Practical fault attack against the Ed25519 and EdDSA signature schemes

Yolan Romailler, Sylvain Pelissier
Kudelski Security
Cheseaux-sur-Lausanne, Switzerland
{yolan.romailler,sylvain.pelissier}@kudelskisecurity.com

***Abstract*—The Edwards-curve Digital Signature Algorithm (EdDSA) was proposed to perform fast public-key digital signatures as a replacement for the Elliptic Curve Digital Signature Algorithm (ECDSA). Its key advantages for embedded devices are higher performance and straightforward, secure implementations. Indeed, neither branch nor lookup operations depending on the secret values are performed during a signature. These properties thwart many side-channel attacks. Nevertheless, we demonstrate here that a single-fault attack against EdDSA can recover enough private key material to forge valid signatures for any message. We demonstrate a practical application of this attack against an implementation on Arduino Nano. To the authors' best knowledge this is the first practical fault attack against EdDSA or Ed25519.**

***Keywords*-EdDSA; Ed25519; fault attack; digital signature**

I. Introduction

Elliptic Curve Cryptography (ECC) can be used to build digital signature algorithms with a smaller key size than the Digital Signature Algorithm (DSA) with the same level of security. The security of such algorithms is generally based on the Discrete Logarithm Problem (DLP), currently the best known algorithms to solve this problem over elliptic curves are less efficient than ones over finite groups. This allowed the fast adoption of ECC to provide security in the embedded ecosystem where resources are constrained. The most widely used signature algorithm is ECDSA [1]. However, its implementations can suffer from some pitfalls. For example, if the random number generator used during the signature process is flawed, one can recover the private key used for the signature operations [2], [3]. Furthermore, some implementations use addition formulas for Montgomery curves which are not complete; for elliptic curves, an addition formula is called complete if it correctly computes the sum of any two points in the group. As a result a point verification must be performed before the signature operation. If this is not performed or a fault changes the base point before the computation, then the process is carried on another curve which may have weaker security properties [4].

After Snowden's revelations, a loss of trust in the NIST curves occurred. Thus public confidence shifted to alternatives such as the Curve25519 already proposed by Bernstein in 2006 [5]. This curve offers a 128-bit security level and is defined over $\mathbb{F}_p$ where $p = 2^{255} - 19$. One of the curve's main advantages is that its scalar multiplication can be implemented without secret dependent branch and lookup, avoiding the risk of side-channel leakage. More recently, a new digital signature algorithm, namely Ed25519 [6], was proposed based on the curve edwards25519, *i.e.*, the Edward twist of Curve25519. This algorithm was then generalized to other curves and called EdDSA [7]. Due to its various advantages [7], EdDSA was quickly implemented in many products and libraries, such as OpenSSH [8]. It was also recently formally defined in RFC 8032 [9].

The authors of EdDSA did not make any claims about its security against fault attacks. However, its resistance to fault attacks is discussed in [10]–[12] and taken into account by Perrin in [13].

Since more and more embedded devices will implement EdDSA we analysed its resistance to fault attacks. We exploited the determinism of the algorithm to build a fault attack and we demonstrated its practicality, in what is—to the extent of our knowledge—the first practical fault attack against Ed25519 or EdDSA. We also studied which countermeasures are necessary to avoid such fault attacks.

This paper is organized as follows: Section II presents EdDSA; Section III reviews previous fault attacks; Section IV describes our attack against EdDSA and how we applied it on the Arduino Nano platform to an Ed25519 implementation. Then Section V presents possible countermeasures and Section VI concludes.

II. Edwards-curve Digital Signature Algorithm

EdDSA is a public-key signature algorithm similar to ECDSA proposed by Bernstein *et al.* [6]. In RFC 8032 [9] EdDSA is defined for two twisted Edwards curves edwards25519 and edwards448; nevertheless EdDSA can be instantiated over other curves. Generally speaking, a point $P = (x, y)$ lies on E, a twisted Edwards curve if it verifies the following formula:

$$ax^2 + y^2 = 1 + dx^2y^2$$

where a, d are two distinct, non-zero elements of the field $\mathbb{K}$ over which E is defined. For instance, edwards25519 is defined over $\mathbb{F}_p$ with $p = 2^{255} - 19$ like Curve25519. As Bernstein *et al.* demonstrated, edwards25519 is birationally equivalent to Curve25519 [14]; thus the difficulty of solving the DLP is equivalent for both curves. Likewise, edwards448

 DOI: 10.1109/FDTC.2017.12

is defined over $\mathbb{F}_q$ with $q = 2^{448} - 2^{224} - 1$ and is built to offer a security level of 224 bits.

