EdDSA for more curves

Daniel J. Bernstein, University of Illinois at Chicago and Technische Universiteit Eindhoven
Simon Josefsson, Simon Josefsson Datakonsult
Tanja Lange, Technische Universiteit Eindhoven
Peter Schwabe, Radboud Universiteit
Bo-Yin Yang, Academia Sinica

2015.07.04

The original specification of EdDSA was suitable only for finite fields $\mathbf{F}_q$ with $q \bmod 4 = 1$. The main purpose of this document is to extend EdDSA to allow finite fields $\mathbf{F}_q$ with any odd q. This document also extends EdDSA to support prehashing, i.e., signing the hash of a message.

Parameters. EdDSA has eleven parameters:

- An odd prime power q. EdDSA uses an elliptic curve over the finite field $\mathbf{F}_q$. Choosing q sufficiently large is important for security, since the size of q constrains the size of ℓ below. There are additional security concerns when q is not chosen to be prime.
- An integer b with $2^{b-1} > q$. EdDSA public keys have exactly b bits, and EdDSA signatures have exactly $2b$ bits.
- A $(b-1)$-bit encoding of elements of the finite field $\mathbf{F}_q$.
- A cryptographic hash function H producing $2b$-bit output. Conservative hash functions are recommended and do not have much impact on the total cost of EdDSA.
- An integer $c \in \{2, 3\}$. Secret EdDSA scalars are multiples of 2^c. The original specification of EdDSA did not include this parameter: it implicitly took $c = 3$.
- An integer n with $c \leq n \leq b$. Secret EdDSA scalars have exactly $n+1$ bits, with the top bit (the 2^n position) always set and the bottom c bits always cleared. The original specification of EdDSA did not include this parameter: it implicitly took $n = b-2$. Choosing n sufficiently large is important for security: standard "kangaroo" attacks use approximately $1.36\sqrt{2^{n-c}}$ additions on average to determine an EdDSA secret key from an EdDSA public key.
- A nonzero square element a of $\mathbf{F}_q$. The usual recommendation for best performance is $a = -1$ if $q \bmod 4 = 1$, and $a = 1$ if $q \bmod 4 = 3$. The original specification of EdDSA did not include this parameter: it implicitly took $a = -1$ (and required $q \bmod 4 = 1$).
- A non-square element d of $\mathbf{F}_q$. The exact choice of d (together with a and q) is important for security, and is the main topic considered in "curve selection".
- An element $B \neq (0, 1)$ of the set $E = \{(x, y) \in \mathbf{F}_q \times \mathbf{F}_q : ax^2 + y^2 = 1 + dx^2y^2\}$. This set forms a group with neutral element $0 = (0, 1)$ under the twisted Edwards addition law

$$(x_1, y_1) + (x_2, y_2) = \left(\frac{x_1y_2 + x_2y_1}{1 + dx_1x_2y_1y_2}, \frac{y_1y_2 - ax_1x_2}{1 - dx_1x_2y_1y_2}\right).$$

- An odd prime ℓ such that $\ell B = 0$ and $2^c\ell = \#E$. The number $\#E$ is part of the standard data provided for an elliptic curve E. Choosing ℓ sufficiently large is important for security: standard "rho" attacks use approximately $0.886\sqrt{\ell}$ additions on average to determine an EdDSA secret key from an EdDSA public key.

- A "prehash" function H'. PureEdDSA means EdDSA where H' is the identity function, i.e., $H'(M) = M$. HashEdDSA means EdDSA where H' generates a short output, no matter how long the message is; for example, $H'(M) = \text{SHA-512}(M)$. The original specification of EdDSA did not include this parameter: it implicitly took H' as the identity function.

EdDSA with prehash parameter H' is also written H'-EdDSA. The signature of M under H'-EdDSA is the signature of $H'(M)$ under PureEdDSA. One can take H' to be a different hash function from H: for example, the SHA-256-Ed25519-SHA-512 signature of M is the Ed25519-SHA-512 signature of SHA-256(M). See below for details of Ed25519-SHA-512.

Encoding and parsing integers. The integer $S \in \{0, 1, \ldots, \ell - 1\}$ below is encoded in little-endian form as a b-bit string $\underline{S}$.

