Internet-Draft D. Brown
Intended status: Experimental BlackBerry
Expires: 2020-04-05 2019-10-03

Elliptic curve 2y^2=x^3+x over field size 8^91+5
<draft-brown-ec-2y2-x3-x-mod-8-to-91-plus-5-04.txt>

Abstract

In elliptic curve cryptography, 2y^2=x^3+x/GF(8^91+5) hedges a remote risk of potential weakness in other curves, if used in multi-curve Diffie--Hellman, for example. This curve features: isomorphism to Miller curves from 1985; low Kolmogorov complexity (little room for secretly embedded trapdoors of Gordon, Young--Yung, or Teske); likeness to a Bitcoin curve; 34-byte keys; prime field; 5*64-bit field arithmetic; easy reduction, inversion, Legendre symbol, and square root; Montgomery ladder or Edwards unified curve arithmetic (Hisil--Carter--Dawson--Wong); multiplication by i (Gallant--Lambert--Vanstone); and string-as-point encoding.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

This document specifies a type of elliptic curve cryptography (ECC) using the curve

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

This curve is useful as part of a multi-curve ECC system that combines a diverse set of curves for extra security.

The extra security in using multiple curves is a strongest-link, multi-layer, fail-safe, defense-in-depth against potential (but not yet known) attacks against one or more of the curves.

Note: Using multiple curves adds a nonzero cost to an ECC system. On a current personal computer, this extra cost includes up to 1 millisecond of runtime and sending an extra 34 bytes, per ECC transaction. In low-end devices, the time may be higher due to slower processors, making the cost might be unaffordable. Even in high-end devices, the benefit-to-cost comparison is quite questionable: is the little extra security (against a potential but unlikely and unknown threat) even worth the cost of extra runtime and traffic? The answer may depend on the data being protected. If the answer is deemed to be "yes", then multi-curve ECC is useful, and curve 2y^2=x^3+x/GF(8^91+5) can contribute to this security.

Comparing single curves when used in isolation, which is current ECC tradition, curve 2y^2=x^3+x/GF(8^91+5) is arguably riskier than the more well-established curves (such as NIST P-256, Curve25519, and even Brainpool).

In traditional single-curve ECC systems, the curve 2y^2=x^3+x/GF(8^91+5) SHOULD NOT be used, due to its risk begin greater than more well-established curves.

Multi-curve ECC is not noticeably more secure than ECC if all of the multiple curves are the actually the same curve. Therefore, a diversity of dissimilar curves is needed to achieve extra security, with each curve hedging against a failure in dissimilar curves.

The curve 2y^2=x^3+x/GF(8^91+5) has features marking it as dissimilar from some of well-established curves: especially lower Kolmogorov complextiy and complex multiplication by i.

1.1. Background

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

1.1.1. Notation

The symbol '^', as used in '2y^2=x^3+x' and '8^91+5' means exponentiation, also known as powering. For example, y^3=yyy, or y*y*y, if * is used for multiplication, and 8^91 = 8*8*...*8, with 91 eights in the product on the right.

> Note: This document does not use '^' the way that C (and similar programming languages) does (as a bit-wise exclusive-or).

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

```
2000000000000000000000000000000000000000000000000000000000000000000005
```

with with 67 zeros between 2 and 5.

> Note: For a lack of a better term, standard ECC terminology uses the a slight misnomer, "scalar multiplication" for the computation dP = P + ... + P for the P a point on the curve, d an integer, + the elliptic curve group addition and law, and the right hand side implying d terms. This suggests calling the integer d a scalar. This is a misnomer, because, in other areas of mathematics, scalars are used to multiply vectors, but elliptic curve scalar multiplication is not really vector multiplication, and risks the suggestion of confusing d(x,y) with (dx,dy). (That said, an elliptic curve group, like any abelian group, is a module over the ring of integers. Since a module is to a ring the analogue of a vector space to a field, the terminology is arguably justifiable.)

1.1.2. Basic features

The underlying field (for defining the curve) is a prime, p=8^91+5. It is very close to a power of two, which is sometimes known as a Crandall prime, making reduction modulo p relatively efficient.

The prime p being slightly larger (not smaller) than a power of two, means that common algorithms for computing inverses, Legendre symbols, and square roots are relatively simple (and slightly more efficient).

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 by the map

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

This 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.

