

Network Working Group
Internet-Draft
Updates: [4492](#), [5246](#) (if approved)
Intended status: Informational
Expires: January 7, 2016

S. Josefsson
SJD AB
M. Pegourie-Gonnard
Independent / PolarSSL
July 6, 2015

Curve25519 and Curve448 for Transport Layer Security (TLS)
draft-ietf-tls-curve25519-01

Abstract

This document specifies the use of Curve25519 and Curve448 for ephemeral key exchange in the Transport Layer Security (TLS) and Datagram TLS (DTLS) protocols. It updates [RFC 5246](#) (TLS 1.2) and [RFC 4492](#) (Elliptic Curve Cryptography for TLS).

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 January 7, 2016.

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 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.

1. Introduction

In [[Curve25519](#)], a new elliptic curve function for use in cryptographic applications was introduced. In [[Ed448-Goldilocks](#)] the Ed448-Goldilocks curve is described. In [[I-D.irtf-cfrg-curves](#)], the Diffie-Hellman functions Curve25519 and Curve448 (using Ed448-Goldilocks curve) are specified.

[RFC4492] defines the usage of elliptic curves for authentication and key agreement in TLS 1.0 and TLS 1.1, and these mechanisms are also applicable to TLS 1.2 [[RFC5246](#)]. The use of ECC curves for key exchange requires the definition and assignment of additional NamedCurve values. This document specifies them for Curve25519 and Curve448, and describes how the values are used to implement key agreement in (D)TLS using these cryptographic primitives.

This document only describes usage of Curve25519 and Curve448 for ephemeral key exchange (ECDHE) in (D)TLS. It does not define its use for signatures, since the primitive considered here is a Diffie-Hellman function; the related signature scheme, EddSA [[I-D.josefsson-eddsa-ed25519](#)], and how it is used in TLS/PKIX, is outside the scope of this document. The use of Curve25519 and Curve448 with long-term keys embedded in X.509 certificates is also out of scope here, but see [[I-D.josefsson-pkix-newcurves](#)].

1.1. Requirements 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. Data Structures and Computations

2.1. Cryptographic computations

All cryptographic computations are done using the Curve25519 and Curve448 functions defined in [[I-D.irtf-cfrg-curves](#)]. In this memo, these functions are considered as a black box that takes as input a (secret key, public key) pair and outputs a public key. Public keys are defined as strings of 32 bytes for Curve25519 and 56 bytes for Curve448. Secret keys are encoded as described in [[I-D.irtf-cfrg-curves](#)]. In addition, a common public key, denoted by G , is shared by all users.

An ECDHE key exchange using Curve25519 goes as follows. Each party picks a secret key d uniformly at random and computes the corresponding public key $x = \text{Curve25519}(d, G)$. Parties exchange their public keys and compute a shared secret as $x_S = \text{Curve25519}(d,$

x_{peer}). ECDHE for Curve448 works similarly, replacing Curve25519 with Curve448. The derived shared secret is used directly as the premaster secret, which is always exactly 32 bytes when ECDHE with Curve25519 is used and 56 bytes when ECDHE with Curve448 is used.

A complete description of the Curve25519 function, as well as a few implementation notes, are provided in [Appendix A](#).

[2.2](#). Curve negotiation and new NamedCurve value

Curve negotiation uses the mechanisms introduced by [[RFC4492](#)], without modification except the following restriction: in the ECPParameters structure, only the named_curve case can be used with Curve25519 or Curve448. Accordingly, arbitrary_explicit_prime_curves in the Supported Curves extension does not imply support for Curve25519 or Curve448, even though the functions happens to be defined using an elliptic curve over a prime field.

The reason for this restriction is that explicit_prime is only suited to the so-called Short Weierstrass representation of elliptic curves, while Curve25519 and Curve448 uses a different representation for performance and security reasons.

This document adds a new NamedCurve value for Curve25519 and Curve448 as follows.

```
enum {
    Curve25519(TBD1),
    Curve448(TBD2),
} NamedCurve;
```

Curve25519 and Curve448 are suitable for use with DTLS [[RFC6347](#)].

Since Curve25519 and Curve448 are Diffie-Hellman functions, and not applicable as signatures algorithms, clients who offer ECDHE_ECDSA ciphersuites and advertise support for Curve25519/Curve448 in the

elliptic_curves ClientHello extension SHOULD also advertise support for at least one curve suitable for ECDSA signatures. Servers MUST NOT select an ECDSA certificate if there are no common curves suitable for ECDSA signing.

