Internet-Draft
Intended status: Experimental
Expires: 2019-10-06

D. Brown
BlackBerry

2019-04-04

Elliptic curve 2y^2=x^3+x over field size 8^91+5

<draft-brown-ec-2y2-x3-x-mod-8-to-91-plus-5-03.txt>

Abstract

This document recommends using a special elliptic curve alongside dissimilar curves, such as NIST P-256, Curve25519, sect283k1, Brainpool, and random curves, as a cryptographic defense against an unlikely, undisclosed attack against mainstream curves. Features of this curve 2y^2=x^3+x/GF(8^91+5) are: isomorphism to Miller curves from 1985; Montgomery form mappable to Edwards; simple field powering for inversion, Legendre symbol, and square roots; efficient endomorphism to speed up Diffie--Hellman with Bernstein's 2-D ladder; 34-byte keys; similarity to Bitcoin curve; hashing-to-point; low Kolmogorov complexity (low risk of backdoor).

Table of Contents

1. Introduction

This document relates to elliptic curve cryptography (ECC). It specifies methods for using the elliptic curve 2y^2=x^3+x over the field of size 8^91+5. It recommends using this curve in combination with a diverse set of curves, as a strongest-link multi-layer defense-in-depth against undisclosed attacks against some subset of curves.

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) use it as bit-wise exclusive-or.

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

200000000000000000000000000000000000000000000000000000000000005

with with 67 zeros between 2 and 5.

1.2. Motivation

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

The best countermeasure against a secretly embedded trapdoor in an elliptic curve is to use a diverse combination of elliptic curves. So, this curve is only recommended for use in such a combination.

The detailed motivations for curve 2y^2=x^3+x over field 8^91+5 are discussed in Appendix B (and in [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 a point into 34 bytes

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 2y^2=x^3+x, the coordinates are x and y are field elements for this curve are integers modulo 8^91+5.

> Note: for practicality, an implementation will often represented 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 finite point (x,y).
>
> Note: In many typical ECC schemes, the 34-byte encoding works well, despite being neither injective nor surjective.

3.1. Encoding a point into bytes

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

In detail: a point (x,y) is encoded into 34 bytes, as follows.

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

0 <= x < 8^91+5.

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

x' =: min(x,p-x) mod 2^272.

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

(x' mod 2^272) = 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 represents applying a complex multiplication by i on the curve, since [i](x,y) = (-x,iy) = (u,v) is also a point on the curve, because 2u^2 = 2(iy)^2 = -2y^2 = -(x^3+x) = (-x)^3 + (-x) = v^3 + v. In many application, 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 elliptic curve algorithms 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. Decoding bytes into a point

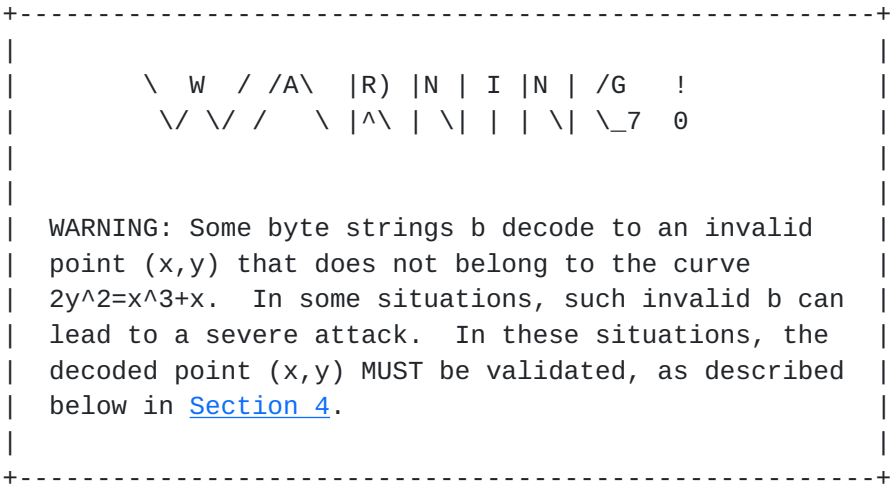
In short: the bytes are little-endian decoded into an integer which becomes the x-coordinate. Public-key validation done if needed. If needed, the y-coordinate is recovered.

