

CFRG
Internet-Draft
Intended status: Informational
Expires: September 25, 2015

A. Langley
Google
R. Salz
Akamai Technologies
S. Turner
IECA, Inc.
March 24, 2015

Elliptic Curves for Security
draft-irtf-cfrg-curves-02

Abstract

This memo describes an algorithm for deterministically generating parameters for elliptic curves over prime fields offering high practical security in cryptographic applications, including Transport Layer Security (TLS) and X.509 certificates. It also specifies a specific curve at the ~128-bit security level and a specific curve at the ~224-bit security level.

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."

This Internet-Draft will expire on September 25, 2015.

Copyright Notice

Copyright (c) 2015 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to [BCP 78](#) and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect

Internet-Draft

cfgrcurve

March 2015

to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1.	Introduction	2
2.	Requirements Language	3
3.	Security Requirements	3
4.	Notation	3
5.	Parameter Generation	4
5.1.	$p = 1 \bmod 4$	4
5.2.	$p = 3 \bmod 4$	5
5.3.	Base points	5
6.	Recommended Curves	6
6.1.	Curve25519	6
6.2.	Goldilocks	7
7.	The curve25519 and curve448 functions	8
7.1.	Test vectors	12
8.	Diffie-Hellman	13
8.1.	Test vectors	14
9.	Acknowledgements	15
10.	References	15
10.1.	Normative References	15
10.2.	Informative References	15
	Authors' Addresses	16

[1.](#) Introduction

Since the initial standardization of elliptic curve cryptography (ECC) in [\[SEC1\]](#) there has been significant progress related to both efficiency and security of curves and implementations. Notable examples are algorithms protected against certain side-channel attacks, various 'special' prime shapes which allow faster modular arithmetic, and a larger set of curve models from which to choose. There is also concern in the community regarding the generation and potential weaknesses of the curves defined in [\[NIST\]](#).

This memo describes a deterministic algorithm for generating cryptographic elliptic curves over a given prime field. The constraints in the generation process produce curves that support constant-time, exception-free scalar multiplications that are

resistant to a wide range of side-channel attacks including timing and cache attacks, thereby offering high practical security in cryptographic applications. The deterministic algorithm operates without any input parameters that would permit manipulation of the resulting curves. The selection between curve models is determined

by choosing the curve form that supports the fastest (currently known) complete formulas for each modularity option of the underlying field prime. Specifically, the Edwards curve $x^2 + y^2 = 1 + dx^2y^2$ is used with primes p with $p \equiv 3 \pmod{4}$, and the twisted Edwards curve $-x^2 + y^2 = 1 + dx^2y^2$ is used when $p \equiv 1 \pmod{4}$.

[2.](#) Requirements Language

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](#) [[RFC2119](#)].

[3.](#) Security Requirements

For each curve at a specific security level:

1. The domain parameters SHALL be generated in a simple, deterministic manner, without any secret or random inputs. The derivation of the curve parameters is defined in [Section 5](#).
2. The trace of Frobenius MUST NOT be in $\{0, 1\}$ in order to rule out the attacks described in [[Smart](#)], [[AS](#)], and [[S](#)], as in [[EBP](#)].
3. MOV Degree: the embedding degree k MUST be greater than $(r - 1) / 100$, as in [[EBP](#)].
4. CM Discriminant: discriminant D MUST be greater than 2^{100} , as in [[SC](#)].

[4.](#) Notation

Throughout this document, the following notation is used:

p Denotes the prime number defining the underlying field.

$\text{GF}(p)$ The finite field with p elements.

A An element in the finite field $GF(p)$, not equal to -1 or zero.

d An element in the finite field $GF(p)$, not equal to -1 or zero.

P A generator point defined over $GF(p)$ of prime order.

$X(P)$ The x-coordinate of the elliptic curve point P on a (twisted) Edwards curve.

$Y(P)$ The y-coordinate of the elliptic curve point P on a (twisted) Edwards curve.

[5.](#) Parameter Generation

This section describes the generation of the curve parameter, namely A , of the Montgomery curve $y^2 = x^3 + Ax^2 + x$. The input to this process is p , the prime that defines the underlying field. The size of p determines the amount of work needed to compute a discrete logarithm in the elliptic curve group and choosing a precise p depends on many implementation concerns. The performance of the curve will be dominated by operations in $GF(p)$ and thus carefully choosing a value that allows for easy reductions on the intended architecture is critical. This document does not attempt to articulate all these considerations.

