Taming the many EdDSAs

Konstantinos Chalkias, François Garillot, Valeria Nikolaenko

(Mysten Labs) (Protocol Labs) (a16z crypto)

https://eprint.iacr.org/2020/1244.pdf appeared at SSR 2020

& (Ed25519 signing attacks) https://github.com/MystenLabs/ed25519-unsafe-libs

for the NIST reading group, Mar 8, 2023

EdDSA signature (high-level)

- Schnorr signatures with deterministic nonce, over a curve Ed25519.
- Signature with fastest verification. Provably secure.
- Widely deployed: OpenSSH, TLS 1.3, Signal, Tor, GnuPG,
  and in blockchains: Corda, Tezos, Stellar, ZCash, Aptos.

- EdDSA high-level:

KeyGen():

- sk - a secret scalar
- pk = sk * B

Sign(sk, M):

- r = H(sk || M)
- R = r * B
- S = r + H(R || pk || M) * sk
- σ = (S, R)

Nonce r is generated deterministically!

Verify(pk, M, σ = (S,R)): S * B == R + H(R || pk || M) * pk

Signature security

- **EUF-CMA**: **E**xistential **U**n**F**orgeability under **C**hosen **M**essage **A**ttacks

  Protecting against the following attack:
  given pk, after asking adaptively for signatures on $m_1,...,m_n$ and getting $\sigma_1,...,\sigma_n$, produce a valid pair (m^*, σ^*) with $m^* \neq m_1,...,m_n$.

An attacker can't produce a signature on a NEW message!

Signature security

- **SUF-CMA**: **S**trong **U**n**F**orgeability under **C**hosen **M**essage **A**ttacks

  Protecting against the following attack:
  given pk, after asking adaptively for signatures on $m_1,...,m_n$ and getting $\sigma_1,...,\sigma_n$, produce a valid pair (m^*, σ^*) with $(m^*, \sigma^*) \neq (m_1, \sigma_1)...,(m_n,\sigma_n)$.

An attacker can't produce a signature on a NEW message +
an attacker can't produce a new signature on an OLD message!

"Bitcoin transaction malleability and MtGox" by C.Decker and R.Wattenhofer ESORICS 2014
~$500,000,000 presumably drained from Mt Gox because the signature was modified on the fly.

Signature security

- **BS**: **B**inding **S**ignature (sig binds to the message)

  Protecting against the following attack:

  produce (pk, m, m’, σ), s.t.

  (m,σ) and (m’,σ) are valid and m ≠ m’

An attacker can’t produce a signature that is valid for two messages, even when an attacker is a signer!

Signature security

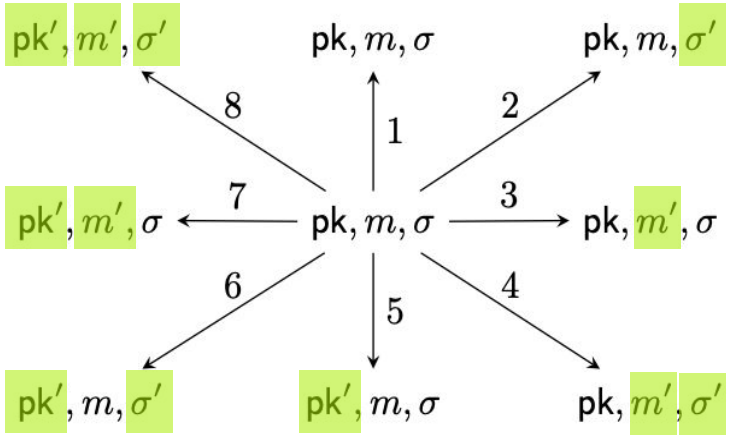
- **SBS**: **S**trongly **B**inding **S**ignatures
  (sig binds to the message and the public key)

  Protecting against the following attack:
  produce (pk, m, pk', m',σ), s.t. (m,σ) is valid under pk, (m',σ') is valid under pk', and (pk,m) ≠ (pk',m')

An attacker can't produce a signature that is valid for two message - public-key pairs, even when an attacker is a signer!

Monero a major bug:
https://www.getmonero.org/2017/05/17/disclosure-of-a-major-bug-in-cryptonote-based-currencies.html

How definitions cover the space of possible malleabilities



(a) Signature transformations
We assume $\mathsf{pk} \neq \mathsf{pk}'$, $m \neq m'$ and $\sigma \neq \sigma'$.

Alteration to (pk, m, σ) triplet	Security property preventing the alteration by a (possibly malicious)	
Fig 1a	signer	public attacker
5	SBS	SBS
3	BS	EUF-CMA
2	uniqueness *	SUF-CMA
4	N/A	EUF-CMA
7	SBS	SBS
6,8	N/A	N/A

(b) Here N/A means that an alteration of this type is expected from the signature scheme and does not concern us in this writing. Note (*) that the EdDSA signatures are deterministic but not unique, i.e. a dishonest signer can always produce multiple signatures for the same message.

