Practical fault attack against the Ed25519 and EdDSA signature schemes

Yolan Romailler ∧ Sylvain Pelissier
Kudelski Security

September 25, 2017

Content

Who are we?

Researchers

We work at Kudelski Security, do research, crypto code reviews, device attacks, IoT security, and more.

We can play with a focused ions beam, laser FI, voltage glitch bench, side-channel bench, chemical decapsulation, ...

Our Lab

Elliptic curve signature schemes

ECDSA

Well known EC signature scheme: ECDSA, over a curve with generator B of order ℓ, using a private key a and a hash function H to sign a message M:

- Generate randomly $k \in [1, \ell - 1]$
- $(x, y) = k \cdot B$
- $R = x \mod \ell$
- $S = k^{-1}(\mathrm{H}(M) + ra) \mod \ell$
- Output (R, S)

Elliptic curve signature schemes

ECDSA

ECDSA can be dangerous to use because its security relies heavily on **cryptographically strong randomness**.

ECDSA has proved to be sensitive to many kinds of fault attacks, side channels and other fun things.

A deterministic version of ECDSA has been published in RFC 6979, getting rid of the randomness by hashing the message.

Elliptic curve signature schemes

The EdDSA scheme

EdDSA is a public-key elliptic curve signature scheme recently standardized in RFC 8032, based on Schnorr's signature.

Its security is based on the ECDLP.

EdDSA works over (twisted) Edwards curves.

Let's say we have such a curve E, with base point $B \neq (0, 1)$ of order ℓ.

The EdDSA scheme

It's pretty good

Some of EdDSA notable features:

- Provides high performances
- "Complete" formulas, *i.e.* no special case
- No randomness required to sign
- Made with side-channel attacks resilience in mind
- Small public keys (32 bytes for Ed25519)
- Small signatures (64 bytes for Ed25519)

The EdDSA scheme

EdDSA uses:

- a curve E, with base point $B \neq (0, 1)$ of order ℓ
- a hash function H that produces a $2b$-bits output (*e.g.* SHA-512 for $b = 256$ bits)
- a private key k that is b-bit long, which get hashed into $\mathrm{H}(k) = (k_0, \ldots, k_{2b-1})$
- an integer a determined from $(k_0, k_1, \ldots, k_{b-1})$ (*i.e.* the first half of the private key)
- a public key A computed from the base point B, such that $A = a \cdot B$

The EdDSA scheme

Signing

Algorithm 1 EdDSA Signature

Require: M, $(k_0, k_1, \ldots, k_{2b-1})$, B and A

1: $a \leftarrow 2^{b-2} + \sum_{3 \leq i \leq b-3} 2^i k_i$
2: $r \leftarrow \mathrm{H}(k_b, \ldots, k_{2b-1}, M) \mod \ell$
3: $R \leftarrow r \cdot B$
4: $h \leftarrow \mathrm{H}(R, A, M)$
5: $S \leftarrow (r + ah) \mod \ell$
6: **return** (R, S)

The EdDSA scheme

Verifying

Amongst its differences with ECDSA, the signature computation is fully deterministic!

A signature is considered valid if $R \in E$, $S \in \{0, \ldots, \ell - 1\}$ and the following equation holds in E:

$$S \cdot B = R + \mathrm{H}(R, A, M) \cdot A$$

Previous work: ECDSA

It leaks the key

Playstation 3 attack: two different messages M_1 and M_2 are signed with the same nonce k and produces signatures S_1 and S_2, resp., then:

$$k = (\mathrm{H}(M_1) - \mathrm{H}(M_2))(S_1 - S_2)^{-1}$$

$$a = (sk - \mathrm{H}(M_1))r^{-1}$$

Deterministic version also proved to be weak against fault attacks.

Our fault model

Let's keep it realistic

If you have the device in your hands, you can fault there:

Require: M, $(k_0, k_1, \ldots, k_{2b-1})$, B and A

1: $a \leftarrow 2^{b-2} + \sum_{3 \leq i \leq b-3} 2^i k_i$
2: $r \leftarrow \mathrm{H}(k_b, \ldots, k_{2b-1}, M) \mod \ell$
3: $R \leftarrow r \cdot B$
4: $h \leftarrow \mathrm{H}(R, A, M)$
5: $S \leftarrow (r + ah) \mod \ell$
6: **return** (R, S)

R is left untouched, S is corrupted to a value S'.

