Internet-Draft
Intended status: Experimental
Expires: 2019-04-07

D. Brown
BlackBerry

2018-10-04

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

<draft-brown-ec-2y2-x3-x-mod-8-to-91-plus-5-02.txt>

Abstract

This document specifies a special elliptic curve with a compact description (see title) and an efficient endormorphism (complex multiplication by i). This curve is only recommended for cryptographic use in a strongest-link combination with dissimilar elliptic curves (e.g. NIST P-256, Curve25519, extension-field curves, etc.). Used in this manner, the curve special features serve as a defense in depth against an unlikely event: a new or secret attack against the other types of elliptic curves.

Status of This Memo

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

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

Copyright Notice

Table of Contents

1. Introduction

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

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

1.1. Background

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

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

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

1.2. Motivation

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

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

2. Requirements Language (RFC 2119)

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

3. Encoding a point into 34 bytes

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

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

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

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

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

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

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

3.1. Encoding a point into bytes

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

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

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

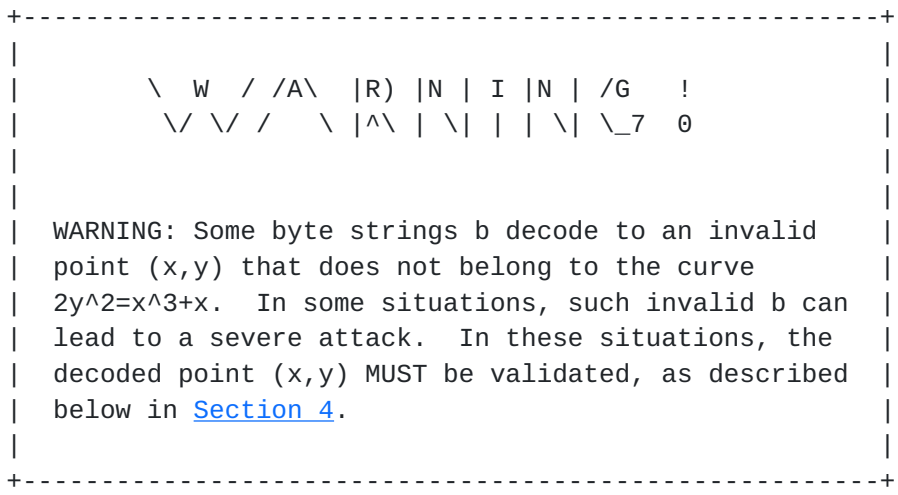
Third, set the byte string b to be the little-endian encoding of the reduced integer x', by finding the unique integers b[i] such that 0<=b[i]<256 and

(x' mod 2^272) = sum (0<=i<=33, b[i]*256^i).

Pseudocode can be found in Appendix C.

3.2. Decoding bytes into a point

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



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

In greater detail: if byte i is b[i], with an integer value between 0 and 255 inclusive, then

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

4. Point validation

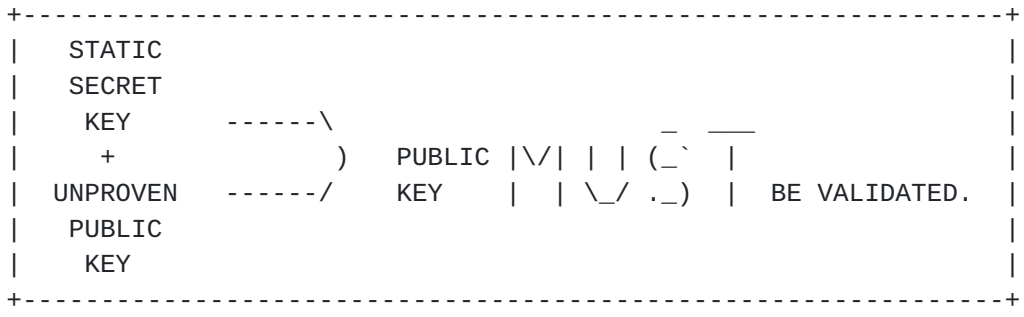
In elliptic curve cryptography, scalar multiplying an invalid public key by a private key risks leaking information about the private key.