SUF-CMA implies EUF-CMA
SBS implies BS

Strongest* signature security:

non-malleability = SUF-CMA + SBS

* for non-unique signatures

Ed25519 group structure

- E is a set of points (x, y) from {0,1,...,q-1}, q is a 255-bits prime that satisfy the equation:

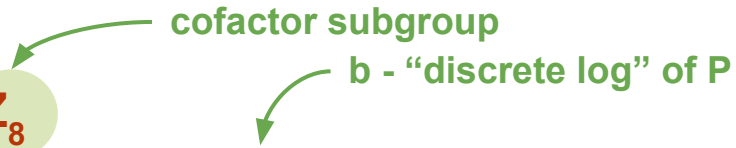
$$-x^2 + y^2 = 1 + d * x^2y^2 \mod q$$

- Addition law determines the group operation:

$$(x_1, y_1) + (x_2, y_2) = \left(\frac{x_1y_2 + x_2y_1}{1 + dx_1x_2y_1y_2}, \frac{y_1y_2 + x_1x_2}{1 - dx_1x_2y_1y_2} \right)$$

- Two special points:
  - Zero point: **O** = (0,1)
  - Base point B ∈ E (generator, specified in the standards): has order L, i.e. L* B = **O**
- Elliptic curve group E ≅ Z_L x Z_8, L is a 253-bits prime

Ed25519 group - cofactor subgroup

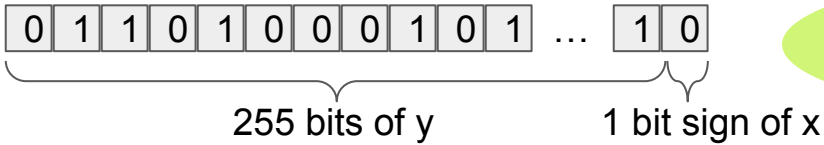


- Elliptic curve group E ≅ Z_L x Z_8
- Any point P ∈ E can be represented P = b*B + t*T
  where b ∈ {0,1,...,L-1} and t ∈ {0,1,2,3,4,5,6,7}
  B is of order L, i.e. L*B = O, and T is of order 8, i.e. 8*T = O.
- If t ≠ 0, P is *mixed order*
- If b = 0, P is *small order*

- When everybody is honest (the signer, the signature relayer), small and mixed order points never show up in EdDSA.
- The co-factor issue can be mitigating by always checking that L*pk = 0, but that is computationally very expensive!

Ed25519 group point serialization

- $P = (x,y) \in (Z_q, Z_q)$, $q = 2^{255}-19$
- Serialized to a 256-bits string as follows:



When y ≥ q, the encoding is non-canonical:
19 non-canonical serializations

- Little-endian format: left to right and from least significant to most significant
- Last bit is 1 iff x is negative.
- x is restored using the curve equation:

$$-x^2 + y^2 = 1 + d * x^2y^2 \mod q$$

Ed25519 - small order points

#	Order	Point	Serialized point
		Canonical serializations	
1	1	$(0, 1)$	010000..0000
2	2	$(0, 2^{255} - 20)$	ECFFFF..FF7F
3	4	$\left(-\sqrt{-1}, 0\right)$	000000..0080
4	4	$\left(\sqrt{-1}, 0\right)$	000000..0000
5	8	...	C7176A..037A
6	8	...	C7176A..03FA
7	8	...	26E895..FC05
8	8	...	26E895..FC85
		Non-canonical serializations	
9	1	$(-0, 1)$	010000..0080
10	2	$(-0, 2^{255} - 20)$	ECFFFF..FFFF
11	1	$(0, 2^{255} - 18)$	EEFFFF..FF7F
12	1	$(-0, 2^{255} - 18)$	EEFFFF..FFFF
13	4	$\left(-\sqrt{-1}, 2^{255} - 19\right)$	EDFFFF..FFFF
14	4	$\left(\sqrt{-1}, 2^{255} - 19\right)$	EDFFFF..FF7F

Table 1: Small order points of Curve25519 in its twisted Edwards form.

#	Canonical serializations
1	0100000000000000000000000000000000000000000000000000000000000000
2	ECFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF7F
3	0000000000000000000000000000000000000000000000000000000000000080
4	0000000000000000000000000000000000000000000000000000000000000000
5	C7176A703D4DD84FBA3C0B760D10670F2A2053FA2C39CCC64EC7FD7792AC037A
6	C7176A703D4DD84FBA3C0B760D10670F2A2053FA2C39CCC64EC7FD7792AC03FA
7	26E8958FC2B227B045C3F489F2EF98F0D5DFAC05D3C63339B13802886D53FC05
8	26E8958FC2B227B045C3F489F2EF98F0D5DFAC05D3C63339B13802886D53FC85

#	Non-canonical serializations
9	0100000000000000000000000000000000000000000000000000000000000080
10	ECFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
11	EEFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF7F
12	EEFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
13	EDFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
14	EDFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF7F

(b) Full serialization of small order points.

