25519, but probably not in an application
  to ECC (i.e. probably in surveys of special primes).

A.2. Risks of new curve-specific attacks

  A risk for all ECC is new curve-specific attacks, especially attacks
  on the elliptic curve discrete logarithm problem. A new
  curve-specific attack could break any ECC using the affected curves.

  The main benefit to ECC of curve 2y^2=x^3+x/GF(8^91+5) is to reduce
  this risk in multi-curve variant of ECC.

    Note: an arguably larger risk, a quantum computer capable of
    running Shor's algorithm, looms over all of ECC. The probability
    of this risk is basically independent of the probability new
    curve-specific attack, but the impacts are heavily dependent, if a
    quantum attack impacts ECC, then the new curve-specific attacks
    are totally moot. Also, even if no quantum attack on ECC emerges,
    but PQC supplements or replaces ECC, then a new curve-specific
    attack becomes much more tolerable. For sake of argument, suppose
    probabilities 1% for a new curve-specific attack by 2030, and 10%
    for a quantum-attack on ECC by 2030. Addressing the 10%
    probability risk is more urgent, but there is still a 90% chance
    that of no-quantum-attack. Assuming that PQC is combined with ECC
    (instead of replacing it) and assuming that the 10% and 1%
    probabilities above are formally independent, then there is 0.9%
    probability that new-curve specific on ECC by 2030 would affect
    PQC+ECC systems, reducing their security to that of PQC only.

A.2.1. What would be considered a "new curve-specific" attack?

  The idea of new curve-specific attacks is now discussed. The
  purpose is to remind the reader of the risks, by comparison to past
  curve-specific attacks, so that a user can estimate the benefits of
  addressing the risk. Ultimately, the reader should make an informed
  as possible decision whether the extra cost of multi-curve is
  warranted.

A.2.2.1. What would be considered a "new" attack?

  The "new" in "new curve-specific attack" means hypothetical and not
  yet published, and hence, either future or hidden. This
  contemplates an adversary with superior cryptanalytic capability
  than current state-of-the-art knowledge.

A.2.2.2. What is, would be, considered a "curve-specific attack"?

  The "curve-specific" in "new curve-specific attack" means that the
  following conditions on the attack are true

    - it affects almost ECC algorithms using the specific curve
      (typically, if the discrete logarithm problem is easy for that
      curve, or in some cases, the decision Diffie--Hellman problem),

    - it does not affect ECC using at least one other curve
      (typically, many other curves), and

    - it would not affect a generic group of the same size of the
      secure ECC group.

  Note: For example, the naive Pollard-rho attack is not
  "curve-specific" because it fails the second condition and third
  condition (it affects all curves and all generic groups of equal
  or smaller size than the attacked curve). The Pohlig--Hellman
  attack (on smooth order groups) is not curve-specific because it
  fails the third condition.

  Note: A side-channel attack on an ECC implementation is not
  necessarily "curve-specific" in the strict sense above, if
  another ECC implementation using the same curve resists the
  attack. Some curves may be more prone than others to side-channel
  attacks, here we refer to that situation "curve-specific
  implementation-vulnerability".

Prime-field curves were affected by two curve-specific attacks (on
the discrete logarithm): the MOV attacks, and the SASS attack, both
from before 2001. For the decision Diffie--Hellman problem, a
generalization of the MOV attack can be considered as
curve-specific.

For non-prime-field curves, more recent curve-specific attacks have
been discovered. (See [GG] for a survey.)

A.2.2.3. Rarity of published curve-specific attacks

The known curve-specific attacks against prime-field curves are rare
in the sense of having negligible probability of affecting a random
curve (over a given prime-field).

Some of these are attacks are also field-specific too.
These attacks somewhat rare among all possible non-prime-field
curves (though in some cases the probability among certain class of
curves is non-negligible).

If the rarity of the known curve-specific attacks carries over to
any new curve-specific attacks, then truly random curves should
resist the new curve-specific attacks, except with negligible
probability. Honestly generated, non-random curves should also
resist the new curve-specific attacks, except in the unfortunate
case the new curve-specific attack is correlated with the honest
curve generation criteria.

A.2.2.4. Correlation of curve-specific efficiency and attacks

Many of the known curve-specific attacks affected previously
proposed curves, and presumably honestly generated curves.

For example, supersingular curves were proposed for their slightly
greater efficiency over ordinary curves, but then turned out to be
vulnerable to the MOV attack. (Similarly, curves vulnerable to the
SASS attack were proposed for slight efficiencies, before the SASS
attack was published.) So, such correlations are not only
plausible, but the real-world pattern for ECC. Accidents have
already happened for such non-random curves.

Worse yet, if a non-random curve is chosen maliciously, a
correlation between a hidden curve-specific attack and some sensible
curve generation criteria might well make it possible for a
maliciously chosen non-random curve to be made vulnerable to a
hidden curve-specific attack.

A.3. Mitigations against new curve-specific attacks

  Because the risk of new curve-specific attack is nonzero, applying
  mitigations against the risk potentially improves security, albeit
  at some cost.

A.3.1. Fixed curve mitigations

  Often, a single fixed curve is used across a system of ECC users,
  generally for reasons of efficiency. This exposes the system to the
  nonzero risk of new curve-specific attacks.

A.3.1.1. Existing fixed-curve mitigations

  Some of the better established fixed curve have sensibly included
  mitigations against the nonzero risk of new curve-specific attacks.

  - NIST curve P-256 has coefficients derived from the output of
    SHA-1, perhaps aiming to avoid any new curve-specific weakness
    that would apply rarely to random curves, although inadequately
    so, because the seed input to the hash is utterly inexplicable,
    and plausibly manipulable.

  - Bernstein's Curve25519 results from a "rigid", non-random design
    process, favoring efficiency over all else, perhaps eliminating
    intentional subversion towards a new curve-specific weakness.

  - Brainpool's curves are derived using hash functions applied to
    nothing-up-my-sleeve numbers, perhaps aiming to mitigate both
    intentional subversion and accidental rare weakness.

    Note: A reasonable inference from these curves is that risk of new
    curve-specific attacks warranted the mitigations used (as listed
    above). The risk may be less now that further time has passed,
    because no other curve-specific attacks against prime-field curves
    arose in the interim. The risk is still not zero, so the
    mitigations may still be warranted.

A.3.1.2. Mitigations used by 2y^2=x^3+x/GF(8^91+5)

  The curve 2y^2=x^3+x/GF(8^91+5) includes similar fixed-curve
  mitigations against the risk of new curve-specific attacks:

  - a short description (low Kolmogorov complexity), aiming to have
    little wiggle for an intentional embedded weakness (somewhat like
    a nothing-up-my-sleeve number used in the Brainpool curves),

  - a set of special efficiencies, such as a curve endomorphism,
    Montgomery form, and fast field operation (somewhat like the
    "rigid" properties of Curve25519 favor efficiency as a mitigation
    to fight off intentional embedded weakness),

  - a prime field, to stay clear of recent curve-specific attacks on
    non-prime-field ECC.

  These mitigations do not suffice to justify its use in single-curve
  ECC (instead of more established non-special curves).

    Note: The mitigations above, like those of NIST P-256 and
    Curve25519, have a cost which consists mostly of a one-time
    computation. The mitigations are somewhat warranted, even if
    multi-curve ECC, because the aim of multi-curve is to hedge the
    risk of curve-specific attacks, so it makes sense for each
    individual curve to include mitigations against this risk.

A.3.2. Multi-curve ECC

  This section further motivates the value of multi-curve ECC over
  single-curve ECC, but does specify a detailed way to do multi-curve
  ECC.

  Multi-curve ECC is only really effective if used with a diverse set
  of curves. Multi-curve ECC SHOULD use a set of curves including the
  three curves:

      NIST P-256, Curve25519, and 2y^2=x^3+x/GF(8^91+5).

  Multi-curve ECC aims to further mitigate the risk of curve-specific
  attack, by securely combining a diverse set of curves. The aim is
  that at least one of the curves used in multi-curve ECC resists a
  new curve-specific attack (if a new attack ever appears). This aim
  is only plausible if the set of curves used is diverse, in features
  or in authorship.

  This curve contributes to the diversity necessary for multi-curve
  ECC, with special technical features distinct from established
  curves NIST P-256 and Curve25519 (and Brainpool):

    - complex multiplication by i (low discriminant, rather than
      high),

    - a greater emphasis on low Kolmogorov descriptional complexity
      (rather than hashed coefficient or efficiency).

A.3.2.1. **Multi-curve ECC is a redundancy strategy**

  Multi-curve ECC is an instance of a strategy often called
  redundancy, applied to ECC. Redundancy is quite general in that it
  can be applied to other types of cryptography, to other types of
  information security, and even to safety systems. Other names for
  redundant strategies include:

    strongest-link, defense-in-depth, hybrid, hedged, composite,
    fail-safe, diversified, resilient, belt-and-suspenders, fault
    tolerant, robust, multi-layer, robustness, compound, combination,
    etc.

A.3.2.2. **Whether to use multi-ECC**

  Multi-curve ECC mitigates the risk of new curve-specific attacks, so
  ought to be used instead of single-curve ECC if affordable, such as
  when

    - the privacy of the data being protected has higher value than
      the extra cost of multi-curve ECC, which may be the case for at
      least financial, medical, or personally-identifying data, and

    - ECC is only a tiny portion of the overall system costs, which
      would be the case if the data is human-generated or high-volume,
      or if ECC is combined with slow or large post-quantum
      cryptography (PQC).

A.3.2.2.1. **Benefits of multi-curve ECC**

  The benefit of multi-curve ECC is difficult to quantify. The aimed
  benefit over single-curve ECC is extra security, in the event of a
  significant curve-specific attack.

  No extra security results if all the curves used are the same. The
  curves must be diverse, so that a potential attack on one is somehow
  unlikely to affect the other. This diversity is difficult to
  assess. Intuitively, a geometric metaphor of a polygon for the
  space of all choices might help. Maximally distant points in a
  polygon tend to be vertices, the extremities of the polygon.
  Translating this intuition suggests choosing curves at the extremes
  of features.