We assume a **single** random byte fault.

THE ATTACK AGAINST EdDSA

LEAKS HALF OF THE KEY

The value of a can be then recovered with

$$a = (S - S')(h - h')^{-1} \mod \ell$$

But the second half of the private key is still unknown!

Even if a is known, **it remains impossible** to compute $r = \mathrm{H}(k_b, \ldots, k_{2b-1}, M)$ for a new message M since the values $k_b, \ldots, k_{2b-1}$ **are not known**.

So EdDSA was well thought and is resistant to such faults, right?

Half a key?

Let's randomize the rest, it's a secret!

In fact you can fake signature!

By selecting r as a **random** number, and computing (R, S) accordingly for any message M we would have upon verification that:

$$S \cdot B = (r + \mathrm{H}(R, A, M)a) \cdot B = R + \mathrm{H}(R, A, M)a \cdot B$$

$$= R + \mathrm{H}(R, A, M) \cdot A$$

The verifier cannot detect this!

THE STANDARD COUNTERMEASURES

FOR SUCH THINGS

- ID, n-plication
- Redundancy
- Post-validation, but validation is costly
- Randomness in the generation of $r = \mathrm{H}(k_b, ..., k_{2b-1}, M)$, makes it non-compliant with the RFC

Another way around single faults

It leaks nothing

We propose to use a so-called "infective countermeasure":

1. Compute $h_1 = \mathrm{H}(R, A, M)$ with an implementation.
2. Compute $h_2 = \mathrm{H}(R, A, M)$ with another implementation.
3. Compute

$$S = (r + h_1 + (a - n_i)h_1 + (n_i - 1)h_2) \mod \ell$$

with n_i a random b-bit number, changed at each signature computation.

Platform and library

We needed something to play with

Arduino Nano board:

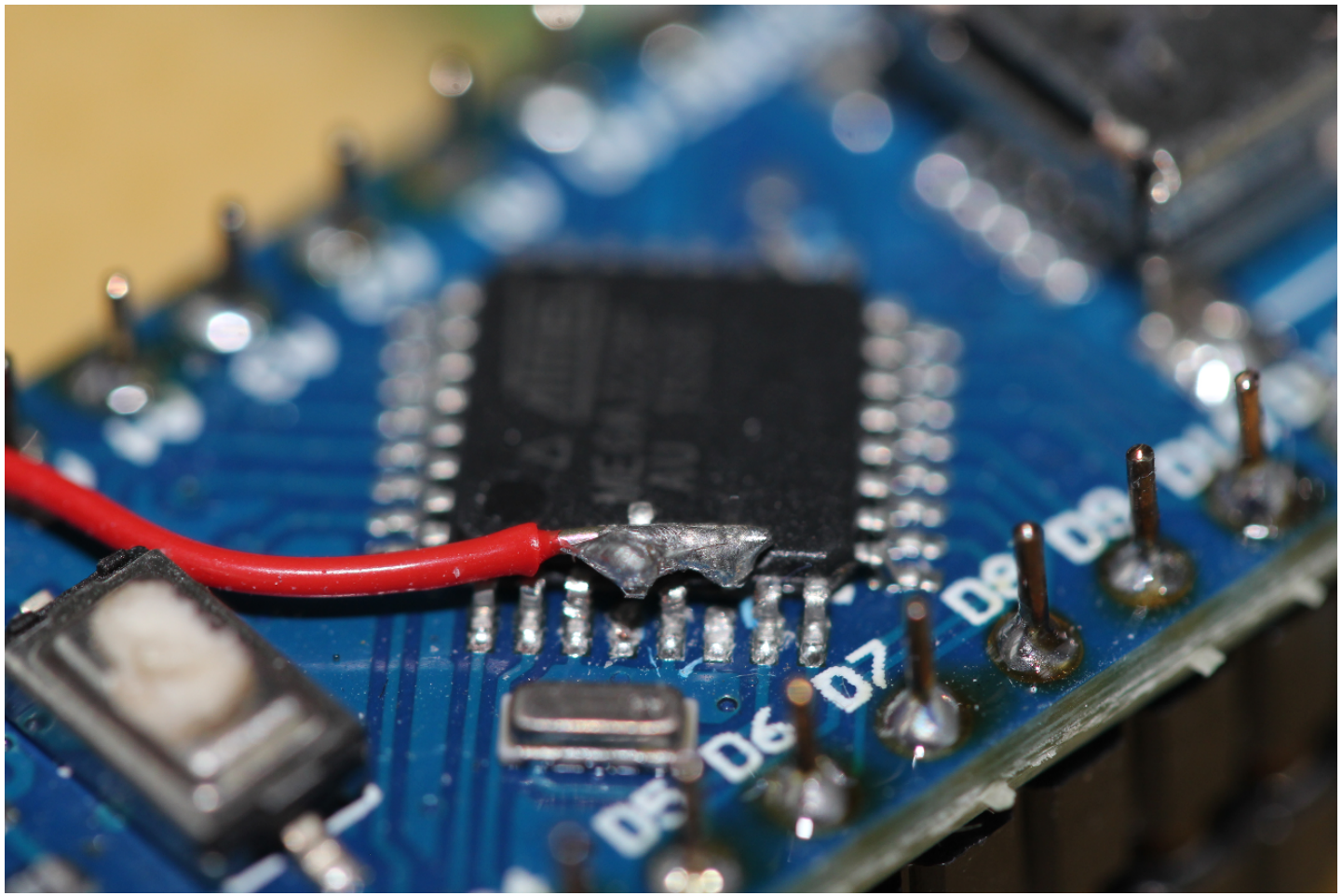
- ATmega328, 8-bit AVR architecture
- 16 MHz Clock speed
- Easy to program thanks to Arduino project