Ed25519 - serialization with non-canonical y

Small order points

#	Order	Point	Serialized point
Canonical serializations			
1	1	$(0, 1)$	010000..0000
2	2	$(0, 2^{255} - 20)$	ECFFFF..FF7F
3	4	$(-\sqrt{-1}, 0)$	000000..0080
4	4	$(\sqrt{-1}, 0)$	000000..0000
5	8	...	C7176A..037A
6	8	...	C7176A..03FA
7	8	...	26E895..FC05
8	8	...	26E895..FC85
Non-canonical serializations			
9	1	$(-0, 1)$	010000..0080
10	2	$(-0, 2^{255} - 20)$	ECFFFF..FFFF
11	1	$(0, 2^{255} - 18)$	EEFFFF..FF7F
12	1	$(-0, 2^{255} - 18)$	EEFFFF..FFFF
13	4	$(-\sqrt{-1}, 2^{255} - 19)$	EDFFFF..FFFF
14	4	$(\sqrt{-1}, 2^{255} - 19)$	EDFFFF..FF7F

Table 1: Small order points of Curve25519 in its twisted Edwards form.

Convention:
x = 0 must be encoded with bit 1 not 0

Non-canonical y

y	$y + 2^{255} - 19$	Valid	Order
0	$2^{255} - 19$	✓	4
1	$2^{255} - 18$	✓	1
2	$2^{255} - 17$	✗	-
3	$2^{255} - 16$	✓	$8 \cdot L$
4	$2^{255} - 15$	✓	$4 \cdot L$
5	$2^{255} - 14$	✓	$8 \cdot L$
6	$2^{255} - 13$	✓	$8 \cdot L$
7	$2^{255} - 12$	✗	-
8	$2^{255} - 11$	✗	-
9	$2^{255} - 10$	✓	$2 \cdot L$
10	$2^{255} - 9$	✓	$8 \cdot L$
11	$2^{255} - 8$	✗	-
12	$2^{255} - 7$	✗	-
13	$2^{255} - 6$	✗	-
14	$2^{255} - 5$	✓	$8 \cdot L$
15	$2^{255} - 4$	✓	$4 \cdot L$
16	$2^{255} - 3$	✓	$8 \cdot L$
17	$2^{255} - 2$	✗	-
18	$2^{255} - 1$	✓	$4 \cdot L$

Table 2: Non-canonically encoded points.

We do not know dlog of these points, hence unlikely, they come up in valid sigs.

× 2 for x=0/1

Handling of “unusual” points and scalars gets tricky

- σ = (S,R) is 512 bits
  - S is a scalar - an integer in {0,1,...,L-1} - 256 bits,
  - R is a serialized point of E - 256 bits.
- pk = A is 256 bits
  - A is a serialized point of E.
- What if
  - S is non-canonical (S >= L) ?
  - R or A are serialized non-canonically (R.y or A.y >= q) ?
  - R or A are small order ?
  - R or A are mixed order ?
  - etc….

Signature Verification (co-factored)

Signature Verification on message M, public key A and signature $\sigma = (R, S)$

1: Reject the signature if $S \notin \{0, \ldots, L-1\}$.
2: Reject the signature if the public key A is one of 8 small order points.
3: Reject the signature if A or R are non-canonical.
4: Compute the hash $\mathsf{SHA512}(R||A||M)$ and reduce it $\mathsf{mod}\ L$ to get a scalar h.
5: Accept if $8(S \cdot B) - 8R - 8(h \cdot A) = 0$.

Non-repudiation & resilience to key substitution attacks

Contract Signing: if company A signed an agreement with company B using a key that allows for repudiation, it can later claim that it signed a completely different deal.

Electronic Voting: malicious voters may pick special keys that allow for repudiation on purpose in order to create friction in the process and deny results, as their signed vote might be verified against multiple candidates.

Transactions: a blockchain transaction of amount X might also be valid for another amount Y, creating potential problems for consensus and dispute resolution.

Non repudiation example


```
{"message1"            : "Send 100 USD to Alice",
 "message1 (UTF-8)"    :
"53656e6420313030205534420746f20416c696365",
 "message2"            : "Send 100000 USD to Alice",
 "message2 (UTF-8)"    :
"53656e642031303030302055534420746f20416c696365",
 "pub_key"             :
"ecffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff7f",
 "signature"          :
"a9d55260f765261eb9b84e106f665e00b867287a761990d7135963ee0a7d59dc\

a5bb704786be79fc476f91d3f3f89b03984d8068dcf1bb7dfc6637b45450ac04"}
```

Blockchain issues

- Forks! 50% of the validators run co-factored, the rest run co-factorless
  - Malicious users can pick small / mixed order keys on purpose
  - Note that such attacks can have financial incentives due to trading opportunities
  - The issue is even more evident when batch verification is utilized

- Transaction malleability on non-canonical S values
  - If SegWit is not applied and the signature is part of the transactionID, then some users won't be able to find their original txID
  - MtGox (one of the biggest exchanges collapses in Bitcoin's history)

Optimizations (canonical S)

L = 2^{252} + 27742317777372353535851937790883648493 (prime)

Hint (a): the 252th bit of s is unlikely to be set for honest users

Hint (b): 240 = 11110000 (binary), 224 = 11100000 (binary)

```
fun is_canonical_s(s_bytes: &[u8]) -> bool {
  return
    if s_bytes[31] & 240 == 0 {  true /* succeed fast */ }
    else if s_bytes[31] & 224 != 0 {  false /* fail fast */ }
    else { full_s_canonicity_check(s_bytes) }
}
```

Optimizations (canonical y-coordinate)

$q = 2^{255} - 19$

Hint (a): all of q's bits, but the 2nd and 5th less significant bits, are set!

