IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2025, No. 4, pp. 1010–1052. DOI:10.46586/tches.v2025.i4.1010-1052

Fault Attacks on ECC Signature Verification

Kevin Schneider[1], Lukas Auer[1,2] and Alexander Wagner[1,2]

[1] Fraunhofer AISEC, Garching, Germany, first.last@aisec.fraunhofer.de
[2] Technical University of Munich, Munich, Germany,

Abstract. Signature verification operations used in secure boot or firmware updates are the foundation of trusted devices. ECC-based signature schemes are preferred for these applications due to their smaller key and signature sizes. Despite their widespread use, we would like to highlight that there is no research available that analyzes the resilience of ECC-based signature verification operations against fault attacks. Therefore, we thoroughly investigate the feasibility of fault attacks on ECC-based signature verification. We cover both theoretical and implementation-specific attacks. We demonstrate that faults in elliptic curve points and parameters allow an adversary to forge signatures in ECGDSA and ECSDSA, while ECDSA and EdDSA remain resilient. The weakness lies in the Weierstraß curves used in the affected schemes. This allows an adversary to perform cryptographic operations on much weaker curves by corrupting at least a single bit. To assess the severity in practice, we evaluate two open-source secure boot implementations—MCUboot and wolfBoot—that use fault injection hardening. Interestingly, these examples do not employ any hardening within the underlying cryptographic libraries. We discovered several attacks on the implementation of the ECDSA and EdDSA verification algorithms. Here, a single instruction skip is sufficient to accept trivially forged signatures. To improve these and future implementations, we propose effective and efficient countermeasures. Our work fills a critical gap to motivate further research for more resilient cryptographic implementations.

Keywords: fault injection · elliptic curve cryptography · signature verification · ECDSA · EdDSA · ECSDSA · ECGDSA · secure boot

1 Introduction

As embedded systems and the Internet of Things are becoming an ever more important part of our daily lives, securing their cryptographic functions against fault attacks (FAs) is more crucial than ever. One of the first documented FAs on signature verification was conducted by Jean-Pierre Seifert [Sei05], who discovered how a fault in the modulus could be used to accept forged RSA signatures. He credits an attack that has been performed on an Xbox game console to execute unsigned code [Hua03] as his inspiration. While fault injection attacks on signature schemes have increasingly been studied over the last two decades, most of the attention has been given to signature generation and much less to the verification process.

Proper signature verification is essential for system security. As exemplified by the Xbox attack, there are scenarios where attackers want to bypass this verification. Secure bootloaders play a critical role in verifying the authenticity of software and firmware, especially in systems that are physically exposed to unauthorized individuals—a common scenario for embedded systems. They ensure that only software signed by the manufacturer is executed and thereby prevent malicious firmware from being run.

In the context of secure bootloaders, an attacker is able to freely choose an arbitrary message, i.e., a firmware image, and a corresponding signature. Without knowledge of

Received: 2025-04-15 Accepted: 2025-06-15 Published: 2025-09-05

the private key corresponding to the public key securely stored on the target device, the attacker is unable to generate a valid signature. Invalid signatures are detected by the bootloader, which aborts the boot process as a result. This is, however, only true if the attacker does not have physical access to the device. If an adversary can inject a fault that tricks a device into erroneously accepting a forged signature, they are able to bypass a secure boot process that is based on such a signature.

In the world of embedded systems and low power SoCs, signature schemes based on elliptic curves (EC) have rapidly superseded earlier systems like DSA and RSA due to their smaller key sizes [Ela20] and, in some cases, better performance. FAs on the verification of EC-based signatures have not yet been widely studied in literature. Previous attacks focused on RSA [Sei05, Mui06, MKS12], hash- [WWO+23], and lattice-based [VBH23, BBK16] schemes.

Many open-source bootloaders for embedded systems like MCUboot [Tru24], wolfBoot [wol24], and OpenTitan [low24] have incorporated software-based countermeasures to defend against FAs. These countermeasures are implemented as precise redundancies, only added where needed to minimize the impact on the system. However, they are limited to the codebase of the bootloader. Any external dependencies, such as cryptographic libraries used for the signature verification, remain unprotected. We identified this pattern in all bootloaders we analyzed. Leaving critical parts of a system unprotected poses a significant threat to overall security. It is thus important to study potential attacks and vulnerabilities in order to properly secure them.

This paper addresses this critical gap by methodically exploring FAs on the verification process of EC-based signature schemes.

1.1 Contributions

Due to the evident threat of FAs on signature verification, the intention is to show how EC-based signature schemes can be manipulated to accept arbitrary or forged signatures. Simultaneously, we will propose hardening techniques and countermeasures to protect a system against the identified attacks. In summary, we make the following contributions.

- We present a novel attack on the ECGDSA and ECSDSA signature schemes, making it possible to forge a signature using a single-bit flip.
- We describe ten FAs on the ECDSA and EdDSA signature verification which allow forged signatures to be accepted, showing that secure bootloaders based on these schemes can be exploited even in the presence of solid hardenings in their core components.
- We demonstrate how the signature schemes can be hardened against the described faults.

2 Preliminaries

This section covers the basics of Elliptic Curve Cryptography (ECC) and the signature schemes analyzed in this paper. It also discusses FAs on embedded systems and common countermeasures.

2.1 Symbols and Notation

Table 1 shows the symbols and notations used throughout this paper.

Table 1: Symbols and notations

Notation / Symbol	Meaning
$\mathbb{F}_q$	A finite field of size q.
$\mathcal{O}$	The identity element of an elliptic curve group.
G	The base (or generator) point that generates a subgroup on a curve.
n	The order of the base point G.
d	A private key, an integer chosen uniformly at random.
P	A public key point on the curve.
$P.x$, $P.y$ / x_p, y_p	The x- and y-coordinates of a point P in affine representation.
$H(m)$	The hash value of a message m.
$[s]Q$	A multiplication of a scalar s with a point Q on an elliptic curve.
$a\|\|b$	A concatenation of the strings or byte arrays a and b.

2.2 Elliptic Curve Cryptography

An elliptic curve E is a curve defined over a field with properties chosen to make it particularly suitable for cryptographic applications. In ECC, an EC is defined over a finite field and can be described by the generalized Weierstraß equation

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6.$$

If the underlying finite field is not of characteristic 2 or 3, we can simplify this equation to the short Weierstraß form [Was08]

$$y^2 = x^3 + ax + b.$$

The solutions for this equation define all points on a given elliptic curve. This set of points on an EC forms a group under addition [Was08] that can be extended to an abelian group by adding the point at infinity $\mathcal{O}$ as the neutral element. To complete the group definition, an addition operation is required.

Elliptic Curve Group Law The addition operation on the group formed with the points on an EC is an essential operation in ECC. Apart from the affine coordinate system with point representation $P = (x_p, y_p)$ used throughout this paper, other systems are commonly used for faster point addition. For example, when working with Jacobian coordinates, a costly inversion can be replaced by cheaper multiplications [Fay14] to improve the overall performance of EC operations. Since the choice of coordinate system does not affect the attacks we discovered, we chose to work with affine coordinates for their simplicity.

The addition group law for $R = P + Q \neq \mathcal{O}$ on a curve in short Weierstraß form is given by

$$x_r = \lambda^2 - x_p - x_q,$$
$$y_r = (x_p - x_r) \cdot \lambda - y_p.$$

If the summands are not equal, i.e., if $x_p \neq x_q$, λ is defined as

$$\frac{y_q - y_p}{x_q - x_p}$$

and if $x_p = x_q$ and $y_p \neq 0$, as

$$\frac{3x_p^2 + a}{2y_p}.$$

2.3 Elliptic Curve Digital Signature Algorithm

The Elliptic Curve Digital Signature Algorithm (ECDSA) is a well-known variant of the Digital Signature Algorithm (DSA) based on EC. It bases its security on the Elliptic Curve Discrete Logarithm Problem (ECDLP), which is defined as the problem of finding a scalar value k such that $R = [k]G$, where R and G are two fixed points on an elliptic curve over a finite field. The operation $[k]G$ is called a scalar multiplication of the EC-point G by an integer k, which is a key operation for all EC-based signature schemes. The ECDLP is significantly harder than the Discrete Logarithm Problem (DLP) used in DSA and thus requires much smaller key lengths for equivalent levels of security. As a consequence, requirements for processing power, storage space, and bandwidth can be reduced [JMV01] which is especially useful for microprocessors and embedded systems.

The verification process for an ECDSA signature is shown in Algorithm 1. Here, G is called the base point (or generator) of the curve, which is part of the curve specification, and P denotes the public key point.

Algorithm 1 ECDSA signature verification

Input: message m, signature pair (r, s)
Output: signature validity

1: **if** r and s not in range $]0; n[$ **then return** false
2: $h = H(m)$, where H is a cryptographic hash function outputting values $< n$
3: $s_{\text{inv}} = s^{-1}$
4: $u_1 = (h \cdot s_{\text{inv}}) \bmod n$
5: $u_2 = (r \cdot s_{\text{inv}}) \bmod n$
6: $R' = [u_1]G + [u_2]P$
7: **return** $R'.x \stackrel{?}{=} r$

The verification algorithms presented for this and the following signature schemes are simplified for better understanding. All schemes analyzed in this paper use a double-scalar multiplication during verification. In the case of ECDSA it is performed in step six. We will commonly refer to the resulting point as the witness to the verification throughout this paper.

2.3.1 Elliptic Curve German Digital Signature Algorithm

The Elliptic Curve German Digital Signature Algorithm (ECGDSA) is a variant of ECDSA with some minor changes. In contrast to ECDSA, the private key is first inverted before multiplying with the base point when calculating the public key $P = [d^{-1}]G$. This avoids the calculation of a multiplicative inverse when signing. The verification procedure is the same as shown in Algorithm 1, except for the calculation of u_1 and u_2 where

$$u_1 = (h \cdot r^{-1}) \bmod n,$$
$$u_2 = (s \cdot r^{-1}) \bmod n.$$

Alongside ECDSA and ECSDSA, the German Federal Office for Information Security (BSI) only standardizes ECGDSA in its technical guideline on EC-based signature schemes [Tec18]. This hints at the internal use of the scheme, which is fortified by its appearance in many technical documents [Sec24] on the German eID. It is, however, unclear from the publicly available documentation where and how it is deployed in this ecosystem.

