The XEdDSA and VXEdDSA Signature Schemes

Trevor Perrin (editor)

Revision 1, 2016-10-20

Contents

1. Introduction

This document describes how to create and verify EdDSA-compatible signatures using public key and private key formats initially defined for the X25519 and X448 elliptic curve Diffie-Hellman functions ([1]–[4]). We refer to this as the "XEdDSA" signature scheme (or "XEd25519" or "XEd448" for specific curves).

XEdDSA enables use of a single key pair format for both elliptic curve Diffie-Hellman and signatures. In some situations it enables using the same key pair for both algorithms.

This document also describes "VXEdDSA" which extends XEdDSA to make it a *verifiable random function*, or VRF (as defined in [5] and [6]). Successful verification of a VXEdDSA signature returns a VRF output which is guaranteed to be unique for the message and public key. The VRF output for a given message and public key is indistinguishable from random to anyone who has not seen a VXEdDSA signature for that message and key.

2. Preliminaries

2.1. Notation

Multiplication of integers a and b modulo prime p is $ab \pmod p$, or $a * b \pmod p$. Division is $a/b \pmod p$ and is calculated as $ab^{-1} \pmod p$. We define $inv(a) \pmod p$ to return $a^{-1} \pmod p$ when a is not 0, and return 0 when a is 0. This may be calculated as $inv(a) = a^{p-2} \pmod p$, per Fermat [7]. The *ceil()* and *floor()* functions round up or down to the nearest integer.

Addition and subtraction of elliptic curve points A and B is $A + B$ and $A - B$. Scalar multiplication of an integer a with an elliptic curve point B yields a new point $C = aB$.

Integer variables are in lower case (x, y). Points and other variables are in upper case (P, Q). Integer constants are in lowercase except for the Montgomery curve constant A, which follows convention.

Byte sequences are in bold ($\boldsymbol{x}$, $\boldsymbol{P}$). A bold integer or elliptic curve point represents a fixed-length byte sequence encoding the value. See Section 2.4 for encoding and decoding details. The concatenation of byte sequences $\boldsymbol{x}$ and $\boldsymbol{P}$ is $\boldsymbol{x} \,||\, \boldsymbol{P}$.

Checking integers or points for equality is done with $a == b$. Checking byte sequences $\boldsymbol{X}$ and $\boldsymbol{Y}$ for equality is done with *bytes_equal*($\boldsymbol{X}$, $\boldsymbol{Y}$).

2.2. Elliptic curve parameters

An elliptic curve used with XEdDSA or VXEdDSDA has the following parameters:

Name	Definition
B	Base point
I	Identity point
p	Field prime
q	Order of base point (prime; q < p; qB = I)
c	Cofactor
d	Twisted Edwards curve constant
A	Montgomery curve constant (see Section 2.6)
n	Nonsquare integer modulo p (see Section 2.6)
$\|p\|$	ceil($\log_2$(p))
$\|q\|$	ceil($\log_2$(q))
b	8 * (ceil((|p| + 1)/8)) (= bitlength for encoded point or integer)

An integer modulo p is a "field element". An integer modulo q is a "scalar".

An elliptic curve is a set of pairs of field elements. Each pair is a "point", the field elements contained in a point are "coordinates", and the coordinates of each point must satisfy some equation which defines the curve. The *on_curve(P)* function returns true if a purported point P satisfies the curve equation.

The elliptic curve also defines an addition operation between points, and an operation for negating points. Together with the identity point, these operations define a group structure on the curve's points. Adding a point P to itself k times $(P + P + P + \dots)$ is scalar multiplication by the scalar k, represented as kP.

XEdDSA and VXEdDSA are defined for twisted Edwards curves consisting of points denoted (x, y). A twisted Edwards curve is *birationally equivalent* to some Montgomery curve consisting of points denoted (u, v) [8]. We will mainly deal with the twisted Edwards curve, so when we discuss the base point B and identity point I, we are referring to points on the twisted Edwards curve.

The *u_to_y* function applies a curve-specific birational map to convert the u-coordinate of a point on the Montgomery curve to the y-coordinate of the equivalent point on the twisted Edwards curve.