    Note: By contrast, in a single-curve ECC, the geometric
    metaphor suggests a central internal point, on the grounds that
    each vertex is more likely to be affected to a special attack.
    Carrying this over to multi-curve suggests that a diverse set
    ought to include a non-extreme curve too.

As always, the benefit of security is really the negative of the
cost of an attack, including the risk.

The contextual benefit of multi-curve ECC therefore depends very
much on the application, involving the assessing both the
probability of attack, and the impact of the attack.

Higher value private data has greater impact if attacked, and
perhaps also higher probability, if the adversary is more motivated
to attack it.

Low probability of attacks are mostly inferred through failed but
extensive cryptanalysis efforts. Normally, this is only intuited,
but approaches to quantifiably estimate these probabilities is
possible too, under sufficiently strong assumptions.

A.3.2.2.2. Costs of multi-curve ECC

The cost of multi-curve ECC is fairly easy to quantify (easier than
quantifying the benefit).

The cost of multi-curve is meant to be compared to the cost of
single-curve ECC.

The cost ratio is approximately the number of curves used. The cost
difference depends on the devices implementing the ECC.

For example, on a current personal computer, the extra cost per ECC
transaction can include up to 1 millisecond of runtime and sending
an extra 30 bytes or more. In low-end devices, the time may be
higher due to slower processors.

The contextual cost of ECC depends on the application context. In
some applications, such as personal messages between two users, the
cost (milliseconds and a few hundred bytes) is affordable relative
to the time users spent writing and reading the messages. In other
applications, such as automated inter-device communication with
frequent brief messages, single-curve ECC may already be a
bottleneck, costing most of the run-time.

A.3.2.3. Applying multi-curve ECC

For key establishment, NIST recently proposed (in a draft amendment
to Special Publication 800-133 on key derivation) a mechanism to
support deriving a single symmetric key from the result of multiple
key establishments. In summary, the mechanism is that the raw ECDH
shared secrets would be concatenated and fed into a hash-based key
derivation function.

An alternative would be to XOR multiple shared symmetric-key
together.

So, multi-curve elliptic curve Diffie--Hellman (ECDH) key agreement
could use one of these mechanism to derive a single key from
multi-curve ECDH.

A mechanism to support sending more than one ECDH public key
(usually ephemeral), with an indication of the curve for each ECDH
key, would also be needed.

For signatures, the simplest approach is to attach multiple
signatures to each message. (For signatures providing message
recovery, then an approach is to apply the results, with outer
signatures recover the inner signed message, and so on.)

A.4. General features of curve 2y^2=x^3+x/GF(8^91+5)

This subsection describes some general features of the curve

   2y^2=x^3+x/GF(8^91+5),

presuming a familiarity with elliptic curve cryptography (ECC).

Each of a set of well-established features, such as Pollard rho
security or Montgomery form, for ECC in general are evaluated and
summarized for the specific curve 2y^2=x^3+x/GF(8^91+5).

   Note: Interoperable ECC requires a few more details than are
   deducible from mathematical description 2y^2=x^3+x/GF(8^91+5) of
   the curve, such encoding points as byte strings. These details
   are discussed in Sections 4, 5, and 6.

A.4.1. Field features

The curve's field of definition, GF(8^91+5), is a finite field, as
is always the case in ECC. (Finite fields are Galois field, and the
field of size is p is written as GF(p).)

The field size is the prime p=8^91+5. (See the appendix for a
Pratt primality certificate.)

In hexadecimal (base 16, big-endian) notation, the number 8^91+5 is

2000000000000000000000000000000000000000000000000000000000000000000005

with with 67 zeros between 2 and 5.

The most recent known curve-specific attacks on
prime-field ECC are from 2000.

Prime fields in ECC tend be more efficient in software than in
hardware.

The prime p is very close to a power of two. Primes very close to a
power of two are sometimes known as Crandall primes. Reduction
modulo p is more efficient for Crandall primes than for most other
primes (or at least random primes). Perhaps Crandall primes are
more resistant to side-channel attacks or implementation faults than
than most other primes.

The fact that p is slightly larger than a power of two -- rather
than slightly lower -- means that powering algorithms to compute
inverses, Legendre symbols, and square roots are simpler and
slightly more efficient (than would be for prime below a 2-power).

A.4.2. Equation features

The curve equation $2y^2=x^3+x$ has Montgomery form,

   $by^2=x^3+ax^2+x,$

with (a,b) = (0,2). This permits the Montgomery ladder scalar point
multiplication algorithm to be used, which makes it relatively
efficient, and also easier to protect against side channels.

The curve $2y^2=x^3+x$ has complex multiplication by i, meaning an
endomorphism

  (x,y) -> (-x,iy).

   Note: Strictly speaking, over some fields, the curve the curve has
   quaternionic multiplication (where it is said to be
   supersingular), in which case, the term "complex multiplication"
   is not the best fit.

The endomorphism permits the Gallant--Lambert--Vanstone (GLV) scalar
multiplication algorithm, which makes it relatively efficient. See
[B4] for a discussion of combining GLV with
Hisil--Wong--Carter--Dawson arithmetic for twisted Edwards
coordinates, as it applies to $2y^2=x^3+x/GF(8^{91}+5)$.

The GLV method can also be combined with Bernstein's two-dimensional
variant of the Montgomery ladder algorithm. See [B1] for some
discussion.

The curve has j-invariant 1728, because it has complex
multiplication by i.

  Note: The j-invariants 0 and 1728 are special in that the curves
  with these j-invariants have more than two automorphisms.
  (Relatedly, over complex numbers, the moduli space of elliptic
  curves is an orbifold, with exactly two non-smooth points, at j=0
  and j=1728.)

A.4.3. Finite curve features

This section describes features of 2y^2=x^3+x/GF(8^91+5) as a finite
curve consisting, the points (x,y) for x,y in GF(p), and also the
point at infinity. In other words, these features are specific to
the combination of both the finite field and the curve equation.

  Note: In algebraic geometry, these points are said to rational
  over k=GF(p), and the set of rational points written as E[k] =
  (2y^2=x^3+x)[GF(8^91+5)], to distinguish from points with
  coordinates in the algebraic closure of k=GF(p).

Many security properties, and a few performance properties, of ECC
are specific to a finite curve.

A.4.3.1. Curve size and cofactor

The curve (of points rational over GF(8^91+5)) has size (order) 72q
for a large prime q, which is, in hexadecimal,

  71C71C71C71C71C71C71C71C71C71C71C7A4ACED12AE9418569B932B8A7B80438A9

    NOTE: Appendix E has a Pratt primality certificate for q.

So, the curve has cofactor 72.

The curve size can verified by implementing the curve's elliptic
curve arithmetic, and scalar multiplying random points on the curve
by the claimed size.

The size can also be partially verified with the complex
multiplication theory and a little big integer arithmetic, as
reviewed below.

The prime p=8^91+5 has p=1 mod 4, so a theorem of Fermat says there
exist integers u and v such that p=u^2+v^2. Numbers u and v can
found using a special case of Cornacchia's algorithm, and are listed
further below.

Complex multiplication theory says that a curve with complex
multiplication by i has size s=(u+1)^2+v^2 = p+2u+1. By negation of
u, and by swapping u and v, there are four possible sizes, p+2u+1,
p-2u+1, p+2v+1, p-2v+1. These are known as the (quadratic) twist
sizes.

Curve 2y^2=x^3+x/GF(8^91+5) has one of these four sizes. In this
case, its size s is divisible by 72, and has large prime factor q =
s / 72.

The following 'bc' program includes values for u and v applicable to
2y^2=x^3+x/GF(8^91+5), verifies these calculations, and outputs q.

```
p = 8^91+5
u = 1043033027901133467787029269772887055144769
v = 655585368017578752283604058587318062281506
if ( p != u^2+v^2 ) { "u and v incorrect" ; halt }
s = (u+1)^2 + v^2
if ( 0 != (s % 72)) { "size not divisible by 72" ; halt}
q = s/72
q
```

Note: Theory only indicates that s has one of four values, so an
extra step is needed to verify which of the four values is the
size. Scalar multiplication by s is a general method. A faster
method, which happens to work for 2y^2=x^3+x/GF(8^91+5), is to
show that only one of the four candidate sizes is divisible by 3,
and then demonstrate a point of order 3 on this curve. Symbolic
calculation with elliptic curve arithmetic show that the point
(x,y) has order 3 if 3x^4 + 1 = 0 in GF(p). The big integer
calculation (-(1+2p)/3)^((p-1)/4) = 1 mod p shows that such an x
exists in GF(p).

Note: The Schoof--Elkies--Atkin (SEA) point-counting algorithm can
compute the size of any general curve, but is slower than methods
for some special curves, which is why Miller had already suggested
in 1985 to use special curves with equation y^2=x^3-ax. Miller
suggested using supersingular curves, but restriction of the
coefficient a, but those curves are insecure. Instead, we have
chosen a different a (in this case a=-1 is isomorphic to
2y^2=x^3+x), which is not supersingular, but still has much easier
point counting than the generic SEA algorithm.

A.4.3.2. Pollard rho security

  The prime q is 267-bit number. The Pollard rho algorithm for
  discrete logarithm to the base G (or any order q point) takes
  (proportional to) sqrt(q) ~ 2^133 elliptic curve operations. The
  curve provides at least 2^128 security against Pollard rho attacks,
  with about 5 bits to spare.

    Note: Arguably, the fact ECC operations are slower than
    symmetric-key operations (such as hashing or block ciphers),
    means that ECC security should be granted a few extra bits,
    perhaps 5-10 bits, of security when trying to match ECC security
    with symmetric-key security. In this case, one might say that
    2y^2=x^3+x/GF(8^91+5) resists Pollard-rho with 2^140 security,
    providing 12 bits of extra security. The extra security can be
    viewed as a safety margin for error, or as an excessive to the
    extent the smaller, and faster curves would more than suffice to
    match 2^128 security of SHA-256 and AES-128.