A strict parser for the bit string $(S_0, S_1, \ldots, S_{b-1})$ computes $S = S_0 + 2S_1 + \cdots + 2^{b-1}S_{b-1}$ and fails if $S \geq \ell$. A liberal parser computes $S = S_0 + 2S_1 + \cdots + 2^{b-1}S_{b-1}$ and then replaces S with $S \bmod \ell$, never failing. A different type of liberal parser computes $S = S_0 + 2S_1 + \cdots + 2^{\lceil \log_2 \ell \rceil - 1}S_{\lceil \log_2 \ell \rceil - 1}$ and then replaces S with $S \bmod \ell$, again never failing. There are other possibilities, such as applying the liberal parser if the top $b - \lceil \log_2 \ell \rceil$ bits are all 0, and otherwise failing; this is simpler than the strict parser and still has a chance of catching malformed signatures (before doing any elliptic-curve operations). If a protocol requires a particular type of parsing (for example, to prevent implementation fingerprinting), then this requirement should be enforced via standard test vectors.

If $S \in \{0, 1, \ldots, \ell - 1\}$ then all of these parsers recover S from $\underline{S}$. The liberal parsers accept a few modified versions of $\underline{S}$, where the modifications add a few multiples of ℓ or set top bits. Such modifications ("malleability") are not relevant to the standard definition of signature security, and similar modifications are possible in most public-key signature systems. Protocol designers are free to assume, after signature verification, that the verified *messages* have been authorized by the signer, but should not assume that the specific *signatures* have been authorized by the signer.

If q is prime then the recommended $(b-1)$-bit encoding of $\mathbf{F}_q$ is the little-endian encoding of $\{0, 1, \ldots, q - 1\}$. Analogous comments apply to parsing this encoding.

Encoding and parsing curve points. The encoding of $\mathbf{F}_q$ is used to define "negative" elements of $\mathbf{F}_q$: specifically, x is negative if the $(b-1)$-bit encoding of x is lexicographically larger than the $(b-1)$-bit encoding of $-x$. In particular, if q is prime and the $(b-1)$-bit encoding of $\mathbf{F}_q$ is the little-endian encoding of $\{0, 1, \ldots, q - 1\}$, then $\{1, 3, 5, \ldots, q - 2\}$ are the negative elements of $\mathbf{F}_q$.

This encoding is also used to define a b-bit encoding of each element $(x, y) \in E$ as a b-bit string $\underline{(x, y)}$, namely the $(b-1)$-bit encoding of y followed by a sign bit; the sign bit is 1 if and only if x is negative.

A parser recovers (x, y) from a b-bit string, while also verifying that $(x, y) \in E$, as follows: parse the first $b - 1$ bits as y; compute $xx = (y^2 - 1)/(dy^2 - a)$; compute $x = \pm\sqrt{xx}$, where the $\pm$ is chosen so that the sign of x matches the bth bit of the string. If xx is not a square then the parsing fails.

(Implementation detail for $p \bmod 4 = 3$: Compute x as $\pm(xx)^{(p+1)/4}$, and then test that $x^2 = xx$. If the test fails then parsing fails. Note that one can compute the $(p+1)/4$th power of any element, not just of squares, but that for nonsquares the result of x^2 is $-xx$. Squaring x is cheaper than a Jacobi-symbol computation for xx.)

(Implementation detail for $p \bmod 8 = 5$: Precompute $i \in \mathbf{F}_q$ with $i^2 = -1$. Compute $X = (xx)^{(p+3)/8}$. If X^2 equals xx, put $x = \pm X$; otherwise, if $-X^2$ equals xx, put $x = \pm iX$; otherwise parsing fails.)

Bit strings as byte strings. Fixed-length bit strings are encoded as fixed-length byte strings in the usual little-endian way. Examples: The 16-bit string $(h_0, h_1, \ldots, h_{15})$ is encoded as the byte $h_0+2h_1+\cdots+2^7h_7$ followed by the byte $h_8+2h_9+\cdots+2^7h_{15}$. The 10-bit string $(h_0, h_1, \ldots, h_8, h_9)$ is encoded as the byte $h_0 + 2h_1 + \cdots + 2^7h_7$ followed by the byte $h_8 + 2h_9$.

It is simplest to take b to be a multiple of 8, so that each element of E is encoded as exactly $b/8$ bytes.

Secret keys and public keys. An EdDSA secret key is a b-bit string k. The hash $H(k) = (h_0, h_1, \ldots, h_{2b-1})$ determines an integer $s = 2^n + \sum_{c \le i < n} 2^i h_i$, which in turn determines the multiple $A = sB$. The corresponding EdDSA public key is $\underline{A}$.