The public-key format for Curve25519 and Curve448 are defined in [[I-D.irtf-cfrg-curves](#)], and in TLS the ECPointFormat enumeration "uncompressed" is used.

[2.3.](#) Public key validation

With the curves defined by [[RFC4492](#)], each party must validate the public key sent by its peer before performing cryptographic computations with it. Failing to do so allows attackers to gain information about the private key, to the point that they may recover the entire private key in a few requests, if that key is not really ephemeral.

Curve25519 was designed in a way that the result of Curve25519(x, d) will never reveal information about d, provided it was chosen as prescribed, for any value of x.

Let's define legitimate values of x as the values that can be obtained as $x = \text{Curve25519}(G, d')$ for some d, and call the other values illegitimate. The definition of the Curve25519 function shows that legitimate values all share the following property: the high-order bit of the last byte is not set.

Since there are some implementations of the Curve25519 function that impose this restriction on their input and others that don't, implementations of Curve25519 in TLS SHOULD reject public keys when the high-order bit of the last byte is set (in other words, when the value of the leftmost byte is greater than 0x7F) in order to prevent implementation fingerprinting.

Other than this recommended check, implementations do not need to ensure that the public keys they receive are legitimate: this is not necessary for security with Curve25519.

3. IANA Considerations

IANA is requested to assign numbers for Curve25519 and Curve448 listed in [Section 2.2](#) to the Transport Layer Security (TLS) Parameters registry EC Named Curve [[IANA-TLS](#)] as follows.

Value	Description	DTLS-OK	Reference
TBD1	Curve25519	Y	This doc
TBD2	Curve448	Y	This doc

4. Security Considerations

The security considerations of [[RFC5246](#)] and most of the security considerations of [[RFC4492](#)] apply accordingly. For the Curve25519 and Curve448 primitives, the considerations in [[I-D.irtf-cfrg-curves](#)] apply.

Curve25519 is designed to facilitate the production of high-performance constant-time implementations of the Curve25519 function. Implementors are encouraged to use a constant-time implementation of the Curve25519 and Curve448 functions. This point is of crucial importance if the implementation chooses to reuse its supposedly ephemeral key pair for many key exchanges, which some implementations do in order to improve performance.

Curve25519 is believed to be at least as secure as the secp256r1 curve defined in [[RFC4492](#)], also known as NIST P-256. While the NIST curves are advertised as being chosen verifiably at random, there is no explanation for the seeds used to generate them. In contrast, the process used to pick Curve25519 is fully documented and rigid enough so that independent verification has been done. This is widely seen as a security advantage for Curve25519, since it prevents the

generating party from maliciously manipulating the parameters.

Another family of curves available in TLS, generated in a fully verifiable way, is the Brainpool curves [[RFC7027](#)]. Specifically, brainpoolP256 is expected to provide a level of security comparable to Curve25519 and NIST P-256. However, due to the use of pseudo-random prime, it is significantly slower than NIST P-256, which is itself slower than Curve25519.

See [[SafeCurves](#)] for more comparisons between elliptic curves.

[5.](#) Acknowledgements

Several people provided comments and suggestions that helped improve this document: Kurt Roeckx, Andrey Jivsov, Robert Ransom, Rich Salz, David McGrew, Simon Huerlimann, Ilari Liusvaara, Eric Rescorla, Martin Thomson.

[6.](#) References

[6.1.](#) Normative References

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

[RFC4492] Blake-Wilson, S., Bolyard, N., Gupta, V., Hawk, C., and B. Moeller, "Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS)", [RFC 4492](#), May 2006.

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", [RFC 5246](#), August 2008.

[RFC6347] Rescorla, E. and N. Modadugu, "Datagram Transport Layer Security Version 1.2", [RFC 6347](#), January 2012.

[I-D.irtf-cfrg-curves]

Langley, A., Salz, R., and S. Turner, "Elliptic Curves for Security", [draft-irtf-cfrg-curves-01](#) (work in progress), January 2015.