> Note: Over the complex numbers, j-invariant 0 and 1728 are special, being the only two non-smooth orbifold points the moduli space of elliptic curves, which also means that the curves have extra symmetry.

The curve 2y^2=x^3+x is not supersingular (as defined over the prime p=8^91+5).

The curve has order 72q for a large prime q, meaning it has cofactor 72, so it is not vulnerable Pohlig--Hellman attack, and it not vulnerable to the Semaev--Araki--Satoh--Smart attack.

The cofactor 72 is divisible by 4 (and also 3), meaning it is isomorphic to a curve with an Edwards equation (and also to cure with a Hessian equation), which may permit yet more efficient implementation (and yet further combination with the GLV method).

The curve has a large embedding degree, so it has no efficient pairing operation. It is therefore also not vulnerable to the Menezes--Okamoto--Vanstone attack.

The best known algorithm to solve the discrete logarithm in the group are Pollard rho algorithms and its variants, (with minor enhancements due to Gallant, Lambert, Vanstone, which take advantage of the extra map for complex multiplication), which takes approximately sqrt(q) point additions to compute a discrete logarithm (with success rate 1/2).

1.1.3. Multi-curve ECC

This document does not specify how to do multi-curve elliptic curve cryptography, but some ideas are sketch without much detail.

Multi-curve Diffie--Hellman key agreement could perhaps compute 10 shared secrets (hashed) with 10 very different curves, and then XOR them together to get one secret. Presumably, as long as one of the Diffie--Hellman secrets is secure, the XOR of 10 is secure. All the Diffie--Hellman private keys (scalars) should be independent and so on.

For signatures, one might just apply multiple signatures, with different curves (and perhaps different signature algorithms).

This document does not specifically recommend which other curves should be combined with 2y^2=x^3+x/GF(8^91+5), but suggests the at least following:

- Use one or more well-established curves, such as NIST P-256 or Curve25519.

- Use one or more curves without complex multiplication, such as NIST P-256 or Curve25519.

- Use one or more pseudo-randomized curves, such as NIST P-256 or Brainpool or something else.

- Use one or more curves whose security features complement those 2y^2=x^3+x/GF(8^91+5) in any other way.

- Use at least three or more curves, since is likely two rather different curves are reasonably less risky than 2y^2=x^3+x/GF(8^91+5).

1.2. Speculative security motivation

The section explain why to use 2y^2=x^3+x/GF(8^91+5) in a set of three of more curves, rather than a set of three or more curves without the curve 2y^2=x^3+x/GF(8^91+5).

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

A lesser motivation for the curve is its special features. A very remote potential catastrophe in ECC would be attack on most curves. In this disaster scenario, perhaps only a few curves survive, saved by some special feature. Complex multiplication by i is perhaps one of those features. Surviving such a disaster would be a fluke, but diversity is perhaps the best possible hedge against this event. More probable than such a disaster would be an attack that exploits precisely the special features of curve 2y^2=x^3+x/GF(8^91+5) which makes it different from better established curves. So, it is only really sensible to use the curve in combination with other very different curves.

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

2. Requirements Language (RFC 2119)

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [BCP14].

3. Encoding points

Elliptic curve cryptography uses points for public keys and raw shared secrets.

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, both of whose coordinates may be deemed as infinity.

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: for practicality, an implementation will often internally represent the x-coordinate as a ratio [X:Z] of field elements. Each field element has multiple representations, but [x:1] can viewed as normal representation of x. (Infinity can be then represented by [1:0], though one must be careful.)

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

3.1. Point encoding process

3.1.1. Summary

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

3.1.2. 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 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, then 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)).