  Gallant, Lambert, Vanstone, show how to speed up Pollard rho
  algorithms when the group has an extra endomorphism, which would
  apply to 2y^2=x^3+x. The speed-up here amounts to a couple of bits
  in the security,

A.4.3.3. Pohlig--Hellman security

  The small cofactor means the curve effectively resists
  Pohlig--Hellman attack (a generic algorithm to solve discrete
  logarithms in any group in time sqrt(m) where m is the largest
  prime factor of the group size).

    Note: Consensus in ECC is to recommend a small factor, such as 1,
    2, 4, or 8, despite the fact that, for random curves, the typical
    cofactor is approximately p^(1/3), which is much larger. The
    small cofactor helps resists Pohlig--Hellman without increasing
    the field size. (A larger field size would be less efficient.)

A.4.3.4. Menezes--Okamoto--Vanstone security

  The curve has a large embedding degree. More precisely, the curve
  size 72q has q with embedding degree (q-1)/2.

  This means that the discrete logarithms to base G (a point of order
  q) resist Menezes--Okamoto--Vanstone attack.

The large embedding degree also means that that no feasible pairings
exist that could be used solve the decision Diffie--Hellman problem
(for points of order q). Similarly, the larger embedding degree
also means, it cannot be used for pairing-based cryptography (and it
would already too small to be used for pairing-based cryptography).

  Note: Intuitively, a near-miss or a close-call could describe this
  curve's resistance to the MOV attack. For about half of all
  primes P, then curve 2y^2=x^3+x is supersingular over GF(P), with
  embedding degree 2, making them vulnerable to the MOV attack
  reduces the elliptic curve discrete logarithm to the finite field
  discrete logarithm over GF(P^2). Miller suggested in 1985 to use
  isomorphic equations, y^2=x^3-ax, without knowing about the 1992
  MOV attack. These special curves would then be vulnerable with
  ~50% chance of being, depending on the prime P. This curve was
  chosen in full knowledge of the MOV attack.

  Note: The near-miss or close-call intuition is misleading, because
  many cryptographic algorithms become insecure based on the
  slightest adjustment to the algorithm.

  Note: The non-supersingularity means that the endomorphism ring is
  commutative. For this curve the endomorphism ring is isomorphic
  to the ring Z[i] of Gaussian integers.

A.4.3.5. Semaev--Araki--Satoh--Smart security

  The fact that the curve size 72q does not equal p, means that the
  curve resists the Semaev--Araki--Satoh--Smart attack.

A.4.3.6. Edwards and Hessian form

  The cofactor 72 is divisible by 4, so the curve isomorphic to a
  curve with an Edwards equation, permitting implementation even more
  efficient than the Montgomery ladder.

  The Edwards form makes possible the Gallant--Lambert--Vanstone
  method that used the efficient endomorphism.

  The cofactor 72 is also divisible by 3, so the curve is isomorphic
  to a curve with a Hessian equation, which is another type of
  equation permitting efficient implementation.

  Note: It is probably too optimistic and speculative to hope that
  future research will show how to take advantage by combining the
  efficiencies of Edwards and Hessian curve equations.

A.4.3.7. Bleichenbacher security

Bleichenbacher's attack against faulty implementations
discrete-log-based signatures fully affects $2y^2=x^3+x/GF(8^{91}+5)$,
because the base point order q is not particularly close to a power
of two. (Some other curves, such as NIST P-256 and Curve25519, have
the base point order is close to a power of two, which provides
built-in resistant to Bleichenbacher's faulty signature attack.)

  Note: Bleichenbacher's attack exploits the signature
  implementation fault of naively reducing uniformly random bit
  strings modulo q, the order of the base point, which results in a
  number biased towards the lower end of the interval [0,q-1].

So, q-uniformization of the pre-message secret numbers is critical
for signature applications of $2y^2=x^3+x/GF(8^{91}+5)$. Various
uniformization methods are known, such as reducing extra large
numbers, repeated sampling, and so on.

A.4.3.8. Bernstein's "twist" security

Unlike Curve25519, curve $2y^2=x^3+x/GF(8^{91}+5)$ is not
"twist-secure", so a Montgomery ladder implementation for static
private keys often requires public-key validation, which is
achievable by computation of a Legendre symbol related to the
received public key.

In particular, a Montgomery ladder x-only implementation that does
not implement public-key validation will process a value x for which
no y satisfying the equation exists in GF(p). More precisely, a y
does exist, but it belongs to the extension field $GF(p^2)$. In this
case, the Montgomery ladder treats x as though it were (x,y) where x
is GF(p) but y is not. Such points belong to a "twist" group, and
this group has order:

  2^2 * 5 * 1526119141 * 788069478421 * 182758084524062861993 *
  345246493045167733003600525204 0328546941

An adversary can exploit this, by finding such invalid x that
correspond to a lower order group element, and thereby try to learn
partial information about a static private key used by a
non-validating Montgomery ladder implementation.

[A.4.3.9](#). **Cheon security**

  Niche applications in ECC involve revealing points [d^e]G for one
  secret number d, and many different integer e, or at least one large
  e. One way such points could be reveal is in protocols that employ
  a static Diffie--Hellman oracle, a function to compute [d]P from any
  point P, which might be applied e times, if e is reasonably small.

  Typical ECDH, to be clear, would never reveal such points, for at
  least two reasons:

    - ECDH is ephemeral, so that the same d is never re-used across
      ECDH sessions (because d is used to compute [d]G and [d]Q, and
      then discarded),

    - ECDH is hashed, so though P=[d]G is sent, the point [d]Q is
      hashed to get k = H([d]Q), and then [d]Q is discarded, so the
      fact that hash is one-way means that k should not reveal [d]Q,
      if k is ever somehow revealed.

  The Brown--Gallant--Cheon q-1 algorithm finds d, given [d^e]G, if
  e|(q-1). It uses approximately sqrt(q/e) elliptic curve operations.
  The Cheon q+1 algorithm finds d, given all the points [d]G, [d^2]G,
  ..., [d^e]G, if e|(q+1), and takes a similar amount of computation.
  These two algorithms rely on factors e of q-1 or q+1, so the
  factorization of these numbers affects the security against the
  algorithm.

  Cheon security refers to the ability to resist these algorithms.

  It is possible seek out special curves with relatively high Cheon
  security, because q-1 and q+1 have no suitable factors e.

  The curve 2y^2=x^3+x/GF(8^91+5) has typical Cheon security in terms
  of the factorization of q-1 and q+1. Therefore, in the niche
  applications that reveal the requisite points, mitigations ought to
  be applied, such as limiting the rate of revealing points, or using
  different value d as much as possible (one d per recipient).

  For 2y^2=x^3+x/GF(8^91+5) the factorization of q-1 and q+1 are:

    q-1 = 2^3 * 101203 * 23810182454264420359 *
          10934784357463776473342498062299965925956115086976992657
  and

    q+1 = 2 * 3 * 11 * 21577 * 54829 * 392473 * 854041 *
          8054201530811151253753936635581206856381779711451564813041

The q-1 and q+1 algorithms convert an oracle for function P -> [d]P
into a way to find d. This may be viewed as a reduction of the
discrete logarithm problem to the problem of computing the function
P -> [d]P for the target d. In other words, computing P -> [d]P is
almost as difficulty as solving the discrete logarithm problem. In
many systems with a static Diffie--Hellman secret d, computing the
function P -> [d]P needs to be difficult, or the security will be
defeated. In these case, an efficient q-1 or q+1 algorithm provides
a security assurance, that the computing P -> [d]P without knowing d
is about as hard as solving the discrete logarithm problem.

A.4.3.10. Reductionist security assurance for Diffie--Hellman

A series of research work, from den Boer, from Maurer and Wolf, and
from Boneh and Lipton, shows that Diffie--Hellman oracle can be used
to solve a discrete logarithm, under certain conditions. In other
words, the discrete logarithm problem can sometimes be reduced to
the Diffie--Hellman problem.

This can be interpreted as a security assurance that Diffie--Hellman
problem is at least as hard the discrete logarithm problem, albeit
perhaps with some gap in the difficulty. This formalized security
assurance supplements the standard conjecture that the
Diffie--Hellman problem is at least as hard as the discrete
logarithm. (A contrarian view is that special conditions under
which such a reduction algorithm is possible might coincide with
special conditions under which the discrete logarithm problem is
easier.)

The general idea is to consider a Diffie--Hellman oracle in a group
of order q to provide multiplication in a special representation
field of order q. Recovering the ordinary field representation from
the special field representation amounts to solving the discrete
logarithm problem.

To recover the ordinary representation, the idea is to construct an
auxiliary group of smooth order, where the group is an algebraic
groups over the field of size q. Solving a discrete logarithm in
the auxiliary group is possible using the Pohlig--Hellman problem,
and solving the discrete logarithm in the auxiliary reveals the
ordinary representation of the field, which, as already noted
reveals the discrete logarithm in the original group.

The most obvious auxiliary groups have orders q-1 and q+1, but these
are not smooth numbers. The next most obvious auxiliary are
elliptic curve groups with complex multiplication by i, but none of
these four group have smooth orders either.

A peculiar strategy to show the existence of an auxiliary group of
smooth order without having any effective means of constructing the
group. This can be done by finding a smooth number in the Hasse
interval of q.

Appendix B. Test vectors

The following are some test vectors, with values explained further
below.

```
00000000000000029352b31395e382846472f782b335e783d325e79322054534554
0000000000000000000000000000000000000000000000000000000000000000117
c8c0f2f404a9fabc91c939d8ea1b9e258d82e21a427b549f05c832cf8d48296ffad7
5f336f56f86de3d52b0eab85e527f2ac7b9d77605c0d5018f5faa4243fd462b1badd
fc023b3f03b469dca32446db80d9b388d753cc77aa4c3ee7e2bb86e99e7bed38f509
8c2b0d58eb27185715a48d6071657273dfbb861e515ac8bac9bfe58f2baa85908221
8c2b0d58eb27185715a48d6071657273dfbb861e515ac8bac9bfe58f2baa85908221
```

The test vectors are explained as follows. (Pseudocode generating
them is supplied in Appendix C.2.)

