Elliptic curve 2y^2=x^3+x over field size 8^91+5
<draft-brown-ec-2y2-x3-x-mod-8-to-91-plus-5-06.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.

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.

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 MUST use it in a combination with other curves, such as Curve25519 or NIST P-256 (as a second layer of defense against unlikely security failures in the other curves).

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

4. Encoding points

Elliptic curve cryptography uses points for public keys and raw shared secret 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, but [x:1] can viewed as normal representation of x. (Infinity can be then represented by [1:0].)

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

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

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

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

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

4.1. Point encoding process

4.1.1. Summary

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

4.1.2. 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.

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)).

4.2. Point decoding process

4.2.1. Summary

The bytes are little-endian decoded into an integer which becomes the x-coordinate. Public-key validation is done when needed. If needed, the y-coordinate is recovered.

4.2.2. 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, as 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).
```

In some specialized applications (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.

5. Point validation

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

Note: For curve 2y^2=x^3+x over 8^91+5, the underlying attacks are slightly milder than is average for a typical elliptic curve.

To avoid leaking information about the private, the public key can be validated, which includes various checks on the public key.

5.1. When to validate

This section specifies three strategies (mandatory, simplified, and minimal) about 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 validation

As a precautionary defense-in-depth, an impelementation MAY opt to apply mandatory validation, meaning every public key (and point) is validated.

5.1.2. Simplified validation

A small, general-purpose, implementation aiming for high speed might not be able to afford the cost of mandatory validation from Section 4.1.1, because each validation costs about 10% of a scalar multiplication.

As a practical middle ground, an impelmentation MAY opt to apply simplified validation, which is the rule is that a distrusted public key is validated before being scalar multiplied by a static secret key.

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

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

Note: Simplified 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 the certification authority has a policy to valid public keys), then validation is not needed.

5.1.3. Minimal validation

An implementation MAY opt to use minimal validation, meaning doing as little point validation as possible, just enough to resist known attack against the implementation.

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.

For example, 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 -- so 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 represented point is valid, otherwise it is not valid.

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

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

The Legendre symbol can be computed using Gauss' quadratic reciprocity law, but this requires implementing modular integer arithmetic for 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, but is perhaps simpler.

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 implicitly include a check that x is valid.

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

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

OPTIONAL: In even rarer situations, it may be necessary to ensure that a point P also has a prime order q = ord(G). The costly method to check this is 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. To be completed.

6. OPTIONAL encodings

The following two encodings are not usually needed to obtain interoperability in the typical ECC applications, such as Diffie--Hellman (or digital signatures). In more specialized application, these encodings can be useful.

6.1. Encoding scalar multipliers

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

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

- An implementation needs to store scalars, because scalars are used at least twice, and must be stored between these two uses. For example, in elliptic curve Diffie--Hellman, Alice has scalar a, sends Bob point aG, keeps scalar a until she receives point B from Bob, to which she then applies aB. (If a is ephemeral, she then deletes a.) An implementation is free to use any encoding of scalar, but implementation are often constructed in modular pieces, and any pieces handling the same scalar need to be able to convey the scalar.

In Diffie--Hellman implementations based on G which has prime order q, where q is approximately p/72, the value of scalar s usually only matters mod q. So, one can reduce s, replacing it by s mod q, making s<q. Since q < 2^267 < 256^34, a value s can be represented in 34 bytes.

Basically, little-endian byte encoding of scalars is recommended, for consistency the little-endian byte encoding of field elements.

6.2. Encoding strings as points

In niche applications, it may be desired to encode an arbtirary string as a point on a curve. Example reasons to encode arbitrary 34-byte strings include:

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

- Hiding the fact that ECC is being used.

To this end, this section sketches a method to reversibly encode any 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. Encoding of variable-length strings is not, and 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.

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, and curve 2y^2=x^3+x has j-invariant 1728.

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.

If y>p-y, replace x by x-i. Solve for s = -i - 3/(i-x). Let r = sqrt(s). 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 constrast 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 ~4:1, and the image set is about one quarter of all 34-byte strings.

7. IANA considerations

This document requires no actions by IANA, yet.

8. Security considerations

No cryptographic algorithm is without risk.

Possible security risks of 2y^2=x^3+x/GF(8^91+5) are listed in this section.

Risk is difficult to estimate, especially aginst possible unknown attacks. Relative risk is slightly easier to estimate, if a comparable cryptographic system is available as a benchmark.