2.3. Elliptic curve conversions

Elliptic curve Diffie-Hellman is often calculated using the Montgomery ladder. This gives a simple and efficient calculation that is naturally resistant to timing side channels. The Montgomery ladder also allows each party's public key to

be a Montgomery u-coordinate. Using a single coordinate instead of the whole point makes public keys smaller without the expense of point decompression.

However, EdDSA signatures are defined on twisted Edwards curves, where a public key is a compressed point consisting of a twisted Edwards y-coordinate and a sign bit s which is either 0 or 1. A twisted Edwards y-coordinate and sign bit provide an alternate representation of a twisted Edwards point, and determine the x-coordinate as specified in [1] or [2].

Converting from a Montgomery u-coordinate to a twisted Edwards point P can be done with the *convert_mont* function, below. This function first masks off excess high bits from u, which is standard practice for Curve25519 Montgomery public keys, and is specified in [4]. The function then applies the curve-specific birational map to compute a twisted Ewards y-coordinate, and finally chooses the sign bit as zero.

```
convert_mont(u):
    u_masked = u (mod 2^|p|)
    P.y = u_to_y(u_masked)
    P.s = 0
    return P
```

Because *convert_mont* doesn't have the Montgomery v, it can't distinguish between the two possibilities for the twisted Edwards sign bit. Forcing the sign bit to zero is an idea from Jivsov [9].

To make private keys compatible with this conversion, we define a twisted Edwards private key as a scalar a where the twisted Edwards public key $A = aB$ has a sign bit of zero (recall that B is the twisted Edwards base point). We allow a Montgomery private key to be any scalar.

Converting a Montgomery private key k to a twisted Edwards public key and private key (A, a) can be done with the *calculate_key_pair* function ("A" here is the public key, not the Montgomery curve constant). This function multiplies the Montgomery private key k by the twisted Edwards base point B, then adjusts the private key if necessary to produce a sign bit of zero, following [9].

```
calculate_key_pair(k):
    E = kB
    A.y = E.y
    A.s = 0
    if E.s == 1:
        a = -k (mod q)
    else:
        a = k (mod q)
    return A, a
```

2.4. Byte sequences

An integer in bold represents a byte sequence of b bits that encodes the integer in little-endian form. An elliptic curve point in bold (e.g. $\boldsymbol{P}$) encodes $\boldsymbol{P.y}$ as an integer in little-endian form of $b\text{-}1$ bits in length, followed by a bit for $\boldsymbol{P.s}$.

2.5. Hash functions

XEdDSA and VXEdDSA require a cryptographic hash function. The default hash function is SHA-512 [10].

We define *hash* as a function that applies the cryptographic hash to an input byte sequence, and returns an integer which is the output from the cryptographic hash parsed in little-endian form. Given *hash* and the curve constants p and b, we define a family of hash functions indexed by nonnegative integers i such that $2^{|p|} - 1 - i > p$.

$\text{hash}_i(\mathbf{X})$:
  return hash($\mathbf{2^b - 1 - i}$ || $\mathbf{X}$)

So $hash_0$ hashes $b/8$ bytes of *0xFF* prior to the input byte sequence $\mathbf{X}$, $hash_1$ changes the first byte to *0xFE*, $hash_2$ changes the first byte to *0xFD*, and so on.

Different $hash_i$ will be used for different purposes, to provide cryptographic domain separation. Note that $hash_i$ will never call *hash* with the first b bits encoding a valid scalar or elliptic curve point, since the first $|p|$ bits encode an integer greater than p. Note also that $hash_0$ is reserved for use by other specifications, and is not used in this document.

2.6. Hashing to a point with Elligator 2

VXEdDSA requires mapping an input message to an elliptic curve point, which is done using the Elligator 2 map [11].

The description in [11] is terse and uses different notation, so we briefly review Elligator 2. The Montgomery curve equation for points (u, v) is $v^2 = u(u^2 + Au + 1) \pmod p$, where A is some curve-specific constant. Elligator 2 maps an integer r onto some u for which $u(u^2 + Au + 1)$ has a square root v modulo p. The following lemma is used.

Lemma (Bernstein, Hamburg, Krasnova, Lange). If u_1 and u_2 are integers modulo p such that $u_2 = -A - u_1$ and $u_2/u_1 = nr^2$ for any r and fixed nonsquare n, then the Montgomery curve equation $v^2 = u(u^2 + Au + 1)$ has a solution for $u = u_1$ or $u = u_2$, or both.