Each line is 34 bytes, representing a non-negative 272-bit integer.
The integer encoding is hexadecimal, with most significant hex
digits on the left, which is to say, big-endian.

  Note: Public keys are encoded as 34-byte strings are
  little-endian. Encoded public keys reverse the order of the bytes
  found in the test vectors. The pseudocode in Appendix C.2 should
  make this clear: since bytes are printed in reverse order.

Each integer is either a scalar (a multiplier of curve points), or
the byte representation of a point P through its x-coordinate or the
x-coordinate of iP (which is the the mod 8^91+5 negation of the
x-coordinate of P).

The first line is a scalar integer x. Its nonzero bytes are the
ASCII representation of the string "TEST 2y^2=x^3+x/GF(8^91+5)",
with the byte order reversed. As a private key, this value of x
would be totally insecure, because it is too small, and like any
test vector, it is public.

The second line is a representation of G, a base point on the curve.

The third line is the representation of z = xG.

The fourth and fifth lines represent updated values of x and z,
obtained after application of the following 100000 scalar
multiplications.

A loop of 50000 iterations is performed. Each iteration consists of
two re-assignments: z = xz and x = zG via scalar multiplications.
In the second assignment, the byte representation of the input point
z is used as the byte representation of an scalar. Similarly, the
output x is the byte representation of the point, which is will used
as as the byte representation of the scalar.

The purpose of the large number of iterations is to catch a bug that
has probability larger than 1/100000 of arising on pseudorandom
inputs. The iterations do nothing to find rarer bugs (such as those
that an adversary can invoke), or silent bugs (side channel leaks).

The sixth and seventh lines are equal to each other. As explained
below, the equality of these lines represents the fact the Alice and
Bob can compute the same shared DH secret. The purpose of these
lines is not to catch any more bugs, but rather a sanity check that
Diffie--Hellman is likely to work.

Alice initializes her DH private key to x, as already computed on
the fourth line of the test vectors (which was the result of 100000
iterations). She then replaces this x by x^900 mod q (where q is
the prime which is the order of the order of the base point G).

Bob sets his private key y as follows. He begins with y being the
34-byte ASCII string whose initial characters are "yet another test"
(not including the quotes, of course). He then reverses the order
of bytes, considers this to be a scalar, and reassigns y to yG.
(So, the y on the left is new, the y on the right is old, they are
not the same, after the assignment.) Another reassignment is done,
as y -> yy, where the on the right side of the equation one y is
treated as a scalar, the other as a point. Finally, Bob's replaces
y by y^900 mod order(G), similarly to Alice's transformation.

The test code in C.2 does not compute x^900 directly. Instead it
uses 900 scalar multiplication by x, to achieve multiplication by
x^900. The same is done for y^900.

Both lines are xyG. The first can be computed as y(xG), and the
second as x(yG). The equality of the two lines can be used to
self-test an implementation, even if the implementation being tested
disagrees with the test vectors above.

Appendix C. Sample code (pseudocode)

This section has sample C code that illustrates well-known elliptic
algorithms, with adaptations specific to 2y^2=x^3+x/GF(8^91+5).

   Warning: the sample code has not been fully hardened against
   side channels or any other implementation attacks; also, no
   independent party has reviewed the sample code.

   Note: The quality of the sample code is similar to pseudocode, not
   reference code, or software. It compiles and runs on my personal
   devices, but has not otherwise been tested for quality.

   Note: Non-standard C language extensions are used the sample code:
   the type __int128, available as an C language extension in the GNU
   C compiler (gcc).

   Note: Non-portable C is used (beyond the non-standard C), for
   convenience. Two's complement integer representation of integers
   is assumed. Bit-shifts negative integers are used, in a way that
   considered non-portable under strict C, even though commonly used
   elsewhere.

   Note: Manually minified C is used: to reduce line and character
   counts, and also to (arguably) aid objective code inspection by
   cramming as much code into a single screen and by not misleading
   reviewers with long comments or variable names.

   Note: Automated tools, such as indent (used as in "gcc -E pseudo.c
   | indent"), can partially revert the C sample code spacing to a
   more conventional style, though other aspects of minification are
   not so easy to remove.

   Note: The minification is not total. It tries to organize the
   code into meaningful units, such as placing single short functions
   on one line or placing all variable declarations on the same line
   with the function parameters. Python-like indentation is kept.
   (Per Lisp styling, the code clumps closing delimiters (that mainly
   serve the compilers.))

   Note: Long sequence expressions, using the C comma operator, in
   place of multiple expression statements, which would be more
   conventional and terminated by semicolons, save some braces in
   control statements, such as "for" loops and "if" conditionals, and
   enable extra initializations in declarations.

C.1. Scalar multiplication of 34-byte strings

  The sample code for scalar multiplication provides an interface for
  scalar multiplication. A function "mulch" takes as input 3 pointer
  to unsigned character strings. The first input is the location
  of the result, the second is the multiplier, and the third is the
  base point.

  Note: The input ordering follows the convention of C assignment
  expressions z=x*y.

  Note: The function name "mulch" is short for multiply character
  strings.

 Mulch returns a Boolean value, indicating success or failure.
 Failure is returned only if validation is requested, and the base
 point is invalid.

 Requesting validation is done implicitly, by comparison of pointers.
 Validation is requested unless the base point is the known valid
 base point G, or if the scalar multiple (2nd input) and the output
 (1st input) pointers are equal, meaning that the scalar multiple
 will be overwritten.

  Note: The motivation here for implicitly requesting validation is
  that if the scalar multiple is really ephemeral, the caller should
  be willing, and eager, to overwrite it as soon as possible, in
  order to achieve forward secrecy. In this case, the need for
  input validation is usually negligible.

 The sample code is to be considered as a single file "pseudo.c".

 The file pseudo.c has two sections. The first section implements
 arithmetic for the field GF(8^91+5). The second section implements
 Montgomery's ladder for curve 2y^2=x^3+x. The two sections are not
 entirely independent. In particular, the field arithmetic section
 is not general-purpose, and could produce errors if used for
 different elliptic curve algorithms, such as Edwards coordinates.

  Note: The scalar multiplication sample code pseudo.c file is
  included into 3 other sample (using a the C preprocessor directive
  #include "pseudo.c").

  Note: Compiler optimizations make a large difference when used on
  the field arithmetic (for versions of the sample code where the
  field and curve arithmetic are in separate source files). This
  suggests that field arithmetic efficiency has room for further
  improvement by hand assembly. (The curve arithmetic might be
  improved by re-writing the source code.) In case, the sample code
  should not be considered to fully optimized.

   Note: Montgomery's ladder might not be the fastest scalar
   multiplication algorithm for 2y^2=x^3+x/GF(8^91+5). Experimental
   C implementations using Bernstein's 2-D ladder algorithm (see
   [B1]) seem about ~10% faster . The experimental code somewhat
   more complicated, and thus more likely to vulnerable to side
   channels or overflows. Even more aggressive C code seems about
   ~20% faster, using Edwards coordinates,
   Hisil--Carter--Dawson--Wong, and Gallant--Lambert--Vanstone, and
   pre-computed windows (see [B4]). Again, these faster methods are
   more complicated, and may be more vulnerable implementation
   attacks. The 10% and 20% gains may be lost upon more thorough
   hardening against implementation attacks, or upon more thorough
   hand-assembly optimizations.

C.1.1. Field arithmetic for GF(8^91+5)

  The field arithmetic sample code, is the first part of the file
  pseudo.c. It implements the field operations used in the Montgomery
  ladder algorithm for elliptic curve 2y^2=x^3+x. For example, point
  decompression is not used in Montgomery ladders, so the square root
  operation is not included the sample code. (The Legendre symbol
  computation is included for validation, and is quite similar to the
  square root operation.)

```
<CODE BEGINS>
#define RZ return z
#define F4j i j=5;for(;j--;)
#define FIX(j,r,k) q=z[j]>>r, z[j]-=q<<r, z[(j+1)%5]+=q*k
#define CMP(a,b) ((a>b)-(a<b))
#define XY(j,k) x[j]*(ii)y[k]
#define R(j,k) (zz[j]>>55*k&((k<2)*M-1))
#define MUL(m,E)\
  zz[0]= m(0,0)E(1,4)E(2,3)E(3,2)E(4,1),\
  zz[1]= m(0,1)m(1,0)E(2,4)E(3,3)E(4,2),\
  zz[2]= m(0,2)m(1,1)m(2,0)E(3,4)E(4,3),\
  zz[3]= m(0,3)m(1,2)m(2,1)m(3,0)E(4,4),\
  zz[4]= m(0,4)m(1,3)m(2,2)m(3,1)m(4,0);\
  z[0]=R(0,0)-R(4,1)*20-R(3,2)*20, z[1]=R(1,0)+R(0,1)-R(4,2)*20,\
  z[2]=R(2,0)+R(1,1)+R(0,2),       z[3]=R(3,0)+R(2,1)+R(1,2),\
  z[4]=R(4,0)+R(3,1)+R(2,2);       z[1]+=z[0]>>55; z[0]&=M-1;
typedef long long i;typedef i*f,F[5];typedef __int128 ii,FF[5];
i M=((i)1)<<55;F O={0},I={1};
f fix(f z){i j=0,q;
  for(;j<5*2;j++) FIX(j%5,(j%5<4?55:53),(j%5<4?1:-5));
  z[0]+=(q=z[0]<0)*5; z[4]+=q<<53; RZ;}
i cmp(f x,f y){i z=(fix(x),fix(y),0); F4j z+=!z*CMP(x[j],y[j]); RZ;}
f add(f z,f x,f y){F4j z[j]=x[j]+y[j]; RZ;}
f sub(f z,f x,f y){F4j z[j]=x[j]-y[j]; RZ;}
f mal(f z,i s,f y){F4j z[j]=y[j]*s; RZ;}
f mul(f z,f x,f y){FF zz; MUL(+XY,-20*XY); {F4j zz[j]=0;} RZ;}
f squ(f z,f x){mul(z,x,x); RZ;}
i inv(f z){F t;i j=272; for(mul(z,z,squ(t,z));j--;) squ(t,t);
  return mul(z,t,z), (sub(t,t,t)), cmp(O,z);}
i leg(f y){F t;i j=270; for(squ(t,squ(y,y));j--;) squ(t,t);
  return j=cmp(I,mul(y,y,t)), (sub(y,y,y),sub(t,t,t)), (2-j)%3-1;}
<CODE ENDS>
```

Field elements are stored as five-element of arrays of limbs. Each
limb is an integer, possibly negative, with array z representing
integer

   z[0] + z[1]*2^55 + z[2]*2^110 + z[3]*2^165 + z[4]*2^220

In other words, the radix (base) is 2^55. Say that z has m-bit
limbs if each |z[i]| < 2^m.