Hint (b): the 8 less significant bits correspond to the decimal number 237

Hint (c): each y-coordinate could start with an "is less than 237" check on the less significant byte, which will succeed with probability 237/256 = 92.5%, then 99.6%.

```
fun is_canonical_y(bytes: &[u8]) -> bool {
    if bytes[0] < 237 { true }
    else {
      for i in 1..=30 {
        if bytes[i] != 255 { return true }
      }
      (bytes[31] | 128) != 255
    }
}
```

Optimizations (faster co-factored verification)

Signature Verification on message M, public key A and signature $\sigma = (R, S)$

1: Reject the signature if $S \notin \{0, \ldots, L-1\}$.
2: Reject the signature if the public key A is one of 8 small order points.
3: Reject the signature if A or R are non-canonical.
4: Compute the hash $\mathsf{SHA512}(R||A||M)$ and reduce it $\mathbf{mod}\ L$ to get a scalar h.
5: Accept if $8(S \cdot B) - 8R - 8(h \cdot A) = 0$.

Method (save a few operations):

- first compute V = SB − R − hA
- then accept if V is one of 8 small order points
  - (or alternatively compute 8V with 3 doublings and check against the neutral element).

Batch verification

cofactored		cofactorless		Example conditions							
[1] single	[2] batch	[3] single	[4] batch	t_A	t_R	pk's order	R's order	[1]+[2]	[1]+[4]	[2]+[3]	[3]+[4]
$\checkmark_1$	$\checkmark_1$	$\checkmark_{1/8}$	$\times_{7/8}$	1	0	mixed	L	ok	FN	ok	FN
$\checkmark_1$	$\checkmark_1$	$\times_{7/8}$	$\checkmark_{1/8}$	1	0	mixed	L	ok	ok	FN	FP
$\checkmark_1$	$\checkmark_1$	$\times_1$	$\checkmark_{1/8}$	0	1	L	mixed	ok	ok	FN	FP
$\checkmark_1$	$\checkmark_1$	$\checkmark_{1/8}$	$\times_{7/8}$	1	1	mixed	mixed	ok	FN	ok	FN
$\checkmark_1$	$\checkmark_1$	$\times_{7/8}$	$\checkmark_{1/8}$	1	1	mixed	mixed	ok	ok	FN	FP

Table 3: Examples of different combinations of t_A and t_R that cause inconsistency between cofactorless single and batch verifications. FN denotes a false negative case, FP denotes a false positive case, ok denotes no discrepancy.

Fast verification of EdDSA signatures

- Antipa et al. considered mutualizing point doublings in a linear combination equivalent to the verification equation.
- Bernstein et al. considered using the trick, but it requires finding a suitable factor for the combination for the linear combination, which requires a GCD, which makes the trick ineffective.
- Pornin managed to find an algorithm for finding this factor that is cost-effective.
- Besides the Pornin implementation, performance gains have been confirmed in BouncyCastle

Antipa, Adrian, et al. "Accelerated verification of ECDSA signatures." *Selected Areas in Cryptography: 12th International Workshop, SAC 2005*
Pornin, Thomas. "Optimized lattice basis reduction in dimension 2, and fast schnorr and EdDSA signature verification." *Cryptology ePrint Archive* (2020).

Test Vectors

#	M	σ	S	A's order	R's order	$8(SB) = 8R + 8(hA)$	$SB = R + hA$
0	..22b6	..0000	$S = 0$	small	small	✓	✓
1	..2e79	..ac04	$0 < S < L$	small	mixed	✓	✓
2	..b9ab	..260e	$0 < S < L$	mixed	small	✓	✓
3	..2e79	..d009	$0 < S < L$	mixed	mixed	✓	✓
4	..f56c	..1a09	$0 < S < L$	mixed	mixed	✓	✗
5	..f56c	..7405	$0 < S < L$	mixed	L	✓ (1)	✗
6	..ec40	..a514	$S > L$	L	L	✓	✓
7	..ec40	..8c22	$S \gg L$ (2)	L	L	✓	✓
8	..8b41	..5f0f	$0 < S < L$	mixed	small (3)	- (3)	- (3)
9	..8b41	..4908	$0 < S < L$	mixed	small (3)	- (3)	- (3)
10	..155b	..ac04	$0 < S < L$	small (4)	mixed	- (4)	- (4)
11	..c06f	..ac04	$0 < S < L$	small (4)	mixed	- (4)	- (4)

Table 4: Conditions satisfied by the test vectors.