3.2. Point decoding process

3.2.1. Summary

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

3.2.2. 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 x or -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.  In some situations, such invalid b can |
|  lead to a severe attack.  In these situations, the  |
|  decoded point (x,y) MUST be validated, as described |
|  below in Section 4.                                 |
|                                                      |
+------------------------------------------------------+
```

In cases where a value for at least one of y, -y, iy, or -iy is needed such as Diffie--Hellman key agreement using some other coordinate system (such as one might need when converting to Edwards coordinates), the candidate value can be obtained by computing a square root:

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

In some cases, 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 replace x by p-x, and adjusts the discrete logarithm appropriately. These steps can be done by the encoder, with the decoder doing nothing.

4. 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 a little milder than the average 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.

4.1. When to validate

This section specifies several strategies.

4.1.1. Mandatory validation

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

4.1.2. 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 groun, an impelmentatio 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.

4.1.4. 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.

Note: the twist of 2y^2=x^3+x/GF(8^91+5) curve has order:

2^2 * 5 * 1526119141 * 788069478421 * 182758084524062861993 *
3452464930451677330036005252040328546941

For example, consider a static hashed-ECDH implementation implemeneted with a Montgomery ladder, such that the static secret key is used at most ten million times hashed-ECDH transactions. Even if exposed to invalid points on the twist, the secruity risk is nearly negligible.

4.2. 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 moduli smaller than 8^91+5.

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

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

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 n = ord(G). The costly method to check this is checking that [n]P = O. An alternative method is to try to solve for Q in the equation [12]Q=P, which involves methods such a division polynomials.

To be completed.

5. OPTIONAL encodings

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

5.1. Encoding scalars

Scalar (integer point multipliers) sometimes needed to be encoding as byte strings, at least internally to an implementation.

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

In Diffie--Hellman only implementations, the scalars s and p-s really have not significant distinction, so all scalars can be represented with 34 bytes.

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.

5.2. 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) for use in password-authenticated key exchange.

- 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 3.2 does not suffice to encode strings, becausse 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]. The 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.

6. IANA Considerations

This document requires no actions by IANA, yet.

7. Security considerations

No cryptographic algorithm is without risk.

Theoretically, therefore, cryptographic risk analysis should be comparative: so that the least risky cryptographic algorithm can be chosen. Practically, however, it is difficult to compare an algorithm to all others.

For practicality, this section lists the most plausible risks of 2y^2=x^3+x/GF(8^91+5), comparing these against a general background of any curve in ECC. To a lesser degree, this section contrasts these risks to a few other well-established and standardized specific curves.

7.1. 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. Perhaps a smaller field, such as that used in Curve25519, has a simpler reduction and overflow-avoidance properties.

- 8^91+5, by virtue of being well-above 256 bits in size, risks its user doing extra, and perhaps unnecessary, computation to protect their 128-bit keys, whereas smaller curves might be faster (as expected) yet still provide enough security. In other words, the extra 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 Kolmogorov 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 much 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. So should have comparable efficiency for basic field arithmetic operations, such as addition, multiplication and reduction. The field 8^95-9 is a little larger, but can still be implemented using five 64-bit words. Being larger, 8^95-9, it 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 bug.

- 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 arugably 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.

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].

To be completed:

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

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

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

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

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

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

7.2. 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.

7.3. Encoding choices

To be completed.

7.4. 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.

7.5. 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 cryptographic forum such as the IACR conferences like Crypto and Eurocrypt. So, it has only 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.

8. References

8.1. Normative References

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

8.2. Informative References

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>

[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. Test vectors

The following are some test vectors.

```
0000000000000000029352b31395e382846472f782b335e783d325e79322054534554
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 big-endian.

> Note: Public keys are encoded as 34-byte strings are little, so one reverses the order of the bytes found in the test vectors. The pseudocode in Appendix C.2 should make this clear.

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, which would serve as a very insecure private key. 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.

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 (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 catch any more bugs, but simply 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 reassigning y with the equation y = yG. (So, the y on the left is new, the y on the right is old, they are not the same.) Then 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. The left side is the new value of y. Finally, Bob's replaces y by y^900 mod order(G), just as Alice did.

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 B. Minimizing trapdoors and backdoors

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

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

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 (secure) elliptic curves using 21 or fewer characters. (But if the specification new interpretations, then new interpretation might also be used to further compress the specification of 2y^2=x^3+x/GF(8^91+5) to below 21 characters.)

The curve is actually 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?].

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

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

This has a shorter field specification. Perhaps an isomorphic curve can be found (one with three terms), so that total length is 20 or fewer characters.

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

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.

Low Kolmogorov complexity is not directly correlated with security of the curve.

Note: Indeed, as shown further below, the very insecure examples exist with lower complexity, by choosing a defective curve equation.

The benefit of low Kolmogorov complexity is an idea, which general to cryptography, sometimes called nothing-up-my-sleeve, or subversion-resistance, or similar. For elliptic curves, the benefit may be stated as the two 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.