The field arithmetic function input order follows the C assignment
order, as input z=x*y, so usually the first input is the location
for the result of the operation. The return value is usually just a
pointer to the result's location, the first input, indicated by the
preprocessor macro RZ.   The functions, inv, cmp, and leg, also
return an integer, which is not a field element, but usually a
Boolean (or for function leg, a value in {-1,0,1}.)

The utility functions are fix and cmp. They are meant to take
inputs with 58-bit limbs, and produce an output with 55-bit
non-negative limbs, with the highest limb, a 53-bit value. The
purpose of fix is to provide a single array representation of each
field element. The function cmp fixes both its inputs, and then
returns a signed comparison indicator (in {-1,0,1}).

The multiplicative functions are mul, squ, inv and leg. They are
meant to take inputs with 58-bit limbs, and produce either an output
with 57-bit limbs, or a small integer output. They try to do this
as follows:

   1. Some of the input limbs are multiplied by 20, then multiplied
      in pairs to 128-bit limbs, and then summed in groups of five
      (with at least one of the pairs having both elements not
      multiplied by 20). The multiplications by 20 should not cause
      64-bit overflow $20*2^{58} < 32*2^{58}=2^{63}$, while the sums of
      128-bit numbers should not cause overflow, because
      $(1+4*20)*2^{58}*2^{58} = 81*2^{116} < 2^7*2^{116} = 2^{123}$.

   2. The five 128-bit limbs are partially reduced to five 57-bit
      limbs.   Each the five smaller limbs is obtained by summing two
      55-bit limbs, extracted from sections of the 128-bit limbs, and
      then summing one or two much smaller values summing to less
      than a 55-bit limb. So, the final limbs in the multiplication
      are a sum of at most three 55-bit sub-limbs, making each final
      limb at most a 57-bit limb.

The additive functions are add, sub and mal. They are meant to take
inputs with 57-bit limbs, and product an output with 58-bit limbs.

The utility and multiplicative function can be used repeatedly,
because they do not lengthen the limbs.

The additive functions potentially increase the limb length, because
they do not perform any reduction on the output. The additive
functions should not be applied repeatedly. For example, if the
output of additive additive function is fed directly as the input to
an additive function, then the final output might have 59-bit limbs.
In this case, if 2nd output might not be evaluated corrected if
given as input to one of the multiplicative functions, an error due
to overflow of 64-bit arithmetic might occur.

The lack of reduction in the additive functions trades generality
for efficiency. The elliptic curve arithmetic code aims to never
send the output of an additive function directly into the input of
another additive function.

  Note: Zeroizing temporary field values is attempted by subtracting
  them from themselves. Some compilers might remove these
  zeroization steps.

  Note: The defined types f and F are essentially the equivalent.
  The main difference is that type F is an array, so it can be used
  to allocate new memory (on the stack) for a field value.

C.1.2. Montgomery ladder scalar multiplication

The second part of the file "pseudo.c" implements Montgomery's
well-known ladder algorithm for elliptic curve scalar point
multiplication, as it applies to the curve $2y^2=x^3+x$.

The sample code, as part of the same file, is a continuation of the
sample code for field arithmetic. All previous definitions are
assumed.

```
<CODE BEGINS>
#define X z[0]
#define Z z[1]
enum {B=34}; typedef void _;typedef volatile unsigned char *c,C[B];
typedef F*e,E[2];typedef E*v,V[2];
f feed(f x,c z){i j=((mal(x,0,x)),B);
  for(;j--;) x[j/7]+=((i)z[j])<<((8*j)%55); return fix(x);}
c bite(c z,f x){F t;i j=((fix(mal(x,cmp(mal(t,-1,x),x),x))), B),k=5;
  for(;j--;) z[j]=x[j/7]>>((8*j)%55); {(sub(t,t,t));}
  for(;--k;) z[7*k-1]+=x[k]<<(8-k); {(sub(x,x,x));} RZ;}
i lift(e z,f x,i t){F y;return mal(X,1,x),mal(Z,1,I),t||
  -1==leg(add(y,x,mul(y,x,squ(y,x)))));}
i drop(f x,e z){return inv(Z)&&mul(x,X,Z)&&(sub(X,X,X)&&sub(Z,Z,Z));}
_ let(e z,e y){i j=2;for(;j--;)mal(z[j],1,y[j]);}
_ smv(v z,v y){i j=4;for(;j--;)add(((e)z)[j],((e)z)[j],((e)y)[j]);}
v mav(v z,i a){i j=4;for(;j--;)mal(((e)z)[j],a,((e)z)[j]);RZ;}
_ due(e z){F a,b,c,d;
  mul(X,squ(a,add(a,X,Z)),mal(d,2,squ(b,sub(b,X,Z))));
  mul(Z,add(c,a,b),sub(d,a,b));}
_ ade(e z,e u,f w){F a,b,c,d;f ad=a,bc=b;
  mul(ad,add(a,u[0],u[1]),sub(d,X,Z)),
  mul(bc,sub(b,u[0],u[1]),add(c,X,Z));
  squ(X,add(X,ad,bc)),mul(Z,w,squ(Z,sub(Z,ad,bc)));}
_ duv(v a,e z){ade(a[1],a[0],z[0]);due(a[0]);}
v adv(v z,i b){V t;
  let(t[0],z[1]),let(t[1],z[0]);smv(mav(z,!b),mav(t,b));mav(t,0);RZ;}
 e mule(e z,c d){V a;E o={{1}};i
b=0,c,n=(let(a[0],o),let(a[1],z),8*B);
  for(;n--;) c=1&d[n/8]>>n%8,duv(adv(a,c!=b),z),b=c;
  let(z,*adv(a,b)); (due(*mav(a,0))); RZ;}
C G={23,1};
i mulch(c db,c d,c b){F x;E p; return
  lift(p,feed(x,b),(db==d||b==G))&&drop(x,mule(p,d))&&bite(db,x);}
<CODE ENDS>
```

This part of the sample code represents points and scalar
multipliers as character strings of 34 bytes.

  Note: Types c and C are used for these 34-byte encodings.
  Following the previous pattern for f and F, type C is an array,
  used for allocating new memory (on the stack) for these arrays.

The conversion functions feed and bite convert
between a 34-byte string and a field value (recall, stored as five
element array, base 2^55).

The conversion functions lift and drop convert between field
elements and the projective line point, so that x <-> (X:1).   The
function lift can also test if x is the x-coordinate of the a point
(x,y) on the curve 2y^2=x^3+x.

  Note: Projective line points are stored in defined types e and E
  (for extended field element).

  Note: The Montgomery ladder can implemented by working with a
  pair of extended field elements.

The raw scalar multiplication function "mule" takes a projective
point (with defined type e), multiplies it by a scalar (encoded as
byte string with defined type c), and then replaces the projective
point by the multiple.

The main loop of mule is written a double-and-always-add, acting on
pair projective line points. Basically it acts on the x-coordinates
of the points nB and (n+1)B, for n changing.

Because the Montgomery ladder algorithm is being used, the "adv"
called by mule function does nothing but swap the two values. With
an appropriate isogeny, this can be viewed as addition operation.

The function "duv" called by mule, does the hard work of finding
(2n)B and (2n+1)B from nB and (n+1)B. It does so, using doubling in
the function "due" and differential addition, in the function "ade".

The functions "due" and "ade" are non-trivial, and use field
arithmetic. They are fairly specific to 2y^2=x^3+x. They try to
avoid repeated application of additive field operations.

The function smv, mav and let are more utilitarian. They are used
for initialization, swapping, and zeroization.

C.1.3. Bernstein's 2-dimensional Montgomery ladder

Bernstein's 2-dimensional ladder is a variant of Montgomery's ladder
that computes aP+bQ, for any two points P and Q, more quickly than
computing aP and bQ separately.

Curve 2y^2=x^3+x has an efficient endomorphism, which allows a point
Q = [i+1]P to compute efficiently. Gallant, Lambert and Vanstone
introduced a method (now called the GLV method), to compute dP more
efficiently, given such an efficient endomorphism. They write d = a
+ eb where e is the integer multiplier corresponding to the
efficient endomorphism, and a and b are integers smaller than d.
(For example, 17 bytes each instead of 34 bytes.)

The GLV method can be combined with Bernstein's 2D ladder algorithm
to be applied to compute dP = (a+be)P = aP + beP = aP + bQ, where
e=i+1.

This algorithm is not implemented by any pseudocode in the version
the draft. (Previous versions had it.)

See [B1] for further explanation and example pseudocode.

I have estimate a ~10% speedup of this method compared to the plain
Montgomery ladder. However, the code is more complicated, and
potentially more vulnerable to implementation-based attacks.

C.1.4. GLV in Edwards coordinates (Hisil--Carter--Dawson--Wong)

It is also possible to convert to Edwards coordinates, and then use
the Hisil--Wong--Carter--Dawson (HWCD) elliptic curve arithmetic.