Test vector repository

- 13 test vectors available in the repository,
- Code for verification on those vectors for all tested libraries,
- Moreover, the code that generated the vectors is available in documented source

https://github.com/novifinancial/ed25519-speccheck

Results

+ See repo for recent contributions: Zig, TweetNacl-C

Library	0	1	2	3	4	5	6	7	8	9	10	11	SUF-CMA	SBS	cofactored
Algorithm 2	✗	✗	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✓	✓	✓
RFC 8032(*) [18]	✓	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✓	✗	✓
FIPS 186-5 [32]	✓	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✓	✗	✓
BoringSSL	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✓	✓	✗	✗
BouncyCastle	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗
CryptoKit	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✓	✓	✗	✗
Dalek	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✓	✓	✗	✗
ed25519-donna	✓	✓	✓	✓	✗	✗	✓	✗	✗	✗	✗	✓	✗	✗	✗
ed25519-java	✓	✓	✓	✓	✗	✗	✓	✓	✗	✗	✓	✗	✗	✗	✗
Go	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✓	✓	✗	✗
LibSodium	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✗
nCipher nShield	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✗
npm	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✓	✓	✗	✗
OpenSSL-3.0	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✓	✓	✗	✗
PyCA	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✓	✓	✗	✗
python-ed25519	✓	✓	✓	✓	✗	✗	✓	✓	✗	✗	✗	✓	✗	✗	✗
ref10	✓	✓	✓	✓	✗	✗	✓	✗	✗	✗	✗	✓	✗	✗	✗
TweetNaCl.js	✓	✓	✓	✓	✗	✗	✓	✓	✗	✗	✗	✓	✗	✗	✗
Zebra	✓	✓	✓	✓	✓	✓	✗	✗	✗	✓	✓	✓	✓	✗	✓

Table 5: Test vector results[8]

(*) The cofactored, recommended, version of the RFC 8032 is used.

Recent BouncyCastle update

RELEASES

Software produced by this site is covered by the following license and was made possible with the help of the following contributors. If you are interested in sponsoring work on Bouncy Castle or getting commercial support for this or prior releases

Release 2.1.1, 18th February 2023

BouncyCastle.Cryptography 2.1.1 Official NuGet Archive.

Release Notes for 2.1.1

Defects Fixed

- Fixed a rounding issue with FF1 Format Preserving Encryption algorithm for certain radices.
- Fixed RFC3394WrapEngine handling of 64 bit keys.
- PkixCertPathValidator: fixed fetching of DateOfCertGen extension.
- PkixCertPathValidator: correctly remove ExtendedKeyUsage from critical extensions (https://github.com/bcgit/bc-csharp/issues/395).
- PkixNameConstraintValidator: fixed special handling of 'serialNumber' in RDNs.

Additional Features and Functionality

- The BIKE implementation has been updated according to the NIST PQC Round 4 modifications.
- The HQC implementation has been updated according to the NIST PQC Round 4 modifications.
- EdDSA verification now conforms to the recommendations of Taming the many EdDSAs, in particular cofactored verification. As a side benefit, Pornin's basis reduction is now used for EdDSA verification, giving a significant performance boost.
- Major performance improvements for Anomalous Binary (Koblitz) Curves.
- Added implementations of Ascon AEAD, Hash and XOF algorithms.
- Added AriaWrapEngine, an implementation of RFC 3394 wrapping for the ARIA cipher.
- User customization of the GCM multiplier has been obsoleted. We recommend no longer supplying a custom multiplier to GcmBlockCipher.
- Several large properties files used by the PQC algorithms have been compressed in order to reduce the size of the assembly.
- Debug symbols have been extracted to a separate snupkg package.
- Major performance improvements for GCM bulk processing when Pclmulqdq, Ssse3 intrinsics available.

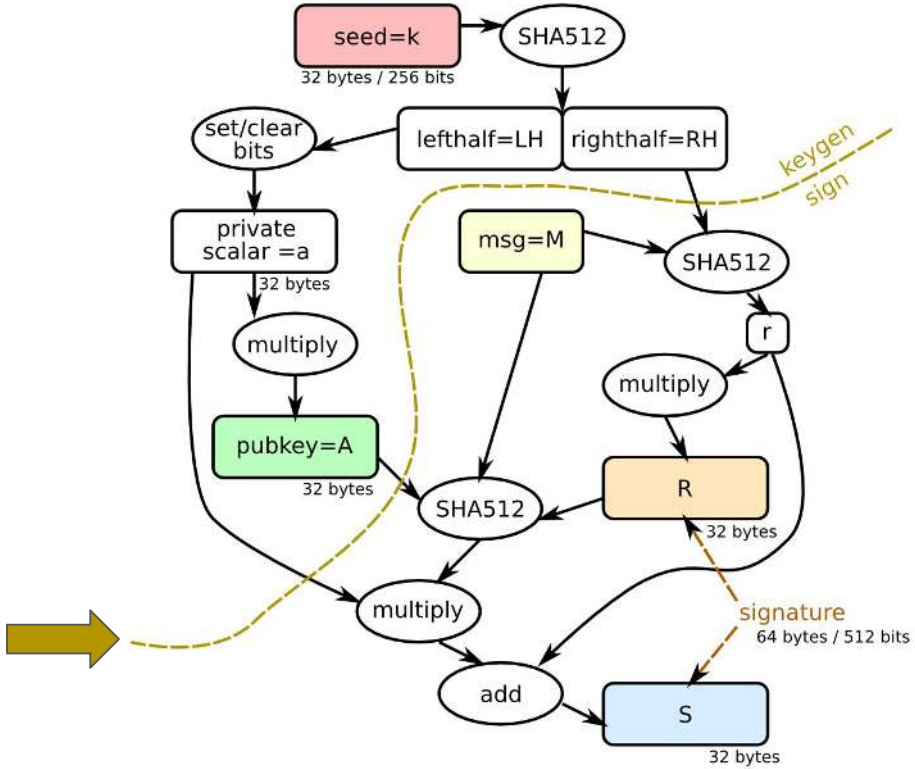
Ed25519 signing api misuse

Taming verification → Taming signing

from (Ed25519 signing attacks) https://github.com/MystenLabs/ed25519-unsafe-libs

Ed25519 api zoo

- seed vs extended privKey
- privKey serialization
  - seed only
  - extended privKey only
  - keyPair
- verification function
  - co-factored vs co-factorless
- signing api
  - sign(privKey, msg)
  - sign(keyPair, msg)
  - sign(privKey, msg, pubKey)
- pre-computations
- batching
- aggregation