Proof. Given u_1 and u_2, define $w_1 = u_1({u_1}^2 + Au_1 + 1)$ and $w_2 = u_2({u_2}^2 + Au_2 + 1)$. We must prove that a solution exists for either $v^2 = w_1$ or $v^2 = w_2$.

We will show that w_2/w_1 is either zero or nonsquare, which implies either that w_2 is zero, which is square, or that one of w_2 or w_1 is square.

$$w_2/w_1 = u_2(u_2^2 + Au_2 + 1) \,/\, u_1(u_1^2 + Au_1 + 1)$$

$$w_2/w_1 = (u_2/u_1)\,(u_2^2 + Au_2 + 1) \,/\, (u_1^2 + Au_1 + 1)$$

Applying $u_2 = -A - u_1$ gives:

$$w_2/w_1 = u_2/u_1$$

Applying $u_2/u_1 = nr^2$ gives:

$$w_2/w_1 = nr^2$$

If r is zero then w_2 must be zero, which is square. If r is nonzero then since r^2 is square and n is nonsquare, w_2/w_1 is nonsquare, which implies one of w_2 or w_1 is square, so the proof is concluded. □

From the lemma it follows that $u_1 = -A/(1 + nr^2)$ and $u_2 = -Anr^2/(1 + nr^2)$. Thus given r, we can easily calculate u_1 and u_2 and use the Legendre symbol [12] to choose whichever value gives a square w. The *elligator2* function implements this map from an integer r to an integer u.

```
elligator2(r):
    u1 = -A * inv(1 + nr^2) (mod p)
    w1 = u1(u1^2 + Au1 + 1) (mod p)
    if w1^((p-1)/2) == -1 (mod p):
        u2 = -A - u1 (mod p)
        return u2
    return u1
```

(The *inv* function is used safely since calling *inv(0)* when $r^2 = -1/n$ will simply map r onto $u=0$, a valid Montgomery u-coordinate.)

To map a byte sequence onto an Edwards point, we hash the byte sequence and parse the hash output to get a field element r and a sign bit s. Elligator 2 converts r to a Montgomery u-coordinate. The birational map converts the Montgomery u-coordinate to an Edwards point. Finally, we multiply the Edwards point by the cofactor c to ensure it lies within the order q subgroup generated by the base point B. The *hash_to_point* function implements these steps.

```
hash_to_point(X):
    h = hash2(X)
    r = h (mod 2^|p|)
    s = floor((h mod 2^b) / 2^(b-1))
    u = elligator2(r)
    P.y = u_to_y(u)
    P.s = s
    return cP
```

3. XEdDSA

The XEdDSA signing algorithm requires the following inputs:

Name	Definition
k	Montgomery private key (integer mod q)
$\boldsymbol{M}$	Message to sign (byte sequence)
$\boldsymbol{Z}$	64 bytes secure random data (byte sequence)

The output is a signature ($\boldsymbol{R} \,||\, \boldsymbol{s}$), a byte sequence of length $2b$ bits, where $\boldsymbol{R}$ encodes a point and $\boldsymbol{s}$ encodes an integer modulo q.

The XEdDSA verification algorithm requires the following inputs:

Name	Definition
$\boldsymbol{u}$	Montgomery public key (byte sequence of b bits)
$\boldsymbol{M}$	Message to verify (byte sequence)
$\boldsymbol{R} \,||\, \boldsymbol{s}$	Signature to verify (byte sequence of 2b bits)

If XEdDSA verification is successful it returns true, otherwise it returns false. Below is the pseudocode for the *xeddsa_sign* and *xeddsa_verify* functions.

```
xeddsa_sign(k, M, Z):
    A, a = calculate_key_pair(k)
    r = hash_1(a || M || Z) (mod q)
    R = rB
    h = hash(R || A || M) (mod q)
    s = r + ha (mod q)
    return R || s


xeddsa_verify(u, M, (R || s)):
    if u >= p or R.y >= 2^|p| or s >= 2^|q|:
        return false
    A = convert_mont(u)
    if not on_curve(A):
        return false
    h = hash(R || A || M) (mod q)
    R_check = sB - hA
    if bytes_equal(R, R_check):
        return true
    return false
```