The security risks of 2y^2=x^3+x/GF(8^91+5) are compared to the risks of a typical generic 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 the purposes of benchmarking, and for selection among selection of a secondary or tertiary cuve in a multi-curve ECC implementation.

Note: For possible security benefits of 2y^2=x^3+x/GF(8^91+5), see Appendix A.

8.1. Field choice

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

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

  Nonetheless, many other standard curves, such as the NIST P-256 and Curve25519, also use special prime field sizes. In this regard, all these special field curves have a similar risk.

  Yet other standard curves, such as the Brainpool curves, use pseudorandom field sizes, reducing their risk to potential special-field attack.

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

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

- 8^91+5 is smaller than some other six-symbol primes: 8^95-9, 9^99+4 and 9^87+4. Therefore, arguably, 8^91+5 fails to absolutely maximize field size relative to decimal exponential complexity. In particular, curves defined over larger field size have better Pollard rho resistance (of the ECDLP).

  Nonetheless, the primes 9^99+4 and 9^87+4 are not close to a power of two, so probably suffer from about two time slower implementation than 8^91+5, which is a significant runtime cost, and perhaps also a security risk (due to implementation bugs).

  The prime 8^95-9 is, just like 8^91+5, very close to a power of two. It may thus have efficiency comparable to 8^91+5 for basic field arithmetic operations, such as addition, multiplication and reduction. The field 8^95-9 is a little larger, but is likely also implementable using five 64-bit words. Being larger, 8^95-9 has a slightly greater risk than 8^91+5 of leading to an arithmetic overflow implementation fault in field arithmetic. Field size 8^95-9 has much less simple powering algorithms for computing field inverses, Legendre symbols, and square roots: so these operations, often important for ECC, may require more code, more runtime, and perhaps more risk of implementation bugs.

- 8^91+5 is smaller than 2^283 (the field size for curve sect283k1 [SEC2], [Zigbee]), and many other five-symbol and four-symbol prime powers (such as 9^97). It provides less resistance to Pollard rho than such larger prime powers. Recent progress in the elliptic curve discrete logarithm problem, [HPST] and [Nagao], is the main reason to prefer prime fields instead of power of prime fields. A second reason to prefer a prime field (including the field of size 8^91+5) over small characteristic fields is the generally better software speed of large characteristic field. (Better software speed is mainly due to general-purpose hardware often having dedicated fast multiplication circuits: special-purpose hardware should make small characteristic field faster.)

- The Kolmogorov complexity of 8^91+5 as six symbols is only minimal for decimal exponential complexity: but it is not minimal if other types of complexity measures are allowed. For example, if we allow the exclamation mark for the factorial operation -- which is quite standard notation! -- primes larger than 8^91+5 expressible in fewer symbols. For example, 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, so would have similar inefficiency and implementability defects to primes like 9^99+4 and 9^87+4. Such inefficiencies could resaonably by the curve choice criteria, ruling out such primes.

Arguably, in traditional mathematical notation, the symbol '^' is not actually written, with operation being marked by the use of superscripts. In this view, using an ASCII character count arguably gives unduly low weight to the factorial operation as compared to exponentiation.

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

8.2. Curve choice

A first risk of using 2y^2=x^3+x is the fact that it is a special curve. It is special in having complex multiplication leading to an efficient endomorphism. Miller, in 1985, already suggested exercising prudence when considering such special curves. Gallant, Lambert and Vanstone found ways to slightly speed up Pollard rho given such an endomorphism, but no other attacks have been found.

Menezes, Okamoto and Vanstone (MOV) found an attack on special elliptic curves, of low embedding degree. The curve 2y^2=x^3+x/GF(8^91+5) is not vulnerable to their attack, but if one changes the underlying to some different primes, say p', the resulting curve 2y^2=x^3+x/GF(p') is vulnerable to their attack for about half of all primes. Because the MOV was later than Miller's caution from 1984, 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).

Many other standard curves, NIST P-256 [NIST-P-256], Curve25519, Brainpool [Brainpool], do not have any efficient complex multiplication endomorphisms. Arguably, these curves comply to Miller's advice to be prudent about special curves.

Yet other (fairly) standard curves do, such as NIST K-283 (used in [Zigbee]) and secp256k1 (see [SEC2] and [BitCoin]). Furthermore, it is not implausible [KKM] that special curves, including those efficient endomorphisms, may survive an attack on random curves.

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

A third risk is a biased ephemeral private key generation in a digital signature scheme. Most standard curves lack this risk because the field size is close to a power of two, and the cofactor is a power of two. Curve 2y^2=x^3+x over 8^91+5 has a base point order which is approximately a power of two divided by nine (because its cofactor is 72=8*9.) As such, it is more vulnerable than typical curves to biased ephemeral keys in a signature scheme.

A fourth risk is a Cheon-type attack. Few standard curves address this risk, and 2y^2=x^3+x over 8^91+5 is not much different.

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.

8.3. Encoding choices

To be completed.

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

Therefore, the final computation of an elliptic curve point, after scalar multiplication, should translate the point to a unique representation, such as the affine coordinates described in this 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) mod p, is one method to compute the inverse in constant time (if the inverse exists).

8.4. General subversion concerns

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

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

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

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

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

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

Note: For any form of ECC, finite field multiplication can be achieved most quickly by using hardware integer multiplication circuits. It is critical that those circuits have no bugs or backdoors. Furthermore, those circuits typically can only multiply integers smaller than the field elements. Larger inputs to the circuits will cause overflows. It is critical to avoid these overflows, not just to avoid interoperability failures, but also to avoid attacks where the attackers supply inputs likely induce overflows [bug attacks], [IT].