An important property of these curves with neutral element $(0, 1)$ is to have an addition law, noted "+", with a complete formula. For both curves, the addition law formula is given by:

$$(x_1, y_1) + (x_2, y_2) = \left(\frac{x_1 y_2 + y_1 x_2}{1 + d x_1 x_2 y_1 y_2}, \frac{y_1 y_2 - a x_1 x_2}{1 - d x_1 x_2 y_1 y_2} \right).$$

This formula can be used for point doubling and points addition, even when the neutral element is involved. Adding a point P to itself n times is defined to be scalar multiplication and is denoted $n \cdot P$.

EdDSA uses a private key k that is b-bit long and a hash function H that produces a $2b$-bits output. One common instance is to use SHA-512 for $b = 256$ bits. An integer a is determined from $H(k) = (h_0, h_1, \ldots, h_{2b-1})$ with

$$a = 2^{b-2} + \sum_{3 \leq i \leq b-3} 2^i h_i \, .$$

The public key A is then computed from the base point $B \neq (0, 1)$ of order ℓ, chosen as per the EdDSA specifications [6], such that $A = a \cdot B$. In practice the public key and the signatures are output according to the encoding defined in RFC 8032. Since there is a one-to-one relation between curve elements and encoded values we do not detail the encoding in our description.

The signature (R, S) of a message M is computed according to Algorithm 1.

Algorithm 1 EdDSA Signature

Require: M, $(h_0, h_1, \ldots, h_{2b-1})$, B and A
1: $a \leftarrow 2^{b-2} + \sum_{3 \leq i \leq b-3} 2^i h_i$
2: $h \leftarrow H(h_b, \ldots, h_{2b-1}, M)$
3: $r \leftarrow h \mod \ell$
4: $R \leftarrow r \cdot B$
5: $h \leftarrow H(R, A, M)$
6: $S \leftarrow (r + ah) \mod \ell$
7: **return** (R, S)

Amongst its differences from ECDSA, the signature computation is deterministic, *i.e.*, for a given message M, if multiple signatures are computed they will all be identical.

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

$$8S \cdot B = 8 \cdot R + 8H(R, A, M) \cdot A \, .$$

Verification without the cofactor 8 is a stronger way to verify a signature [7]. The algorithm's performance, ease of implementation, small key and signature sizes, all pave the way for EdDSA's rapid adoption, specially in embedded devices.

III. Fault attacks

The principle of fault attacks is to disturb the normal behaviour of a device making it output erroneous results or bypass certain operations. If the fault happens during the computation of sensitive operations, the erroneous outputs may be used to recover sensitive information.

One of the simplest fault attacks consists in varying the supplied voltage leading an unprotected processor to exhibit non-expected behaviour. This attack, called voltage glitch, has been successful in the past and remains possible against unprotected microcontrollers as shown against the Arduino Nano platform during the second Riscure challenge [15]. This method is easy to implement and non-invasive and thus we have chosen it to practically demonstrate our attack. Another simple fault attack is called clock glitching and consists in varying the speed of the clock signal of a mircocontroller. Although this attack has been demonstrated to be successful [16] we did not try it during our investigation. These attacks have been widely studied and nowadays most modern secure devices integrate countermeasures against them. Other advanced fault attacks exist and are actively used such as electromagnetic [17], body biasing [18], optical [19] and laser fault injection [20]. Since a basic voltage glitch was sufficient to demonstrate our attack we did not attempt more advanced fault injections, but our attack is perfectly compatible with those methods.

The first fault attack against a public-key algorithm was proposed by Boneh *et al.* [21] against RSA. They showed that injecting a single fault during the modular exponentiation computation is enough to recover the private key. This was later practically realized by Amüller *et al.* [22] on a smartcard using voltage glitch fault injection. They used power profiling to find the correct timing for the fault injection and then overdrove the supplied voltage during a short period to disturb one of the operations. They also tested the same scenario with various countermeasures.