The value $(A-2)/4$ is used in several of the elliptic curve point arithmetic formulas. For simplicity and performance reasons, it is beneficial to make this constant small, i.e. to choose A so that $(A-2)$ is a small integer which is divisible by four.

[5.1.](#) $p \equiv 1 \pmod{4}$

For primes congruent to $1 \pmod{4}$, the minimal cofactors of the curve and its twist are either $\{4, 8\}$ or $\{8, 4\}$. We choose a curve with the latter cofactors so that any algorithms that take the cofactor into account don't have to worry about checking for points on the twist, because the twist cofactor is larger.

To generate the Montgomery curve we find the minimal, positive A value such $(A-2)$ is divisible by four and where the cofactors are as desired. The "find1Mod4" function in the following Sage script

returns this value given p:

```
def findCurve(prime, curveCofactor, twistCofactor):
    F = GF(prime)

    for A in xrange(1, 1e9):
        if (A-2) % 4 != 0:
            continue

        try:
            E = EllipticCurve(F, [0, A, 0, 1, 0])
        except:
            continue

        order = E.order()
        twistOrder = 2*(prime+1)-order

        if (order % curveCofactor == 0 and is_prime(order // curveCofactor) and
            twistOrder % twistCofactor == 0 and is_prime(twistOrder // twistCofactor)):
            return A

def find1Mod4(prime):
    assert((prime % 4) == 1)
    return findCurve(prime, 8, 4)
```

Generating a curve where $p = 1 \bmod 4$

[5.2.](#) $p = 3 \bmod 4$

For a prime congruent to $3 \bmod 4$, both the curve and twist cofactors can be 4 and this is minimal. Thus we choose the curve with these cofactors and minimal, positive A such that $(A-2)$ is divisible by four. The "find3Mod4" function in the following Sage script returns this value given p :

```
def find3Mod4(prime):
    assert((prime % 4) == 3)
    return findCurve(prime, 4, 4)
```

Generating a curve where $p = 3 \bmod 4$

[5.3.](#) Base points

The base point for a curve is the point with minimal, positive u value that is in the correct subgroup. The "findBasepoint" function in the following Sage script returns this value given p and A :

```
def findBasepoint(prime, A):
    F = GF(prime)
    E = EllipticCurve(F, [0, A, 0, 1, 0])

    for uInt in range(1, 1e3):
        u = F(uInt)
        v2 = u^3 + A*u^2 + u
        if not v2.is_square():
            continue
        v = v2.sqrt()

        point = E(u, v)
        order = point.order()
        if order > 8 and order.is_prime():
            return point
```

Generating the base point

[6.](#) Recommended Curves

[6.1.](#) Curve25519

For the ~128-bit security level, the prime $2^{255}-19$ is recommended for performance on a wide-range of architectures. This prime is congruent to 1 mod 4 and the above procedure results in the following Montgomery curve, called "curve25519":

p $2^{255}-19$

A 486662

order $2^{252} + 0x14def9dea2f79cd65812631a5cf5d3ed$

cofactor 8

The base point is $u = 9$, $v = 14781619447589544791020593568409986887264606134616475288964881837755586237401$.

This curve is isomorphic to the twisted Edwards curve $-x^2 + y^2 = 1 - d \cdot x^2 \cdot y^2$ where:

p $2^{255}-19$

d 37095705934669439343138083508754565189542113879843219016388785533085940283555

order $2^{252} + 0x14def9dea2f79cd65812631a5cf5d3ed$

cofactor 8

X(P) 15112221349535400772501151409588531511454012693041857206046113283949847762202

Y(P) 46316835694926478169428394003475163141307993866256225615783033603165251855960

The isomorphism maps are:

$$\begin{aligned}(u, v) &= ((1+y)/(1-y), \sqrt{-1} \cdot \sqrt{486664} \cdot u/x) \\ (x, y) &= (\sqrt{-1} \cdot \sqrt{486664} \cdot u/v, (u-1)/(u+1))\end{aligned}$$

The Montgomery curve defined here is equal to the one defined in [[curve25519](#)] and the isomorphic twisted Edwards curve is equal to the one defined in [[ed25519](#)].