2.4 Elliptic Curve Schnorr Digital Signature Algorithm

While Schnorr initially proposed using a subgroup of a finite group with prime order, he also noted that it is possible to implement the signature scheme on any finite group with an efficient multiplication algorithm, where the discrete logarithm is infeasible to compute, and even suggested the use of elliptic curves [Sch91]. Apart from this note, further development of this idea in the scientific literature is limited to applications for multi-party signatures [BN06, NRS21, KG20]. Over the years, many standardizations [Tec18, ISO18], informal descriptions, and implementations have been developed. Consequently, there exists no consensus on how Elliptic Curve Schnorr Digital Signature Algorithm (ECSDSA) is performed in detail. Schnorr signatures have some advantages over ECDSA, such as a security proof that is based on weaker assumptions [PS00, FKP16] and easier application in compact multi-party signature schemes [MPSW19].

Algorithm 2 shows a simplified version of the verification procedure, as described in the BSI guideline TR-03111 [Tec18].

Algorithm 2 ECSDSA signature verification

Input: message m, signature pair (r, s)
Output: signature validity

1: **if** r not in range of hash function or s not in range $]0; n[$ **then return** false
2: $Q = [s]G + [r]P$
3: **if** $Q = \mathcal{O}$ **then return** false
4: $r' = H(m||Q.x)$, where "||" denotes concatenation
5: **return** $r \stackrel{?}{=} r'$

2.5 Edwards Digital Signature Algorithm

Edwards Digital Signature Algorithm (EdDSA) is a variant of a Schnorr signature based on twisted Edwards curves. Twisted Edwards curves are considered faster than curves in Montgomery [BBJ+08] or Weierstraß [BL07] form because group operations on such curves with well-chosen parameters can be calculated considerably faster since fewer finite field operations are needed. Another advantage of twisted Edwards curves is the unified addition formula, which works both for the addition of distinct points and point doubling operations. EdDSA is notable for its high speed and natural resistance to side-channel analysis attacks, making it especially interesting for embedded applications [BDL+11]. Algorithm 3 shows the verification procedure for an EdDSA signature.

Algorithm 3 EdDSA signature verification

Input: message m, signature pair (R, s) with scalar s and encoded point R
Output: signature validity

1: **if** s not in range $[0; n[$ **then return** false
2: $h = H(R||P||m)$
3: $Q1 = [s]G$
4: $Q2 = R + [h]P$
5: **return** $Q1 \stackrel{?}{=} Q2$

2.6 Fault Attacks

For the scope of our work, we define faults as deviations in a hardware component from its intended function. They can be classified as permanent, intermittent, or transient based on the duration of their impact [CP95] and the part of the system that is directly affected (often the instruction pipeline, registers, SRAM, or flash). A fault model (FM) refers to the consequences of these faults that an adversary could exploit in an attack.

For instance, an attacker might be able to set or reset a bit [RSDT13], set the value of a byte to a random value, or skip one or even multiple [MDP+20] instructions [BH22]. The FM depends on the targeted chip and the attacker's means of injecting faults. The methods used range from cost-effective clock [KSV13] and voltage [KJP14] glitching, and contactless electromagnetic fault injection (EMFI) [CH17] to highly sophisticated and precise laser systems [DBC+18, LBC+15] that require access to the die.

2.7 Hardenings against Fault Attacks

The implementation of countermeasures against FAs can be done in both hardware and software. The discussion will mainly focus on software-based countermeasures since they can be deployed at a more fine-grained level and tailored to specific algorithms. In contrast, hardware-based countermeasures are generally more generic and do not require fine-tuning to fit a specific firmware. In Section 5, we will demonstrate how the signature verification of ECDSA and EdDSA can be hardened in software using a few simple checks to prevent the attacks covered in Section 4.3.

2.8 Secure Bootloaders

A secure bootloader ensures that a device only boots firmware considered trusted. This trust is established through digital signatures, which are verified against a public key securely stored on the device. Manufacturers often use this technology to securely provide updates for their devices while protecting end-users from malicious actors in the supply chain.

The secure bootloaders considered in this paper are MCUboot [Tru24] and wolfBoot [wol24]. They are both designed to be deployed on embedded systems and rely on ECDSA for the verification of boot images. Only wolfBoot supports the more recent EdDSA, which is implemented in the wolfSSL library and can optionally be enabled. We will use this library alongside Mbed TLS, which is used in MCUboot, to build proof of concepts of our attacks on the bootloaders. A successful attack in the library will also result in the erroneous verification of a forged signature in the bootloader incorporating it, thereby enabling an adversary to maliciously boot arbitrary images. The attacks on the implementations shown in Section 4.3 were all tested on the aforementioned widely used open-source libraries by manually injecting the described faults through a debugger.

3 Related Work

To the best of our knowledge, this is the first study of FAs in the verification of signature schemes based on elliptic curves. To understand the impact of possible fault attacks, it is important to first analyze the runtimes of the most efficient algorithms and attacks that currently exist for solving the ECDLP. Since it is possible to target curve parameters, some special cases are also considered, which are usually not possible if those parameters are well-chosen. Being able to solve the ECDLP efficiently will completely break all cryptographic protocols based on elliptic curves by allowing anyone to derive the private key from the public key point.

As of now, the most efficient attacks that are applicable to every curve are based on the Baby-Step Giant-Step [Sha71, BL13] or Pollard's rho algorithm [Pol78]. They have an asymptotic complexity of $\mathcal{O}(\sqrt{n})$, where n is the order of the curve and belong to the category of collision-finding algorithms.

Other approaches to solving the ECDLP are based on summation polynomials. They were first described by [Sem04], who proposed their use in an index calculus algorithm for elliptic curves. Subsequently, extensive research has been conducted on this subject

[Gau09, MM18, APS18]. Previous approaches also suggested pairing-based methods [MOV93] or Xedni calculus [Sil00]. However, as of now, none of these approaches are significantly better on generic curves than Pollard's rho method [GG16, GG14, APS18].

Nevertheless, it is possible to solve the ECDLP much quicker in specific scenarios. We will refer to these special cases as weak curves, as they are not suitable for use in cryptographic schemes because of the weakened ECDLP. The Pohlig-Hellman algorithm [PH78] cleverly exploits curves with non-prime orders. It works by breaking down the computation of the discrete logarithm into smaller problems that can be solved more easily. By finding the discrete logarithm modulo each prime factor of the curve order separately and then combining the results using the Chinese Remainder Theorem (CRT), the overall computation time can be significantly reduced. This method is particularly effective when the curve order has small prime factors, as its runtime complexity is given by $\mathcal{O}(p)$, where p is the biggest prime factor of the curve order. Curves defined over composite fields ($\mathbb{F}_{2^k}$ where k is not prime) are susceptible to the Weil-descent attack [GHS02], which transforms the original ECDLP to a DLP over the composite field's base field, which can be solved far more efficiently in some cases.

FAs on signature verification have already been studied for the following signature schemes. Seifert [Sei05] presented an attack on the RSA cryptosystem, which requires faulting the $\mathcal{O}(log\ log\ N)$ least significant bits in the public modulus N to make it prime. Calculating the corresponding private key is trivial on such a modulus, as it is simply the inverse of the public key. This private key corresponding to the faulted modulus can then be used to sign arbitrary messages. Muir [Mui06] improved this attack by not requiring the faulted modulus to be easily factorizable and not necessarily prime. This significantly lowered the restrictions on the required data faults in N. Michéle et al. also targeted the modulus, but exploited the underlying structures managing it by faulting bits in pointers to the modulus in memory or variables defining its size [MKS12]. Targeting Winternitz one-time signatures (WOTS), Wagner et al. [WWO$^+$23] have demonstrated how the hash-based signature schemes LMS, XMSS, and SPHINCS$^+$ can be attacked to accept forged signatures. They start by using a valid signature to calculate a forged signature for their own message. This forged signature will not have a valid checksum, which is resolved through a fault injection on the checksum mechanism of the WOTS scheme.

FAs on the signing process of elliptic curve-based signature schemes were already described more than two decades ago. Biehl et al. [BMM00] demonstrate how a single bit fault in the input point of a scalar point multiplication over a curve in Weierstraß form is enough to considerably simplify the ECDLP, as calculations on such a faulted point are carried out on a likely different and potentially weaker curve. Schmidt et al. [SM09] have shown an attack that reconstructs the private key by repeatedly leaking parts of the ephemeral key through a fault during the scalar multiplication of the witness calculation. Preceding approaches also target modular multiplication with the private key [BBB$^+$16] or curve parameters to extract private key information through a side-channel [CSC$^+$21].

4 Attacking EC Signature Verification

This section describes our FAs on the verification process of various elliptic curve-based signature schemes. First, attacks on point multiplications are described, which are based on faults in points or curve parameters and pose a threat to every signature scheme based on Weierstraß curves. Afterward, primarily implementation-specific FAs targeting specific steps of the verification algorithm are presented.

4.1 Attacker Model

For all presented attacks, we assume a system which verifies an EC signature in software. The goal of the attacks is to trick the system into accepting a forged signature for an arbitrary message. The attacker only knows the fixed public data, i.e., the public key point and the corresponding elliptic curve and field parameters. They can freely choose the message to be verified and the corresponding signature pair, but none of the public data. The attacker has physical control of the system to inject faults into it during the EC signature verification. This allows them to break the system's control flow and data integrity assumptions.

In the secure bootloader context, the attacker replaces a system's firmware with their own malicious firmware and forged signature. Successful attacks inject faults to cause the targeted system to erroneously accept the forged signature and boot the unauthentic firmware.

4.2 Faulting Elliptic Curve Points and Parameters

Biehl et al. [BMM00] have shown that a fault in a generator point on an elliptic curve significantly impacts the security of the signing process in EC signature schemes by allowing an adversary to leak private key bits. We suggest introducing a similar fault during the verification phase to enable the verification of forged signatures. The underlying idea behind this attack is to weaken the ECDLP, which is the foundation of every EC signature scheme. A single bit flip in the coordinate of a point used in such a scheme can be enough to significantly weaken it.