The first fault attack against ECC was proposed by Biehl *et al.* [4]. They showed that if the base point B is corrupted during signature operations, and if the ECC implementation does not check whether it lies on the curve or not, then the computation will be performed on another curve where the DLP may be easily solved using the Pohlig-Hellman algorithm. This attack was later improved by Ciet *et al.* [23]. They showed that the faulted base point value can be recovered from the faulty output. They also showed that a fault occurring in the system parameters such as the field definition or the curve coefficient values can lead to recovery of the secret key. Another attack was proposed by Schmidt and Medwed [24] against the ECDSA algorithm. Their fault model was to skip a loop step during a scalar multiplication operation. The main difficulty in applying fault attacks to ECDSA is its use of a random nonce for each signature. However, for EdDSA the same message can be signed

several times and the signature will not change, since it does not rely on random nonces, but on deterministic generation of the nonces. We used this to our advantage in order to mount our fault attack. This property was also used to build side-channel analysis against deterministic digital signature schemes by Seuschek *et al.* [25].

IV. Attack against EdDSA

Our attack is based on faulting operation 5 of Algorithm 1 above during the computation of the signature. If the output of the hash is faulted and changed to the value $h' \neq h$ then the faulty signature will be (R, S') *i.e.*, only the second part of the signature is changed. The value of a can be then recovered with

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

This property has already been noticed in [11]–[13] and holds for other variants of the EdDSA scheme [13], as well as other deterministic signature schemes [10], [26]. The value of h can be computed from R, A and M which are known, however, the value h' has to be known or guessed. This limitation can be overcome if the fault model is characterized properly and the faulted value can be guessed in a post-processing phase. For example, we considered the fault model to be a random byte $e \in \{1, 2, ..., 255\}$ injected at a random offset i after the hash computation, *i.e.*, $h' = 2^{8i}e \oplus h$. For example, $i \in \{0, 1, ..., 31\}$ for edwards25519 and $i \in \{0, 1, ..., 57\}$ for edwards448. In the first case, all the $255 \cdot 32 = 8160$ possibilities can be tested for h' until the computation of $a \cdot B$ matches the public key value A. The complete fault verification algorithm for this fault model is given in Algorithm 2.

Algorithm 2 Ed25519 fault post-processing

Require: M, A, (R, S) and (R, S')
1: $h \leftarrow H(R, A, M)$
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`

If the previous algorithm outputs `ERROR`, then the fault injected did not correspond to a single random byte error. Other fault models can be considered depending on the hardware, and the computation power of the attacker. For instance, a random multi-byte error can be considered and in this case the attacks iterates until a larger value of e. Another possibility is a random byte error occurring before the hash computation, *e.g.*, on the value of the message M. The attack still applies, except that it needs some tuning: for the later example, a hash computation $H(R, A, M')$, for M' the faulty message candidate, must be performed at step 6 of Algorithm 2 and thus the cost of the error detection grows significantly with the size of the message.

In addition, if the message M is not known, the attack is still feasible as only the value $h - h'$ is used during the computation of a. In our fault model, the difference will be in the range $\{-255 \cdot 2^{8i}, -254 \cdot 2^{8i}, ..., -2^{8i}, 2^{8i}, ..., 254 \cdot 2^{8i}, 255 \cdot 2^{8i}\}$ for a given offset i. Thus we can test all of the 511 values for all the possible offsets until a valid value a is found.

Even if a is known, it remains impossible to compute $r = H(h_b, ..., h_{2b-1}, M)$ for a new message M since the values $h_b, ..., h_{2b-1}$ are not known. This was considered as a structural resistance by Barenghi and Pelosi [10]. However, by selecting r as a random number, and computing a new (R, S) similarly for any message M we have:

$$8S \cdot B = 8(r + H(R, A, M)a) \cdot B = 8 \cdot R + 8H(R, A, M)a \cdot B$$

$$= 8 \cdot R + 8H(R, A, M) \cdot A$$

The verification equation still holds. Thus it is possible to forge valid signatures for any message. This demonstrates that the design of EdDSA is sensitive to fault attacks and great care should be taken when faults are part of the threat model.