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

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

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

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



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

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

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

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

   a) s is a secret

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

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

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

   e) the direct value of sP has been requested

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Note that the twist has order:

2^2 * 5 * 1526119141 * 788069478421 * 182758084524062861993 * 345246493045167733003600525204032854694 1

OLD TEXT BELOW:

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

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

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

ATTEMPT (TO BE CONFIRMED):

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

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

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

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

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

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

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

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

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

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

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

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

5. OPTIONAL encodings

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

5.1. Encoding scalar multipliers as 34 bytes

To be completed.

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

The main application is to signatures.

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

5.2. Encoding 34 bytes into a point (sketch)

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

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

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

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

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

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

Given any random field element r, compute

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

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

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

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

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

This map is reversible.

To be completed.

6. Cryptographic schemes

To be completed, or even removed!

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

6.1. Diffie--Hellman key agreement

To be completed.

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

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

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

6.2. Signatures

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

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

To be completed.

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

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

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

sG = rR + hQ

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

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

s = rk + hd mod n

where n is the order of G.

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

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

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

6.3 Menezes--Qu--Vanstone key agreement

To be completed.

7. IANA Considerations

This document requires no actions by IANA, yet.

8. Security considerations

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

8.1. Field choice

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

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

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

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

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

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

- 8^91+5, is smaller than 2^283 (the field size for curve sect283k1 [SEC2], [Zigbee]), and many other five-symbol and four-symbol powers of primes (such as 9^97). So, it less to provide less resistance to Pollard rho. Recent progress in the elliptic curve discrete logarithm problem, [HPST] and [Nagao], is the main reason to prefer prime fields instead of power of prime fields. A second reason to prefer prime field 8^91+5 (and other large characteristic fields) over small characteristic fields, is the generally better software speed of large characteristic fields: which arises because most software is implemented on a general purpose hardware processor that has fast multiplication circuits. (This speed advantage probably does not apply for hardware.)

See [B1] for further discussion.

8.2. Curve choice

A first risk of using 2y^2=x^3+x is the fact that it is a special curve, with complex multiplication leading to an efficient endomorphism. Many other standard curves, NIST P-256 [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 curve lack this risk because the field is close to a power of two, and the cofactor is a power of two.

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

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

8.3. Encoding choices

To be completed.

8.4. General subversion concerns

Although the main motivation of curve 2y^2=x^3+x over 8^91+5 is to minimize the risk of subversion via a backdoor ([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 user is encouraged to take an especially skeptical viewpoint of curve 2y^2=x^3+x over 8^91+5. So, it is expected that skeptical users of the curve will either

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

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

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

8.5. Concerns about 'aegis'

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

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

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

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

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

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

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

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

Obvious cautions to the reader:

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

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

9. References

9.1. Normative References

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

9.2. Informative References

To be completed.

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

[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: Projective coordinates leak. Pointcheval, Smart, Malone-Lee?

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

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

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

[HPST] Y. Huang, C. Petit, N. Shinohara and T. Takagi. On Generalized First Fall Degree Assumptions, IACR ePrint 2015. <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

In most cases, probablistic primality tests, if conducted properly, can ensure more than adequate security. But recent work of Albrecht and others [AMPS] has shown the combination of adversarially chosen prime and improper probabilistic primality tests can result in attacks.

Three countermeasures to the attacks in [AMPS] are (1) using proper probabilistic prime tests, (2) using provable prime tests, and (3) using nothing-up-my-sleeve primes which seem immune to the [AMPS] attack.

It seems that field size 8^91+5 should already resist [AMPS] based on the third countermeasure (its especially compact representation). This document cannot really accomplish the first countermeasure, because the first countermeasure requires fresh randomness for each test. This document can help with the second countermeasure by providing primality certificate.

A primality certificate is essentially a rapidly verifiable proof of primality. More precisely, it involves some rigorous logic and and some calculations. The calculations, although too tedious to be done by hand, have a cost that is polynomial time, and usually amounting only to some number of modular expontiations, with the number comparable to the bit length of the prime being tested.