	privkey	pubkey	signature
NaCl (20110221)	a RH (64 bytes)	A (32 bytes)	R (32 bytes) M S (32 bytes)
SUPERCOP (20110704)	k A	A	R S M
python-ed25519 (0.7)	k to_seed() k A to_bytes()	A	R S (64 bytes)

Image credits: Brian Warner, “Ed25519 keys”, blogpost, https://blog.mozilla.org/warner/2011/11/29/ed25519-keys/

FIPS 186-5

sign(keyPair, msg)

The secure use of digital signatures depends on the management of an entity's digital signature key pair. Key pair management requirements for EdDSA are the same as for ECDSA, which are provided in Section 6.2.2.

7.6 EdDSA Signature Generation

EdDSA signatures are deterministic. The signature is generated using the hash of the private key and the message using the procedure below or an equivalent process.

Inputs:

1. Bit string M to be signed
2. Valid public-private key pair (d, Q) for domain parameters D
3. H: SHA-512 for Ed25519 or SHAKE256 for Ed448
4. For Ed448, a string *context* set by the signer and verifier with a maximum length of 255 octets; by default, *context* is the empty string

Output: The signature $R \,||\, S$, where R is an encoding of a point and S is an octet string of a given length of a little-endian encoded value.

Process:

As specified in IETF RFC 8032, the EdDSA signature of a message M under a private key d is

RFC 8032 - EdDSA (generic description)

sign(privKey, msg, pubKey)

3.3. Sign

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

Where is the pubKey??

The PureEdDSA signature of a message M under a private key k is the 2*b-bit string ENC(R) || ENC(S). R and S are derived as follows. First define r = H(h_b || ... || h_(2b-1) || M) interpreting 2*b-bit strings in little-endian form as integers in {0, 1, ..., 2^(2*b) - 1}. Let R = [r]B and S = (r + H(ENC(R) || ENC(A) || PH(M)) * s) mod L. The s used here is from the previous section.

RFC 8032 - EdDSA (concrete instances)

sign(privKey, msg)

5.1.6. Sign

The inputs to the signing procedure is the private key, a 32-octet string, and a message M of arbitrary size. For Ed25519ctx and Ed25519ph, there is additionally a context C of at most 255 octets and a flag F, 0 for Ed25519ctx and 1 for Ed25519ph.

1. Hash the private key, 32 octets, using SHA-512. Let h denote the resulting digest. Construct the secret scalar s from the first half of the digest, and the corresponding public key A, as described in the previous section. Let prefix denote the second half of the hash digest, h[32],...,h[63].

2. Compute SHA-512(dom2(F, C) || prefix || PH(M)), where M is the message to be signed. Interpret the 64-octet digest as a little-endian integer r.

Original Paper

sign(keyPair, msg) ?

sign(privKey, msg, pubKey) ?

High-speed high-security signatures

Daniel J. Bernstein[1], Niels Duif[2], Tanja Lange[2],
Peter Schwabe[3], and Bo-Yin Yang[4]

which in turn determines the multiple $A = aB$. The corresponding EdDSA public key is $\underline{A}$. Bits $h_b, \ldots, h_{2b-1}$ of the hash are used as part of signing, as discussed in a moment.

The signature of a message M under this 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 $S = (r + H(\underline{R}, \underline{A}, M)a) \bmod \ell$. The signature of M under k is then the $2b$-bit string $(\underline{R}, \underline{S})$, where $\underline{S}$ is the b-bit little-endian encoding of S. Applications wishing to pack data into every last nook and cranny should note that the last three bits of signatures are always 0 because ℓ fits into $b - 3$ bits.

50+ affected
Ed25519 libs

r/crypto

Posts Wiki

Posted by u/KryptosPi 9 months ago

106

Initial impact report about this week's EdDSA Double-PubKey Oracle attack in 40 affected crypto libs

Last week MystenLabs crypto research revealed a list of 26 Ed25519 libraries that could expose the private key if the signing function was misused either accidentally or on purpose. **As of today and thanks to community contribution, this list currently includes 40 libraries** from almost every programming language, some of them very popular with 100+ Github stars.

← **Tweet**

Steven Galbraith
@EllipticKiwi

A hazard for deterministic signatures: if you put the public key in the hash (to generate the challenge in the sigma protocol) then you better check it is the correct public key!!

Kostas Kryptos @kostascrypto · Jun 18, 2022

Preliminary juicy impact report for the recent EdDSA Double PubKey Sign Oracle misuse exploit.