8.5. Concerns about '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 compare to more established curves.

Furthermore, it has not been submitted for a publication with peer review to any formally peer-reviewed academic cryptographer forum such as the IACR conferences like Crypto and Eurocrypt. So, it has most like been reviewed by very few eyes.

Arguably, other reviewers have little incentive to study it critically, for several reasons. The looming threat of a quantum computer has diverted many researchers towards studying post-quantum cryptography, such as supersingular isogeny Diffie--Hellman. The past disputes over NIST P-256 and Curve25519 (and several other alternatives) have perhaps tired some reviewers, many of whom reasonably wish to concentrate on deployment of ECC.

So, under the metric of aegis, as 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 over 8^91+5 at least 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.

Obvious cautions to the reader:

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

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

9. References

9.1. Normative References

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

9.2. Informative References

To be completed.

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

[AMPS] Martin R. Albrecht, Jake Massimo, Kenneth G. Paterson, and Juraj 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>

[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] (((To do:))) fill in this reference.

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

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

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

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

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

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

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

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

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

[HPST] Y. Huang, C. Petit, N. Shinohara and T. Takagi. On Generalized First Fall Degree Assumptions, IACR ePrint 2015. <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] (((To do:))) Yung and Young, generalized Gordon's ideas into Secretly-embedded trapdoor ... also known as a backdoor.

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

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

Note: Later sections (especially 4, 5, 6, 8, A, B, C, and D) cover some relatively routine ECC details about how 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) SHOULD NOT be used in single-curve ECC.

It 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. (Others go even further, dismissing prudence against special curves as myth [ref-tba].)

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 attakc" 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 situtation "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, some asymptotically polynomial-time. (To be completed.)

A.2.2.3. Rarity of published curve-specific attacks

To be completed.

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

To be completed.

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.3.1.2. 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 ouptut of SHA-1, perhaps aiming to avoid any new curve-specific weakness that would appply 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-specifc 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. Migitations 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 compelxity), 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 discrimiant, 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 signficant 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.

To be completed.

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 mechansim 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 Mongtomgery 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

200000000000000000000000000000000000000000000000000000000000000000005

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.4.3. 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, given an endomorphism

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

Note: Strictly speaking, over some fields, the curve would be supersingular, in which the term "complex mutliplication" is not used, because the curve then has quaternionic multiplication.

The endomorphism permits the Gallant--Lambert--Vanstone (GLV) scalar multiplication algorithm, which makes it relatively efficient. (The GLV method can also be combined with Bernstein's two-dimensional variant of the Montgomery ladder algorithm.)

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.4.4. 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 alebraic closure of k=GF(p).

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

A.4.4.1. 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 certifcate 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. It can be partially verified using the complex multiplication theory, and a little big integer arithmetic.

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 and swapping u and v, there are four possible sizes, p+2u+1, p-2u+1, p+2v+1, p-2v+1 (sometimes known as the 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 = 104303302790113346778702926977288705144769
v = 65558536801757875228360405858731806281506
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, specific to 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 demostrate 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 suggested special curves 1985.

A.4.4.2. Pollard rho security

The prime q is 267-bit number. The Pollard rho algorithm for discrete logarithem 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 operartions (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 endormorphism, which would apply to 2y^2=x^3+x. The speed-up here amounts to a couple of bits in the security,

A.4.4.3. 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.4.4.2. 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.4.4.3. 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.4.4.4. 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 permmitting efficient implementation.

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

A.4.4.5. 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 Bleicenbacher's faulty signature attack.)

Note: Bleichenbacher's attack exploits the signature implmentation 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.4.4.6. 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 comptuation 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 satsifying the equation exists in GF(p). More precsiely, 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 *
3452464930451677330036005252040328546941
```

