Network Working Group
Internet-Draft
Intended status: Informational
Expires: February 13, 2015

W. Ladd
Grad Student UC Berkley
R. Salz
Akamai Technologies
S. Turner
IECA, Inc.
August 12, 2014

The Curve25519 Function
draft-turner-thecurve25519function-01

Abstract

This document specifies the Curve25519 function, an ECDH (Elliptic-Curve Diffie-Hellman) key-agreement scheme for use in cryptographic applications. It was designed with performance and security in mind. This document is based on information in the public domain.

the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

1. Introduction

This document specifies the Curve25519 function, an ECDH (Elliptic-curve Diffie-Hellman) key-agreement scheme for use in cryptographic applications. It was designed with performance and security in mind. This document is based on information in the public domain.

This document provides a stable reference for the Curve25519 function [Curve25519] to which other specifications may refer when defining their use of Curve25519. It specifies how to use Curve25519 for key exchange. This document defines the algorithm, the "wire format" (how to serialize and parse bytes sent over a network, for example), and provides some implementation guidance to avoid known side-channel timing exposures.

This document does not specify the use of Curve25519 in any other specific protocol, such as TLS (Transport Layer Security) or IPsec (Internet Protocol Security). It does not specify how to use Curve25519 for digital signatures.

Readers are assumed to be familiar with the concepts of elliptic curves, modular arithmetic, group operations, and finite fields [RFC6090] as well as rings [Curve25519].

1.1. Terminology

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 [RFC2119].

2. Notation and Definitions

The following notation and definitions are used in this document (notation is to the left of the ":"):

A: A value used in the elliptic-curve equation E.

E: An elliptic-curve equation.

p: A prime.

GF(p): The field with p elements.

_#: Subscript notation, where # is a number or letter.

Q

=: Assignment.

^: Exponentiation.

+, -, *, /: Addition, subtraction, multiplication, and division, respectively.

Note that all operations are performed modulo p.

3. The Curve25519 Function

Let p = 2^255 - 19. Let E be the elliptic curve with the equation y^2 = x^3 + 486662 * x^2 + x over GF(p).

Each element x of GF(p) has a unique little-endian representation as 32 bytes x[0] ... x[31], such that x[0] + 256 * x[1] + 256^2 * x[2] + ... + 256^31 * x[31] is congruent to x modulo p, and x[31] is minimal. Implementations MUST only produce points in this form. On receiving a point, implementations MUST mask the leftmost bit of byte 31 to zero. This is done to preserve compatibility with point formats which reserve the sign bit for use in other protocols and increase resistance to implementation fingerprinting. Implementations MUST reject numbers in the range [2^255-19, 2^255-1], inclusive.

Let X denote the projection map from a point (x,y) on E, to x, extended so that X of the point at infinity is zero. X is surjective onto GF(p) if the y coordinate takes on values in GF(p) and in a quadratic extension of GF(p).

Then Curve25519(s, X(Q)) = X(sQ) is a function defined for all integers s and elements X(Q) of GF(p). Proper implementations use a restricted set of integers for s and only x-coordinates of points Q defined over GF(p). The remainder of this document describes how to compute this function quickly and securely, and use it in a Diffie-Hellman scheme.

4. Implementing the Curve25519 Function

Let s be a 255 bits long integer, where s = sum s_i * 2^i with s_i in {0, 1}.

Computing Curve25519(s, x) is done by the following procedure, taken from [Curve25519] based on formulas from [Mont]. All calculations are performed in GF(p), i.e., they are performed modulo p. The parameter a24 is a24 = (486662 - 2) / 4 = 121665.

```
    x_1 = x
    x_2 = 0
    z_2 = 1
    x_3 = x
    z_3 = 1
    For t = 254 down to 0:
        // Conditional swap; see text below.
    (x_2, x_3) = cswap (s_t, x_2, x_3)
    (z_2, z_3) = cswap (s_t, z_2, z_3)
        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 (s_t, x_2, x_3)
    (z_2, z_3) = cswap (s_t, z_2, z_3)
    Return x_2 * (z_2^(p - 1))
```

In 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 s. It is also important that the arithmetic used not leak information about the integers modulo p (such as having b * c distinguishable from c * c).