Dozens of cryptography libraries vulnerable to private key theft

Ben Dickson 28 June 2022 at 15:38 UTC
Updated: 29 June 2022 at 07:34 UTC

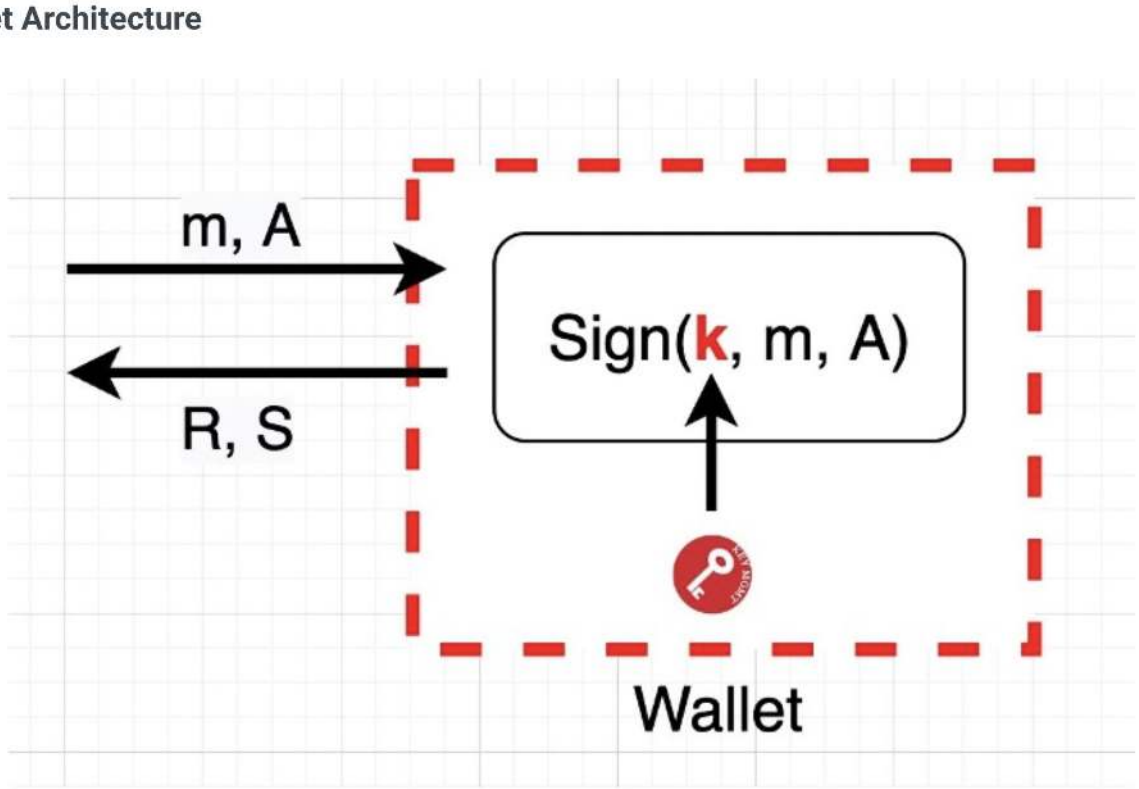
Encryption API Hacking News

Signing mechanism security shortcomings exposed

A poor implementation of Ed25519, a popular digital signature algorithm, has left dozens of cryptography libraries vulnerable to attacks.

According to Konstantinos Chalkias, a cryptographer at MystenLabs who discovered and reported the vulnerability, attackers could exploit the bug to steal private keys from cryptocurrency wallets.

Vulnerable apis



The above image presents a common wallet architecture, including mobile wallet, hardware wallet, cloud wallet (run on a server), etc. scenarios. In the wallet architecture, the private key is generally stored in a safe place which can not be accessed by outside but can participate in signature computing.

Image credits: Safeheron, “Analysis On Ed25519 Use Risks: Your Wallet Private Key Can Be Stolen”, blogpost, https://tinyurl.com/safeheron

The attack

Mainly affects **signing apis that take the pubKey as input**

- EdDSA signatures are deterministic, and thus for the same input message M to be signed, a unique signature output that includes two elements, a curve point R and a scalar S, is returned.
- Algorithmic detail: signer's public key is involved in the deterministic computation of the S part of the signature only, **but not in the R value**.
- Thus, if an adversary could somehow use the signing function as an Oracle (that expects arbitrary public keys as inputs), then it is possible for the same message to **receive 2 signatures sharing the same R and only differ on the S part.**
- Unfortunately, when this happens, one can easily extract the private key