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.4.4.7. 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, becasue 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 *
        1093478435746377647334249806229965925956115086976992657
```

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 logartithm 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.

To be completed.

A.4.4.8 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 receover 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.

To be completed.

Appendix B. Test vectors

The following are some test vectors.

```
000000000000000029352b31395e382846472f782b335e783d325e79322054534554
00000000000000000000000000000000000000000000000000000000000000000117
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 samem, 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, adaptations specific to 2y^2=x^3+x/GF(8^91+5).

As a warning: the sample code has not been fully hardened against side channels or any other implementation attacks; also, no independent party has reivewed 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 intializations 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 muliplier, 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 charcater 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 implemetns 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 muliplication 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 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. 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 implemenatioon attacks, or upon more thorough hand-assembly optimizations.

To be completed.

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 sigend 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 addtive 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 multipilcative 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 Montogomery 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 differntial 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)

To be completed.

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

The HCDW 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.

I have implemented this, finding ~20% speed-up over my implementation of the Montgomery ladder. However, 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 compatibilty 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 fees each shared secret keys into SHA-256. This demonstrates their equality. It also illusrates 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 invavlid 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.)

C.5. Elligator i

To be deleted (or completed).

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

This is INCOMPATIBLE with previous samples of code above, and is taken from an earlier version of experimental code.

Pseudocode (to be verified):

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

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

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

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

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 specificaion 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 isomorophic 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,

To be completed.

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 an 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.

To be completed.

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 commmon acceptance, here is a slightly edited session of the program "bc" (versions of which are standardized in POSIX).

```
   <CODE BEGINS>
   $ BC_LINE_LENGTH=71 bc
   bc 1.06.95
   Copyright ... Free Software Foundation, Inc.
   ...
     p=8^91+5 ; p; obase=16; p
  151771007205135083665582961470587414581438034300948400097797844510851\
    89728165691397
  200000000000000000000000000000000000000000000000000000000000000000005
     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 specifcation 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 benefit of low Kolmogorov complexity to cryptography is well known, but very informal. The general benefit is believed to a form of subversion-resistance, where the attacker is the designer of the cryptography.

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 prevent low Kolmogorov complexity, essentially by the low probability of the attack, and the independence of attack's success from the natural Kolmogorov complexity.

- Even more hypothetically, there may yet exist undisclosed classes of weak curves, or attacks, for which 2y^2=x^3+x/GF(8^91+5) is lucky enough to avoid. This would be a fluke. A real-world example is prime-order, or low cofactor curves, which are are among all curves, but which better resist the Pohlig--Hellman attack.

Of course, 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 Komogorov complexity in ECC

To be completed.

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

To be completed.

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 correleated 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 simlar 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.

To be completed.

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=1453023029482044854944519555964740294049.

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

Thanks to John Goyo and various other BlackBerry employees for past technical review, and to Gaelle Martin-Cocher and Takashi Suzuki for encouraging work on I-D. Thanks to David Jacobson for sending Pratt primality certificates.

Author's Address

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