A. Attack simulation

To demonstrate our attack, we implemented it in Python for the curve edwards25519. We based our code on the original implementation of Bernstein [27]. The program randomly generates a correct signature, then generates a fault corresponding to a random byte inserted at a random offset at the output of the hash function. Finally we applied Algorithm 2 on the faulted results to recover a. As soon as the correct value for a has been found we forge a new signature for a different message and we verify the signature. On a common laptop with an Intel Core i5-3320M processor, on average our program takes 6.27s running on a single core to recover the value a. An example of the program execution is given in Appendix A. Our program was not meant to be efficient but it demonstrates that this attack is practical. Our code is available online [28]. The same method applies for other curve like edwards448.

B. Practical verification

We also tested our attack against the Arduino Nano platform running the Cryptographic Library from Arduino-Libs [29]. The Arduino Nano board is based on the ATmega328 microcontroller and is fully compatible with the Arduino software. The library implements the Ed25519 algorithm, with H being SHA-512 solely in software *i.e.*no hardware accelerator is available on this platform. We created a small program performing a signature computation with a fixed secret key when a character is sent on the serial connection and prints its signature. Our test code is also publicly available [28]. We noticed in the code performing the last SHA-512 operation, that the `SHA512::finalize` function performs a loop to copy the resulting hash in big endian with the `htobe64` function. Faulting this loop could allow us to have the required fault to perform the attack. If the fault consists in one random byte, the value of h' can be recovered when trying all the $32 \cdot 255 = 8160$ possibilities offline.

To have timing information about the different operations, we used the GPIO pins of the Arduino Nano to indicate when an operation starts and finishes. The resulting traces are shown in Figure 1.

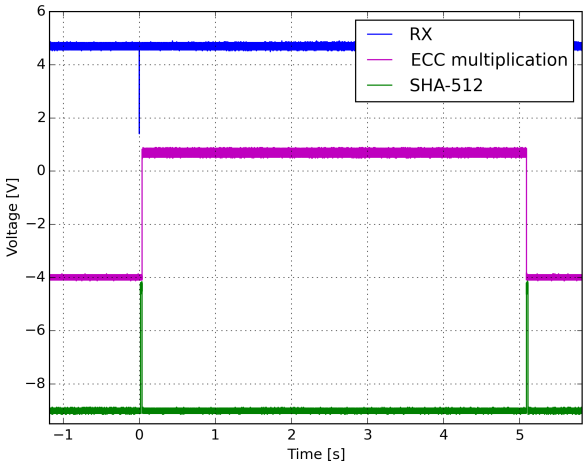

Figure 1: Signature timing

The RX signal is the serial connection signal and it indicates the beginning of the signature operation. The whole signature operation lasts 5.11s on average and one can remark that most of the signature time is spent in the scalar multiplication.

To perform glitches, we unsoldered the VCC pins 4 and 6 of the TQFP package from the PCB and we supplied them externally with a pulse generator as shown in Figure 2.

AVCC pin 18 was also disconnected to avoid any internal power supply. The microcontroller worked properly even when the VCC was lower than 3.9V. We used this voltage level supply to perform our experiments. We also modified the code and set a GPIO pin to be high during the critical

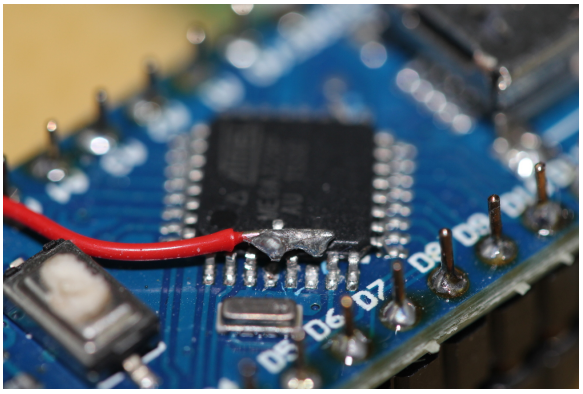

Figure 2: VCC external connection

copy loop at the end of the second SHA-512 operation. This step takes about 141.66µs. The GPIO signal was used as a trigger to launch the glitch during the loop operation. We performed glitches during the time where the GPIO pin was high only. The parameters for a glitch are its width, *i.e.*, the time when the VCC is set to be low, and its depth, *i.e.*, the level where the VCC was lowered during the glitch. We kept the slope of the glitch as steep as possible, i.e. the glitch is only a few ns. After empirical tests, we discovered that when the glitch width was around 30ns and the depth 0V we obtained faults matching our fault model. This allowed us to recover the value a derived from the private key, as shown in Appendix B. We show in Figure 3 the signals we obtained during this experiment as well as a zoom on the voltage glitch form that gave these results.