Such a faulted point is almost certainly not on the original curve and can be assumed to be an arbitrary pair of elements of a field $\mathbb{F}_q$. The curve parameter $b \in \mathbb{F}_q$ is the coefficient of the monomial of degree 0 in the polynomial equation of a short Weierstraß curve. This means that for every set of values x, y and a, there exists exactly one value b such that the pair of field elements (x, y) describes a point on the curve defined by the parameters a and b. When b is not used in the equations of the group law, it is implicitly assumed when adding points on a Weierstraß curve. EC point calculations on a faulted point will thus be performed on the curve of the faulted point instead of the original curve.

A well-chosen Weierstraß curve will have a prime order that is close to the underlying finite field's order. Consequently, it is extremely likely that a random bit-flip will greatly decrease the security of the curve, as the Pohlig-Hellman algorithm efficiently solves the ECDLP over curves with an order that consists of many small prime factors. This raises the question of how likely it is to get a sufficiently weak curve through a fault. We are searching for a property in the curve order known as smoothness, which specifies the largest prime factor of a number. Any integer that has only prime factors less than or equal to n is called n-smooth.

We assume that the order of a random curve is randomly distributed in the curve's underlying finite field. Solving the ECDLP during the attack gets easier the smoother the curve order is, meaning that the largest prime factor of the curve order needs to be as small as possible. To find out how likely it is to encounter numbers of a particular smoothness, we can use the Dickman function. The Dickman function estimates the proportion of smooth numbers up to a given bound. In other words, it can be used to estimate the probability that a randomly selected number below a given bound is n-smooth.

Figure 1 shows the probabilities that a random number of a specific size is n-smooth. It contains both experimental results, whose generation is explained in Appendix A.1, and the expected distribution according to the Dickman function. The experimental results we obtained closely align with the Dickman function. This indicates that the order of random Weierstraß curves can in fact be assumed to be random and can be used to estimate the likelihood of finding a fault of a certain quality.

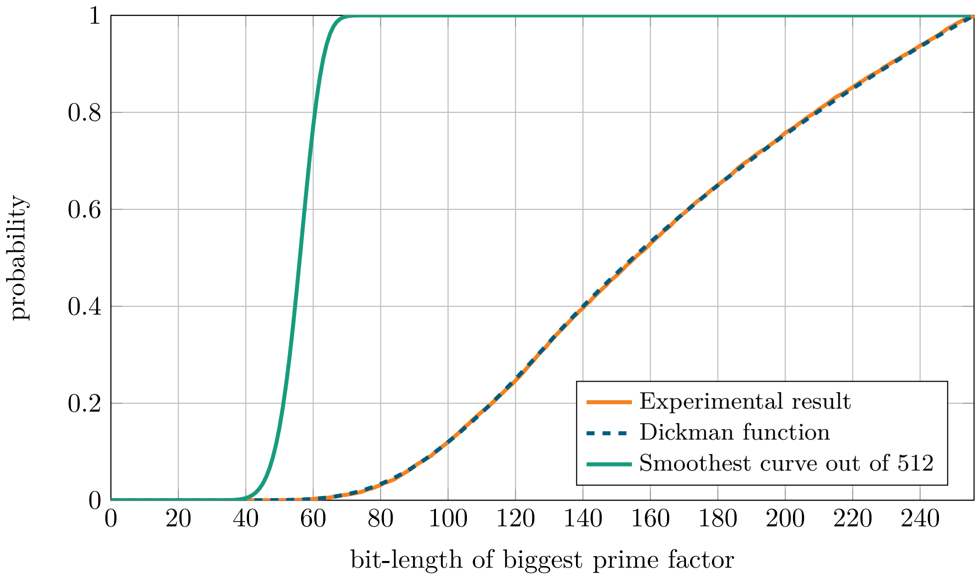

Figure 1: Smoothness of random 256-bit curve orders

We can see that the expected bit length of the biggest prime factor of a random 256-bit integer is approximately 160 bits. We, however, do not need to restrict ourselves to a single, randomly chosen fault. Since a point on a 256-bit curve consists of two 256-bit coordinates, we have a total of 512 single-bit faults we can possibly introduce at the target point. These 512 candidates can then be checked to find the fault that results in a point with the smoothest order. The resulting curve orders and their smoothness can be calculated ahead of time for the base point of a commonly used curve and thus do not necessarily contribute to the complexity of the attack. Nevertheless, this process is easily performed by a modern processor in a matter of minutes, as the only noteworthy computation is the prime factorization of 512 256-bit integers.

Figure 1 also displays the probabilities of finding a curve order of particular smoothness in one of these 512 fault candidates. For example, with 512 single-bit flip faults on a random point of a 256-bit curve, there is an 80% chance of finding a point with a curve order that is 2^{60} smooth.

The complexity of solving the ECDLP for such a curve is even lower than that. When solving using Pohlig-Hellman in combination with the Pollard-Rho algorithm, a solution can be found in $\mathcal{O}(\sqrt{p})$, where p is the biggest prime factor of the curve order. Consequently, if a solution exists for our example, it will likely be found after 2^{30} iterations. The subsequent sections will extensively examine the impact of these FAs on different signature algorithms.

4.2.1 ECGDSA

The general idea behind this attack is to reduce the problem of forging a signature to a single ECDLP over a faulted point. A well-chosen fault will result in a point on a curve with a smooth order, meaning that the ECDLP can efficiently be solved to find the remaining part of the forged signature.

As described in Section 2.3.1, a valid signature satisfies the constraint

$$
\begin{aligned}
u_1 &= (r^{-1} \cdot H(m)) \bmod n, \\
u_2 &= (r^{-1} \cdot s) \bmod n, \\
r &= ([u_1]G - [u_2]P).x \bmod n.
\end{aligned}
$$

Targeting the public key point P with a fault enables the efficient calculation of the ECDLP for the factor u_2 in scalar multiplication. It is sufficient to flip a single bit in one of P's coordinates to change the curve it lies on to another, significantly weaker curve. This new faulted point P' ideally has an order that can be factorized into many small prime numbers, making the ECDLP efficiently solvable over the faulted curve using the Pohlig-Hellman attack. With the factor u_2, it is possible to calculate the needed value for s with a fixed value for r. The fact that r can have a fixed value implies that the missing point $[u_2]P$ can be calculated with

$$[u_2]P' = [u_1]G - R.$$

This assumption is, however, not completely true since the faulted public key point P' will be on a different curve than the generator G. The point addition equations naturally assume that both summands are on the same curve and consequently break when violating this constraint. The calculation of the missing point thus requires additional steps. We will derive a solution using equations for affine coordinates; the forged signature will, however, still be applicable when using projective or Jacobian coordinates.

We start with the point addition equation ($R = P + Q$):

$$\begin{aligned} \lambda &= \frac{y_q - y_p}{x_q - x_p}, \\ x_r &= \lambda^2 - x_q - x_p. \end{aligned}$$

For the problem described above, we are interested in x_q depending on x_r and x_p. Since x_q can only have two possible values for a fixed y_q, it can be eliminated from the equation for now. The point addition equation is thus solved for y_q:

$$y_q = \sqrt{(x_r + x_p + x_q) \cdot (x_q - x_p)^2} + y_p.$$

Using the curve equation, y_q can be represented using x_q, a, and b:

$$\sqrt{x_q^3 + a \cdot x_q + b} = \sqrt{(x_r + x_p + x_q) \cdot (x_q - x_p)^2} + y_p.$$

The curve parameters used here depend on the replaced point and must be the ones belonging to the faulted point. This can then be transformed into an equation yielding a polynomial of degree six after all known variables have been replaced:

$$\begin{aligned} 0 =& x_p^6 + x_p^2 x_q^4 - 2b x_p^3 + 2(x_p^3 + a x_p) x_q^3 + y_p^4 - (x_p^4 - 2a x_p^2 - a^2 - 2b x_p) x_q^2 + \\ & (x_p^4 - 4x_p^3 x_q + 6 x_p^2 x_q^2 - 4 x_p x_q^3 + x_q^4) x_r^2 - 2(x_p^3 - x_p x_q^2 + 2x_q^3 - (x_p^2 - a) x_q + \\ & (x_p^2 - 2x_p x_q + x_q^2) x_r + b) y_p^2 + b^2 - 2(x_p^5 + a x_p^3 - b x_p^2 - ab) x_q + 2(x_p^5 - x_p x_q^4 + \\ & (x_p^2 - a) x_q^3 - b x_p^2 + (2x_p^3 + 2a x_p - b) x_q^2 - (3x_p^4 + a x_p^2 - 2b x_p) x_q) x_r. \end{aligned} \tag{1}$$

This polynomial is easily solved by a computer algebra system, finally resulting in the x-coordinate of the point we are looking for. The y-coordinate of this point must be restored to obtain the desired point, since it was eliminated to simplify the equation. The two candidates for this value can be found with the curve equation using the curve parameters of the faulted point. The correct point out of the two candidates is then identified by performing a point addition to validate that $R = P + Q$ is satisfied.

All of this can be used to construct the final attack on the scheme:

1. choose arbitrary $r \in \{1, \ldots, n-1\}$
2. calculate $u_1 = r^{-1} \cdot H(m) \bmod n$

3. find X such that $r = ([u_1]G + X).x$ where X is on the same curve as P'
4. find u_2 such that $[u_2]P' = X$ through Pohlig-Hellman
5. calculate $s = u_2 \cdot r$

It is noteworthy that neither step 3 nor step 4 are guaranteed to have a solution. Consequently, the procedure will sometimes need to be repeated, starting from the first step, in order to find a solution. In general, this does not increase the runtime complexity of the attack since it is quite likely that a solution exists for an arbitrary r.

Experimental results show that a point X in step 3 can be found for around 63% of all scenarios when all other values in the equation are randomized.