Signing. The PureEdDSA signature of a message M under a secret key k is defined as follows. Define $r = H(h_b, \ldots, h_{2b-1}, M) \in \{0, 1, \ldots, 2^{2b} - 1\}$; here we interpret $2b$-bit strings in little-endian form as integers in $\{0, 1, \ldots, 2^{2b} - 1\}$. Define $R = rB$. Define $S = (r + H(\underline{R}, \underline{A}, M)s) \bmod \ell$. The signature of M under k is then the $2b$-bit string $(\underline{R}, \underline{S})$.

(Implementation detail: To save time in the computation of rB, the signer can replace r with $r \bmod \ell$ before computing rB.)

The EdDSA signature of a message M under a secret key k is defined as the PureEdDSA signature of $H'(M)$. In other words, EdDSA simply uses PureEdDSA to sign $H'(M)$.

The signer has the power to make different choices of r, creating other strings $(\underline{R}, \underline{S})$ that are not *the* signature of M but that will nevertheless pass verification. This option complicates testing of the signing procedure; leads to a complete break of the signature scheme if the signer repeats the value r for different messages M and the same key s; and also leads to a complete break of the signature scheme if r is even slightly guessable. This option is therefore not recommended.

Verification. Verification of an alleged PureEdDSA signature on a message M under a public key works as follows. The verifier parses the key as $\underline{A}$ for some $A \in E$, and parses the alleged signature as $(\underline{R}, \underline{S})$ for some $R \in E$ and $S \in \{0, 1, \ldots, \ell - 1\}$. The verifier computes $H(\underline{R}, \underline{A}, M)$ and then checks the group equation $2^cSB = 2^cR + 2^cH(\underline{R}, \underline{A}, M)A$ in E. The verifier rejects the alleged signature if the parsing fails or if the group equation does not hold.

EdDSA verification for a message M is defined as PureEdDSA verification for $H'(M)$.

Cofactorless verification. A "cofactorless verifier" instead checks the equation $SB = R + H(\underline{R}, \underline{A}, M)A$ and rejects alleged signatures where this equation does not hold. Any alleged signature that passes cofactorless verification will also pass verification. The signature of a message will pass cofactorless verification, so it will also pass verification. However, a signer using a secret key outside the above signing procedure can create strings that pass verification without passing

cofactorless verification. The EdDSA security goal protects verification against attackers, so it also protects cofactorless verification against attackers.

Selecting hash functions. SHAKE256, part of the forthcoming SHA-3 hash-function standard, is designed to straightforwardly generate arbitrary output lengths at a 2^{256} security level. (Efficiency note: SHAKE256 internally generates outputs as 136-byte blocks, so a single SHAKE256 block handles b as large as 544.) Choosing SHAKE256 with $2b$-bit output as the hash function inside EdDSA has the benefit of allowing multiple curve sizes to be handled directly by a single hashing module. SHA-3 has many other advantages over SHA-2, not summarized here.

However, in recognition of today's widespread deployment of SHA-512, this document instead presents EdDSA parameters that use SHA-512 for hashing. SHA-512 was originally defined to produce only one output length, 512 bits, but can easily be reused to produce shorter output lengths by truncation, and longer output lengths by concatenation.

FIPS 180-4 defines SHA-512/224 as a 224-bit truncation of a modified version of SHA-512. The modification consists of replacing the SHA-512 IV with an IV defined by hashing the string "`SHA-512/224`". FIPS 180-4 defines SHA-512/256 similarly.

This document defines SHA-512/832 as a concatenation of two 416-bit truncations of modified SHA-512 hash outputs, where the first modification uses an IV obtained as in FIPS 180-4 from the string "`SHA-512/832part0/2`", and the first modification uses an IV obtained as in FIPS 180-4 from the string "`SHA-512/832part1/2`". This document similarly defines SHA-512/912 as a concatenation of two 456-bit truncations of modified SHA-512 hash outputs. This document also defines SHA-512/1056 as a concatenation of four 264-bit truncations of modified SHA-512 hash outputs, with strings "`SHA-512/1056part0/4`" etc.

Examples: Ed25519-SHA-512 and SHA-512-Ed25519-SHA-512. Ed25519-SHA-512 is PureEdDSA with the following parameters: q is the prime $2^{255}-19$; $b = 256$; the 255-bit encoding of $\mathbf{F}_{2^{255}-19}$ is the usual little-endian encoding of $\{0, 1, \ldots, 2^{255}-20\}$; H is SHA-512; $c = 3$; $n = 254$; $a = -1$; $d = -121665/121666$; B is the point $(\ldots 202, 4/5) \in E$; and ℓ is the prime

$$2^{252} + 27742317777372353535851937790883648493.$$

SHA-512-Ed25519-SHA-512 is the same except with $H' =$ SHA-512; i.e., the SHA-512-Ed25519-SHA-512 signature of M is the Ed25519-SHA-512 signature of SHA-512(M).