4. VXEdDSA

The VXEdDSA signing algorithm takes the same inputs as XEdDSA. The output is a pair of values. First, a signature ($\boldsymbol{V} \mid\mid \boldsymbol{h} \mid\mid \boldsymbol{s}$), which is a byte sequence of length $3b$ bits, where $\boldsymbol{V}$ encodes a point and $\boldsymbol{h}$ and $\boldsymbol{s}$ encode integers modulo q. Second, a VRF output byte sequence $\boldsymbol{v}$ of length equal to b bits, formed by multiplying the V output by the cofactor c.

The VXEdDSA verification algorithm takes the same inputs as XEdDSA, except with a VXEdDSA signature instead of an XEdDSA signature. If VXEdDSA verification is successful, it returns a VRF output byte sequence $\boldsymbol{v}$ of length equal to b bits; otherwise it returns false.

Below is the pseudocode for the *vxeddsa_sign* and *vxeddsa_verify* functions.

```
vxeddsa_sign(k, M, Z):
    A, a = calculate_key_pair(k)
    B_v = hash_to_point(A || M)
    V = aB_v
    r = hash_3(a || V || Z) (mod q)
    R = rB
    R_v = rB_v
    h = hash_4(A || V || R || R_v || M) (mod q)
    s = r + ha (mod q)
    v = hash_5(cV) (mod 2^b)
    return (V || h || s), v


vxeddsa_verify(u, M, (V || h || s)):
    if u >= p or V.y >= 2^|p| or h >= 2^|q| or s >= 2^|q|:
        return false
    A = convert_mont(u)
    B_v = hash_to_point(A || M)
    if not on_curve(A) or not on_curve(V):
        return false
    if cA == I or cV == I or B_v == I:
        return false
    R = sB - hA
    R_v = sB_v - hV
    h_check = hash_4(A || V || R || R_v || M) (mod q)
    if bytes_equal(h, h_check):
        v = hash_5(cV) (mod 2^b)
        return v
    return false
```

5. Curve25519

The Curve25519 elliptic curve specified in [4] can be used with XEdDSA and VXEdDSA, giving XEd25519 and VXEd25519. This curve defines the following parameters.

Name	Definition
B	convert_mont(9)
I	(x=0, y=1)
p	2^{255} - 19
q	2^{252} + 27742317777372353535851937790883648493
c	8
d	-121665 / 121666 (mod p)
A	486662
n	2
$\|p\|$	255
$\|q\|$	253
b	256

The twisted Edwards curve equation is $-x^2 + y^2 = 1 + dx^2y^2$. The *u_to_y* function implements the birational map from [4] by calculating $y = (u - 1) * inv(u + 1) (mod p)$.

XEd25519 signatures are valid Ed25519 signatures [1] and vice versa, provided the public keys are converted with the birational map.

Ed25519 allows implementations some flexibility in accepting or rejecting certain invalid signatures (e.g. with s unreduced, or checking the verification equation with or without cofactor multiplication). XEdDSA precisely specifies verification, so may differ from some Ed25519 implementations in accepting or rejecting such signatures (just as some Ed25519 implementations may differ from each other).

The particular verification steps chosen by XEdDSA include rejecting s if it has excess bits but not requiring it to be fully reduced, and checking verification without cofactor multiplication. These choices align with existing Ed25519 code, and lead to simpler implementations.

6. Curve448

The Curve448 elliptic curve specified in [4] can be used with XEdDSA and VXEdDSA, giving XEd448 and VXEd448. This curve defines the following parameters.

Name	Definition
B	convert_mont(5)
I	(x=0, y=1)
p	2^{448} - 2^{224} - 1
q	2^{446} - 138180668098951153520073867485154268803366924748821786098945475038 85
c	4
d	39082 / 39081 (mod p)
A	156326
n	-1
$\|p\|$	448
$\|q\|$	446
b	456

The twisted Edwards curve equation is $x^2 + y^2 = 1 + dx^2y^2$. The *u_to_y* function implements the birational map from [4] by calculating *y = (1 + u) * inv(1 - u) (mod p)*.