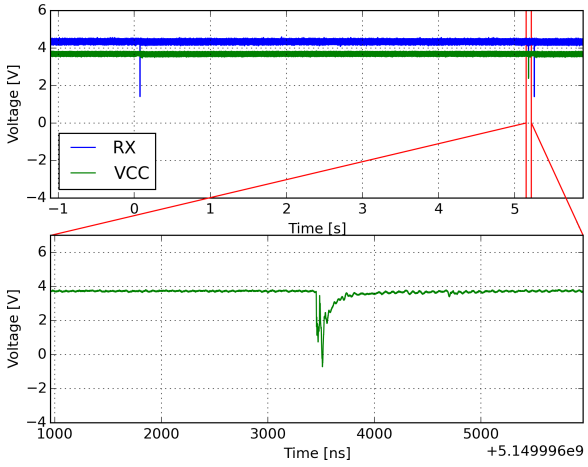

Figure 3: VCC glitch form

During a normal attack scenario, the attacker would not be able to modify the code to add GPIO synchronization point. The attacker has to be synchronized with an external or internal event, for example the serial communication or side channel leakage such as power consumption or electromagnetic emanations. In the Ed25519 signature implementation,

the value a is derived from k which is time independent of the value of k. Then the value r is computed with SHA-512. This operation is also time constant when the size of M does not change. Then $R = r \cdot B$ is computed. In ArduinoLibs this operation is CPU cycle constant. Finally $H(R, A, M)$ is computed. Hence, the global timing for the fault injection remains fairly constant, varying only with small fluctuations of the internally generated clock if used. The attacker may choose to synchronize her glitch with the serial communication to the Arduino board and does not need any additional steps to synchronize with side channel leakages. This means that as soon as the timing of the hash operation to be faulted is known, it can be repeated on other similar devices to recover their secrets. The only large variations in the timings could come from microcontroller interrupts or clock division circuitry.

We synchronized our glitch with the serial communication which initiates the signature operation, with a delay based on the previously gained timing information. In this case some jitter is introduced by the serial communication interruptions but glitching several times with the same delay allows to fault the final SHA-512 copy loop and thus to obtain the desired faulted signature. Appendix B gives some examples of the faults obtained during these experiments, as well as the private key material recovered. ince this approach was successful we did not use more advanced methods to synchronize our fault injection with other events.

V. Countermeasures

Besides generic countermeasures against fault attacks, we can highlight three ways to avoid being vulnerable to this attack:

- Using a random r value is completely compatible with the verification process and undetectable upon validation, the sole way to detect the difference is to sign twice the same message and check if the same signature is produced, since it would then give different signatures (both of which are still validated by the public key without issue). However, doing so brings back a random number generator in the implementation of the signature scheme and renders it non-compliant with the newly released RFC 8032 [9]. This approach has already been implemented in the XEdDSA and VXEdDSA signature algorithms of the Signal protocol [13].
- Validating the signature at the end of the process would catch a fault but it would likely not protect effectively against scenarios where an attacker can inject faults in the system at will, as is typically the case in embedded systems. Besides, validation is a longer operation than the signature operation and thus is inefficient.
- Since our attack targets the output of the second hash function, one can think of securing this computation. For example it may be possible to double the computation and compare both results to resist a single fault injection. However, the attack may be applied to the input of the hash function and thus the input has to be protected as well. This approach does not prevent other attack paths, or other fault models.
- Last, but not least, we propose to compute S in a way which would prevent faults located in both the hash and the message to allow the recovery of the private material a:
  1) Compute $h_1 = H(R, A, M)$ with an (hardware) implementation.
  2) Compute $h_2 = H(R, A, M)$ 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.