It may not always be possible to find a solution for u_2 in step 4, because the curve order is no longer prime. Since the point X was not generated by the point P', there is no guaranteed solution for the ECDLP in some cases. If the order of X does not divide the order of P', it cannot be in a subgroup of the one generated by P', as follows from Lagrange's theorem. When working with curves in short Weierstraß form, parameters are usually chosen so that the generator is of prime order. Using Lagrange's theorem again, we can conclude that the group of such a generator has no subgroups. Therefore, all points in this group act as generators. This is, however, not true for the point P', which was chosen to be on a curve with an order that consists of many prime factors. Consequently, the subgroup generated by P' might not contain all points on the curve. If the point X that was generated randomly by fixing an arbitrary value for r does not belong to the same group as P', we are not able to find a solution for the ECDLP.

Experimental results show that a solution for the ECDLP is found in around 75% of all cases. Luckily, we can already sort out these cases by looking at the points' curve orders. Thus, no time-expensive operations are required when searching for a suitable r.

Consequently, after roughly two tries, we can expect to find a fitting value for r. When looking at the attack's runtime in big O notation, the runtime complexity is reduced to solving the ECDLP in step 4. This part of an attack is also commonly referred to as an offline phase since no access to the targeted device is needed to calculate a solution. Once a signature has been forged, the actual attack on the device begins, which is why this step does not contribute to the complexity of the FA itself.

4.2.2 ECSDSA

As mentioned in Section 2.4, there is no definite standard for elliptic curve Schnorr signatures. The following attack description will thus not be applicable to every variation of this signature scheme. We will focus on the ECSDSA specification [Tec18] of the BSI, also described in Section 2.4. Finally, we will explain to which other variations the attack is applicable.

The general idea behind the attack is the same as in the previous scheme, as we are again trying to reduce the problem to an ECDLP over a faulted point. The process of getting to that point is, however, a bit different.

As a reminder, the verification is done by first reconstructing a point chosen during signing with

$$Q = [s]G + [r]P. \tag{2}$$

As r and s constitute the signature, we have the liberty to choose their values within the prescribed limits. With the attack in the previous section in mind, it is fair to assume that we can find a point Q with a specific x-coordinate using a fault on the generator point G or the public key point P. This is done by fixing one of the summands to some arbitrary value, finding the missing point to fulfill the equation, and finally solving the ECDLP over that point.

During the next verification step, a hash is calculated, which is then compared to the signature value r:

$$r \stackrel{?}{=} H(m||Q.x).$$

We generally cannot reverse the hash function and thus start the attack by choosing some point Q to calculate the value of r through the hashing operation described above. Looking at Eq. (2), we can see that we already fixed values for Q, r, and consequently $[r]P$, meaning that the only free variable left is s. Finding a value s that fulfills this equation is generally not possible because of the difficulty of the ECDLP, which is why the point G needs to be faulted to end up on a weaker curve. Since we are now again working with an addition between two points on different curves, the resulting point Q of this operation is going to be some element in $\mathbb{F}_q^2$ that is neither on the original curve nor on the one of the faulted point G'. We can repurpose Eq. (1) to find a point $Q1$ such that $Q.x = (Q1 - [r]P).x$. This point $Q1$ can then be used to find a value s so that $X = [s]G'$ using the Pohlig-Hellman algorithm if G' is a faulted point on a sufficiently weak curve.

In summary, a signature can be forged with the following steps:

1. choose arbitrary $Q \in \mathbb{F}_q^2$
2. calculate $r = H(m||Q.x)$
3. calculate $Q2 = [r]P$
4. find $Q1$ so that $Q.x = (Q1 + Q2).x$ using the polynomial derived in Section 4.2.1
5. find s so that $[s]G' = Q1$ through Pohlig-Hellman

However, this attack is not possible for variants of EC-Schnorr signatures that include the y-coordinate of Q in the hash. The ISO standard 14888-3 [ISO18] describes two variants of ECSDSA: normal and optimized. While the optimized version is similar to the one presented by the BSI, the normal variant constructs the hash using both x- and y-coordinates of Q: $H(Q.x||Q.y||m)$.

Looking back at step 4 of the attack on ECSDSA as standardized by the BSI, it is extremely unlikely ($\sim 1/n$) that the y-coordinates of Q and $Q1 + Q2$ will be equal. This is because we cannot know on which curve $Q1 + Q2$ is going to be when selecting a point Q since they are on distinct curves. Since the non-optimized variant in the ISO standard includes the y-coordinate in the final hash, these coordinates also need to be equal for the signature to be valid.

We believe that it is not possible to overcome this issue. Once the point Q has been arbitrarily chosen, the value of r is fixed (apart from changes in the signed message, which can be ignored since it can only be used for a brute-force attack). The closest we can get to a solution from this point is the attack described above, which can at least be used to forge a signature with the correct value for $Q.x$. This only leaves the option of finding a better value for Q than arbitrarily selecting it. Doing this would require a strong correlation between the coordinates of point Q based on the curve parameters of both summands of the addition. Experimental results have, however, shown that the addition of random points on two distinct but fixed curves will result in random elements in $\mathbb{F}_q^2$ without any apparent pattern. We thus conclude that a FA on the EC points or parameters does not weaken such signature schemes.

4.2.3 ECDSA

Unlike with ECGDSA, a single- or few-bit fault in one of the elliptic curve points is not easily exploitable for attacking ECDSA. The main difference between the verification in both schemes is the calculation of the witness R'. In ECDSA, the witness is recomputed using

$$R' = [h \cdot s_{\text{inv}}]G + [r \cdot s_{\text{inv}}]P.$$

The equation differs from the one for ECGDSA only in how the values r and s are utilized. To understand why this small difference poses a challenge for our attack, we need to take a look at the options we have to approach the attack. We will briefly describe why we believe that ECDSA is not susceptible to such faulty point attacks here and provide more details in the appendix.

For the attack on ECGDSA, we started by fixing an arbitrary value for the signature value r. The idea behind this was to eliminate variables until the problem is reduced to a single ECDLP. Trying the same for ECDSA results in a problem that is much harder to solve, as both factors of the scalar multiplications are still dependent on the signature value s. Even if both points are faulted to lie on weak curves, we will not be able to leverage this weakness, as the points do not share a common group. Fixing a value for s also fixes $[h \cdot s_{\text{inv}}]G$ but results in the same issue of having to solve over points that do not share the same group. This approach is described in more detail in Appendix A.2.

The failure of the previous attempt to break the signature scheme can be attributed to the interdependency of the summands in the final addition and the distinct curves on which these points reside. The differing group orders of the faulty points can, however, also facilitate finding a solution instead of posing an additional challenge. The idea behind this approach is to remove the interdependence between the summands by finding distinct fitting values for s (s_1 and s_2), which are used as a factor in the respective multiplications:

$$R' = [h \cdot s_1]G + [r \cdot s_2]P.$$

Of course, we cannot have two different values for s, but since the orders of the groups generated by the faulted points G' and P' are very likely not identical, we can find a common value for s that results in s_1 and s_2 when taken modulo the respective group orders. A fitting value is easily found through the CRT. This value will, however, almost certainly be too big to fit into the signature. Finding a value for s that satisfies all constraints proved to be equally hard as solving the ECDLP on an unfaulted point. Appendix A.3 outlines the constraints for s and approximates the computational effort to find it.

We thus conclude that attacks on ECDSA targeting elliptic curve points, or curve parameters, are not feasible under the assumed FM.

4.2.4 EdDSA

In contrast to the signature schemes discussed in the previous sections, EdDSA does not operate on elliptic curves in Weierstraß form but on twisted Edwards curves. The previous attacks were all based on faulting points, which implicitly resulted in the elliptic curve point multiplications being performed on another (hopefully weaker) curve. This worked because the curve parameter b on a short Weierstraß curve is not used in either the addition or doubling operation. When the group law of an elliptic curve does not use all curve parameters, we can say that the curve has *implicit curve parameters*.

Since the group law for points on Weierstraß curves naturally differs from the group law on twisted Edwards curves, a fault as discussed in the previous sections might not necessarily be possible on such curves. As the process of doubling and adding distinct points is based on the same unified equation on twisted Edwards curves, it is sufficient to only examine this single operation defined through

$$x_r = \frac{x_p y_q + y_p x_q}{1 + d x_p x_q y_p y_q}, \qquad y_r = \frac{y_p y_q - a x_p x_q}{1 - d x_p x_q y_p y_q}.$$

From observation alone, it is evident that using faulty points that are not on the expected curve will not produce any useful results. In contrast to Weierstraß group law, the curve of the supplied points is not assumed but explicitly specified, and all parameters are required for the calculation. While the resulting x-coordinate is independent of the parameter a of the original curve, the equation for the y-coordinate includes both curve parameters a and d. The scalar multiplication employed in EdDSA propagates and amplifies the erroneous value in the resulting y-coordinate to the x-coordinate, because the subsequent double or add operation will utilize this faulty value as its input. As a consequence, each intermediate result of the scalar multiplication will be a point that is almost certainly on another curve than its summands. The multiplication can thus no longer be described through group operations since the underlying group changes with the supplied points between each addition. Consequently, the result of any scalar multiplication with a point that is not located on the expected curve will be a practically random element in $\mathbb{F}_q^2$ located on a curve that is distinct to the original one and the one of the faulted input.

To summarize, we believe that an attack similar to the one based on faulty points on Weierstraß curves is not applicable to curves in twisted Edwards form. While most twisted Edwards curves over $\mathbb{F}_q$ are $\mathbb{F}_q$-isomorphic to a curve in Weierstraß form [CS18], the non-linearity in a translation between these models eliminates the property of implicit curve parameters that we need for this attack to work.

4.2.5 Limitations

Attacking implementations as described in this section is not always possible. In principle, an attacker can be fully oblivious to the implementation of the algorithm as long as they are able to set or unset a bit in the public key or base point. This does, however, not mean that every implementation of an algorithm is vulnerable.

Most implementations we analyzed use incomplete equations (e.g., the Jacobian doubling formula from [CMO98] in Mbed TLS) that do not include the curve parameter b, likely due to their performance benefits. A major advantage of complete formulas [RCB16] is that they are naturally more resistant against side-channel attacks and easier to implement. In addition, they are not vulnerable to the faulty point attacks presented in this paper. Nevertheless, both wolfSSL (v5.7.0) and Mbed TLS (v3.5.2) use incomplete formulas but lack support for ECGDSA and ECSDSA. The most popular open-source projects that support these schemes are Botan[1], Crypto++[2], and libecc[3]. libecc implements both complete and incomplete equations but uses complete equations by default. Botan (v3.7.1) and Crypto++ (v8.9) only implement incomplete equations.