XEd448 differs from EdDSA [2] in choice of hash function. XEd448 uses SHA-512, whereas [2] recommends a 912-bit hash ($912 = 2b$). If the hash function is secure, outputs larger than 512 bits don't add security with Curve448, so XEd448 makes a simpler choice.

XEd448 may differ from other proposed instantiations of EdDSA [13] which use the "4-isogenous" curve from [4] rather than the "birationally equivalent" curve. Mapping from the Montgomery form Curve448 to the isogenous curve is more complicated.

7. Performance considerations

This section contains an incomplete list of performance considerations.

Faster signing: Calling *calculate_key_pair* for every XEdDSA signature roughly doubles signing time compared to EdDSA, since *calculate_key_pair* performs an additional scalar multiplication $E = kB$. VXEdDSA signing is more expensive, so the impact is proportionally less. To avoid this cost signers may cache the (non-secret) point E.

Pre-hashing: Except for XEdDSA verification, the signing and verification algorithms hash the input message twice. For large messages this could be expensive, and would require either large buffers or more complicated APIs.

To prevent this, APIs may wish to specify a maximum message size that all implementations must be capable of buffering. Protocol designers can specify "pre-hashing" of message fields to fit within this. Designers are encouraged to use pre-hashing selectively, so as to limit the potential impact from collision attacks (e.g. pre-hashing the attachments to a message but not the message header or body).

8. Security considerations

This section contains an incomplete list of security considerations.

Random secret inputs: XEdDSA and VXEdDSA signatures are randomized, they are not deterministic in the sense of [1] or [14]. The caller must pass in a new secret and random 64 byte value each time the signing function is called.

Deterministic signatures are designed to prevent reuse of the same nonce r with different messages, as this reveals the private key a. Consider two XEdDSA signatures $(\mathbf{R} \,||\, \mathbf{s_1})$ and $(\mathbf{R} \,||\, \mathbf{s_2})$ such that:

$$s_1 = r + h_1 a \pmod q$$
$$s_2 = r + h_2 a \pmod q$$

The private key a can be calculated as $a = (s_1 - s_2)/(h_1 - h_2) \pmod q$.

A deterministic signing scheme hashes $\mathbf{M}$ with a long-term secret to calculate r, instead of taking r from a random number generator. Because $\mathbf{M}$ is also hashed to calculate h the probability that different h get the same r is small. However, if the same message is signed repeatedly, a glitch that affects the calculation of h could cause this to happen (an observation due to Benedikt Schmidt). Also, repeated use of the same r might make it easier to recover information about r through side-channel analysis.

So XEdDSA and VXEdDSA preserve the idea of calculating r by hashing a long-term secret key and the message, but add a random value into the calculation

for greater resilience.

Constant time. The signing algorithms must not perform different memory accesses or take different amounts of time depending on secret information. This is typically achieved by "constant time" implementations that execute a fixed sequence of instructions and memory accesses, regardless of secret keys or message contents.

Particular care should be taken with the *calculate_key_pair* function due to its use of conditional branching. The *hash_to_point* function also uses conditional branching (within *elligator2*) and should be made constant time, even though it only handles the message, not secret keys.

Key reuse: It is safe to use the same key pair to produce XEdDSA and VXEdDSA signatures.

In theory, under some circumstances it is safe to use a key pair to produce signatures and also use the same key pair within certain Diffie-Hellman based protocols [15]. In practice this is a complicated topic requiring careful analysis, and is outside the scope of the current document.

9. IPR

This document is hereby placed in the public domain.

10. Acknowledgements

Special thanks to Mike Hamburg, creator of the Curve448 elliptic curve and Elligator 2, who was helpful in explaining them and in discussing several design questions.

Much thanks also to Moxie Marlinspike, David J. Wu, Robert Ransom, Peter Schwabe, Benedikt Schmidt, Samuel Neves, and Christian Winnerlein, all of whom contributed significantly to design discussions.

Thanks to Joseph Bonneau, Henry Corrigan-Gibbs, and Tom Ritter for review and editorial feedback, and thanks to Joe for drawing our attention to the discrete-log VRF [16].

Thanks to Daniel Bernstein and Tanja Lange for suggesting the naming scheme, and for discussion.