6.2. Goldilocks

For the ~224-bit security level, the prime $2^{448}-2^{224}-1$ is recommended for performance on a wide-range of architectures. This prime is congruent to 3 mod 4 and the above procedure results in the following Montgomery curve, called "curve448":

p $2^{448}-2^{224}-1$

A 156326

order $2^{446} -$
0x8335dc163bb124b65129c96fde933d8d723a70aad873d6d54a7bb0d

cofactor 4

The base point is $u = 5$, $v =$
3552939267855681752641275020637833348089763993877142718318808984351 \\
69088786967410002932673765864550910142774147268105838985595290606362.

This curve is isomorphic to the Edwards curve $x^2 + y^2 = 1 + d \cdot x^2 \cdot y^2$ where:

p $2^{448}-2^{224}-1$

d 611975850744529176160423220965553317543219696871016626328968936415
0 \ 87860042636474891785599283666020414768678979989378147065462815
545017

order $2^{446} -$
0x8335dc163bb124b65129c96fde933d8d723a70aad873d6d54a7bb0d

X(P) 345397493039729516374008604150537410266655260075183290216406970
2816 \ 45695073672344430481787759340633221708391583424041788924124
567700732

Y(P) 363419362147803445274661903944002267176820680343659030140745099
5903 \ 06164083365386343198191849338272965044442230921818680526749
009182718

The isomorphism maps are:

$$\begin{aligned}(u, v) &= ((y-1)/(y+1), \sqrt{156324} * u/x) \\ (x, y) &= (\sqrt{156324} * u/v, (1+u)/(1-u))\end{aligned}$$

That curve is also 4-isogenous to the following Edward's curve $x^2 + y^2 = 1 + d * x^2 * y^2$, called "Ed448-Goldilocks":

p $2^{448} - 2^{224} - 1$

d -39081

order $2^{446} -$
0x8335dc163bb124b65129c96fde933d8d723a70aad873d6d54a7bb0d

cofactor 4

The 4-isogeny maps between the Montgomery curve and the Edward's curve are:

$$\begin{aligned}(u, v) &= (x^2/y^2, (2 - x^2 - y^2) * x/y^3) \\ (x, y) &= (4 * v * (u^2 - 1) / (u^4 - 2 * u^2 + 4 * v^2 + 1), (u^5 - 2 * u^3 - 4 * u * v^2 + u) / (u^4 - 2 * u^2 + 4 * v^2 + 1))\end{aligned}$$

7. The curve25519 and curve448 functions

The "curve25519" and "curve448" functions performs scalar multiplication on the Montgomery form of the above curves. (This is used when implementing Diffie-Hellman.) The functions take a scalar and a u-coordinate as inputs and produce a u-coordinate as output. Although the functions work internally with integers, the inputs and outputs are 32-byte strings and this specification defines their encoding.

U-coordinates are elements of the underlying field $GF(2^{255-19})$ or $GF(2^{448-2^{224}-1})$ and are encoded as an array of bytes, u , in little-endian order such that $u[0] + 256 * u[1] + 256^2 * u[2] + \dots + 256^n * u[n]$ is congruent to the value modulo p and $u[n]$ is minimal. When receiving such an array, implementations of curve25519 (but not

curve448) MUST mask the most-significant bit in the final byte. This is done to preserve compatibility with point formats which reserve the sign bit for use in other protocols and to increase resistance to implementation fingerprinting.

For example, the following functions implement this in Python, although the Python code is not intended to be performant nor side-channel free. Here the "bits" parameter should be set to 255 for curve25519 and 448 for curve448:

```
def decodeLittleEndian(b, bits):
    return sum([b[i] << 8*i for i in range((bits+7)/8)])

def decodeUCoordinate(u, bits):
    u_list = [ord(b) for b in u]
    if bits % 8:
        u_list[-1] &= (1<<(bits%8))-1
    return decodeLittleEndian(u_list, bits)

def encodeUCoordinate(u, bits):
    u = u % p
    return ''.join([chr((u >> 8*i) & 0xff) for i in range((bits+7)/8)])
```

(EDITORS NOTE: [draft-turner-thecurve25519function](#) also says "Implementations MUST reject numbers in the range $[2^{255-19}, 2^{255}-1]$, inclusive." but I'm not aware of any implementations that do so.)