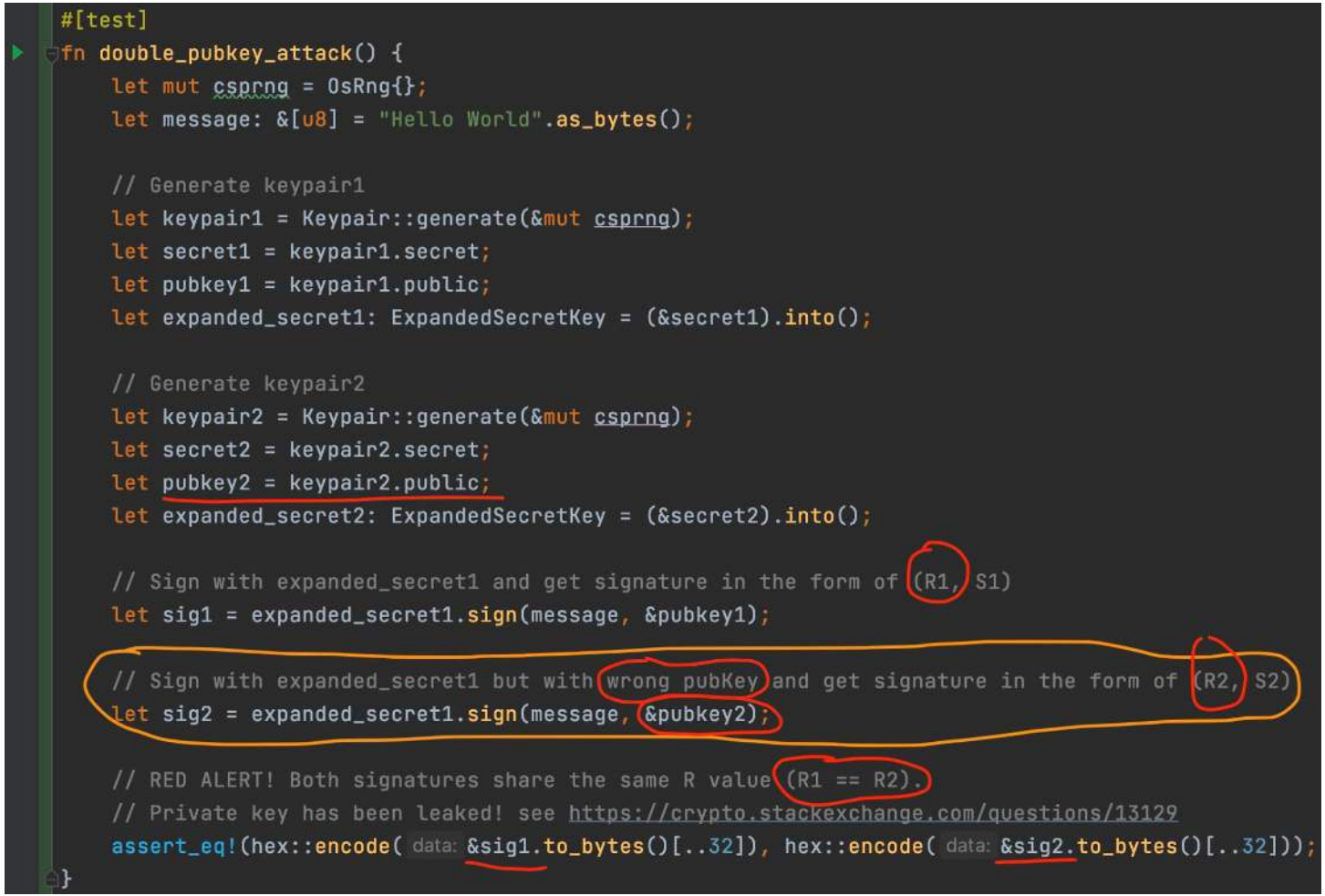
Hint: slightly easier to stay undetected in co-factored verification (even when signer verifies sigs before submitting it).

Trick: replace the original pubKey with a mixed order pubKey (function of the 1st)

Attacks in practice

Rust: [ed25519-exploit]

Python: [ed25519-vuln]
by Bill Buchanan



Most common issues

A fintech api had a **DB table storing `<userID, pubKey>`**, while the only data stored in the TCB (securely with integrity protection) was the private key.

Flow: external users invoked a public api `sign(userID, msg)`

a. then internally the server did a DB request to retrieve the stored pubKey for this user,
b. and finally there was an HSM call to `sign(userID, msg, pubKey)`.
c. The HSM retrieved the private key by `userID`, and returned the `signature`.
d. The administrators thought that even malicious employees cannot extract privKeys, but the issue was many employees had write access to the DB `<userID, pubKey>`, so the application couldn't guarantee someone didn't change the original pubKey.

Feedback: the api misleads devs to believe that it's harmless to have different integrity protection between privKey and pubKey and think that the worst case scenario is a signature failure.

Another spectacular failure - **key rotation**:

- For a few msecs, the DB still stored the old `<userID, pubKeyOld>`,
- this allowed a narrow window for frontrunning attacks: get a signature asap, before the DB gets updated with the new pubKey

Aftermath

“Taming the many EdDSAs” (verification rules)

- tested against 20+ libs (even hardware vendors)
- proposed test-vectors (now used in industry, ie Bouncycastle cryptography lib)
- explained why “optionality” can cause compatibility + security issues
- tricks for improved performance

“50+ vulnerable ed25519 apis” (signing function)

- due to paper, RFC, standards and library inconsistencies
- domino effect from original implementations (performance Vs security)
- 48 libs still vulnerable (incl. popular ones)

Q & A

Taming the many EdDSAs

Konstantinos Chalkias, François Garillot, Valeria Nikolaenko

(original paper) https://eprint.iacr.org/2020/1244.pdf

(Ed25519 signing attacks) https://github.com/MystenLabs/ed25519-unsafe-libs

for the NIST reading group, Mar 8, 2023