Another problem we might encounter when tampering with EC points are on-curve tests. These checks verify that the curve equation holds for a given point. All libraries supporting ECGDSA and ECSDSA we checked verify that a point is on the expected curve upon deserialization. However, only libecc performs additional checks at a later point. Consequently, in Crypto++ and Botan, a fault can still be injected at any point between the initialization of the EC points and first use during verification.

Further, implementations that utilize Shamir's trick [Ava05] for a combined calculation of the double scalar multiplication are likely not vulnerable. While this optimization can speed up verification times, it is not always possible to make use of it because of strict memory constraints on some embedded systems. With Shamir's trick the multiplications can be performed simultaneously bit-by-bit or with a sliding window. In each step both input points can be added to the intermediate result value depending on the respective scalar factors. To forge a signature an attacker generally needs to be able to reverse the

[1] https://botan.randombit.net
[2] https://www.cryptopp.com
[3] https://github.com/libecc/libecc

steps of the verification routine. This is, however, not easily done in this case, since each addition of points on distinct curves garbles the intermediate result.

Since libecc is the only library we know of that is targeted towards embedded systems while also implementing ECGDSA and ECSDSA, it is hard to assess the real-world impact of these attacks. Nevertheless, it is vital to be aware of these attacks when implementing the aforementioned algorithms, as they are bound to be applicable to implementations that are targeted towards strongly resource-constrained systems.

4.3 Attacking Elliptic Curve Signature Implementations

Instead of faulting EC points or parameters, it is also possible to directly attack specific steps of the verification algorithms. These attacks are less generic and generally require the attacker to have access to the source code or binary image that implements the verification. Nevertheless, we try to keep the attacks general and applicable to every common implementation by using a generous FM as a foundation for our analysis. To demonstrate the practicality of the attacks, we examine Mbed TLS and wolfSSL, which are integrated in the secure bootloaders MCUboot and wolfBoot, respectively. A successful verification of a forged signature in these cryptographic libraries enables an adversary to undermine the secure boot process by booting arbitrary, self-signed images. Although both bootloaders offer hardenings against FAs, the verification executed within the libraries is entirely unprotected, rendering the countermeasures ineffective. We chose ECDSA and Ed25519 for analysis, as they are the most prevalent signature schemes based on ECC and serve as the default options for MCUboot and wolfBoot, respectively.

The emphasis is on 32- and 64-bit architectures of non-specialized general-purpose CPUs, which is exactly what the aforementioned bootloaders were built for. Consequently, specialized ISAs, such as those including big number registers or hardware accelerators for cryptographic operations, are not further considered but could still be vulnerable to the described attacks.

The following sections include detailed descriptions of all possible FAs on the popular elliptic curve signature algorithms ECDSA and EdDSA, according to a common FM. We will describe these attacks on an algorithmic level without focusing on a specific implementation. This allows us to cover all faults that need to be accounted for when designing a hardened verification algorithm against this FM. We practically verified the feasibility in either Mbed TLS or wolfSSL with simulated faults and included source code snippets in the Appendix that explain how the attack was carried out. Identifying all possible attacks that an implementation might be subjected to allows us to later suggest a hardened version of the signature scheme.

4.3.1 Fault Model

Fault injections can result in a variety of arbitrarily complex effects on the targeted system. For a usable FM, it is thus necessary to abstract these effects. Through glitches on the clock signal [BGV11, KH14] or supply voltage [KH14], or EMFI [LG19, DDRT12] it is possible to skip or temporarily replace instructions. The effects of such a replacement depend on the original instruction, the processor's execution pipeline, architecture, etc., meaning an attacker has only a very limited ability to control the outcome. Skipping occurs when an instruction is replaced with one that has no significant effect, such as writing to a register that is later overwritten before being read. Additionally, lasers [RSDT13] or EMFI [OGST+15, OGSM16] can be used to precisely set or unset single bits in registers or flash memory. With these capabilities in mind, we can define a more abstract FM that incorporates specifics of the cryptographic libraries and target architectures. This abstraction allows us to work on an algorithmic level, which is required to identify weaknesses that potentially affect all implementations. The goal of this model is to include all possible and useful faults an attacker could inject in order to accept forged signatures. Although we are confident that this model covers the majority of potential faults in common implementations, we must emphasize that we cannot guarantee complete coverage due to unaccounted-for specifics in certain implementations or hardware configurations.

```
typedef struct mbedtls_mpi {
    /** Pointer to limbs.
     * This may be \c NULL if \c n is 0.  */
    mbedtls_mpi_uint *MBEDTLS_PRIVATE(p);

    /** Sign: -1 if the mpi is negative, 1 otherwise.  */
    signed short MBEDTLS_PRIVATE(s);

    /** Total number of limbs in \c p.  */
    unsigned short MBEDTLS_PRIVATE(n);
}
mbedtls_mpi;
```

Figure 2: *bigint* definition in Mbed TLS. Snippet taken from file `include/bignum.h` at version v3.5.2

For well-chosen curves, solving the ECDLP has a runtime complexity of around $\mathcal{O}(\sqrt{n})$. Consequently, to achieve 128-bit security, a 256-bit curve needs to be used. Nowadays, processors usually work on architectures with a word size of 64 bits or less. The arithmetic operations that need to be performed for any of the elliptic curve signature schemes are, however, often performed on integers of sizes between 196 and 512 bits. Since these values are too large for a single word in any modern general-purpose architecture, the calculation is split into multiple steps by software. To store these large integer values, implementations need to use *bigint* structures that typically support integers of almost arbitrary and variable size. Sometimes, this is realized through a structure containing a pointer to an arbitrarily long array of words or even just bytes (often called limbs), an integer specifying the word count, and a sign flag. An exemplary definition of a *bigint* structure as used in Mbed TLS is shown in Fig. 2.

Fault attacks can target data directly or the managing structure. Consequently, various effects can be achieved when targeting them; however, not all of them are sensible in the context of verifying signatures. Faults that modify the value of a *bigint* without significantly reducing its size or overwriting it with a specific value are not beneficial in this context. Faulting a 256-bit chunk of memory to a specific value is usually not feasible with current fault injection methods. Except for the attack outlined in Section 4.2, we are certain that attackers cannot exploit single- or few-bit changes in the values used during signature verification in the schemes we studied.

Reducing the size of a *bigint* structure is however easily done when targeting the integer specifying the number of limbs. It is often enough to unset a single bit in order to set the length specifier and also the value of the whole *bigint* to 0. Alternatively, setting the length specifier to 1 can drastically reduce the size of a *bigint*, possibly enabling a brute-force attack.

In addition to altering the data used in calculations, faults can also influence the control flow of a program. We can generalize and abstract these effects to make them easier to work with. For instance, skipping a compare, jump, or load instruction might allow an attacker to execute or skip a specific part of the code. Apart from the obvious target that is the final comparison of each signature verification algorithm, these faults can be abstracted to a few desirable effects for an attacker during arithmetic operations. Since arithmetic operations are performed on integers that do not fit into a single register, they consist of many steps implemented in software and not just a single processor instruction. Before executing these expensive operations, some implementations try to take shortcuts, for example, by checking if any of the arguments are equal to 0 or the neutral element when working with elliptic curve points. An attacker can deliberately trigger these shortcuts through a fault by skipping a branch or compare instruction. This makes it possible to directly return any of the input values given to a function as the result of the calculation. Often at the beginning of a calculation the result is initialized with one of the arguments, or 0. Consequently, we need to assume that an attacker may be able to return any of the arguments or 0 as the result of any calculation.

To summarize, we need to consider the following capabilities of an attacker:

- setting any big integer value to 0
- reducing the size of any big integer value to a single word
- altering a single or at most a few bits of any big integer value
- returning one of the inputs as the result of an arithmetic operation
- returning 0, or a freshly initialized value (e.g., neutral element of EC-group) from any arithmetic operation
- skipping any step or operation during verification entirely

Under the extended FM, an attacker is also able to target a program's data structures by introducing faults to data pointers, similarly to the attack of Michéle et al. on structures managing the modulus in RSA [MKS12]. For example, it is sufficient to flip a single bit in a pointer to fully overwrite the original value of a variable with an unrelated value. If this new value is a known constant or if it can at least be predicted with good reliability, some additional attacks on the schemes are possible. In almost all attacks in this section where we assume a fault that sets a variable to 0 or where one of the arguments of an arithmetic operation is directly returned, we can also use a fault that replaces a specific value with an unrelated constant for the same effect. An example of how an attack under the extended FM could look like is provided in Appendix A.11.

4.3.2 ECDSA

Using the FM derived in the previous section, we can now take a closer look at the verification algorithm of ECDSA. In particular, we will sequentially inspect every operation and analyze the effects of skipping a particular step or returning one of the steps' inputs or some constant instead of the actual value. Additionally, there is the possibility of setting any variable to one of the constants of the FM.

The following steps outline the verification algorithm for ECDSA:

1. verify that $0 < r < n$ and $0 < s < n$
2. calculate message hash $h = H(m)$
3. calculate modular inverse of signature proof $s_{\text{inv}} = s^{-1}$
4. calculate $u_1 = (h \cdot s_{\text{inv}}) \bmod n$
5. calculate $u_2 = (r \cdot s_{\text{inv}}) \bmod n$
6. calculate $R' = [u_1]G + [u_2]P$
7. verify that $r \stackrel{?}{=} R'.x \bmod n$

The FAs will be presented in reverse order, starting from the final step, to highlight the common theme that impacts step 6.

4.3.2.1 Faulting the Final Comparison

Targeted Step: $r \stackrel{?}{=} R'.x \bmod n$ [7]
Fault Type: control flow

The most obvious target for a FA is the last step of the scheme, which only consists of a compare operation between big integers. Introducing a fault here can be done in various ways, for example, by exiting the comparison function loop early or skipping the jump instruction of the final if-condition or one of the subroutines of the comparison completely.

We cannot cover all possibilities here since the attack surface is highly dependent on the specific implementation and even the compiler used. Appendix A.4 describes an attack on Mbed TLS that skips a single instruction in order to accept an arbitrary forged signature.