Scalars are assumed to be randomly generated bytes. For curve25519, in order to decode 32 bytes into an integer scalar, set the three least significant bits of the first byte and the most significant bit of the last to zero, set the second most significant bit of the last byte to 1 and, finally, decode as little-endian. This means that resulting integer is of the form $2^{254} + 8 * \{0, 1, \dots, 2^{(251)} - 1\}$. Likewise, for curve448, set the two least significant bits of the first byte to 0, and the most significant bit of the last byte to 1. This means that the resulting integer is of the form $2^{447} + 4 * \{0, 1, \dots, 2^{(445)} - 1\}$.

Internet-Draft

cfrgcurve

March 2015

```
def decodeScalar25519(k):
    k_list = [ord(b) for b in k]
    k_list[0] &= 248
    k_list[31] &= 127
    k_list[31] |= 64
    return decodeLittleEndian(k_list,255)

def decodeScalar448(k):
    k_list = [ord(b) for b in k]
    k_list[0] &= 252
    k_list[55] |= 128
    return decodeLittleEndian(k_list,448)
```

To implement the "curve25519(k, u)" and "curve448(k, u)" functions (where "k" is the scalar and "u" is the u-coordinate) first decode "k" and "u" and then perform the following procedure, taken from [[curve25519](#)] and based on formulas from [[montgomery](#)]. All calculations are performed in GF(p), i.e., they are performed modulo p. The constant a24 is $(486662 - 2) / 4 = 121665$ for curve25519, and $(156326 - 2) / 4 = 39081$ for curve448.

Internet-Draft

cfrgcurve

March 2015

```
x_1 = u
x_2 = 1
z_2 = 0
x_3 = u
z_3 = 1
swap = 0

For t = bits-1 down to 0:
    k_t = (k >> t) & 1
    swap ^= k_t
    // Conditional swap; see text below.
    (x_2, x_3) = cswap(swap, x_2, x_3)
    (z_2, z_3) = cswap(swap, z_2, z_3)
    swap = k_t

    A = x_2 + z_2
    AA = A^2
    B = x_2 - z_2
    BB = B^2
    E = AA - BB
    C = x_3 + z_3
    D = x_3 - z_3
    DA = D * A
    CB = C * B
    x_3 = (DA + CB)^2
    z_3 = x_1 * (DA - CB)^2
    x_2 = AA * BB
    z_2 = E * (AA + a24 * E)

// Conditional swap; see text below.
(x_2, x_3) = cswap(swap, x_2, x_3)
(z_2, z_3) = cswap(swap, z_2, z_3)
```

Return $x_2 * (z_2^{(p-2)})$

(TODO: Note the difference in the formula from Montgomery's original paper. See <https://www.ietf.org/mail-archive/web/cfrg/current/msg05872.html>.)

Finally, encode the resulting value as 32 or 56 bytes in little-endian order.

When implementing this procedure, due to the existence of side-channels in commodity hardware, it is important that the pattern of memory accesses and jumps not depend on the values of any of the bits of "k". It is also important that the arithmetic used not leak information about the integers modulo p (such as having $b*c$ be distinguishable from $c*c$).

The cswap instruction SHOULD be implemented in constant time (independent of "swap") as follows:

```
cswap(swap, x_2, x_3):
    dummy = swap * (x_2 - x_3)
    x_2 = x_2 - dummy
    x_3 = x_3 + dummy
    Return (x_2, x_3)
```

where "swap" is 1 or 0. Alternatively, an implementation MAY use the following:

```
cswap(swap, x_2, x_3):
    dummy = mask(swap) AND (x_2 XOR x_3)
    x_2 = x_2 XOR dummy
    x_3 = x_3 XOR dummy
    Return (x_2, x_3)
```

where "mask(swap)" is the all-1 or all-0 word of the same length as x_2 and x_3 , computed, e.g., as $\text{mask}(\text{swap}) = 1 - \text{swap}$. The latter version is often more efficient.

