A modern signature scheme with message recovery: Abe–Okamoto signatures with EdDSA elements

Satō Shinichi*

April 17, 2020

Abstract

This paper revisits the Abe–Okamoto signature scheme to present a version of their signature scheme augmented with modern best practices, with major influences taken from EdDSA. Implementation guidance is given on how to reuse existing Ed25519 code.

1 Introduction

1.1 Message recovery and EdDSA

Message recovery in the context of a digital signature schemes minimizes the amount of information transferred. This is an important property for very space-constrained applications, such as QR codes which might identify and validate some kind of ticket. For very short messages, it even may become entirely unnecessary to send a message at all. Instead, the message can be contained entirely in the message recovering signature. This is of particular interest for product keys in software protection schemes. Though the scheme as presented here with Ed25519 will likely be too large for software protection schemes nonetheless. Requiring users to input 64 bytes (103 characters in base 32) will likely not be acceptable for typical applications.

Nyberg and Rueppel note that hash functions can be done away with for signature schemes based on discrete logarithms.[11] Unlike their scheme, however, the Abe–Okamoto scheme, which this paper is based on, relies on several distinct hash functions (modeled as random oracles for the proof); negating this particular advantage of message recovery.[3]

EdDSA is a signature scheme that uses twisted Edwards curves introduced by Bernstein et al. in [5]. It popularized deterministic nonce generation as a safeguard against failure to generate sufficiently random numbers, a technique introduced by Barwood in a post to `sci.crypt`.[4] Recently, however, efforts have been made to re-introduce randomness into EdDSA in a variant called XEdDSA such that the result is at least as secure as EdDSA, to additionally guard against power analysis attacks and fault injection.[12] Subsequent research

*The author is not currently affiliated with any company or academic institution. Contact: `sato@airmail.cc`

by Samwel et al. has found that the mitigations against power-relate attacks in XEdDSA are insufficient as-is.[13] An internet draft by Preuss Mattson et al. takes the results of continued analysis of these mitigations and proposes a new version of these mitigations.[8]

While the scheme presented this paper will not be adopting these additional mitigations, in my view it is nonetheless worth pointing out the existence of deterministic nonces with additional random noise considering that one of the applications of message recovery is in the context of embedded scenarios, which see themselves particularly vulnerable to power-related attacks.

1.2 Paper organization

This paper first reviews the EdDSA and the Abe–Okamoto signature schemes, comparing them side by side. Care has been taken to use the same variable names for both when they refer to the same concept.

It then proceeds to merge the ideas contained in EdDSA into the Abe–Okamoto signature scheme, explaining why none of them impact the security proof in [3].

It then closes by giving the algorithm of the new scheme accompanied by specific advice on how to implement this scheme given an existing implementation of Ed25519 or Ed448.

2 Comparison of the two schemes

Common parameters:

q An odd prime power. The elliptic curve is defined over the finite field $\mathbb{F}_q$.

B The base point of the curve.

ℓ The order of the subgroup generated by B ($\ell B = 0$; $h\ell = \#E$). ℓ must be prime.

b An integer so that $2^{b-1} > q$, i.e. at least the bit size of a field element in $\mathbb{F}_q$.

h The cofactor of the elliptic curve.

c The number of bits in the cofactor, i.e. $c = \log_2 h$.

n An integer so that $c \leq n \leq b$.

2.1 Key generation

2.1.1 EdDSA

The keys are generated as follows:

1. Generate the secret key k, a b-bit string, and save it.
2. Apply a hash function H that generates $2b$-bit output to k to obtain a': $a' = H(k)$

3. Trim a' to obtain a: Set the n-th bit of a' to 1. Clear the bottom c bits of a' to zero. Store the result of this in a.
4. Perform scalar multiplication to get public key point $A = aB$.
5. Encode public key point A to byte string $\underline{A}$, and save it.

2.1.2 Abe–Okamoto

1. Select scalar k uniformly at random such that $0 \leq a < \ell$.
2. Encode scalar k to byte string $\underline{a}$ and save it.
3. Perform scalar multiplication to get a public key point $A = -aB$.
4. Encode public key point A to byte string $\underline{A}$, and save it.