4.3.2.2 Faulting the Curve Order

Targeted Step: $r \stackrel{?}{=} R'.x \bmod n$ [7]
Fault Type: data corruption

Since the curve order is exclusively used as the divisor in the modulo operations of the schemes, faulting it to 0 will most likely result in a crash. Significantly reducing its size, however, greatly reduces the number of possible values for r and allows an attacker to forge a signature through an exhaustive search. This can be achieved by faulting the limb counter of the *bigint* structure holding the curve order value. Reducing this value to 1 often requires more than one bit to be flipped, requiring a precise multi-bit fault, for example, injected using a laser or electromagnetic pulses. Depending on the implementation, it might also be possible to reduce the size of the *bigint* holding n by skipping an instruction while the value is loaded from memory or during the modulo operation itself.

The required signature can be pre-computed for a specific fault, requiring only a small brute-force of fault injection parameters on the target device to find the exact fault needed. The fault needs to be introduced at any point before the last modulo operation. Faulting the curve order earlier might change the signature needed for a successful validation, but it has no effect otherwise. It is advisable to introduce the fault before executing the verification function, as it allows for a more extended time frame compared to a precisely timed fault between routine steps.

In the case of Mbed TLS, a limb of a *bigint* has the size of a word. For instance, for a 32-bit microprocessor, this means that we can use the aforementioned fault to ensure that $n' < 2^{32}$. wolfSSL has even smaller limbs that are only byte-sized, thus requiring a much smaller brute-force. Regardless of the limb size, a valid signature over this new curve order is then found through brute-force, which is possible in a reasonable timespan due to the small size of n' compared to n. This is done by iteratively testing different signature pairs $\{r, s\}$ with $r < n'$, until a solution is found:

1. set $r = 0$ and $s = 1$
2. increment r
3. if $r \stackrel{?}{=} n'$ set $r = 1$ and increment s
4. verify signature using faulted curve order n'; if signature is invalid, go to step 2.
5. return $\{r, s\}$

With the more or less random value n', instead of the actual curve order, it is hard to say if a solution exists for every possible set of fixed parameters (curve parameters, message, hash algorithm, etc.). It is, however, not far-fetched to assume that the resulting values for $R.x \bmod n'$ are randomly distributed in the range $0, \ldots, n' - 1$ or at least extremely close to. Under this assumption, we can expect to find a value for r for every possible value of s when looking for accepted signature values under the faulted curve order. This means that a solution can be found in $n'/2$ iterations on average.

This assumption is further reinforced by the practical evaluation of the attack in Mbed TLS described in Appendix A.6.

4.3.2.3 Eliminating Summands of the Point Addition

Targeted Step:	$R' = [u_1]G + [u_2]P$	[6]
Fault Type:	control flow	

According to the FM, attacking the point addition operation can have three possibly desirable outcomes. It can result in the function returning either summand as the resulting point or leaving it completely unchanged. In the latter case, the usefulness of such a fault depends on the value the result point is initialized with. If the value of this point is constant or easily predictable, a signature for any message can be trivially generated by setting r to that value. In most implementations, however, this point's coordinates are initialized with 0, which is not a valid value for r.

A fault returning either of the summands as the result of the addition can always be used to accept an easily forged signature.

If $[u_1]G$ is returned, a fitting signature can be trivially generated. Since r is now no longer used to produce the witness $R'.x$, we can simply generate an arbitrary witness by fixing a value for s. The missing part r of the signature is then set to $R'.x$.

If $[u_2]P$ is returned, another approach is needed since both r and s are included in the factor for the scalar multiplication with P. To be more exact, we need to find a solution for $r = ([r \cdot s_{\text{inv}}]P).x$. Because the calculation of $u_2 = (r \cdot s_{\text{inv}})$ is done modulo n, a suitable value for r or s_{inv} can always be determined to satisfy the equation, regardless of the fixed values of other variables. We can do this because the generator of a well-chosen curve used for ECDSA will always have a prime order, meaning that the modulo operation does not only form a residue class ring but also a finite field. This guarantees the existence of a multiplicative inverse for every field element. With a fixed value for r and u_2, we can thus always find an s such that $r \cdot s^{-1} = u_2$.

$$\begin{aligned} 1 &= r \cdot r^{-1} && | \cdot s^{-1} \\ s^{-1} &= r \cdot r^{-1} \cdot s^{-1} && | \, u_2 = r \cdot s^{-1} \\ s^{-1} &= r \cdot u_2 \\ s &= r^{-1} \cdot u_2 \end{aligned}$$

We can summarize this with the following steps, which can generate a signature that will be accepted by the faulted scheme:

1. choose arbitrary u_2 from $\{1, \ldots, n-1\}$
2. calculate $R' = [u_2]P$ and take $r = R'.x$
3. calculate $s = r^{-1} \cdot u_2$

Alternatively, we can take a shortcut to find a solution without needing to perform any calculations. Setting $r = s = P.x$ always results in a valid solution for the faulted signature scheme, as follows trivially from the steps above.

A concrete example for both fault variants where either summand is returned from the addition can be found in Appendix A.5.

Additional Faults to Eliminate Summands The idea behind both of these attacks can also be applied to many other FAs that manage to eliminate one of the summands in step 6. To eliminate the first summand, an attacker may also perform any of the following attacks to end up with $u_1 = 0$:

- fault the limb count of its *bigint* structure to 0
- return 0 in the multiplication of h and s_{inv} in step 4
- skip the fourth step to leave u_1 at its initialized value, which is often 0
- set $h = 0$ by faulting the limb count of its *bigint* structure to 0
- skip hash calculation to leave h at its initialized value, which is often 0

To eliminate the second summand, an attacker may also set $u_2 = 0$ through any of the following faults:

- fault the limb count of its *bigint* structure to 0
- return 0 in the multiplication of r and s_{inv} in step 5
- skip the fifth step to leave u_2 at its initialized value, which is often 0
- set $s_{\text{inv}} = 0$ by faulting the limb count of its *bigint* structure to 0

Additionally, each scalar point multiplication in step 6 can be skipped entirely. This can have the same effect as returning the respective other summand from the addition, because the variable holding the resulting point is left in its freshly initialized state, which is often the neutral element. Alternatively, a fault on the point multiplication might result in $R' = G + [u_2]P$ or $R' = [u_1]G + P$, for which exploitation can be easily derived from the similar scenarios described in this section by including the additional points in the calculation of the witness.

4.3.2.4 Faulting u_2 Calculation

Targeted Step:	$u_2 = (r \cdot s_{\text{inv}}) \bmod n$	[5]
Fault Type:	control flow	

The multiplication in the fifth step of the verification algorithm was already targeted in the previous section, where the constant 0 was returned instead of the actual result. We can, however, also fault the multiplication to only return s_{inv} from this operation instead of $r \cdot s_{\text{inv}}$. Doing this, r is fully eliminated from the recomputation of the witness R'. As a result, the new faulted value of the recomputed witness can be used to simply set $r = R'.x$. In detail, a signature is generated as follows:

1. calculate $h = H(m)$
2. choose arbitrary s from range $\{1, \ldots, n-1\}$
3. calculate $R' = [h \cdot s_{\text{inv}}]G + [s_{\text{inv}}]P$
4. set $r = R'.x$

We were not able to find a fault that could enable such an attack in the libraries we analyzed. Nevertheless, it is included as it is possible under the assumed FM.

4.3.2.5 Faulting u_1 Calculation

Targeted Step: $u_1 = (h \cdot s_{\text{inv}}) \bmod n$ [4]
Fault Type: control flow

Similar to the previous FA, we can also target the multiplication in the fourth step and only return h instead of the actual result $h \cdot s_{\text{inv}}$. This has the effect that the first summand $[u_1]G$ in step 6 is now no longer dependent on the signature value s. Consequently, we can apply almost the same attack as for the fault in attack 4.3.2.3 that completely eliminated the first summand. The only difference is that we need to consider the constant point $[u_1]G = [h]G$ in the equation, resulting in the following attack:

1. choose arbitrary u_2 from $\{1, \ldots, n-1\}$
2. calculate $R' = [h]G + [u_2]P$
3. take $r = R'.x$
4. calculate $s = r^{-1} \cdot u_2$

We were not able to find a fault that could enable such an attack in the libraries we analyzed. Nevertheless, it is included as it is possible under the assumed FM.

4.3.3 EdDSA

While we deemed attacks on twisted Edwards curves using faulted points infeasible in Section 4.2.4, we were able to identify five ways in which common implementations can be attacked. We assume the same FM as in the last section.

The verification algorithm for an EdDSA signature (R, s) as described in Section 2.5 is slightly inefficient and is thus modified in most implementations to save the decompression of the point R. This is done by calculating $Q = [s]G - [h]P$ and comparing its compressed representation to the encoded value of R directly from the signature. The final resulting algorithm was originally proposed by Bernstein et al. [BDL+11] for the fast verification of a single signature and consists of the following steps:

1. verify that $0 \leq s < n$
2. calculate $h = H(R||P||m)$
3. calculate $Q = [s]G - [h]P$
4. verify that $R \stackrel{?}{=} Q$

All cryptographic libraries we analyzed used this implementation. We therefore also assume it for our FAs.

4.3.3.1 Faulting the Final Comparison

Targeted Step: $R \stackrel{?}{=} Q$ [4]
Fault Type: control flow

Regardless of which variant of the algorithm is implemented, it is possible to simply introduce a fault into the final step of comparing the resulting points. Again, successful FAs can come in many variations and are highly dependent on the implementation. Appendix A.7 shows how this attack could be realized in wolfSSL.

4.3.3.2 Skipping Calculation of Witness Q

- Targeted Step: $Q = [s]G - [h]P$ [3]
- Fault Type: control flow

By skipping the third step entirely, we are left with the freshly initialized point Q. Most implementations will initialize a point with the identity element of the curve. We can thus simply use this identity element as our point R in the signature and successfully verify any message after such a fault. This attack is still effective, even if the point is not initialized at all but contains some other constant and predictable value that, again, can be used as the point R in our signature. Appendix A.8 shows how this attack could be realized in wolfSSL.