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

cswap(s_t, x_2, x_3) dummy = s_t * (x_2 - x_3) x_2 = x_2 - dummy x_3 = x_3 + dummy Return (x_2, x_3)

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

```
  dummy = mask(s_t) AND (x_2 XOR x_3)
  x_2 = x_2 XOR dummy
  x_3 = x_3 XOR dummy
```

where mask(s_t) is the all-1 or all-0 word of the same length as x_2 and x_3, computed, e.g., as mask(s_t) = 1 - s_t. The latter version is often more efficient.

5. Use of the Curve25519 function

The Curve25519 function can be used in an ECDH protocol as follows:

Alice generates 32 random bytes in f[0] to f[31]. She masks the three rightmost bits of f[0] and the leftmost bit of f[31] to zero and sets the second leftmost bit of f[31] to 1. This means that f is of the form 2^254 + 8 * {0, 1, ..., 2^(251) - 1} as a little-endian integer.

Alice then transmits K_A = Curve25519(f, 9) to Bob, where 9 is the number 9.

Bob similarly generates 32 random bytes in g[0] to g[31], applies the same masks, computes K_B = Curve25519(g, 9) and transmits it to Alice.

Alice computes Curve25519(f, Curve25519(g, 9)); Bob computes Curve25519(g, Curve25519(f, 9)) using their generated values and the received input.

Both of them now share K = Curve25519(f, Curve25519(g, 9)) = Curve25519(g, Curve25519(f, 9)) as a shared secret. Alice and Bob can then use a key-derivation function, such as hashing K, to compute a key.

6. Test Vectors

The following test vectors are taken from [NaCl]. All numbers are shown as little-endian hexadecimal byte strings:

Alice's private key, f:

```
 77 07 6d 0a 73 18 a5 7d 3c 16 c1 72 51 b2 66 45
 df 4c 2f 87 eb c0 99 2a b1 77 fb a5 1d b9 2c 2a
```

Alice's public key, Curve25519(f, 9):

```
 85 20 f0 09 89 30 a7 54 74 8b 7d dc b4 3e f7 5a
 0d bf 3a 0d 26 38 1a f4 eb a4 a9 8e aa 9b 4e 6a
```

Bob's private key, g:

```
 5d ab 08 7e 62 4a 8a 4b 79 e1 7f 8b 83 80 0e e6
 6f 3b b1 29 26 18 b6 fd 1c 2f 8b 27 ff 88 e0 eb
```

Bob's public key, Curve25519(g, 9):

```
 de 9e db 7d 7b 7d c1 b4 d3 5b 61 c2 ec e4 35 37
 3f 83 43 c8 5b 78 67 4d ad fc 7e 14 6f 88 2b 4f
```

Their shared secret, K:

```
 4a 5d 9d 5b a4 ce 2d e1 72 8e 3b f4 80 35 0f 25
 e0 7e 21 c9 47 d1 9e 33 76 f0 9b 3c 1e 16 17 42
```

7. Security Considerations

Curve25519 meets all standard assumptions on DH and DLP difficulty.

In addition, Curve25519 is twist secure: the co-factor of the curve is 8, that of the twist is 4. Protocols that require contributory behavior must ban outputs K_A = 0, K_B = 0 or K = 0.

Curve25519 is designed to enable very high performance software implementations, thus reducing the cost of highly secure cryptography to a point where it can be used more widely.

8. IANA Considerations

None.

9. Acknowledgements

We would like to thank Tanja Lange (Technische Universiteit Eindhoven) for her review and comments.

10. References

10.1. Normative References

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

[Mont] 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>.

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

[RFC6090] McGrew, D., Igoe, K., and M. Salter, "Fundamental Elliptic Curve Cryptography Algorithms", RFC 6090, February 2011.

10.2. Informative References

[NaCl] Bernstein, D., "Cryptography in NaCl", 2013, <http://cr.yp.to/highspeed/naclcrypto-20090310.pdf>.

Authors' Addresses

Watson Ladd
Grad Student UC Berkley

Email: watsonbladd@gmail.com

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

Phone: +1-617-714-6169
Email: rsalz@akamai.com

Sean Turner
IECA, Inc.
Suite 106
Fairfax, VA 22031
USA

Phone: +1-703-628-3180
Email: turners@ieca.com