2.1.3 Differences in key generation

Abe–Okamoto does not use a hash function with trimming, instead selecting a secret scalar directly uniformly at random. It also *negates* the public key point (or the scalar before scalar multiplication).

The reason the n-th bit of a' is set to 1 is to help prevent kangaroo attacks; the difficulty for curves with a cofactor relates to the difference between $n-c$,[7] so that a large n is beneficial. This issue only affects curves with $h! = 0$, such as all curves in (twisted or untwisted) Edwards form and Montgomery form. No special precautions like this must be taken for curves with a prime order (including the Ristretto groups).

2.2 Signing

Common parameters:

M The message to be signed (bit string).

k The private key (scalar).

2.2.1 EdDSA

Given a hash function H that produces $2b$-bit outputs:

1. Hash k. Save two halves of the hash in $\mathcal{H}$: $\mathcal{H} = H(k)$
2. Use the first $\lceil \frac{b}{8} \rceil$ bytes to re-generate a, the public key A and its encoding $\underline{A}$ as in key generation.
3. Use the second $\lceil \frac{b}{8} \rceil$ bytes to generate the nonce scalar prefix r_{prefix}. $r_{prefix} = \mathcal{H}[32..64]$
4. Generate a nonce scalar r by hashing r_{prefix} and M, then trimming it as for key generation above (optionally reducing it modulo ℓ): $r = H(r_{prefix}||M)_{trimmed} \mod \ell$
5. Generate a nonce point R with scalar multiplication: $R = rB$. Encode R to the byte string $\underline{R}$.

6. Generate a hash C by applying H to the nonce point, the public key and the message: $C = H(\underline{R}||\underline{A}||M)$
7. Generate scalar $S = (r + a \cdot C) \mod \ell$. Encode it to the byte string $\underline{S}$.

The signature is $(\underline{R}, \underline{S})$.

2.2.2 Abe–Okamoto

Parameters:

u The bit length of a serialized point, i.e. $u = |\mathcal{E}(P)|$ for any given point P.

u_1, u_2 such that $u_1 + u_2 = k$. It is desirable to choose u_2 such that it is a multiple of 8, i.e. byte-aligned, which makes it possible for M to be byte-aligned.

F_1 A random oracle function that maps u_2-bit strings to u_1-bit strings.

F_2 A random oracle function that maps u_1-bit strings to u_2-bit strings. This parameter determines the length of M that can be encoded.

H A random oracle function that maps $u_1 + u_2$-bit strings to $|\ell|$-bit strings.

The signature is calculated thus:

1. Get a k-bit string M': $M' = F_1(M)||(F_2(F_1(M)) \oplus M)$.
2. Select a nonce scalar r uniformly at random such that $0 \leq r < \ell$.
3. Generate a nonce point R with scalar multiplication: $R = rB$. Encode the X-coordinate of R as a bit string $\underline{R}$.
4. Generate a value $t = \underline{R} \oplus M'$
5. Generate a hash C by applying H to t: $C = H(t)$
6. Generate scalar $S = (r + a \cdot C) \mod \ell$. Encode it to the byte string $\underline{S}$.

The signature is now $(t, \underline{S})$.

2.2.3 Differences in signing

The most obvious difference is the generation of M' and xoring with the X-coordinate of the shared nonce point. Everything else is identical *in concept*:

The nonce scalar r can be generated using EdDSA methodology, i.e. deterministically based on (half of a hash of) the secret key and the message, reduced modulo ℓ and then trimmed. The security argument goes the same as for deterministic nonces with elliptic curve signature schemes in general, detailed in [10]: Abe–Okamoto signatures only require that the nonce scalar be chosen uniformly at random. In fact, Abe–Okamoto signatures are a variant of Schnorr signatures,[9] so that the arguments in [10] can be applied wholesale for Abe–Okamoto signatures.