In greater 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.



In cases, where a value for at least 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.

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

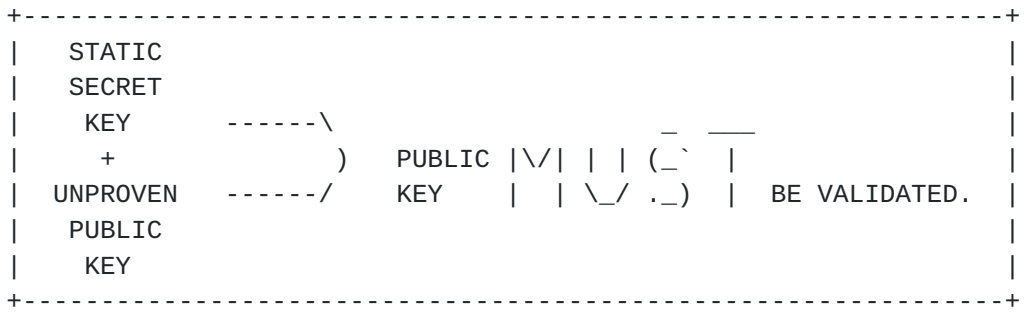
4.1.1. Precautionary mandatory validation

Every public key (and point) MAY be validated, as an extra precaution (i.e., defense in depth.

4.1.2. Simplified validation

If small but general implementation aims for high speed, then the implementation might not be able to the cost mandatory public key validation.

It SHOULD follow at least the rule that an distrusted public key is validated before combination with a static secret key.



4.1.3. Relatively safe cases of non-validation

In some application an implementation of ECC seem not to suffer an invalid curve attack. This section lists these situations.

Note: The main reason to omit public key validation is save time.

We classify these situations at two levels: safe, if it seems that no harm is possible, and relatively safe, if it there the attack is both no worse than another attack and requires something else to be broken.

To be verified.

If the secret key is ephemeral, the public key is trusted (signed) and a Montgomery ladder is used, then omitting validation of the public key seems relatively safe. This is mostly because if the holder of the public key is the attacker, then the trust system has been broken. Furthermore, the attacker, as the intended recipient in a typical communication, should already be able to receive any data hidden by the secret key.

To be extended.

4.1.4. Minimal validation

To maximize efficiency, an implementation may wish to minimize the amount of validation done down to the point of only resisting a known
attack.

To be completed.

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

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 * 345246493045167733003600525204032854694l

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

Upon decoding the 34 bytes into x, the next step is to compute z=2(x^3+x). Then one checks if z has a nonzero square root (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 compute the Legendre symbol using powering in the field: (z/p) = z^((p-1)/2) = z^(2^272+2). This will have value 0,1 or p-1 (which is equivalent to -1).

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

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 scalar multipliers as 34 bytes

To be completed.

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

The main application is to signatures.

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

5.2. Encoding 34 bytes into a point (sketch)

In niche applications, it may be desired to encode arbitrary bytes as
points.

> Note: This type of encoding is sometimes called hashing to a curve.
>
> Note: Diffie--Hellman key exchange or digital signatures do not require encoding of arbitrary byte strings.

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

> Note: the point encoding described earlier does a different job. It encodes every point as a byte string. The task here is the opposite: to encode every byte string as a point.
>
> Note: Encoding a byte string as a point yields biased elliptic curve points, but has the advantage that the byte-strings are unbiased.

The encoding is called Elligator i, (see also [B1]), and is just a minor variation of the Elligator 2 construction [Elligator]. Elligator 2 fails for curves with j-invariant 1728, which includes $2y^2=x^3+x$, so a minor tweak is made, obtain Elligator i.

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

Given any random field element r, compute

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

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

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

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

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

The Elligator i map is reversible.