Example: Ed41417-SHA-512/832 and SHA-512-Ed41417-SHA-512/832. Ed41417-SHA-512/832 is PureEdDSA with the following parameters: q is the prime $2^{414}-17$; $b = 416$; the 415-bit encoding of $\mathbf{F}_{2^{414}-17}$ is the usual little-endian encoding of $\{0, 1, \ldots, 2^{414}-16\}$; H is SHA-512/832; $c = 3$; $n = 414$; $a = 1$; $d = 3617$; B is the point $(\ldots 165, 34) \in E$; and ℓ is the prime

$$2^{411} - 3336414086375514252081017769409838517898472720041120858959475 9.$$

SHA-512-Ed41417-SHA-512/832 is the same except with $H' =$ SHA-512.

Examples: Ed448-SHA-512/912 and SHA-512-Ed448-SHA-512/912. Ed448-SHA-512/912 is PureEdDSA with the following parameters: q is the prime $2^{448} - 2^{224} - 1$; $b = 456$; the 455-bit encoding of $\mathbf{F}_{2^{448}-2^{224}-1}$ is the usual little-endian encoding of $\{0, 1, \ldots, 2^{448} - 2^{224} - 2\}$; H is

SHA-512/912; $c = 2$; $n = 448$; $a = 1$; $d = -39081$; B is the point $(\ldots 495, 19) \in E$; and ℓ is the prime

$$2^{446} - 138180668098951153520073867485154268803366924748821786098945475038 85.$$

SHA-512-Ed448-SHA-512/912 is the same except with $H' = \text{SHA-512}$.

One can compress the 114-byte signatures here to 113 bytes by merging the 114th byte, which is always between 0 and 63, into the 57th byte, which is always 128 or 0 according to the sign of x. However, the 114-byte format has the advantage of sharing E-parsing code between public keys and signatures.

Examples: Ed521-SHA-512/1056 and SHA-512-Ed521-SHA-512/1056. Ed521-SHA-512/1056 is PureEdDSA with the following parameters: q is the prime $2^{521} - 1$; $b = 528$; the 527-bit encoding of $\mathbf{F}_{2^{521}-1}$ is the usual little-endian encoding of $\{0, 1, \ldots, 2^{521} - 2\}$; H is SHA-512/1056; $c = 2$; $n = 521$; $a = 1$; $d = -376014$; B is the point $(\ldots 324, 12) \in E$; and ℓ is the prime

$$2^{519} - 337554763258501705789107630418782636071904961214051226618635150085779108655765.$$

SHA-512-Ed521-SHA-512/1056 is the same except with $H' = \text{SHA-512}$.

Security notes on prehashing. PureEdDSA is resilient to collisions in the underlying hash function H. HashEdDSA is not resilient to collisions in H': if the attacker finds messages M_1 and M_2 with $H'(M_1) = H'(M_2)$, and convinces the legitimate H'-EdDSA signer to sign M_1, then the attacker can forge the same signature as a signature of M_2. Modern hash functions are designed to resist collisions, and in principle it should be safe to design signature systems to rely on this, but it is more conservative to design signature systems so that collisions serve merely as early-warning signals. PureEdDSA is therefore recommended by default.

Any protocol allowing multiple signature schemes needs to authenticate the choice of signature scheme, so that signatures under one scheme cannot be confused with signatures under another. For example, if a public key A were allowed to be used with both PureEdDSA and H'-EdDSA then an attacker could forge a signature of M under H'-EdDSA by obtaining a signature of $H'(M)$ under PureEdDSA. There are many other "cross-protocol" attacks in the literature; it is an error to simply certify a public key A without also certifying the signature scheme to be used for A.

The main motivation for HashEdDSA is the following storage issue (which is irrelevant to most well-designed signature applications). Computing the PureEdDSA signature of M requires reading through M twice from a buffer as long as M, and therefore does not support a small-memory "Init-Update-Final" interface for long messages. Every common hash function H' supports a small-memory "Init-Update-Final" interface for long messages, so H'-EdDSA signing also supports a small-memory "Init-Update-Final" interface for long messages. Beware, however, that analogous streaming of *verification* for long messages means that verifiers pass along forged packets from attackers, so it is safest for protocol designers to split long messages into short messages to be signed; this splitting also eliminates the storage issue.