The (encoded) public key $\underline{A}$ can be added into the random oracle function H during the generation of S. EdDSA adds $\underline{A}$ into the hash (key-prefixing) to

eliminate key-substitution attacks and avoid issues regarding attacking multiple public keys simultaneously; the single-key security of all Schnorr-like signatures (such as the Abe–Okamoto scheme) implies multi-key security *if key-prefixing is used.*[5][6] This is a straight security improvement and should be incorporated into all modern instantiations of Abe–Okamoto and other Schnorr-like systems. Moving to a key-prefixed variant does not affect the proof in [3] either, as they assume that $H(t)$ (c in their paper) is random in the first place. The properties of a hash function as a stand-in for a random oracle is not affected by an additional input.

Unlike EdDSA, it is not feasible to add the message into C because the receiver needs to arrive at the same nonce point R before it can actually recover the message (which it would require for the EdDSA method of generating C), putting the figurative cart before the horse.

H doesn't *need* to output u-bit output since a reduction modulo ℓ immediately follows, so we can use EdDSA's H as a stand-in for the H in Abe–Okamoto here. Abe–Okamoto seem to implicitly assume that the order of the curve is a power of two. The proof in [3] wants H to be a random oracle, but using a hash function with reduction is the closest approximation.

2.3 Verification

2.3.1 EdDSA

1. Extract $\underline{R}$ and $\underline{S}$ from the signature byte string.
2. Decode $\underline{R}$ as a point R. Verify that it is a point on the curve. If this fails, return $\perp$.
3. Decode $\underline{S}$ as a scalar S. Verify that it is below ℓ. If this fails, return $\perp$.
4. Decode the encoded public key $\underline{A}$ as a point A. Verify that it is a point on the curve. If this fails, return $\perp$.
5. Check that $hSB = hR + h \cdot H(\underline{R}||\underline{A}||M) \cdot A$. (EdDSA as originally specified would ignore the cofactor h.) If this equation is true, return ok, else return $\perp$.

2.3.2 Abe–Okamoto

1. Extract t and $\underline{S}$ from the signature byte string.
2. Decode $\underline{S}$ as a scalar S. Verify that it is below ℓ. If this fails, return $\perp$.
3. Decode the encoded public key $\underline{A}$ as a point A. Verify that it is a point on the curve. If this fails, return $\perp$.
4. Recover the alleged hash $C = H(t)$.
5. Recover the alleged nonce point $R = (SB + CA)$. Encode the X-coordinate of R as a bit string $\underline{R}$.
6. Recover the alleged M' by xoring t and $\underline{R}$: $M' = t \oplus \underline{R}$

7. Recover the candidate for the message M by xoring the right-hand side of M' with the output of F_2 for the left-hand side: $M = M'_{u_2} \oplus F_2(M'^{u_1})$
8. Verify the equation $M'^{u_1} = F_1(M)$. If this equation is true, return M, else return $\perp$.

2.3.3 Differences in verification

The algorithms share a common core: Recovering a point with the sum of two scalar multiplications. We can make this match EdDSA more closely by also hashing in $\underline{A}$ for the multiplicand of the public key for reasons outlined in the section about signing.

Regarding the X-coordinate, both ends must reach the same encoding of the nonce point R or message recovery will fail.

For minimal code changes to an existing implementation as well as support for Decaf (Ed448) and Ristretto (Ed25519), it is possible to use the Decaf or Ristretto point serialization of R as $\underline{R}$ instead of its X-coordinate. Depending on the format, this directly impacts the encoding length of the point, and thus u_1 and u_2, which indirectly affects F_1 and F_2. The proof in [3] makes no assumptions about the Y-coordinate being a secret, hence it is safe to use point encodings that contain information about the whole point, rather than just the X-coordinate and having to go through extraordinary measures to reach the same X-coordinate with Decaf and Ristretto. What would invalidate the proof, however, is when the value xored against M' includes information *other* than that contained in the shared point.

In particular, this means that having "dead bits" that are not covered by point information encoding (be it just the encoded X-coordinate, affine point encoding with sign bit or Ristretto) causes them to not be part of the verification process. The excess bits in a byte when encoding a point thus allow *storage* of bits of the encoded message M' that are not covered by a point encoding, but verifying that part of it would become impossible. Therefore, the encoding of a point dictates the maximum length of the encoded message M'. Exact counting of the available bits is required.