To be completed.

6. IANA Considerations

This document requires no actions by IANA, yet.

7. Security considerations

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

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.

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

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

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

- 8^91+5 is smaller than some other six-symbol primes: 8^95-9, 9^99+4 and 9^87+4. Arguably, 8^91+5 fails to maximize field size, and thus potential Pollard rho resistance of the ECDLP, among six-symbol primes. 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 cost. The prime 8^95-9 is just a little below a power of two, so should have comparable efficiency for basic field arithmetic. The field 8^95-9 is a little larger, but can still be implemented using five 64-bit words. Being larger, 8^85-9, it has a slightly greater risk than 8^91+5 of leading to an arithmetic overflow implementation fault in field arithmetic. Also, field size 8^91+5 has very simple powering algorithms for computing field inverses, Legendre symbols, and square roots, all because it is just slightly above a power of two. For field size 8^85-9, these powering algorithms require more complicated algorithms.

- 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.)

See [B1] for further discussion.

7.2. Curve choice

A first risk of using 2y^2=x^3+x is the fact that it is a special curve, with complex multiplication leading to an efficient endomorphism. Many other standard curves, NIST P-256 [NIST-P-256], Curve25519, Brainpool [Brainpool], do not have any efficient endomorphisms. Yet some standard curves do, NIST K-283 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 over 8^91+5 was (seemingly) first described to the public in 2017 [AB]. So, it has a very low age.

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

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

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

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

- 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 [Gordon] into
Secretly-embedded trapdoor ... also known as a backdoor.

Appendix A. Test vectors

To be completed.

Appendix B. Motivation: minimizing the room for backdoors

To be completed.

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

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

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

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

Appendix C. Pseudocode

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

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

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

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

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

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

C.1. Byte encoding

Pseudocode for byte representation encoding process is

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

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

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

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

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

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

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

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

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

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

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

C.2. Byte decoding

Pseudocode for decoding is:

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

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

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

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

C.3. Fermat inversion

Projective coordinates help avoid costly inversion steps during scalar multiplication.

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

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

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

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

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

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

Pseudocode for Fermat inversion is:

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

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

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

C.4. Branchless Legendre symbol computation

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

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

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

C.5. Field multiplication and squaring

To be completed.

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

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

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

C.6. Field element partial reduction

To be completed.

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

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

TO DO: add notes answering these questions:

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

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

C.7. Field element final reduction

To be completed.

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

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

Pseudocode that aims for final reduction is the following:

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

C.8. Scalar point multiplication

Work in progress.

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

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

> Note: The following pseudocode is not entirely consistent with previous pseudocode examples.
>
> Note and Warning: The following pseudocode uses secret indices to access (small) arrays. This has a risk of cache-timing attacks.

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

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

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

TO DO:

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

C.9. Diffie--Hellman pseudocode

To be completed.

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

C.10. Elligator i

To be completed.

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

Pseudocode (to be verified):

```
<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 two primes involved for 2y^2=x^3+x/GF(8^91+5) should already resist [AMPS] because compact representation of these primes.

For further assurance, this section provides Pratt primality certificates for the two primes.

> Note: Recall that every prime p has a unique Pratt certificate, which consists of the factorization of p into primes, and, recursively, Pratt primality certificates for each of those primes. Verification of a Pratt certificates is also recursive: it uses the factorization data to conduct Fermat and Lehmer tests, which together verify primality.

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: Writing % for modular reduction (with lower precedence than exponentiation ^), a first step in verifying the Pratt certificate is a Fermat pseudoprime test for each prime in the list, meaning the all the numbers below are 1:

```
2^(b-1)%b, 2^(c-1)%c, 3^(d-1)%d, 2^(e-1)%e, 2^(f-1)%f, 3^(g-1)%g,
2^(h-1)%h, 5^(i-1)%i, 6^(j-1)%j, 3^(k-1)%k, 2^(l-1)%l, 3^(m-1)%m,
2^(n-1)%n, 5^(o-1)%o, 2^(p-1)%p, 3^(q-1)%q, 6^(r-1)%r, 2^(s-1)%s,
2^(t-1)%t, 6^(u-1)%u, 7^(v-1)%v, 2^(w-1)%w, 2^(x-1)%x, 14^(y-1)%y,
3^(z-1)%z, 5^(A-1)%A, 3^(B-1)%B, 7^(C-1)%C, 3^(D-1)%D, 7^(E-1)%E,
5^(F-1)%F, 2^(G-1)%G, 2^(H-1)%H, 2^(I-1)%I, 3^(J-1)%J, 2^(K-1)%K,
2^(L-1)%L, 10^(M-1)%M, 5^(N-1)%N, 10^(O-1)%O, 2^(P-1)%P,
10^(Q-1)%Q, 6^(R-1)%R, 7^(S-1)%S, 5^(T-1)%T, 3^(U-1)%U, 5^(V-1)%V,
2^(W-1)%W, 2^(X-1)%X, 3^(Y-1)%Y, 7^(Z-1)%Z.
```

Note: Each Fermat base above was chosen as the minimal possible value. These bases can be deduced from b,c,...,Z by searching bases 2,3,4,... until a Fermat is found. The results of these search are included above for convenience.

Note: A second step to verifying a Pratt certificate is to apply Lehmer's theorem to each Fermat psuedoprime. To prove b,c,d,...,Z are prime, it now suffices to verify that all of following 154 modular exponentiations result in a value different from 1.