  This works since:
  - If $h_1 = h_2 = h$, we have $S = r + ah \mod \ell$.
  - If $h_1 \neq h_2$, and a fault occurred on h_1:
  $$\begin{aligned}(S - S') &= (r + h + (a - n_0)h + (n_0 - 1)h) \\ &\quad - (r + h'_1 + (a - n_1)h'_1 + (n_1 - 1)h) \\ &= h - h'_1 + ah - ah'_1 - n_1h + n_1h'_1 \\ &= (a + 1 - n_1)(h - h'_1) \mod \ell,\end{aligned}$$
  the introduction of the random n_1 hinders the recovery of a.
  - If $h_1 \neq h_2$, and a fault occurred on h_2:
  $$\begin{aligned}(S - S') &= (r + h + (a - n_0)h + (n_0 - 1)h) \\ &\quad - (r + h + (a - n_1)h + (n_1 - 1)h'_2) \\ &= n_1h - h - n_1h'_2 + h'_2 \\ &= (n_1 - 1)(h - h'_2) \mod \ell,\end{aligned}$$
  is no longer linked with the private material a at all.

  This is similar to the so-called "fault infective computations" introduced by [30]. Thus this method prevents faults in h_1 or h_2 from outputting a value which leaks the actual value of a, except if two same faults are performed on both h_1 and h_2. This is considered hard to do—especially if both implementations differ.

We believe that the impact of using poor randomness in EdDSA is far less than for ECDSA and either our first or last proposal should be preferred to prevent fault attacks.

VI. Conclusion

We demonstrated the first known practical fault attack against EdDSA and Ed25519, which shows that those algorithms, although featuring excellent protections against many classes of attacks by design, are vulnerable to fault attacks.

Thus, as with other algorithms, fault protections must be built in, when deployed on embedded devices. It is important to note that our work is unlikely to—and indeed should not—hinder the prevalence of these well thought out and well designed algorithms in most cryptographic protocols, since fault attacks are generally not part of their threat model. That being said, as we have shown, mitigations are possible when faults are part of the model which may allow their safe use without further changes.

ACKNOWLEDGMENTS

The authors would like to thank Andrew McLauchlan for his help setting up the glitch bench and conducting the practical experiment, as well as Karine Villegas for her help designing a novel countermeasure. We would also like to thank the multiple reviewers for their insightful comments on our paper.

REFERENCES

[1] NIST. FIPS 186-4 Digital Signature Standard (DSS), 2013.

[2] Phong Nguyen and Igor Shparlinski. The Insecurity of the Elliptic Curve Digital Signature Algorithm with partially known nonces. *Designs, codes and cryptography*, 30(2):201–217, 2003.

[3] Filippo Valsorda. Exploiting ECDSA Failures in the Bitcoin Blockchain. *HITBSecConf - Malaysia*, 2014.

[4] Ingrid Biehl, Bernd Meyer, and Volker Müller. Differential Fault Attacks on Elliptic Curve Cryptosystems. In Mihir Bellare, editor, *Advances in Cryptology - CRYPTO 2000: 20th Annual International Cryptology Conference Santa Barbara, California, USA, Proceedings*, 2000.

[5] Daniel J. Bernstein. Curve25519: new Diffie-Hellman speed records. In *International Workshop on Public Key Cryptography*, pages 207–228. Springer, 2006.

[6] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 124–142. Springer, 2011.

[7] Daniel J. Bernstein, Simon Josefsson, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. EdDSA for more curves. Cryptology ePrint Archive, Report 2015/677, 2015. URL `http://eprint.iacr.org/2015/677`.

[8] Ianix. Things that use Ed25519, 2017. URL `https://ianix.com/pub/ed25519-deployment.html`.

[9] Ilari Liusvaara and Simon Josefsson. Edwards-Curve Digital Signature Algorithm (EdDSA). RFC 8032, January 2017. URL `https://rfc-editor.org/rfc/rfc8032.txt`.

[10] Alessandro Barenghi and Gerardo Pelosi. A note on fault attacks against deterministic signature schemes. In *International Workshop on Security*, pages 182–192. Springer, 2016.

[11] Benedikt Schmidt. [curves] EdDSA specification, 2016. URL `https://moderncrypto.org/mail-archive/curves/2016/000768.html`.

[12] Maarten Baert. Ed25519 leaks private key if public key is incorrect #170, 2014. URL `https://github.com/jedisct1/libsodium/issues/170`.