6.2. Informative References

- [Curve25519]
Bernstein, J., "Curve25519: New Diffie-Hellman Speed Records", LNCS 3958, pp. 207-228, February 2006, <http://dx.doi.org/10.1007/11745853_14>.
- [Ed448-Goldilocks]
Hamburg, , "Ed448-Goldilocks, a new elliptic curve", June 2015, <<https://eprint.iacr.org/2015/625>>.
- [IANA-TLS]
Internet Assigned Numbers Authority, "Transport Layer Security (TLS) Parameters", <<http://www.iana.org/assignments/tls-parameters/tls-parameters.xml>>.
- [SafeCurves]
Bernstein, D. and T. Lange, "SafeCurves: choosing safe curves for elliptic-curve cryptography.", January 2014, <<http://safecurves.cr.yp.to/>>.
- [RFC7027] Merkle, J. and M. Lochter, "Elliptic Curve Cryptography (ECC) Brainpool Curves for Transport Layer Security (TLS)", [RFC 7027](#), October 2013.
- [EFD] Bernstein, D. and T. Lange, "Explicit-Formulas Database: XZ coordinates for Montgomery curves", January 2014, <<http://www.hyperelliptic.org/EFD/g1p/auto-montgom-xz.html>>.
- [NaCl] Bernstein, D., "Cryptography in NaCl", March 2013, <<http://cr.yp.to/highspeed/naclcrypto-20090310.pdf>>.

- [I-D.josefsson-pkix-newcurves]
Josefsson, S., "Using Curve25519 and Curve448 Public Keys in PKIX", [draft-josefsson-pkix-newcurves-00](#) (work in progress), June 2015.
- [I-D.josefsson-eddsa-ed25519]
Josefsson, S. and N. Moller, "EdDSA and Ed25519", [draft-josefsson-eddsa-ed25519-02](#) (work in progress), February

2015.

[Appendix A](#). The curve25519 function

[A.1](#). Formulas

This section completes [Section 2.1](#) by defining the Curve25519 function and the common public key G. It is meant as an alternative, self-contained specification for the Curve25519 function, possibly easier to follow than the original paper for most implementors.

[A.1.1](#). Field Arithmetic

Throughout this section, P denotes the integer $2^{255}-19 = 0x7FFED$. The letters X and Z , and their numbered variants such as x_1 , z_2 , etc. denote integers modulo P , that is integers between 0 and $P-1$ and every operation between them is implicitly done modulo P . For addition, subtraction and multiplication this means doing the operation in the usual way and then replacing the result with the remainder of its division by P . For division, " X / Z " means multiplying (mod P) X by the modular inverse of Z mod P .

A convenient way to define the modular inverse of Z mod P is as $Z^{(P-2)} \text{ mod } P$, that is Z to the power of $2^{255}-21 \text{ mod } P$. It is also a practical way of computing it, using a square-and-multiply method.

The four operations $+$, $-$, $*$, $/$ modulo P are known as the field operations. Techniques for efficient implementation of the field operations are outside the scope of this document.

[A.1.2](#). Conversion to and from internal format

For the purpose of this section, we will define a Curve25519 point as a pair (X, Z) where X and Z are integers mod P (as defined above). Though public keys were defined to be strings of 32 bytes, internally they are represented as curve points. This subsection describes the conversion process as two functions: `PubkeyToPoint` and `PointToPubkey`.

Input: a public key b_0, \dots, b_{31}
Output: a Curve25519 point (X, Z)

1. Set $X = b_0 + 256 * b_1 + \dots + 256^{31} * b_{31} \bmod P$
2. Set $Z = 1$
3. Output (X, Z)

PointToPubkey:

Input: a Curve25519 point (X, Z)
Output: a public key b_0, \dots, b_{31}

1. Set $x1 = X / Z \bmod P$
2. Set b_0, \dots, b_{31} such that
 $x1 = b_0 + 256 * b_1 + \dots + 256^{31} * b_{31} \bmod P$
3. Output b_0, \dots, b_{31}

A.1.3. Scalar Multiplication

We first introduce the DoubleAndAdd function, defined as follows (formulas taken from [EFD]).

DoubleAndAdd:

Input: two points $(X2, Z2), (X3, Z3)$, and an integer mod P : $X1$
Output: two points $(X4, Z4), (X5, Z5)$

Constant: the integer mod P : $a24 = 121666 = 0x01DB42$

Variables: $A, AA, B, BB, E, C, D, DA, CB$ are integers mod P

1. Do the following computations mod P :

$$\begin{aligned} A &= X2 + Z2 \\ AA &= A^2 \\ B &= X2 - Z2 \\ BB &= B^2 \\ E &= AA - BB \\ C &= X3 + Z3 \\ D &= X3 - Z3 \\ DA &= D * A \\ CB &= C * B \\ X5 &= (DA + CB)^2 \\ Z5 &= X1 * (DA - CB)^2 \\ X4 &= AA * BB \\ Z4 &= E * (BB + a24 * E) \end{aligned}$$

2. Output $(X4, Z4)$ and $(X5, Z5)$

This may be taken as the abstract definition of an arbitrary-looking function. However, let's mention "the true meaning" of this function, without justification, in order to help the reader make more sense of it. It is possible to define operations "+" and "-" between Curve25519 points. Then, assuming $(X2, Z2) - (X3, Z3) = (X1, 1)$, the DoubleAndAdd function returns points such that $(X4, Z4) = (X2, Z2) + (X2, Z2)$ and $(X5, Z5) = (X2, Z2) + (X3, Z3)$.