4.3.3.3 Fault Addition to Return First Summand Only

- Targeted Step: $Q = [s]G - [h]P$ [3]
- Fault Type: control flow

With a fault in the addition of the third step, a solution for the signature is trivially found. When only the first summand $[s]G$ is returned, a signature only needs to satisfy $R = [s]G$ to be accepted. Since R and s make up the supplied signature, we can simply choose any value for s in $\{0, \ldots, n-1\}$ and calculate the value of R.

While this attack is feasible regarding our FM, none of our analyzed implementations allowed for such an attack. Since we do not know of any implementations that deploy countermeasures against FAs on elliptic curve signature verification algorithms, this is still a valuable insight. While the analyzed implementations may be resistant to this attack by chance, it needs to be considered when designing generic hardenings for this scheme.

As a side note, if newly allocated EC points are initialized with the neutral element of the curve, it might be possible to achieve the same effect by preventing the loading of the public key point into the initialized point. This is, however, unlikely and not done in wolfSSL, as there is no need to initialize a point with some specific constant if it is later overwritten. The same is also true for faulting the value of the hash to 0, which will also effectively eliminate the second summand from the addition.

4.3.3.4 Faulting Scalar Multiplication with Hash

- Targeted Step: $Q = [s]G - [h]P$ [3]
- Fault Type: control flow

Similar to the previous fault, we try to eliminate the dependency on the signature point R from the witness Q. Instead of targeting the addition in step 3, it is also possible to inject a fault into the scalar point multiplication of the hash and the public key. To be specific, we want to eliminate the hash from the equation as it depends on R, effectively reducing the calculation of $[s]G - [h]P$ to $[s]G - P$. Again, a solution for this faulted problem is trivially found by choosing a valid value for s and calculating the second part of the signature with $R = [s]G - P$. Appendix A.9 shows how this attack could be realized in wolfSSL.

4.3.3.5 Fault Hash Calculation

- Targeted Step: $h = H(R||P||m)$ [2]
- Fault Type: control flow

Continuing the scheme of attacks, we can inject a fault to remove the dependency on R on the right side of the equation even earlier. If we can exclude the point R from the hash calculated in the second step, we are left with $h = H(P||m)$. Again, we can calculate the missing value of R after fixing an arbitrary value for s with $R = [s]G - [H(P||m)]P$. Appendix A.10 shows how this attack could be realized in wolfSSL.

4.4 Practical Verification of Faults

To verify the correctness of our attacks on ECGDSA and ECSDSA, we implemented the signature schemes, as well as the attacks, in SageMath. Using this setup, we are able to forge a signature for a 192-bit curve in less than six minutes on a single core of a current-generation CPU.

We tried to perform all attacks on ECDSA and EdDSA described in Sections 4.3.2 and 4.3.3 on the cryptographic libraries used in MCUboot and wolfBoot. Except for the attacks described in the attacks 4.3.2.4, 4.3.2.5, and 4.3.3.3, to which none of the analyzed implementations were susceptible, we provide descriptions on how this attack could be realized.

We created a proof of concept for each attack to practically verify the feasibility of the fault. For both ECDSA and EdDSA, a wrapper application was written that verifies a given signature using the cryptographic implementations in Mbed TLS (v3.5.2) or wolfSSL (v5.7.0).

To verify an attack, an arbitrary message was chosen, and the signature value was initialized either with the forged signature or with an arbitrary invalid signature. After constructing these examples, we executed each one to verify that the signatures were rejected without the presence of faults. In the final step, we attached a debugger to the programs to manually simulate the fault required to accept the invalid signature and verified that the signature was in fact accepted.

5 Countermeasures

This section describes countermeasures against the attacks discovered in Section 4. The suggested mitigation for the attacks on ECGDSA and ECSDSA is straightforward and effective but comes with a noteworthy increase in runtime. Meanwhile, the attacks on ECDSA and EdDSA, which are based on exploitable weaknesses in common implementations, require carefully selected checks but come at a negligible runtime cost for our basic FM.

5.1 Hardening Weierstraß Group Law

Our attacks on ECGDSA and ECSDSA are based on a weakness in the group law of Weierstraß curves that infers the curve parameter b from the supplied point. Implementations for these operations favor incomplete addition formulas since they are slightly more efficient. The complete addition formula introduced by Renes et al. [RCB16] has the advantage of including the curve parameter b in the calculation. The attacks covered in Sections 4.2.1 and 4.2.2 are successfully mitigated by using this formula. A fault in the base or public key point would then result in seemingly arbitrary pairs of field elements X when used in a scalar point multiplication. To benefit from such a fault, we must be able to find a value r such that $[r]G' = X$, which is not efficiently possible with currently known methods since the calculation will no longer represent a valid EC operation. Unfortunately, using a complete group law can result in a runtime increase of up to 44% [RCB16]. Given that the elliptic curve operations represent the most expensive steps in signature verification within these schemes, it is reasonable to infer an equal impact on the overall verification process' runtime.

Alternatively, an on-curve test can be used to verify that the calculated witness point is still on the expected curve. This calculation only requires a few basic field operations and will thus have a negligible impact on the overall runtime and is already implemented in most implementations. While this test is not as solid as the aforementioned alternative, as it can easily be targeted by another fault, it is much more performant and easier to add to an existing implementation.

5.2 Hardening Signature Verification

This section focuses on implementing simple software-based countermeasures to enhance the security of the ECDSA and EdDSA signature schemes against the fault models described in Section 4.3.1, aiming to minimize the attack surface without significant overhead. We also discuss how these hardenings can be adjusted to protect against the extended fault model, which assumes an attacker with advanced faulting capabilities.

Generic approaches that aid in building resilience against FAs have already been studied in both hard- and software. Modular redundancy [GRRV06], sensors that monitor the environment [YPA08, FBCP10, BTD+13], error correcting codes [GRRV06], and shielding [CDG+14] have been suggested as countermeasures implemented in hardware.

Given that these signature algorithms and cryptographic libraries are often used on general-purpose microcontrollers without hardware protection measures, our hardened schemes are designed to operate without any supporting hardware-based countermeasures. However, it is highly recommended to combine hard- and software countermeasures, as it significantly raises the required complexity of attacks.

There already exist various approaches that try to generically harden control flow [BR23, GRRV06, TMS+13] and data integrity [TMS+13] in software. To guarantee completeness and full protection against the attacks discovered in Section 4.3, we are going to only rely on incorporating additional checks and redundancies. These ensure that operations do not return known values, such as input values or the identity element, by comparing operation results with constants or input values and aborting the verification if a match is found. This allows us to fortify the verification procedures against the identified attacks while incurring minimal overhead.

Considering the extended FM described in Section 4.3.1, additional countermeasures are required. Appendix A.11 describes how this FM can be handled through redundant computations and a technique similar to blinding.

While perfect security is unattainable, these hardened schemes significantly increase the time and resources required for an attacker to succeed, such as necessitating multiple time-sensitive sequential faults.

5.2.1 Hardened ECDSA

In this section, ECDSA is hardened through a minimal set of boundary checks to prevent the attacks identified in Section 4.3.2. Each step for which we identified an attack is augmented with one or multiple checks that verify that the respective attacks have not been performed. The final comparison step must be securely executed under the consideration of the attacker's faulting capabilities. For our assumed FM, this means that the check needs to be performed redundantly. We will not go into further detail on how this can be implemented, as wolfBoot and MCUboot already provide a way of securely implementing such checks with compiler optimizations and target architecture in mind. All of these considerations finally result in the hardened scheme:

1. calculate message hash $h = H(m)$
2. invert signature proof: $s_{\text{inv}} = s^{-1} \pmod{n}$
3. calculate $u_1 = (h \cdot s_{\text{inv}}) \bmod n$
    - **check** $u_1 \neq h$ (Attack 4.3.2.5)

4. calculate $u_2 = (r \cdot s_{\text{inv}}) \bmod n$
   - **check** $u_2 \neq s_{\text{inv}}$ (Attack 4.3.2.4)
5. calculate $Q1 = [u_1]G$
   - **check** $Q1 \neq G$ (Attack 4.3.2.3)
6. calculate $Q2 = [u_2]P$
   - **check** $Q2 \neq P$ (Attack 4.3.2.3)
7. recover random point $R' = Q1 + Q2$
   - **check** $R' \neq e$ (Attack 4.3.2.3)
   - **check** $R' \neq Q1$
   - **check** $R' \neq Q2$
8. check $r \stackrel{?}{=} R'.x$ (**redundant check**) (Attack 4.3.2.1)

A pitfall of this strategy is the possibility of rejecting valid signatures. For example, if the signature value r equals 1, the check in step 4 will immediately reject this signature. These special cases need to be considered when generating a signature to guarantee that the signature will be accepted by the verifier. In case it is not possible to adjust the signing procedure to these new requirements, one might also get away with ignoring them, since the number of special cases that will falsely get rejected is so small ($\sim 1/n$) that it is almost impossible to encounter such a case by chance.

5.2.2 Hardened EdDSA

Similar to the previous section, we present a hardened variant of EdDSA. We again realized the hardenings by adding simple checks to detect the attacks identified in Section 4.3.3. As with ECDSA, the comparison in the last step needs to be performed redundantly under consideration of the FM.

1. check $0 < s < n$
2. **check** $R \neq e$ (Attack 4.3.3.2)
3. calculate $h = H(R)$
   - **check** that h is not equal to initial state of hash function (Attack 4.3.3.5)
4. calculate $h = H\_update(h, P||m) \bmod q$
5. calculate $Q1 = [s]G$
6. calculate $Q2 = [h]P$
   - **check** $Q2 \neq P$ (Attack 4.3.3.4)
7. calculate $Q = Q1 - Q2$
   - **check** $Q \neq Q1$ (Attack 4.3.3.3)
8. check $R \stackrel{?}{=} Q$ (**redundant check**) (Attack 4.3.3.1)

To ensure resilience against the fault described in attack 4.3.3.5, we would need to recompute the hash to make sure the signature point R was properly digested. We can, however, be a bit more efficient than that by only hashing R first. Now we can verify that the initial state of the hash function has changed, indicating that the hashing operation has not been skipped, and continue calculating the hash with the remaining values P and m.