Fortunately, EdDSA was already implemented in Arduino Libs.

Our code is open source, if you want to play.
`https://github.com/kudelskisecurity/EdDSA-fault-attack`

The device

It's small, and it's slow



In practice

Let's bruteforce the error's offset

We fault $h = \mathrm{H}(R, A, M)$ when signing `M = 74657374`:

```
R =    b18b67af0d1bcc4786322748d682c6eef1590f
       ee77e3ba1eccaf71856ce481f3
S =    95635ccb2af746eba982d8d8674d12468db804
       dc8403ea5ddafe3a32dc0f6105
S' =   2d210d14c162d508379562b745004f23b5b163
       13b1bab7b5408c0d586358f200
```

We need to bruteforce the **offset** of the error, to recover the faulted h' value.
(Assuming a single byte fault occurred!)

Require: M, A, (R, S) and (R, S')

1: $h \leftarrow \mathrm{H}(R, A, M)$
2: $i \leftarrow 0$
3: **for** $i < 32$ **do**
4: $\quad e \leftarrow 1$
5: $\quad$ **for** $e < 256$ **do**
6: $\quad\quad h' \leftarrow 2^{8i}e \oplus h$
7: $\quad\quad a \leftarrow (S - S')(h - h')^{-1} \mod \ell$
8: $\quad\quad$ **if** $a \cdot B == A$ **then**
9: $\quad\quad\quad$ **return** a
10: $\quad\quad$ **end if**
11: $\quad\quad e \leftarrow e + 1$
12: $\quad$ **end for**
13: $\quad i \leftarrow i + 1$
14: **end for**
15: **return** ERROR

Actual results

We got half of the key, and we use it

So we recover the first half of the secret key, once we've found the correct h': $a = (S - S')(h - h')^{-1} \mod \ell$

```
a =     110ce4cd00b3bc0c677cd52ac368710a8519e8
        3a17dc00a0e21c6b43aee142f
```

And we can actually use it for signing:

```
def signwitha(m, pk, a):
    r = random.randint(1, 2**256)
    R = scalarmult(B,r)
    S = (r+Hint(encodepoint(R)+pk+m)*a) % ℓ
    return encodepoint(R) + encodeint(S)
```

Conclusion

Here we are

- EdDSA is a really nice EC signature algorithm
- But it might not be a good fit on embedded devices
- Simple faults allow **partial private key recovery**
- Which allows to produce **valid signatures**!
- Ironically, its **determinism** is what doomed it
- **Open question**: what is the best way to **counter** it?