Singular cubics, though not formally elliptic curves, are arguably among the same class of object, and can be described similarly, using equations and so. For smooth singular curves (irreducible cubics) a group can be define, 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. Worse, a low Kolmogorov complexity curve can be described that suffers from three attacks simultaneously: y^2=x^3+1/GF(2^127-1). To be completed.

Of course, the weak cubics are vulnerable to extremely well-known attacks, so when estimating the bits of information in the Kolmogorov complexity of curves that resist known attacks, we can ignore such examples. The point of these examples, however, is to demonstrate that there exists known attacks that affect curves of low Kolmogorov complexity, and therefore secret attacks might have the same property.

So, it is sensible to disclaim any resistance to secret attacks of such a nature. For this reason, 2y^2=x^3+x/GF(8^91+5) should be used with other elliptic curves.

Appendix C. Pseudocode

This section uses a C-like pseudocode to demonstrate both the well-known algorithms one can use implement this curve, and some details particular to this curve.

Note: Some implementers, such as C programmers, may prefer such pseudocode over the wordy and formulaic specifications given earlier in this draft. Besides the principles and algorithms are well-known, so I have opted to put the pseudocode in a more runnable form than traditional language-agnostic pseudocode.

Note: The pseudocode is not standard C (e.g., it uses non-standard C type __int128), not portable, not thoroughly hardened against side channels or any other implementation attacks.

Note: The pseudocode is highly constricted to minimize line and character counts, with Python-like indentation and Lisp-like clumping of closing delimiters. Tools may exist that can put transform the pseudocode into more conventional C indentation. The pseudocode borrows various yet further C brevities: some idiomatic and conventional, some altogether peculiar. Anything too indecipherable deserves explanation in a future revision of this draft.

Note: this pseudocode has not yet received any independent review.

C.1. Scalar multiplication of 34-byte strings

The pseudocode for scalar multiplication provides an interface for scalar multiplication. A function takes as input 3 pointer to unsigned character strings; it also returns a Boolean value, indicating success or failure.

The pseudocode is to be consider to form a single file, pseudo.c, which is then include into other 3 pieces pseudocode: one to generate test vectors, one to demo a command-line Diffie--Hellman, one to demo public-key validation and twist insecurity of the curve.

The file pseudo.c has two sections, one for field arithmetic, and one form scalar multiplication using Montgomery's ladder.

Note: I have been able to improve the speed of Montgomery's ladder by ~10% using Bernstein's 2-D ladder. I have also been to improve the speed by ~20% using Gallant--Lambert--Vanstone and Edwards coordinates. These improvements are not likely to carry through to a proper optimization regime, since I never used any assembly optimizations. Also these improvements involve more complex algorithms, which may suffer higher risk of implementation attacks.

To be completed.

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

The field arithmetic pseudocode, is the first part of the file pseudo.c, implements all the necessary field operations to implement a Montgomery for elliptic curve 2y^2=x^3+x. This means that it does not include a square computation: instead it has a Legendre symbol computation.

Note: The Legendre symbol is used for public-key validation. The pseudocode implements field inversion and the Legendre symbol using exponentiation, with the aim of being simple and constant-time. Alternative algorithms for these tasks are known to experts.

```
<CODE BEGINS>
#define RZ return z
#define B 34
#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)), !j;}
<CODE ENDS>
```

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

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

- type ii is 128-bit integer

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

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

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

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

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

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

TO DO: add notes answering these questions:

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

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

Note: The partial reduction technique used in the multiplication pseudocode is sometimes known as lazy reduction. It aims to do just enough calculation to avoid overflow errors, and thus it may be regarded as attempt at optimization.

To be completed.

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

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

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

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

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

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

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

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

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

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

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

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

C.1.2. Montgomery ladder scalar multiplication

The pseudocode below, 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.

Again, the pseudocode is a continuation of the pseudocode for field arithmetic, and all previous definitions are assumed.

```
<CODE BEGINS>
#define X z[0]
#define Z z[1]
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||
  leg(mal(y,2,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;
  mal(X,2,mul(X,squ(a,add(a,X,Z)),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>
```

The pseudocode function mulch -- which multiplies byte string (character) representations of point b by the byte string representation of integer d -- omits public key validation of the input point b if the base of scalar multiplication is the chosen fixed base, or if the input integer d and output point db have the same location.