3 The merged Abe–Okamoto scheme with EdDSA elements

Common parameters reiterated here for convenience:

q An odd prime power. The elliptic curve is defined over the finite field $\mathbb{F}_q$. For Edwards25519, $q = 2^{255} - 19$. For Ed448, $q = 2^{448} - 2^{224} - 1$.

b An integer so that $2^{b-1} > q$, i.e. at least the bit size of a field element in $\mathbb{F}_q$. This is normally the bit length of q rounded up to the nearest multiple of 8. For Ed25519, $b = 256$. For Ed448, $b = 456$.

n An integer so that $c \leq n \leq b$. For Ed25519, $n = b - 2 = 254$. For Ed448, $n = b - 9 = 447$.

B The base point of the curve.

ℓ The order of the subgroup generated by B ($\ell B = 0$; $h\ell = \#E$). ℓ must be prime.

c The number of bits in the cofactor, i.e. $c = \log_2 h$. For Ed25519, $c = 3$. For Ed448, $c = 2$.

$\mathcal{E}$, $\mathcal{D}$ A point encoding function and a point decoding function, respectively. The point encoding function encodes a point from an internal representation to a sequence of bytes. The point decoding function decodes a point from a sequence of bytes to an internal representation, returning $\perp$ if the encoded point (byte sequence) is invalid. This may simply encode and decode the X-coordinate of a byte string, or be a more involved point serialization format such as standard affine point compression (as specified in RFC 8032 for Ed25519 and Ed448) or Decaf/Ristretto.

3.1 Key generation

1. Generate a secret key k, a b-bit string, and save it.
2. Apply a hash function H that generates $2b$-bit output to k to obtain a': $a' = H(k)$
3. Trim a' to obtain a: Set the n-th bit of a'. Clear all bits between b and n of a'. Clear the bottom c bits of a' to zero. Store the result of this in a.
4. Perform scalar multiplication to get the public key point $-A$: $-A = -aB$. This can be accomplished by either negating B before multiplication, negating a before multiplication or negating the resulting point.
5. Encode public key point $-A$ to byte string $\underline{A}$.

3.2 Signing

Static parameters:

u The bit length of a serialized point, i.e. $u = |\mathcal{E}(P)|$ for any given point P.

u_1, u_2 such that $u_1 + u_2 = k$. It is desirable to choose u_2 such that it is a multiple of 8, i.e. byte-aligned, which makes it possible for M to be byte-aligned.

F_1 A hash function that produces u_1-bit digests.

F_2 A hash function that produces u_2-bit digests. This parameter determines the length of M that can be encoded. Note that $F_1! = F_2! = H$.

H A hash function that produces $2b$-bit length digests. For Ed25519, this is SHA-512. For Ed448, this is SHAKE256 with 114 bytes of output.

It is strongly recommended to have domain separation strings in the implementation of F_1, F_2 and H. Additionally, all of F_1, F_2 and H *should* also take a context string for application-level domain separation. Given different domain separation strings (and optionally a context string), F_1 and F_2 can be implemented as truncated verisons of H invoked with different domain separation and context strings.

The signature for a message M given secret key k is calculated thus:

1. Compute the u-bit string M': $M' = F_1(M)||(F_2(F_1(M)) \oplus M)$
   For efficiency, observe that $F_1(M)$ needs to be computed only once.
2. Hash k. Save two halves of the hash in $\mathcal{H}$: $\mathcal{H} = H(k)$
3. Use the first $\lceil \frac{b}{8} \rceil$ bytes to re-generate a, the public key A and its encoding $\underline{A}$ as in key generation.
4. Use the second $\lceil \frac{b}{8} \rceil$ bytes to generate the nonce scalar prefix r_{prefix}. $r_{prefix} = \mathcal{H}[32..64]$
5. Generate a nonce scalar r by hashing r_{prefix} and M, then trimming it as for key generation above (optionally reducing it modulo ℓ): $r = H(r_{prefix}||M)_{trimmed} \mod \ell$
6. Generate a nonce point R with scalar multiplication: $R = rB$. Encode R to the byte string $\underline{R}$: $\underline{R} = \mathcal{E}(R)$
7. Generate a u-bit value t: $t = \underline{R} \oplus M'$
8. Generate a hash C by applying H to t and the public key: $C = H(t||\underline{A})$
9. Generate scalar $S = (r + a \cdot C) \mod \ell$. Encode it to the byte string $\underline{S}$.