[13] Trevor Perrin. The XEdDSA and VXEdDSA Signature Schemes, revision 1, 2016. URL `https://whispersystems.org/docs/specifications/xeddsa/`.

[14] Daniel J. Bernstein, Peter Birkner, Marc Joye, Tanja Lange, and Christiane Peters. Twisted edwards curves. Cryptology ePrint Archive, Report 2008/013, 2008. URL `http://eprint.iacr.org/2008/013`.

[15] Eloi Sanflix and Andres Moreno. RHME2 CTF challenges and solutions. *Insomni'hack*, 2017.

[16] Brett Giller. Implementing practical electrical glitching attacks. *Black Hat Europe*, 2015.

[17] Nicolas Moro, Amine Dehbaoui, Karine Heydemann, Bruno Robisson, and Emmanuelle Encrenaz. Electromagnetic fault injection: towards a fault model on a 32-bit microcontroller. In Wieland Fischer and Jorn-Marc Schmidt, editors, *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2013 Workshop on*, pages 77–88. IEEE Computer Society, 2013.

[18] Noemie Beringuier-Boher, Marc Lacruche, David El-Baze, Jean-Max Dutertre, Jean-Baptiste Rigaud, and Philippe Maurine. Body biasing injection attacks in practice. In *Proceedings of the Third Workshop on Cryptography and Security in Computing Systems*, CS2 '16, pages 49–54, 2016.

[19] Sergei Skorobogatov and Ross Anderson. *Optical Fault Induction Attacks*, pages 2–12. Springer Berlin Heidelberg, 2003.

[20] Falk Schellenberg, Markus Finkeldey, Nils Gerhardt, Martin Hofmann, Amir Moradi, and Christof Paar. Large laser spots and fault sensitivity analysis. In *2016 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 203–208, 2016.

[21] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the importance of checking cryptographic protocols for faults. In Walter Fumy, editor, *Advances in Cryptology EUROCRYPT 97*, volume 1233 of *LNCS*, pages 37–51. Springer Berlin Heidelberg, 1997.

[22] Christian Aumüller, Peter Bier, Peter Hofreiter, Wieland Fischer, and Jean-Pierre Seifert. Fault attacks on RSA with CRT: Concrete Results and Practical Countermeasures. Cryptology ePrint Archive, Report 2002/073, 2002. URL `http://eprint.iacr.org/2002/073`.

[23] Mathieu Ciet and Marc Joye. Elliptic curve cryptosystems in the presence of permanent and transient faults. Cryptology ePrint Archive, Report 2003/028, 2003. URL `http://eprint.iacr.org/2003/028`.

[24] Jörn-Marc Schmidt and Marcel Medwed. A fault attack on ecdsa. In *Proceedings of the 2009 Workshop on Fault Diagnosis and Tolerance in Cryptography*, FDTC '09, pages 93–99. IEEE Computer Society, 2009.

[25] Hermann Seuschek, Johann Heyszl, and Fabrizio De Santis. A Cautionary Note: Side-Channel Leakage Implications of Deterministic Signature Schemes. In *Proceedings of the Third Workshop on Cryptography and Security in Computing Systems*, CS2 '16, pages 7–12. ACM, 2016.

[26] Thomas Pornin. Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA). RFC 6979, August 2013. URL `https://rfc-editor.org/rfc/rfc6979.txt`.

[27] Daniel J. Bernstein. EdDSA software, 2017. URL `https://ed25519.cr.yp.to/python/ed25519.py`.

[28] Sylvain Pelissier and Yolan Romailler. Eddsa-glitch-attack, 2017. URL `https://github.com/kudelskisecurity/EdDSA-fault-attack`.

[29] ArduinoLibs. Cryptographic library, 2016. URL `https://rweather.github.io/arduinolibs/crypto.html`.

[30] Yen Sung-Ming, Seungjoo Kim, Seongan Lim, and SangJae Moon. RSA speedup with residue number system immune against hardware fault cryptanalysis. In *International Conference on Information Security and Cryptology*, pages 397–413. Springer, 2001.

APPENDIX

We propose here an example to test the attack and its different phases, as well as practical values obtained when performing the fault attack. This is also part of our python code [28].