Taking the "+" operation as granted, we can define multiplication of a Curve25519 point by a positive integer as $N * (X, Z) = (X, Z) + \dots + (X, Z)$, with N point additions. It is possible to compute this operation, known as scalar multiplication, using an algorithm called the Montgomery ladder, as follows.

ScalarMult:

Input: a Curve25519 point: $(X, 1)$ and a 255-bits integer: N

Output: a point $(X1, Z1)$

Variable: a point $(X2, Z2)$

0. View N as a sequence of bits b_{254}, \dots, b_0 , with b_{254} the most significant bit and b_0 the least significant bit.
1. Set $X1 = 1$ and $Z1 = 0$
2. Set $X2 = X$ and $Z2 = 1$
3. For i from 254 downwards to 0, do:
 - If $b_i == 0$, then:
 - Set $(X2, Z2)$ and $(X1, Z1)$ to the output of `DoubleAndAdd((X2, Z2), (X1, Z1), X)`
 - else:
 - Set $(X1, Z1)$ and $(X2, Z2)$ to the output of `DoubleAndAdd((X1, Z1), (X2, Z2), X)`
4. Output $(X1, Z1)$

[A.1.4.](#) Conclusion

We are now ready to define the Curve25519 function itself.

Curve25519:

Input: a public key P and a secret key S

Output: a public key Q

Variables: two Curve25519 points (X, Z) and $(X1, Z1)$

1. Set $(X, Z) = \text{PubkeyToPoint}(P)$
2. Set $(X1, Z1) = \text{ScalarMult}((X, Z), S)$
3. Set $Q = \text{PointToPubkey}((X1, Z1))$
4. Output Q

The common public key G mentioned in the first paragraph of [Section 2.1](#) is defined as $G = \text{PointToPubkey}((9, 1))$.

[A.2.](#) Test vectors

The following test vectors are taken from [\[NaCl\]](#). Compared to this reference, the private key strings have been applied the `ClampC` function of the reference and converted to integers in order to fit the description given in [\[Curve25519\]](#) and the present memo.

The secret key of party A is denoted by S_a , its public key by P_a , and similarly for party B. The shared secret is SS .

```
S_a = 0x6A2CB91DA5FB77B12A99C0EB872F4CDF
      4566B25172C1163C7DA518730A6D0770
```

```
P_a = 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
```

```
S_b = 0x6BE088FF278B2F1CFDB6182629B13B6F
      E60E80838B7FE1794B8A4A627E08AB58
```

```
P_b = 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
```

```
SS = 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
```

[A.3.](#) Side-channel considerations

Curve25519 was specifically designed so that correct, fast, constant-time implementations are easier to produce. In particular, using a Montgomery ladder as described in the previous section ensures that, for any valid value of the secret key, the same sequence of field operations are performed, which eliminates a major source of side-channel leakage.

However, merely using Curve25519 with a Montgomery ladder does not prevent all side-channels by itself, and some point are the responsibility of implementors:

1. In step 3 of `SclarMult`, avoid branches depending on b_i , as well as memory access patterns depending on b_i , for example by using safe conditional swaps on the inputs and outputs of `DoubleAndAdd`.
2. Avoid data-dependant branches and memory access patterns in the implementation of field operations.

Techniques for implementing the field operations in constant time and

with high performance are out of scope of this document. Let's mention however that, provided constant-time multiplication is available, division can be computed in constant time using exponentiation as described in [Appendix A.1.1](#).

If using constant-time implementations of the field operations is not convenient, an option to reduce the information leaked this way is to replace step 2 of the SclarMult function with:

Josefsson & Pegourie-GonnExpires January 7, 2016

[Page 10]

Internet-Draft Curve25519 and Ed448-Goldilocks for TLS

July 2015

- 2a. Pick Z uniformly randomly between 1 and P-1 included
- 2b. Set $X2 = X * Z$ and $Z2 = Z$

This method is known as randomizing projective coordinates. However, it is no guaranteed to avoid all side-channel leaks related to field operations.

Side-channel attacks are an active research domain that still sees new significant results, so implementors of the Curve25519 function are advised to follow recent security research closely.

Authors' Addresses

Simon Josefsson
SJD AB

Email: simon@josefsson.org

Manuel Pegourie-Gonnard
Independent / PolarSSL

Email: mpg@elzevir.fr