[7.1](#). Test vectors

curve25519:

```

Input scalar:
  a546e36bf0527c9d3b16154b82465edd62144c0ac1fc5a18506a2244ba449ac4
Input scalar as a number (base 10):
  31029842492115040904895560451863089656472772604678260265531221036453811406496
Input U-coordinate:
  e6db6867583030db3594c1a424b15f7c726624ec26b3353b10a903a6d0ab1c4c
Input U-coordinate as a number:
  34426434033919594451155107781188821651316167215306631574996226621102155684838
Output U-coordinate:
  c3da55379de9c6908e94ea4df28d084f32eccf03491c71f754b4075577a28552

Input scalar:
  4b66e9d4d1b4673c5ad22691957d6af5c11b6421e0ea01d42ca4169e7918ba0d
Input scalar as a number (base 10):
  35156891815674817266734212754503633747128614016119564763269015315466259359304
Input U-coordinate:
  e5210f12786811d3f4b7959d0538ae2c31dbe7106fc03c3efc4cd549c715a493
Input U-coordinate as a number:
  8883857351183929894090759386610649319417338800022198945255395922347792736741
Output U-coordinate:
  95cbde9476e8907d7aade45cb4b873f88b595a68799fa152e6f8f7647aac7957

```

curve448:

```

Input scalar:
  3d262fddf9ec8e88495266fea19a34d28882acef045104d0d1aae121
  700a779c984c24f8cdd78fbff44943eba368f54b29259a4f1c600ad3
Input scalar as a number (base 10):
  5991891753738964027837560161452132561572308560850261299268914594686 \
  22403380588640249457727683869421921443004045221642549886377526240828
Input U-coordinate:
  06fce640fa3487bfda5f6cf2d5263f8aad88334cbd07437f020f08f9
  814dc031ddbd38c19c6da2583fa5429db94ada18aa7a7fb4ef8a086
Input U-coordinate as a number:
  3822399108141073301162299612348993770314163652405713251483465559224 \
  38025162094455820962429142971339584360034337310079791515452463053830
Output U-coordinate:
  ce3e4ff95a60dc6697da1db1d85e6afbdf79b50a2412d7546d5f239f
  e14fbaadeb445fc66a01b0779d98223961111e21766282f73dd96b6f

```

Input scalar:

```
203d494428b8399352665ddca42f9de8fef600908e0d461cb021f8c5
38345dd77c3e4806e25f46d3315c44e0a5b4371282dd2c8d5be3095f
```

Input scalar as a number (base 10):

```
6332543359069705927792594815348623723825251552520289610564040013321 \
22152890562527156973881968934311400345568203929409663925541994577184
```

Input U-coordinate:

```
0fbcc2f993cd56d3305b0b7d9e55d4c1a8fb5dbb52f8e9a1e9b6201b
165d015894e56c4d3570bee52fe205e28a78b91cdfbde71ce8d157db
```

Input U-coordinate as a number:

```
6227617977583254444629220684312341806495903900248112997616251537672 \
28042600197997696167956134770744996690267634159427999832340166786063
```

Output U-coordinate:

```
884a02576239ff7a2f2f63b2db6a9ff37047ac13568e1e30fe63c4a7
ad1b3ee3a5700df34321d62077e63633c575c1c954514e99da7c179d
```

8. Diffie-Hellman

The "curve25519" function can be used in an ECDH protocol as follows:

Alice generates 32 random bytes in $f[0]$ to $f[31]$ and transmits $K_A = \text{curve25519}(f, 9)$ to Bob, where 9 is the u-coordinate of the base point and is encoded as a byte with value 9, followed by 31 zero bytes.

Bob similarly generates 32 random bytes in $g[0]$ to $g[31]$ and computes $K_B = \text{curve25519}(g, 9)$ and transmits it to Alice.

Alice computes $\text{curve25519}(f, K_B)$; Bob computes $\text{curve25519}(g, K_A)$ using their generated values and the received input.

Both now share $K = \text{curve25519}(f, \text{curve25519}(g, 9)) = \text{curve25519}(g, \text{curve25519}(f, 9))$ as a shared secret. Both MUST check, without leaking extra information about the value of K , whether K is the all-zero value and abort if so (see below). Alice and Bob can then use a key-derivation function, such as hashing K , to compute a key.

The check for the all-zero value results from the fact that the curve25519 function produces that value if it operates on an input corresponding to a point with order dividing the co-factor, h , of the curve. This check is cheap and so MUST always be carried out. The check may be performed by ORing all the bytes together and checking

whether the result is zero as this eliminates standard side-channels in software implementations.