Typically, generation of the primality certificate is much more costly than verifying it: generation is not polynomial-time. For example, Pratt certificates require the factorization of p-1, which is typically expensive for large primes. Nevertheless, for the prime 8^91+5, software exists that can generate a Pratt certificate in minutes on a personal computer. Yet other kinds of primality certificates (those using elliptic curves) can be generated even more quickly, except their verification requires an elliptic curve implementation, and uses less elementary rigor in their proofs.

For these reasons, a primality certificate for primes of size 8^91+5 is nearly redundant. Nonetheless, it does not hurt to provide the proofs within this curve, if only for completeness.

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 concatentation is used to indicate multiplication. For example, f = 1+aab = 1+2*2*(1+2) = 13. This brevity was only possible by a fluke: only 52 integers were needed, obviating the need for multi-letter variable names.

Note: the information above can suffice as a Pratt certificate for the primality of Z, but only if the following further sequence of computations are done. (The information suffices because it deducibly implies the sequence of computations.)

Writing % for modular reduction (with lower precedence than exponention ^), verify that following 51 modular exponentiations all result in value 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.

This shows that b,c,...,Z are Fermat pseudoprimes to the Fermat bases indicated (for example, Z is a Fermat pseudoprime to Fermat base 7).

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.

Verify that a is prime (because it is just two).

Lehmer's theorem provides the Lehmer test that a Fermat pseudoprime is prime: if the Fermat base raised to each integer power of the form (pseudoprime-1)/(a prime factor) is not congruent to 1 modulo the pseudoprime. Consequently, to prove b,c,d,...,Z are prime, it suffices to do all the necessary Lehmer tests, which means 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: Each base above is the same as the base in the previous Fermat pseudoprime stage, and each list of prime factors is deduced from the definition of positive integers b,c,...,Z. The consistency between these stages of the proof must be verified for rigor. For example, the Fermat base for Z was 7, and the factorization of Z-1 was aabzUVY, so we must test 7^((Z-1)/a)%Z, 7^((Z-1)/b)%Z, ..., 7^((Z-1)/Y)%Z.

This proves that a,b,...,Z are primes.

Verify that Z=8^91+5 to conclude that 8^91+5 is prime.

Note: the Pratt certificate is essentially unique for each prime. The presentation above is for illustrative purposes: the formatting is not intended for an automated verification. A sensible automation of the verification would simply generate the 154 Lehmer tests from integer definitions and the Fermat pseudoprime tests, rather than rely on the listing provided above. With only slightly greater cost, the Fermat pseudoprime test can be derived from integers, by a separate search for each Fermat base.

Note: A reader who wishes to verify, with greatest certainty, that 8^91+5 is prime, would be probably be most convinced by running a provable prime test entirely independent of this document and the primality certificate given in this section.

Note: the most expensive step in generating the Pratt certificate for 8^91+5 was factoring the integer 8^91+4 = Z-1 = aabzUVY. The other integers were generated in reverse alphabetical order, as Y,X,...,c,b,a, with each integer appearing as an (seemingly) prime factor of one less than number previously computed. All subsequent integer factorizations took time negligible compared to the factorization of Z-1.

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. If the reader has a tool to generate a Pratt certificate (with decimal notation), the reader should be able to find these numbers in the Pratt certifcate for 8^91+5. (Since Pratt certificate generation is slower than for other primality certificate types, some tools require special configuration to generate a Pratt certificate.)

Note: the Pratt certificate for 8^91+5 might not be shortest possible primality certificate (under some measure of length), but optimizing the shortness of a primality certificate seems to add little value.

D.2 Pratt certificate for size of the large elliptic curve subgroup

Using the same verification strategy of the previous strategy, but now with 56 variables a,b,...,z,A,B,...,Z,!,@,#,$, with new definitions:

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 use of punctuation for variable names !,@,#,$, does not scale (or fit into most programming languages), so is really just a hack to avoid a multiplication operator.

A routine but tedious verification (Fermat and Lehmer tests) converts the information above into a proof that $ is prime. (The information above, obtained by repeated factorization, is also routine, but more computatinally expensive, because it involves integer factorization.)

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

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 me Pratt primality certificates (generated with mathetmatic, and re-generated by me).

Author's Address

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