A. Example

We are using Ed25519. Let the key pair be:

```
k = 523b05d3a02887f67eef8bec3f723dc2c17
    73200d779fa8d1f5f2afbd84ef529
A = 35e7d4d097deb97470d03ff2ae9b515d090
    a54fc76f50e95bac0d21a86bd5d3c
```

We sign the message "test" with the secret key `k` and we obtain thus:

```
M = 74657374
R = b18b67af0d1bcc4786322748d682c6eef15
    90fee77e3ba1eccaf71856ce481f3
S = 95635ccb2af746eba982d8d8674d12468db
    804dc8403ea5ddafe3a32dc0f6105
```

Now, one can verify the signature `R|S` against the message "test" using `pk`, for `|` the concatenation operation and see that it is valid.

Now, here is a faulted signature for the same message and key pair:

```
M = 74657374
R' = b18b67af0d1bcc4786322748d682c6eef15
     90fee77e3ba1eccaf71856ce481f3
S' = 2d210d14c162d508379562b745004f23b5b
     16313b1bab7b5408c0d586358f200
```

As one can see, the `R` and `R'` values are identical, while the `S` and `S'` values differ, this means there is a problem and indeed the signature `R'|S'` is not valid against the message "test".

We can thus compute `a` as shown in Section IV and we obtain through the explained brute-force computation using Algorithm 2 that there is effectively a single octet error, which occurred at offset 5 of the hash `H(R,A,M)`, which leads us to:

```
a = 110ce4cd00b3bc0c677cd52ac368710a851
    9e83a17dc00a0e21c6b43aee142f
```

Using that `a` we can now sign other messages of our choice, using random `r` values. For example the following would be a signature that is valid under `A` of the message "anothermessage":

```
M = 616e6f746865726d657373616765
R = da406c4bd799398a53aadedc1539a188e54
    3822724c8b836ef6e8e14ed1a17a5
S = 580cb166d5529434bdbec155b6d59b98b1a
    de1cd79d57de7c987b99e55bd770c
```

while the actual signature when signing using `k` would have been:

```
R =  7636c2bcf86567843a5c4a04f896f5041da
     6021aff5de1f1442c6e83bda5b93b
S =  a363ef93ae1a31cd993000d180955180f81
     2ea505b67f0f8b82493b296a52a09
```

yet both are valid for verifying the message "anothermessage" using the same `A` as public key.

B. Practical faults on Arduino Nano

During our experiments, the public key was:

```
A =  9c74125c9a2bcfbc6107bcdb00b79b93811
     2b3ec2fa4db67b7128f2a8cc9da43
```

We performed a voltage glitch during the signature of the message "test1" padded with zeros. Here is some example of the fault we obtained:

```
M =  7465737431000000000000000000000000
     00000000000000000000000000000000
R =  d49bd03b98b85afbcff46bfacf9af673662
     a253871ec9a56d035aefad2166e54
S1 = 49f230f6967071855663997a5fe0562f73c
     aec8cceaa202a8ba39c1cee807c0b
R =  d49bd03b98b85afbcff46bfacf9af673662
     a253871ec9a56d035aefad2166e54
S2 = b2811f21f6e1f7d92d393621bc0584ee7b2
     89a357448ec54bd3d7f21d42aee0f
R =  d49bd03b98b85afbcff46bfacf9af673662
     a253871ec9a56d035aefad2166e54
S3 = 67a7f6fc4a0270d5b2e238777b4453cd973
     3e4f4052b5b097c21fc3062b8b008
R =  d49bd03b98b85afbcff46bfacf9af673662
     a253871ec9a56d035aefad2166e54
S4 = 58b12c240028c2e854a1f373b874732e655
     b1780fca01001fe12691b93879500
R =  d49bd03b98b85afbcff46bfacf9af673662
     a253871ec9a56d035aefad2166e54
S5 = 4b4934ee480c076ab4b19ed375044f4e601
     ccb7a751b4b81cf8c607d9122e00b
```

We ran our program on this signature, and we found an error at offset 6 for $S1$, 4 for $S2$, 2 for $S3$ and 1 for $S4$. $S5$ was not corresponding to our fault model, since more than one byte was faulted. For the first four signatures, the program could recover the value:

```
a =  ba1a351f0a13d9c58d23b60fa34ce0a4dd1
     7ba37da6bc6a41fc6ed4541f4682
```