8.1. Test vectors

curve25519:

Alice's private key, f:

77076d0a7318a57d3c16c17251b26645df4c2f87ebc0992ab177fba51db92c2a

Alice's public key, curve25519(f, 9):

8520f0098930a754748b7ddcb43ef75a0dbf3a0d26381af4eba4a98eaa9b4e6a

Bob's private key, g:

5dab087e624a8a4b79e17f8b83800ee66f3bb1292618b6fd1c2f8b27ff88e0eb

Bob's public key, curve25519(g, 9):

de9edb7d7b7dc1b4d35b61c2ece435373f8343c85b78674dadfc7e146f882b4f

Their shared secret, K:

4a5d9d5ba4ce2de1728e3bf480350f25e07e21c947d19e3376f09b3c1e161742

curve448:

Alice's private key, f:

9a8f4925d1519f5775cf46b04b5800d4ee9ee8bae8bc5565d498c28d

d9c9baf574a9419744897391006382a6f127ab1d9ac2d8c0a598726b

Alice's public key, curve448(f, 5):

9b08f7cc31b7e3e67d22d5aea121074a273bd2b83de09c63faa73d2c

22c5d9bbc836647241d953d40c5b12da88120d53177f80e532c41fa0

Bob's private key, g:

1c306a7ac2a0e2e0990b294470cba339e6453772b075811d8fad0d1d

6927c120bb5ee8972b0d3e21374c9c921b09d1b0366f10b65173992d

Bob's public key, curve448(g, 5):

3eb7a829b0cd20f5bcfc0b599b6feccf6da4627107bdb0d4f345b430

27d8b972fc3e34fb4232a13ca706dcb57aec3dae07bdc1c67bf33609

Their shared secret, K:

fe2d52f1ca113e5441538037dc4a9d4cb381035fb4a990ac50ac4333

63dc072301d1d4f2e82883b35103be96068c11e7c84b8fff74bb6ab0

9. Acknowledgements

This document merges "[draft-black-rpgecc-01](#)" and "[draft-turner-thecurve25519function-01](#)". The following authors of those documents

wrote much of the text and figures but are not listed as authors on this document: Benjamin Black, Joppe W. Bos, Craig Costello, Mike Hamburg, Patrick Longa, Michael Naehrig and Watson Ladd.

The authors would also like to thank Tanja Lange and Rene Struik for their reviews.

10. References

10.1. Normative References

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", [BCP 14](#), [RFC 2119](#), March 1997.

10.2. Informative References

- [AS] Satoh, T. and K. Araki, "Fermat quotients and the polynomial time discrete log algorithm for anomalous elliptic curves", 1998.
- [EBP] ECC Brainpool, "ECC Brainpool Standard Curves and Curve Generation", October 2005, <<http://www.ecc-brainpool.org/download/Domain-parameters.pdf>>.
- [NIST] National Institute of Standards, "Recommended Elliptic Curves for Federal Government Use", July 1999, <<http://csrc.nist.gov/groups/ST/toolkit/documents/dss/NISTReCur.pdf>>.
- [S] Semaev, I., "Evaluation of discrete logarithms on some elliptic curves", 1998.
- [SC] Bernstein, D. and T. Lange, "SafeCurves: choosing safe curves for elliptic-curve cryptography", June 2014, <<http://safecurves.cr.yp.to/>>.
- [SEC1] Certicom Research, "SEC 1: Elliptic Curve Cryptography", September 2000, <http://www.secg.org/collateral/sec1_final.pdf>.
- [Smart] Smart, N., "The discrete logarithm problem on elliptic curves of trace one", 1999.

[curve25519]

Bernstein, D., "Curve25519 -- new Diffie-Hellman speed records", 2006,
<<http://www.iacr.org/cryptodb/archive/2006/PKC/3351/3351.pdf>>.

[ed25519] Bernstein, D., Duif, N., Lange, T., Schwabe, P., and B. Yang, "High-speed high-security signatures", 2011,
<<http://ed25519.cr.yp.to/ed25519-20110926.pdf>>.

[montgomery]

Montgomery, P., "Speeding the Pollard and elliptic curve methods of factorization", 1983,
<<http://www.ams.org/journals/mcom/1987-48-177/S0025-5718-1987-0866113-7/S0025-5718-1987-0866113-7.pdf>>.

Authors' Addresses

Adam Langley
Google
345 Spear St
San Francisco, CA 94105
US

Email: agl@google.com

Rich Salz
Akamai Technologies
8 Cambridge Center
Cambridge, MA 02142
US

Email: rsalz@akamai.com

Sean Turner
IECA, Inc.
3057 Nutley Street
Suite 106
Fairfax, VA 22031
US

Email: turners@ieca.com