The reason for the latter omission of public key validation is the integer d is overwritten presumably the caller of mulch intended to use d only once, so that d is likely to be an ephemeral secret, largely obviating the need to validate b.

In other words, the caller of mulch can control whether public key validation is done by choosing the locations of db, b, b appropriately. (An alternative would be for mulch to include a flag to indicate whether b needs to be validated. Instead, the pseudocode tries to make mulch do the sensible choice for Diffie--Hellman if the caller forgets whether public key validation is necessary.)

The pseudocode files tv.c, dhe.c and pkv.c, define in the sections below, demonstrate the use of mulch, and its features regarding public key validation.

In case, mulch returns a Boolean-valued integer indicating whether b was valid. If validation was requested by the interface, and b is invalid, then mulch return false (0), and the memory location db should remain unaltered.

> Note: the pseudocode makes types c and C volatile, with the aim that the C compiler will preserve attempts to zeroize values of this type. Such zeroization steps in the pseudocode do add clutter to the code, but have usually been delimited by parentheses or braces to indicate their implementation-specific purpose.

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 Pseudocode for test vectors

The following pseudocode, describing the contents of a file tv.c, includes the previously defined file pseudo.c, and stdio.h, and then generates some test vectors.

```
<CODE BEGINS>
#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 j=1e5/2,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)/1e5);//else//debug
  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>
```

To be completed: Explain this properly, if possible.

The test vectors should output this:

```
0000000000000000029352b31395e382846472f782b335e783d325e79322054534554
00000000000000000000000000000000000000000000000000000000000000000117
c8c0f2f404a9fabc91c939d8ea1b9e258d82e21a427b549f05c832cf8d48296ffad7
5f336f56f86de3d52b0eab85e527f2ac7b9d77605c0d5018f5faa4243fd462b1badd
fc023b3f03b469dca32446db80d9b388d753cc77aa4c3ee7e2bb86e99e7bed38f509
8c2b0d58eb27185715a48d6071657273dfbb861e515ac8bac9bfe58f2baa85908221
8c2b0d58eb27185715a48d6071657273dfbb861e515ac8bac9bfe58f2baa85908221
```

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

The following code, representing a file dhe.c, is a bilingual: being valid C and bash script.

As a bash script, it will compile the C code as dhe, then run it twice, once as Alice and once as Bob, piping the ephemeral public keys, and writing the resulting Diffie--Hellman agreed secret keys into pipes. The agreed secret keys are fed into SHA-256 to demonstrate their equality, but also to show the typical way to use DH agree keys (to hash them rather than use them directly).

This pseudocode assumes a Linux-like system.

```
<CODE BEGINS>
#include "pseudo.c" /* dhe.c (also a bash script)
: demos ephemeral DH, also creates, clobbers files dhba dha dhb
: -- Dan Brown, BlackBerry, '19 */
#include <stdio.h>
_ get(c p,_*f){if(f)while(!fread((_*)p,B,1,f));}
_ put(c p,_*f){if(f)fwrite((_*)p,B,1,f),fflush(f); bite(p,O);}
int main (_){C s="/dev/urandom",p="EPHEMERAL s => OK if p INVALID";
  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 -O3 dhe.c -o 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>
```

C.4 Pseudocode for public-key validation and twist insecurity

The following pseudocode, describing a file pkv.c, demonstrates the public-key validation features of mulch from pseudo.c, by deliberately supplying invalid points to mulch. It also demonstrates how to turn PKV on and off using the mulch interface.

It also demonstrates the need for PKV despite using the Montgomery by finding points of low order on the twist of the curve, and showing that such points can leak bits of the secret multiplier.

It further demonstrates the order of the curve, and complex multiplication by i, and the fact the 34-byte representation of points is unaffected by multiplication by i.

```
<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>
```

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 pseudocode above.

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>
```

D. Primality proofs and certificates

Recent work of Albrecht and others [AMPS] has shown the combination of adversarially chosen prime and improper probabilistic primality tests can result in attacks.

The adversarial primes are generally result of an exhaustive search, and therefore 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 compact representation of these primes:

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

The [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.

D.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.

D.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, to Gaelle Martin-Cocher for encouraging submission of this I-D. Thanks to David Jacobson for sending Pratt primality certificates.

Author's Address

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