The HWCD arithmetic can be combined with the GLV techniques to
obtain a scalar multiplication potentially more efficient than
Bernstein's 2-dimensional Montgomery. The downside is that it may
require key-dependent array look-ups, which can be a security risk.

My implementation of this (see [B4]) gives a ~20% speed-up over my
implementation of the Montgomery ladder. Of course, this speed-up
may disappear upon further optimization (e.g. assembly), or further
security hardening (safe table lookup code).

C.2. Sample code for test vectors

The following sample code describes the contents of a file "tv.c",
with the purpose of generating the test vectors in Appendix B.

```
<CODE BEGINS>
//gcc tv.c -o tv -O3 -flto -finline-limit=200;strip tv;time ./tv
#include <stdio.h>
#include "pseudo.c"
#define M mulch
void hx(c x){i j=B;for(;j--;)printf("%02x",x[j]);printf("\n");}
int main (void){i n=1e5,j=n/2,wait=/*your mileage may vary*/7000;
  C x="TEST 2y^2=x^3+x/GF(8^91+5)",y="yet another test",z;
  M(z,x,G); hx(x),hx(G),hx(z);
  fprintf(stderr,"%30s(wait=~%ds, ymmv)","",j/wait);
  for(;j--;)if(fprintf(stderr,"\r%7d\r",j),!(M(z,x,z)&&M(x,z,G)))
   j=0*printf("Mulch fail rate ~%f :(\n",(2*j)/n);//else//debug
  fprintf(stderr,"\r%30s                     \r",""),hx(x),hx(z);
  M(y,y,G);M(y,y,y);
  for(M(z,G,G),j=900;j--;)M(z,x,z);for(j=900;j--;)M(z,y,z);hx(z);
  for(M(z,G,G),j=900;j--;)M(z,y,z);for(j=900;j--;)M(z,x,z);hx(z);}
<CODE ENDS>
```

It includes the previously defined file pseudo.c, and the standard
header file stdio.h.

The first for-loop in main aims to terminate in the event of the bug
such that the output of mulch is an invalid value, not on the curve
2y^2=x^3+x.

Of the 100,000 scalar multiplication in this for-loop, the aim is
that 50,000 include public-key validation. All 100,000 include a
field-inversion, to encode points uniquely as 34-byte strings.

The second and three for-loops aims to test the compatibility with
Diffie--Hellman, by showing the 900 applications of scalar
multipliers x and y are the same, whether x or y is applied first.

The 1st line comment suggest possible compilation commands, with
some optimization options. The run-time depends on the system, and
should be slower on older and weaker systems.

Anecdotally, on a ~3 year-old personal computer, it runs in time as
low as 5.7 seconds, but these were under totally uncontrolled
conditions (with no objective benchmarking). (Experience has shown
that on a ~10 year-old personal computer, it could be ~5 times
slower.)

C.3. Sample code for a command-line demo of Diffie--Hellman

The next sample code is intended to demonstrate ephemeral (elliptic
curve) Diffie--Hellman: (EC)DHE in TLS terminology.

The code can be considered as a file "dhe.c". It has both C and
bash code, intermixed within comments and strings. It is bilingual:
a valid bash script and valid C source code. The file "dhe.c" can
be made executable (using chmod, for example), so it can be run as a
bash script.

```
<CODE BEGINS>
#include "pseudo.c" /* dhe.c (also a bash script)
: demos ephemeral DH, also creates, clobbers files dhba dha dhb
: -- Dan Brown, BlackBerry, '20 */
#include <stdio.h>
_ get(c p,_*f){f&&fread ((_*)p,B,1,f)||mulch(p,p,G);}
_ put(c p,_*f){f&&fwrite((_*)p,B,1,f)&&fflush(f); bite(p,O);}
int main (_){C p="not validated",s="/dev/urandom" "\0"__TIME__;
  get(s,fopen((_*)s,"r")), mulch(p,s,G), put(p,stdout);
  get(p,stdin),            mulch(s,s,p), put(s,stderr);} /*'
[ dhe.c -nt dhe ]&&gcc -O2 dhe.c -o dhe&&strip dhe&&echo "$(<dhe.c)"
mkfifo dh{a,b,ba} 2>/dev/null || ([ ! -p dhba ] && :> dhba)
./dhe <dhba 2>dha | ./dhe >dhba 2>dhb &
sha256sum dha & sha256sum dhb  # these should be equal
(for f in dh{a,b,ba} ; do [ -f $f ] && \rm -f $f; done)# '*/
<CODE ENDS>
```

Run as a bash script, file "dhe.c" will check if it needs compile
its own C code, into an executable named "dhe". Then the bash
script file "dhe.c" runs the compiled executable "dhe" twice. One
run is Alice's, and the other Bob's.

Each run of "dhe" generates an ephemeral secret key, by reading the
file "/dev/urandom". Each run then writes to "stdout", the
ephemeral public key. Each run then reads the peer's ephemeral
public key from "stdin". Each run then writes to "stderr" the
shared Diffie--Hellman secret. (Public-key validation is mostly
unnecessary, because the ephemeral is only used once, so it is
skipped by using the same pointer location for the ephemeral secret
and final shared secret.)

The script "dhe.c" connects the input and output of these two using
pipes. One pipe is generated by the shell command line using the
shell operator "|". The other pipe is a pipe name "dhab", created
with "mkfifo". The script captures the shared secrets from each run
by redirecting "stderr" (as file descriptor 2), to files "dha" and
"dhb", which will be made named pipes if possible.

The scripts feeds each shared secret keys into SHA-256. This
demonstrates their equality. It also illustrates a typical way to
use Diffie--Hellman, by deriving symmetric keys using a hash
function. In multi-curve ECC, hashing a concatenation of such
shared secrets (one for each curve used), could be done instead.

C.4. Sample code for public-key validation and curve basics

The next sample code demonstrates the public-key validation issues
specific to 2y^2=x^3+x/GF(8^91+5). It also demonstrates the order
of the curve. It also demonstrates complex multiplication by i, and
the fact the 34-byte representation of points is unaffected by
multiplication by i.

The code can be considered to describe a file "pkv.c". It uses the
"mulch" function by including "pseudo.c".

```
<CODE BEGINS>
#include <stdio.h>
#include "pseudo.c"
#define M mulch // works with +/- x, so P ~ -P ~ iP ~ -iP
void hx(c x){i j=B;for(;j--;)printf("%02x",x[j]);printf("\n");}
int main (void){i j;// sanity check, PKV, twist insecurity demo
  C y="TEST 2y^2=x^3+x/GF(8^91+5)",z="zzzzzzzzzzzzzzzzzzzz",
  q = "\xa9\x38\x04\xb8\xa7\xb8\x32\xb9\x69\x85\x41\xe9\x2a"
  "\xd1\xce\x4a\x7a\x1c\xc7\x71\x1c\xc7\x71\x1c\xc7\x71\x1c"
  "\xc7\x71\x1c\xc7\x71\x1c\x07", // q=order(G)
  i = "\x36\x5a\xa5\x56\xd6\x4f\xb9\xc4\xd7\x48\x74\x76\xa0"
  "\xc4\xcb\x4e\xa5\x18\xaf\xf6\x8f\x74\x48\x4e\xce\x1e\x64"
  "\x63\xfc\x0a\x26\x0c\x1b\x04", // i^2=-1 mod q
  w5= "\xb4\x69\xf6\x72\x2a\xd0\x58\xc8\x40\xe5\xb6\x7a\xfc"
  "\x3b\xc4\xca\xeb\x65\x66\x66\x66\x66\x66\x66\x66\x66\x66"
  "\x66\x66\x66\x66\x66\x66\x66"; // w5=(2p+2-72q)/5
  for(j=0;j<=3;j++)M(z,(C){j},G),hx(z); // {0,1,2,3}G, but reject 0G
  M(z,q,G),hx(z); // reject qG; but qG=O, under hood:
  {F x;E p;lift(p,feed(x,G),1);mule(p,q);hx(bite(z,p[1]));}
  for(j=0;j<0*25;j++){F x;E p;lift(p,feed(x,(C){j,1}),1);mule(p,q);
  printf("%3d ",j),hx(bite(z,p[1]));}// see j=23 for choice of G
  for(j=3;j--;)q[0]-=1,M(z,q,G),hx(z);// (q-{1,2,3})G ~ {1,2,3}G
  M(z,i,G),hx(z); i[0]+=1,M(z,i,G),M(z,i,z),hx(z);// iG~G,(i+1)^2G~2G
  M(w5,w5,(C){5}),hx(w5);// twist, ord(w5)=5, M(z,z,p) skipped PKV(p)
  M(G,(C){1},w5),hx(G);// reject w5 (G unch.); but w5 leaks z mod 5:
  for(j=10;j--;)M(z,y,G),z[0]+=j,M(z,z,w5),hx(z);}
<CODE ENDS>
```

The sample code demonstrates the need for public-key validation
even when using the Montgomery ladder for scalar multiplication. It
does this by finding points of low order on the twist of the curve.
This invalid points can leak bits of the secret multiplier. This
is because the curve 2y^2=x^3+x/GF(8^91+5) is not fully "twist
secure". (Its twist security is typical of that of a random curve.)

Appendix D. Minimizing trapdoors and backdoors

The main advantage of curve 2y^2=x^3+x/GF(8^91+5) over almost all
other elliptic curves is its Kolmogorov complexity is almost minimal
among curves of sufficient resistance to the Pollard rho attack on
the discrete logarithm problem.

See [AB] and [B1] for some details.

D.1. Decimal exponential complexity

The curve can be described with 21 characters:

```
  2  y  ^  2  =  x  ^  3  +  x  /  G  F  (  8  ^  9  1  +  5  )
  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21
```

Those familiar with ECC will recognize that these 21 characters
suffice to specify the curve up to the level of detail needed to
describe the cost of the Pollard rho algorithm, as well as many
other security properties (especially resistance to other known
attacks on the discrete logarithm problem, such as Pohlig--Hellman
and Menezes--Okamoto--Vanstone).