The signature is now $(t, \underline{S})$.

3.3 Verification

The signature $(t, \underline{S})$ for a given encoded public key $\underline{S}$ is verified (with recovery of the message) thus:

1. Extract t and $\underline{S}$ from the signature byte string. If $b > k$, ensure that the top bits of t are clear. If $|\underline{S}| > |S|$ (i.e. the bit length of the encoding of S has "dead bits"), ensure that the top bits of $\underline{S}$ are clear.
2. Decode $\underline{S}$ as a scalar S. If $S \geq \ell$, return $\perp$.
3. Decode the encoded public key $\underline{A}$ as a point A. If the decoding fails or A does not actually refer to a valid point on the curve, return $\perp$.
4. Recover the alleged hash $C = H(t||\underline{A})$.
5. Recover the alleged nonce point $R = (SB + CA)$. Encode R to the byte string $\underline{R}$: $\underline{R} = \mathcal{E}(R)$
6. Recover the alleged value of M' by xoring t and $\underline{R}$: $M' = t \oplus \underline{R}$
7. Recover the candidate for the message M by xoring the right-hand side (recall that M' was formed by a concatenation) of M' with the output of F_2 for the left-hand side: $M = M'_{u_2} \oplus F_2(M'^{u_1})$
8. If $M'^{u_1} = F_1(M)$, return M, else return $\perp$.

4 Notes on changing existing EdDSA implementations to fit

When using Ed25519 in particular, note how using compressed affine point encoding (regular Ed25519 point encoding) yields a perfectly byte-aligned 256 bits for encoding M: A field element in $\mathbb{F}_{2^{255}-19}$ requires 255 bits to encode; the sign bit for the other coordinate makes for the 256th bit. On the other hand, beware when using Ristretto, which requires no sign bit and thus cannot cover a perfectly byte-aligned M: Additionally, the X-coordinate method cannot be used with Ristretto because a point *may* decompress to different coordinates (but an identical point). Instead, the point encoding must be used, not the X-coordinate; a (computationally wasteful) alternative is to encode and re-encode the point in Ristretto until the encoding yields the same coordinates every time.

5 Discussion: Efficiency

This scheme is not efficient for messages whose length does not exceed the size of u if the alternative short Schnorr signatures are considered. Short Schnorr signatures, brought up by Schnorr in the paper introducing Schnorr signatures themselves, compress the signature by transmitting $H(\underline{R}||M)$ instead of $\underline{R}$, where for a signature aiming at the equivalent of b-bit symmetric security, only b bits of H would be used, bringing the total down to a size of $3b$.[1] However, it has never been possible to fully prove the security of short Schnorr signatures with a tight security reduction.

The (less than) b bits available with full message recovery with Abe–Okamoto is approximately equally efficient with regards to space, but less efficient than short Schnorr signatures with regards to computation due to several hash functions being involved. Considering that EdDSA is a relative of Schnorr signatures, it is trivial to change EdDSA to instead use short Schnorr signatures, especially considering that the hash is a byproduct of EdDSA signature verification.[2]

Message recovery thus becomes only attractive for *partial* message recovery. Partial message recovery can be achieved by splitting the message M in two parts M_1 and M_2, where the bit length of M_1 is u_2. M_1 is compressed into and recovered from the signature. M_2 is sent along with the signature, and hashed as part of the hash value C after all other inputs.[3]

Table 1 illustrates the sizes in bits (rounded up to the next multiple of 8) when using Edwards25519 given a message M of bit length $|M|$ and a signature z of bit length $|z|$. Observe in particular how the size of a short Schnorr signature plus the size of a message is always equal to a size of an Abe–Okamoto signature with full message recovery. In fact, for messages that are smaller than u_2 bits, short Schnorr signatures end up having a lower total entirely.