11. References

[1] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang, "High-speed high-security signatures," Journal of Cryptographic Engineering, vol. 2, no. 2, 2012. https://ed25519.cr.yp.to/ed25519-20110705.pdf

[2] D. J. Bernstein, S. Josefsson, T. Lange, P. Schwabe, and B.-Y. Yang, "EdDSA for more curves." Cryptology ePrint Archive, Report 2015/677, 2015. http://eprint.iacr.org/2015/677

[3] D. J. Bernstein, "Curve25519: New Diffie-Hellman Speed Records," in Public Key Cryptography - PKC 2006: 9th International Conference on Theory and Practice in Public-Key Cryptography, New York, NY, USA, April 24-26, 2006. Proceedings, Berlin, Heidelberg: Springer Berlin Heidelberg, 2006. https://cr.yp.to/ecdh/curve25519-20060209.pdf

[4] A. Langley, M. Hamburg, and S. Turner, "Elliptic Curves for Security." Internet Engineering Task Force; RFC 7748 (Informational); IETF, Jan-2016. http://www.ietf.org/rfc/rfc7748.txt

[5] S. Micali, M. Rabin, and S. Vadhan, "Verifiable Random Functions," in Proceedings of the 40th Annual Symposium on the Foundations of Computer Science, 1999. https://people.csail.mit.edu/silvio/Selected%20Scientific%20Papers/Pseudo%20Randomness/Verifiable_Random_Functions.pdf

[6] Y. Dodis and A. Yampolskiy, "A Verifiable Random Function with Short Proofs and Keys," in Public Key Cryptography - PKC 2005: 8th International Workshop on Theory and Practice in Public Key Cryptography, Les Diablerets, Switzerland, January 23-26, 2005. Proceedings, Berlin, Heidelberg: Springer Berlin Heidelberg, 2005. https://www.cs.nyu.edu/~dodis/ps/short-vrf.pdf

[7] Wikipedia, "Fermat's little theorem — Wikipedia, The Free Encyclopedia." 2016. https://en.wikipedia.org/w/index.php?title=Fermat%27s_little_theorem

[8] D. J. Bernstein, P. Birkner, M. Joye, T. Lange, and C. Peters, "Twisted Edwards Curves." Cryptology ePrint Archive, Report 2008/013, 2008. http://eprint.iacr.org/2008/013

[9] A. Jivsov, "Compact representation of an elliptic curve point," Internet Engineering Task Force; Internet Engineering Task Force, Internet-Draft draft-jivsov-ecc-compact-05, Sep. 2014. https://tools.ietf.org/html/draft-jivsov-ecc-compact-05

[10] NIST, "FIPS 180-4. Secure Hash Standard (SHS)," National Institute of Standards & Technology, Gaithersburg, MD, United States, 2012. http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf

[11] D. J. Bernstein, M. Hamburg, A. Krasnova, and T. Lange, "Elligator: Elliptic-curve points indistinguishable from uniform random strings." Cryptology

ePrint Archive, Report 2013/325, 2013. http://eprint.iacr.org/2013/325

[12] Wikipedia, "Legendre symbol — Wikipedia, The Free Encyclopedia." 2016. https://en.wikipedia.org/w/index.php?title=Legendre_symbol

[13] I. Liusvaara and S. Josefsson, "Edwards-curve Digital Signature Algorithm (EdDSA)," Internet Engineering Task Force; Internet Engineering Task Force, Internet-Draft draft-irtf-cfrg-eddsa-08, Oct. 2016. https://tools.ietf.org/html/draft-irtf-cfrg-eddsa-08

[14] T. Pornin, "Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA)." Internet Engineering Task Force; RFC 6979 (Informational); IETF, Aug-2013. http://www.ietf.org/rfc/rfc6979.txt

[15] J. P. Degabriele, A. Lehmann, K. G. Paterson, N. P. Smart, and M. Strefler, "On the Joint Security of Encryption and Signature in EMV." Cryptology ePrint Archive, Report 2011/615, 2011. http://eprint.iacr.org/2011/615

[16] M. S. Melara, A. Blankstein, J. Bonneau, E. W. Felten, and M. J. Freedman, "CONIKS: Bringing key transparency to end users." Cryptology ePrint Archive, Report 2014/1004, 2014. http://eprint.iacr.org/2014/1004