  Note: The letters GF mean Galois Field, and are quite traditional
  mathematics, and every elliptic curve in cryptographic needs to
  use some notation for the finite field.

We may therefore describe the curve's Kolmogorov complexity as 21
characters.

  Note: The idea of low Kolmogorov complexity is hard to specify
  exactly. Nonetheless, a claim of nearly minimal Kolmogorov
  complexity is quite falsifiable. The falsifier need merely
  specify several other (secure) elliptic curves using 21 or fewer
  characters. (But if the other curves use a different
  specification language, then a fair comparison should re-specify
  2y^2=x^3+x/GF(8^91+5) in this specification language.)

D.1.1. A shorter isomorphic curve

The curve is isomorphic to a curve specifiable in 20 characters:

   y^2=x^3-x/GF(8^91+5)

Generally, isomorphic curves have essentially equivalently hard
discrete logarithm problems, so one could argue that curve
2y^2=x^3+x/GF(8^91+5) could be rated as having Kolmogorov complexity
at most 20 characters.

Isomorphic curves, however, may differ slightly in security, due to
issues of efficiency, and implementability. The 21-character
specification uses an equation in Montgomery form, which creates an
incentive to use the Montgomery ladder algorithm, which is both safe
and efficient [Bernstein?].

D.1.2. Other short curves

Allowing for non-prime fields, then the binary-field curve known as
sect283k1 has a 22-character description:

   y^2+xy=x^3+1/GF(2^283)

This curve was formerly one of the fifteen curves recommended by
NIST. Today, a binary curve is curve is considered risky, due to
advances in elliptic curve discrete logarithm problem over extension
fields, such as 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 (but sect283k1's short description should
help mitigate that risk).

This has a longer overall specification than curve
2y^2=x^3+x/GF(8^91+5), but the field part is shorter field
specification. Perhaps an isomorphic curve can be found (one with
three terms), so that total length is 21 or fewer characters.

A non-prime field tends to be slower in software. A non-prime field
is therefore perhaps riskier due to some recent research on
attacking non-prime field discrete logarithms and elliptic curves,

D.1.3. Converting DEC characters to bits

The units of characters as measuring Kolmogorov complexity is not
calibrated as bits of information. Doing so formally would be very
difficult, but the following approach might be reasonable.

Set the criteria for the elliptic curve. For example, e.g. prime
field, size, resistance (of say 2^128 bit operations) to known
attacks on the discrete logarithm problem (Pollard rho, MOV, etc.).
Then list all the possible ECC curve specification with Kolmogorov
complexity of 21 characters or less. Take the base two logarithm of
this number. This is then a calibrated estimate of the number of
bits needed to specify the curve. It should be viewed as a lower
bound, in case some curves were missed.

D.1.4. Common acceptance of decimal exponential notation

The decimal exponentiation notation used in to measure decimal
exponential complexity is quite commonly accepted, almost standard,
in mathematical computer programming.

For example, as evidence of this common acceptance, here is a
slightly edited session of the program "bc" (versions of which are
standardized in POSIX).

```
 <CODE BEGINS>
 $ BC_LINE_LENGTH=70 bc
 bc 1.06.95
 Copyright ... Free Software Foundation, Inc.
 ...
   p=8^91+5 ; p; obase=16; p

15177100720513508366558296147058741458143803430094840009779784451085\
  189728165691397
20000000000000000000000000000000000000000000000000000000000000000005
   define v(b,e,m){
     auto a; for(a=1;e>0;e/=2){
     if(e%2==1) {a=(a*b)%m;}
     b=(b^2)%m;}
   return(a);}
   v(571,p-1,p)
 1
   x = (1*256) + (23*1)
   v(2*(x^3+x),(p-1)/2,p)
 1
   y = (((p+1)/2)*v(2*(x^3+x),(p+3)/8,p))%p
   (2*y^2)%p == (x^3+x)%p
 1
   (2*y^2 -(x^3+x))%(8^91+5)
 0
 <CODE ENDS>
```

Note: Input lines have been indented at least two extra spaces,
and can be pasted into a "bc" session. (Pasting the output lines
causes a few spurious results.)

The sample code demonstrates that "bc" directly accepts the
notations "8^91+5" and "x^3+x": parts parts of the curve
specification "2y^2=x^3+x/GF(8^91+5)", which goes to show how much
of the notation used in this specification is commonly accepted.

Note: Defined function "v" implements modular exponentiation,
with returning v(b,e,m) returning (b^e mod m). Then, "v" is used
to show that p=8^91+5 is a Fermat pseudoprime to base 571
(evidence that p is prime). The value x defined is the
x-coordinate of the recommend base point G. Then, another
computation with "v" shows that 2(x^3+x) has Legendre symbol 1,
which implies (assuming p is prime) that there exists y with
2y^2=x^3+x, namely y = (1/2)sqrt(2(x^3+x)). The value of y is
computed, again using "v" (but also a little luck). The curve
equation is then tested twice with two different expressions,
somewhat similar to the mathematical curve specification
2y^2=x^3+x/GF(8^91+5).

D.2. General benefits of low Kolmogorov complexity to ECC

The intuitive benefit of low Kolmogorov complexity to cryptography
is well known, but very informal and heuristic. The general benefit
is believed to be a form of subversion-resistance, where the
attacker is the designer of the cryptography. The idea is that low
Kolmogorov complexity thwarts that type of subversion which causes
high Kolmogorov complexity. Exhaustive searches for weaknesses
would seem to require relatively high Kolmogorov complexity,
compared to lowest complexity non-weak examples in the search.

Often, fixed numbers in cryptographic algorithms with low Kolmogorov
complexity are called "nothing-up-my-sleeve" numbers. (Bernstein et
al. uses terms in "rigid", for a very similar idea, but with an
emphasis on efficiency instead of compressibility.)

For elliptic curves, the informal benefit may be stated as the
following gains.

- Low Kolmogorov complexity defends against insertion of a keyed
  trapdoor, meaning the curve can broken using a secret trapdoor,
  by an algorithm (eventually discovered by the public at large).
  For example, the Dual EC DRBG is known to capable of having such
  a trapdoor. Such a trapdoor would information-theoretically
  imply an amount of information, comparable the size of the
  secret, to be embedded in the curve specification. If the
  calibrated estimate for the number of bits is sufficiently
  accurate, then such a key cannot be large.

- Low Kolmogorov complexity defends against a secret attack
  (presumably difficult to discover), which affects a subset of
  curves such that (a) whether or not a specific curve is affected
  is a somewhat pseudorandom function of its natural
  specification, and (b) the probably of a curve being affected
  (when drawn uniformly from some sensible of curve
  specification), is low. For an example of real-world attacks
  meeting the conditions (a) and (b) consider the MOV attack.
  Exhaustively finding curve meeting these two conditions is
  likely to result in relatively high Kolmogorov complexity.
  The supply of low Kolmogorov complexity curves is so low that
  the probability of any falling into the weak class is low.

- Even more hypothetically, there may yet exist undisclosed
  classes of weak curves, or attacks, which $2y^2=x^3+x/GF(8^{91}+5)$
  avoids somehow. A real-world example is prime-order, or low
  cofactor curves, which are rare among all curves, but which
  better resist the Pohlig--Hellman attack. If this happens, then
  it should be considered a fluke.

Low Kolmogorov complexity is not a panacea. The worst failure would
be attacks that increase in strength as Kolmogorov complexity gets
lower. Two examples illustrate this strongly.

D.2.1. Precedents of low Kolmogorov complexity in ECC

The standard curves sect283k1 and Curve25519 can be argued as
mitigating the risk of manipulated designed-in weakness, by virtue
of the low Kolmogorov complexity.

Furthermore, the Brainpool curves also make use of low Kolmogorov
complexity, except that a hash function is used in the computation
of the curve coefficients. Arguably, since a hash function is very
useful in cryptography, for key derivation and message digesting,
the complexity of the hash function should be discounted from the
overall Kolmogorov complexity of the Brainpool curve coefficients.

D.3. **Risks of low Kolmogorov complexity**

   Low Kolmogorov complexity is not a panacea for cryptography.

   Indeed, it may even add its own risks, if some weakness are
   positively correlated with low Kolmogorov complexity, making some
   attacks stronger.

   In other words, choosing low Kolmogorov complexity might just
   accidentally weaken the cryptography. Or worse, if attackers find
   and hold secret such weaknesses, then attackers can intentionally
   include the weakness, by using low Kolmogorov serving as a cover,
   thereby subverting the algorithm.

   Evidence of positive correlations between curve weakness and low
   Kolmogorov complexity might help assess this risk.

   In general cryptography (not ECC), the shortest cryptography
   algorithms may be the least secure, such as the identity function as
   an encryption function.

   Within ECC, however, some minimum threshold of complexity must be
   met for interoperability. But curve size is positively correlated
   with security (via Pollard rho) and negatively correlated with
   complexity (at least for fields, larger fields needs larger
   specifications). Therefore, there is a somewhat negative
   correlation between Pollard rho security of ECC and Kolmogorov
   complexity of the field size.

   Beyond field size in ECC, there is some negative correlations in the
   curve equation.

   Singular cubics have equations that look very similar to those
   commonly used elliptic curves. For smooth singular curves
   (irreducible cubics) a group can be defined, using more or less the
   same arithmetic as for a elliptic curve. For example
   $y^2=x^3/GF(8^{91}+5)$ is such a cubic. The resulting group has an easy
   discrete logarithm problem, because it can be mapped to the field.

   Supersingular elliptic curves can also be specified with low
   Kolmogorov complexity, and these are vulnerable to MOV attack,
   another negative correlation.

   Combining the above, a low Kolmogorov complexity elliptic curve,
   $y^2=x^3+1/GF(2^{127}-1)$, with 21-character decimal exponential
   complexity, suffers from three well-known attacks:

     1. The MOV (Menezes--Okamato--Vanstone) attack.

2. The Pohlig--Hellman attack (since it has 2^127 points).

3. The Pollard rho attack (taking 2^63 steps, instead of the 2^126 of exhaustive).

Had all three attacks been unknown, an implementer seeking low Kolmogorov complexity, might have been drawn to curve y^2=x^3+1/GF(2^127-1). (This document's curve 2y^2=x^3+x/GF(8^91+5) uses 1 more character and is much slower since, the field size has twice as many bits.)