Method	$\lvert M\rvert$	$\lvert z\rvert$	M **bits in** z	M **bits outside**	**Total**
Abe–Okamoto	≤ 128	512	$\lvert M\rvert$	0	512
	> 128	512	u_2	$\lvert M\rvert - 128$	$512 + \lvert M\rvert - 128$
Short Schnorr	≤ 128	384	0	$\lvert M\rvert$	$384 + \lvert M\rvert$
	> 128	384	0	$\lvert M\rvert$	$384 + \lvert M\rvert$
EdDSA/Schnorr	≤ 128	512	0	$\lvert M\rvert$	$512 + \lvert M\rvert$
	> 128	512	0	$\lvert M\rvert$	$512 + \lvert M\rvert$

Table 1: Signature sizes for full and partial message recovery

References

[1] *Efficient Identification and Signatures for Smart Cards*, volume 435 of *Lecture Notes in Computer Science*. Springer, New York, NY, 1990.

[2] *Faster Batch Forgery Identification*, volume 7668 of *Lecture Notes in Computer Science*. Springer, Berlin, Heidelberg, 2012.

[3] Masayuki Abe and Tatsuaki Okamoto. A signature scheme with message recovery as secure as discrete logarithm. In *Advances in Cryptology – ASIACRYPT'94*, volume 1716 of *Lecture Notes in Computer Science*, pages 378–389. Springer, Berlin, Heidelberg, 1999.

[4] George Barwood. Digital signatures using elliptic curves, message 32f519ad.19609226@news.dial.pipex.com. posted to newsgroup `sci.crypt`, 1997. URL: `https://groups.google.com/d/msg/sci.crypt/SalLSLBBTe4/xtYNGDe6irIJ`.

[5] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2:77–89, 2012.

[6] Daniel J. Bernstein. Multi-user Schnorr security, revisited, 2015. URL: `https://ed25519.cr.yp.to/multischnorr-20151012.pdf`.

[7] Daniel J. Bernstein and Simon Josefsson and Tanja Lange and Peter Schwabe and Bo-Yin Yang. EdDSA for more curves, 2015. URL: `https://ed25519.cr.yp.to/eddsa-20150704.pdf`.

[8] J. Preuss Mattsson and E. Thormarker and S. Ruohomaa. Deterministic ECDSA and EdDSA Signatures with Additional Randomness. Internet-Draft draft-mattsson-cfrg-det-sigs-with-noise-02, Internet Engineering Task Force (IETF) Secretariat, March 2020. URL: `https://tools.ietf.org/html/draft-mattsson-cfrg-det-sigs-with-noise-02`.

[9] Masayuki Abe and Tatsuaki Okamoto and Koutarou Suzuki. Message recovery schemes from sigma-protocols. *NTT Technical Review*, 6, 2008.

[10] David M'Raïhi, David Naccache, David Pointcheval, and Serge Vaudenay. Computational alternatives to random number generators. In *Selected Areas in Cryptography. SAC 1998*, volume 1556 of *Lecture Notes in Computer Science*, pages 72–80. Springer, Berlin, Heidelberg, 1999.

[11] Kaisa Nyberg and Rainer A. Rueppel. Message recovery for signature schemes based on the discrete logarithm problem. In *Advances in Cryptology – EUROCRYPT'94*, volume 1716 of *950*, pages 182–193. Springer, Berlin, Heidelberg, 1994.

[12] Trevor Perrin. Finalizing xeddsa. posted to modern crypto `curves` mailing list, 2016. URL: `https://moderncrypto.org/mail-archive/curves/2016/000779.html`.

[13] Niels Samwel, Lejla Batina, Guido Bertoni, Joan Daemen, and Ruggero Susella. Breaking ed25519 in wolfssl. Cryptology ePrint Archive, Report 2017/985, 2017. URL: `https://eprint.iacr.org/2017/985`.