```
2^((b-1)/a)%b, 2^((c-1)/a)%c, 3^((d-1)/a)%d, 3^((d-1)/b)%d,
2^((e-1)/a)%e, 2^((e-1)/c)%e, 3^((f-1)/a)%f, 3^((f-1)/b)%f,
3^((g-1)/a)%g, 2^((h-1)/a)%h, 2^((h-1)/b)%h, 5^((i-1)/a)%i,
5^((i-1)/e)%i, 6^((j-1)/a)%j, 6^((j-1)/c)%j, 3^((k-1)/a)%k,
2^((l-1)/a)%l, 2^((l-1)/f)%l, 3^((m-1)/a)%m, 3^((m-1)/b)%m,
3^((m-1)/f)%m, 2^((n-1)/a)%n, 2^((n-1)/c)%n, 5^((o-1)/a)%o,
5^((o-1)/b)%o, 5^((o-1)/f)%o, 2^((p-1)/a)%p, 2^((p-1)/l)%p,
3^((q-1)/a)%q, 3^((q-1)/g)%q, 6^((r-1)/a)%r, 6^((r-1)/a)%r,
2^((s-1)/a)%s, 2^((s-1)/b)%s, 2^((t-1)/a)%t, 2^((t-1)/k)%t,
6^((u-1)/a)%u, 6^((u-1)/b)%u, 6^((u-1)/c)%u, 7^((v-1)/a)%v,
7^((v-1)/c)%v, 7^((v-1)/k)%v, 2^((w-1)/a)%w, 2^((w-1)/s)%w,
2^((x-1)/a)%x, 2^((x-1)/b)%x, 2^((x-1)/i)%x, 14^((y-1)/a)%y,
14^((y-1)/c)%y, 14^((y-1)/o)%y, 3^((z-1)/a)%z, 3^((z-1)/b)%z,
3^((z-1)/u)%z, 5^((A-1)/a)%A, 5^((A-1)/t)%A, 3^((B-1)/a)%B,
3^((B-1)/d)%B, 3^((B-1)/h)%B, 7^((C-1)/a)%C, 7^((C-1)/c)%C,
7^((C-1)/u)%C, 3^((D-1)/a)%D, 3^((D-1)/v)%D, 7^((E-1)/a)%E,
7^((E-1)/e)%E, 7^((E-1)/f)%E, 5^((F-1)/a)%F, 5^((F-1)/A)%F,
2^((G-1)/a)%G, 2^((G-1)/B)%G, 2^((H-1)/a)%H, 2^((H-1)/D)%H,
2^((I-1)/a)%I, 2^((I-1)/c)%I, 2^((I-1)/x)%I, 3^((J-1)/a)%J,
3^((J-1)/c)%J, 3^((J-1)/e)%J, 3^((J-1)/j)%J, 2^((K-1)/a)%K,
2^((K-1)/b)%K, 2^((K-1)/q)%K, 2^((K-1)/r)%K, 2^((L-1)/a)%L,
2^((L-1)/b)%L, 2^((L-1)/J)%L, 10^((M-1)/a)%M, 10^((M-1)/b)%M,
10^((M-1)/d)%M, 10^((M-1)/t)%M, 5^((N-1)/a)%N, 5^((N-1)/b)%N,
5^((N-1)/d)%N, 5^((N-1)/p)%N, 5^((N-1)/w)%N, 10^((O-1)/a)%O,
10^((O-1)/b)%O, 10^((O-1)/m)%O, 10^((O-1)/C)%O, 2^((P-1)/a)%P,
2^((P-1)/b)%P, 2^((P-1)/e)%P, 2^((P-1)/K)%P, 10^((Q-1)/a)%Q,
10^((Q-1)/b)%Q, 10^((Q-1)/c)%Q, 10^((Q-1)/f)%Q, 10^((Q-1)/g)%Q,
10^((Q-1)/E)%Q, 6^((R-1)/a)%R, 6^((R-1)/b)%R, 6^((R-1)/P)%R,
7^((S-1)/a)%S, 7^((S-1)/b)%S, 7^((S-1)/c)%S, 7^((S-1)/M)%S,
5^((T-1)/a)%T, 5^((T-1)/I)%T, 5^((T-1)/O)%T, 3^((U-1)/a)%U,
3^((U-1)/d)%U, 3^((U-1)/u)%U, 3^((U-1)/G)%U, 3^((U-1)/S)%U,
5^((V-1)/a)%V, 5^((V-1)/b)%V, 5^((V-1)/n)%V, 5^((V-1)/u)%V,
5^((V-1)/H)%V, 5^((V-1)/T)%V, 2^((W-1)/a)%W, 2^((W-1)/b)%W,
2^((W-1)/f)%W, 2^((W-1)/L)%W, 2^((W-1)/N)%W, 2^((W-1)/Q)%W,
2^((W-1)/R)%W, 2^((X-1)/a)%X, 2^((X-1)/f)%X, 2^((X-1)/F)%X,
2^((X-1)/W)%X, 3^((Y-1)/a)%Y, 3^((Y-1)/u)%Y, 3^((Y-1)/X)%Y,
7^((Z-1)/a)%Z, 7^((Z-1)/b)%Z, 7^((Z-1)/z)%Z, 7^((Z-1)/U)%Z,
7^((Z-1)/V)%Z, 7^((Z-1)/Y)%Z.
```

Note: The verifier should verify that each base in the Fermat and Lehmer test are equal. For example, the Fermat base for Z was 7, and the factorization of Z-1 was aabzUVY, so the Lehmer theorem test 7^((Z-1)/a)%Z, 7^((Z-1)/b)%Z, ..., 7^((Z-1)/Y)%Z.

Note: The final step is to verify that Z=8^91+5.

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=156151230751602494064296769877 9, X=167553393621084508180871720014384259, Y=145302302948204485494451955596474029 4049.

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.

Note: The same process (Fermat and Lehmer tests) verifies that $ is prime. This process includes search for bases for each of the prime. These bases have not been included in the certificate.

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

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

```
    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 Za=1+ab3oHOT Zb=1+a3SUX
    Zc=1+abNYZb Zd=1+a4kzFZc Ze=1+a3QZaZd
```

When parsing this, say 2nd last item Zd=1+a4kzFZc, the string Zd on the left of = defines a new variables, while the string on the right = gives its value. The string a4kzFZc can be unambiguously parsed as a^4*k*z*F*Zc, scanning from left to right for previous variables, or integers as powers.

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