Had an attacker known one of three attacks, the attacker could found y^2=x^3+1/GF(2^127-1), proposed it, touted its low Kolmogorov complexity, and maybe successfully subverted the system security.

Note: The curve y^2=x^3+1/GF(2^127-1) not only has low decimal exponential complexity, it also has high efficiency: fast field arithmetic and fairly fast curve arithmetic (for its bit lengths). So high efficiency can also be positively correlated with weakness.

It can be argued, that pseudorandomized curves, such as NIST P-256 and Brainpool curves, are an effective way mitigate such attacks positively correlated with low complexity. More precisely, strong pseudorandomization somewhat mitigates the attacker's subversion ability, by reducing an easy look up of the weakest curve to an exhaustive search by trial and error, intuitively implying a probable high Kolmogorov complexity (proportional the rarity of the weakness).

It can be further argued that all major known weak classes of curves in ECC are positively correlated with low complexity, in that the weakest curves have very low complexity. No major known weak classes of curves imply an increase in Kolmogorov complexity, except perhaps Teske's class of curves.

In defense of low complexity, it can be argued that the strongest way to resist secret attacks is to find the attacks.

For these reasons, this specification suggests to use curve 2y^2=x^3+x/GF(8^91+5) in multi-curve elliptic curve cryptography, in combination with at least one pseudo-randomized curve.

D.4. Alternative measures of Kolmogorov complexity

Decimal exponential complexity arguably favors decimal and the exponentiation operators, rather than the arbitrary notion of compressibility.

Allowing more arbitrary compression schemes introduces another
possible level of complexity, the compression scheme itself,
somewhat defeating the purpose of nothing-up-sleeve number. An
attacker might be able to choose a compression scheme among
many that somehow favors a weak curve.

Despite this potential extra complexity, one can still seek a
measure more objective than decimal complexity. To this end, in
[B3], I adapted the Godel's approach for general recursive
functions, which breaks down all computation into succession,
composition, repetition, and minimization.

The adaption is a miniature programming language called Roll to
describe number-related functions, including constant functions. A
Roll program for the constant function that always return 8^91+5 is:

```
<CODE BEGINS>
8^91+5 subs 8^91+1 in +4
8^91+1 subs 2^273 in +1
2^273 subs 273 in 2^
273 subs 17 in *16+1
17  subs  1 in *16+1
*16+1 roll +16 up 1
+16 subs +8 in +8
+8  subs +4 in +4
+4  subs +2 in +2
2^ roll *2 up  1
1 subs  in +2
*2 roll +2 up  0
+2 subs +1 in +1
0 subs   in +1
<CODE ENDS>
```

A Roll program has complexity measured in its length in number of
words (space-separated substrings). This program has 68 words.
Constants (e.g. field sizes) can be compared using roll complexity,
the shortest known length of their implementations in Roll.

In [B3], several other ECC field sizes are given programs. The only
prime field size implemented with 68 or fewer words was 2^521-1.
(The non-prime field size (2^127-1)^2 has 58-word "roll" program.)
Further programming effort might produce shorter programs.

   Note: Roll programs have a syntax implying some redundancy.
   Further work may yet establish a reasonable normalization for roll
   programs, resulting in a more calibrated complexity measure in
   bits, making the units closed to a universal kind of Kolmogorov
   complexity.

[Appendix E](#). **Primality proofs and certificates**

  Recent work of Albrecht and others [[AMPS](#)] has shown the combination
  of an adversarially chosen prime, and users using improper
  probabilistic primality tests can make user vulnerable to an attack.

  The adversarial primes in their attack are typically the result of
  an exhaustive search. These bad primes would therefore typically
  contain an amount of information corresponding to the length of
  their search, putting a predictable lower bound on their Kolmogorov
  complexity.

  The two primes involved for 2y^2=x^3+x/GF(8^91+5) should perhaps
  already resist [[AMPS](#)] because of the following compact
  representation of these primes:

    p = 8^91+5
    q = #(2y^2=x^3+x/GF(8^91+5))/72

  This attack [[AMPS](#)] can also be resisted by:

   - properly implementing probabilistic primality test, or
   - implementing provable primality tests.

  Provable primality tests can be very slow, but can be separated into
  two steps:

  -- a slow certificate generation, and

  -- a fast certificate verification.

  The certificate is a set of data, representing an intermediate step
  in the provable primality test, after which the completion of the
  test is quite efficient.

  Pratt primality certificate generation for any prime p, involves
  factorizing p-1, which can be very slow, and then recursively
  generating a Pratt primality certificate for each prime factor of
  p-1. Essentially, each prime has a unique Pratt primality
  certificate.

  Pratt primality certificate verification of (p-1), involves search
  for g such that 1 = (g^(p-1) mod p) and 1 < (g^((p-1)/q) mod p) for
  each q dividing p-1, and then recursively verifying each Pratt
  primality certificate for each prime factor q of p-1.

In this document, we specify a Pratt primality certificate as a
sequence of (candidate) primes each being 1 plus a product of
previous primes in the list, with certificate stating this product.

Although Pratt primality certificate verification is quite
efficient, an ECC implementation can opt to trust 8^91+5 by virtue
of verifying the certificate once, perhaps before deployment or
compile time.

E.1. Pratt certificate for the field size 8^91+5

Define 52 positive integers, a,b,c,...,z,A,...,Z as follows:

 a=2 b=1+a c=1+aa d=1+ab e=1+ac f=1+aab g=1+aaaa h=1+abb i=1+ae
 j=1+aaac k=1+abd l=1+aaf m=1+abf n=1+aacc o=1+abg p=1+al q=1+aaag
 r=1+abcc s=1+abbbb t=1+aak u=1+abbbc v=1+ack w=1+aas x=1+aabbi
 y=1+aco z=1+abu A=1+at B=1+aaaadh C=1+acu D=1+aaav E=1+aeff F=1+aA
 G=1+aB H=1+aD I=1+acx J=1+aaacej K=1+abqr L=1+aabJ M=1+aaaaaabdt
 N=1+abdpw O=1+aaaabmC P=1+aabeK Q=1+abcfgE R=1+abP S=1+aaaaaaabcM
 T=1+aIO U=1+aaaaaduGS V=1+aaaabbnuHT W=1+abffLNQR X=1+afFW
 Y=1+aaaaauX Z=1+aabzUVY.

  Note: variable concatenation is used to indicate multiplication.
  For example, f = 1+aab = 1+2*2*(1+2) = 13.

  Note: One must verify that Z=8^91+5.

  Note: The Pratt primality certificate involves finding a generator
  g for each the prime (after the initial prime). It is possible to
  list these in the certificate, which can speed up verification by
  a small factor.

   (2,b), (2,c), (3,d), (2,e), (2,f), (3,g), (2,h), (5,i), (6,j),
   (3,k), (2,l), (3,m), (2,n), (5,o), (2,p), (3,q), (6,r), (2,s),
   (2,t), (6,u), (7,v), (2,w), (2,x), (14,y),(3,z), (5,A), (3,B),
   (7,C), (3,D), (7,E), (5,F), (2,G), (2,H), (2,I), (3,J), (2,K),
   (2,L),(10,M), (5,N), (10,O),(2,P), (10,Q),(6,R), (7,S), (5,T),
   (3,U), (5,V), (2,W), (2,X), (3,Y), (7,Z).

Note: The decimal values for a,b,c,...,Y are given by: a=2, b=3,
c=5, d=7, e=11, f=13, g=17, h=19, i=23, j=41, k=43, l=53, m=79,
n=101, o=103, p=107, q=137, r=151, s=163, t=173, u=271, v=431,
w=653, x=829, y=1031, z=1627, A=2063, B=2129, C=2711, D=3449,
E=3719, F=4127, G=4259, H=6899, I=8291, J=18041, K=124123,
L=216493, M=232513, N=2934583, O=10280113, P=16384237, Q=24656971,
R=98305423, S=446424961, T=170464833767, U=115417966565804897,
V=4635260015873357770993, W=1561512307516024940642967698779,
X=167553393621084508180871720014384259,
Y=145302302948204485494451955596474@294@49.

E.2. Pratt certificate for subgroup order

Define 56 variables a,b,...,z,A,B,...,Z,!,@,#,$, with new
values:

a=2 b=1+a c=1+a2 d=1+ab e=1+ac f=1+a2b g=1+a4 h=1+ab2 i=1+ae
j=1+a2d k=1+a3c l=1+abd m=1+a2f n=1+acd o=1+a3b2 p=1+ak q=1+a5b
r=1+a2c2 s=1+am t=1+ab2d u=1+abi v=1+ap w=1+a2l x=1+abce y=1+a5e
z=1+a2t A=1+a3bc2 B=1+a7c C=1+agh D=1+a2bn E=1+a7b2 F=1+abck
G=1+a5bf H=1+aB I=1+aceg J=1+a3bc3 K=1+abA L=1+abD M=1+abcx N=1+acG
O=1+aqs P=1+aqy Q=1+abrv R=1+ad2eK S=1+a3bCL T=1+a2bewM U=1+aijsJ
V=1+auEP W=1+agIR X=1+a2bV Y=1+a2cW Z=1+ab3oHOT !=1+a3SUX @=1+abNY!
#=1+a4kzF@ $=1+a3QZ#

Note: numeral after variable names represent powers. For example,
f = 1 + a2b = 1 + 2^2 * 3 = 13.

The last variable, $, is the order of the base point, and the order
of the curve is 72$.

Note: Punctuation used for variable names !,@,#,$, would not scale
for larger primes. For larger primes, a similar format might work
by using a prefix-free set of multi-letter variable names.
E.g. replace, Z,!,@,#,$ by Za,Zb,Zc,Zd,Ze:

Acknowledgments

Author's Address

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