5.2.3 Overhead of Hardenings

Before discussing the hardenings' performance overheads, it is important to consider common optimizations. Both ECDSA and EdDSA perform a costly [BDL+11] double-scalar multiplication during verification. To make the two multiplications more efficient, they are often combined into a single operation where the factors are read simultaneously or processed with a sliding window [Ava05]. This approach is roughly 40% faster than two single-scalar multiplications [BCL17].

The optimization of steps 5, 6, and 7 in both of our hardened schemes makes the checks involving the intermediate points $Q1$ and $Q2$ inapplicable, because they are never calculated during the optimized computation. This is, however, not a concern as the parallel processing of factors should eliminate the possibility of fully omitting one of them. In this case, the aforementioned checks do not need to be considered.

Upon disclosure of the attacks described in this paper, wolfSSL implemented our suggested checks as optional hardenings starting with release 5.7.6. To confirm that our basic checks do not significantly impact the runtime of the verification, we measured their performance impact in wolfSSL 5.7.6 on an STM32F415 MCU. Table 2 shows the number of CPU cycles needed for the verification of a signature with and without hardenings in place for both ECDSA and EdDSA. As expected, our countermeasures have a minimal impact on the overall runtime.

Table 2: Cycle count for signature verification in wolfSSL 5.7.6 with and without countermeasures (CMs)

Signature algorithm	Cycle count	Cycle count increase of CMs	
		Absolute	Relative
Ed25519	1.9M	+20.09k	+1.03%
Ed25519 lowresource*	109.6M	+23.97k	+0.02%
ECDSA P-256	40.2M	+16.27k	+0.04%
ECDSA P-256 lowresource*	136.5M	+31.42k	+0.02%

* The *lowresource* build flag disables optimizations such as Shamir's trick that decrease runtime but increase memory requirements.

All countermeasures have a small memory overhead because they only save intermediate values or the results of duplicate calculations for later comparison.

6 Discussion

The attacks presented in Section 4 pose a serious threat to the integrity of the analyzed signature schemes, and clearly demonstrate that signature verification needs to be protected as well as signing. Table 3 shows a summary of all described attacks and compares the contingent complexity of the offline phase required to forge a signature.

Edwards curves are known for their natural resistance against side-channel analysis attacks due to their unified addition and multiplication equation. Our findings indicate that they also demonstrate superior resilience against attacks exploiting faulty points or curve parameters compared to Weierstraß curves by using all curve coefficients during point multiplication. While undoubtedly more complex than some of the attacks on the implementation we found, this shortcoming of Weierstraß curves is easily overlooked when designing or hardening a signature scheme. We demonstrated that such an attack on the curve parameters or points used during verification is infeasible for ECDSA and are certain

Table 3: Attack overview and comparison based on the complexity of the offline phase.

Section	Signature Scheme	Attack Target	Offline Complexity
4.2.1	ECGDSA	bit-flip in P	$\mathcal{O}(\sqrt{p})^*$
4.2.2	ECSDSA	bit-flip in G	$\mathcal{O}(\sqrt{p})^*$
4.3.2.1		final compare	negligible
4.3.2.2		curve order	$\mathcal{O}(2^l)^\dagger$
4.3.2.3	ECDSA	point addition	
4.3.2.4		u_2 calculation‡	negligible
4.3.2.5		u_1 calculation‡	
4.3.3.1		final compare	
4.3.3.2		calculation of Q	
4.3.3.3	EdDSA	point addition‡	negligible
4.3.3.4		hash multiplication	
4.3.3.5		hash calculation	

* p is the biggest prime factor of the faulted curve order ($\sim 2^{60}$ for a 256-bit curve)

† l is either the bit-length of a byte or a word depending on the implementation

‡ none of the analyzed implementations were vulnerable to this attack

that similar attacks cannot be performed on Edwards curves. We thus recommend using Edwards curve-based signature algorithms on systems that might be physically accessed by an adversary, such as for secure bootloaders used in embedded systems.

The remaining attacks on ECDSA and EdDSA have shown how simple it is to make a device accept forged signatures by only skipping a single instruction. It is apparent that all analyzed libraries supporting those schemes do not implement meaningful defensive measures since we did not run into any difficulties while performing our attacks on hardware with simulated faults. Even though many secure bootloaders, like MCUboot and wolfBoot, are well aware of the threat of FAs and thus have implemented software-based countermeasures, they are still using unprotected libraries that are easily exploited.

As a short-term solution, we recommend executing the verification functions twice and comparing the results for equality. This already significantly improves the security by requiring at least two faults, but comes at the cost of doubling the verification process's runtime. On a highly performant web server that needs to handle thousands of TLS handshakes per minute, such an increase in the runtime of signature verification would greatly impact the overall performance of the server. In contrast, an IoT device that verifies a single signature to ensure the authenticity of a firmware image might prioritize security over saving runtime for a single operation, because it is also much more susceptible to fault attacks to begin with. For a more sustainable solution, these problems need to be addressed in the cryptographic library itself.

The hardened schemes presented in Section 5.2 efficiently protect the verification algorithms from an adversary regarding the specified FM. As a consequence, an attacker would need to expand their FM which considerably increases the required time and resources for a successful attack. It is thus important to note that these hardening techniques are in no way a universal solution to preventing FAs. Instead, they should act as a guideline on how to harden a scheme and should be individually adjusted to the target platform, corresponding FM, and consequently the required level of security.

6.1 Future Work

While we explained why faulty point attacks on ECDSA are infeasible in Section 4.2, we were unable to prove this claim. We attributed the resilience of the scheme against this kind of attack to the interdependent factors of the EC points used in the double-scalar multiplication, which did not allow us to reduce the problem to a single ECDLP over a weak curve. The security of EC schemes on Weierstraß curves would thus greatly benefit from a formalized description of the property that determines whether such an attack is possible.

We have shown that such an attack is not possible on twisted Edwards curves, which we attribute to the fact that all curve parameters are used in their unified addition equation. It would be interesting to formally prove this property and its consequences for twisted Edwards curves. Taking this one step further, one could also analyze whether this weakness exists for every curve representation that does not utilize all curve parameters in its group operation.

We demonstrated that current state-of-the-art signature schemes are susceptible to fault attacks during verification, which allow an adversary to forge and verify invalid signatures. Further work on this topic could thus focus on designing a signature scheme that is resilient to FA targeting the verification procedure by design. It is well known that Edwards curves are resilient to side-channel attacks. In this paper we demonstrated that they are also resilient against fault attacks that target EC points or curve parameters, which is why we suggest building the foundations of such a scheme on these curves. A scheme that is resilient against fault attacks would likely require some kind of redundancy built into it, which would cause losses in performance and thus not constitute a great general-purpose scheme. However, in cases where security is prioritized over maximizing performance, such a purpose-built scheme could simplify the process of hardening a system against FAs and ultimately improve security.

Until such a purposeful scheme is designed, existing cryptographic libraries need to be developed that implement optional hardenings for commonly used schemes like EdDSA and ECDSA as suggested in Section 5.2.

7 Conclusion

We have demonstrated that even a single bit-flip fault in an elliptic curve point or a curve parameter is sufficient to fully compromise the signature schemes ECGDSA and ECSDSA. This weakness was enabled by the underlying Weierstraß curves used in many signature schemes. Meanwhile, the more common ECDSA scheme proved to be resilient against such an attack because of the interdependence of the factors used in the multi-scalar point multiplication. We have further shown that EdDSA is not susceptible to this attack, as the underlying Edwards curves do not possess this faulty point weakness. Given the many other advantages of Edwards curve-based signature schemes already highlighted by others in terms of security [IHH+20] and performance [BBJ+08, BL07], we can only recommend such schemes in regard to FAs on the verification routine.

Apart from the faults on elliptic curves, we discovered ten implementation-based attacks on ECDSA and EdDSA allowing forged signatures to be erroneously accepted. This demonstrates the ease of accepting invalid signatures in unhardened implementations. We want to emphasize the importance of hardening all critical components of a system against FAs. For example, secure bootloaders deploy comprehensive software-based hardenings in the core of the firmware. The hardenings can, however, not raise the complexity of an attack on the system if critical parts of it, like the cryptographic libraries, are left completely unprotected.

Although our suggested countermeasures can mitigate these vulnerabilities, we acknowledge that they are interim solutions for an issue requiring a more comprehensive approach. Therefore, further research in this area is essential to developing a solution that effectively safeguards devices vulnerable to FAs on cryptographic algorithms. While software-based hardenings are a cost-effective way to add resilience against FAs to a system, they should always be accompanied by hardware-based countermeasures to achieve the best possible fault resistance.

It is important to keep in mind that no solution can ever provide perfect security against every fault model. The single purpose of all these efforts is to increase the complexity of the attacks required to compromise the security of a system. With this rising complexity, an adversary is forced to invest more time and resources in order to successfully breach the system. We can consider a system to be secure when the cost of breaching it outweighs the potential reward.

Acknowledgements

The authors thank the anonymous reviewers for their comments that significantly improved the quality of this paper. This work was supported by the Bavarian Ministry of Economic Affairs, Regional Development and Energy, in the project ”Trusted Electronics Center Bavaria”.

References

[APS18] Alessandro Amadori, Federico Pintore, and Massimiliano Sala. On the discrete logarithm problem for prime-field elliptic curves. *Finite Fields and Their Applications*, 51:168–182, 2018.

[Ava05] Roberto Maria Avanzi. The complexity of certain multi-exponentiation techniques in cryptography. *Journal of Cryptology*, 18(4):357–373, September 2005.

[BBB+16] Alessandro Barenghi, Guido M. Bertoni, Luca Breveglieri, Gerardo Pelosi, Stefano Sanfilippo, and Ruggero Susella. A Fault-Based Secret Key Retrieval Method for ECDSA: Analysis and Countermeasure. *ACM Journal on Emerging Technologies in Computing Systems*, 13(1):8:1–8:26, 2016.

[BBJ+08] Daniel J. Bernstein, Peter Birkner, Marc Joye, Tanja Lange, and Christiane Peters. Twisted Edwards curves. In Serge Vaudenay, editor, *AFRICACRYPT 08*, volume 5023 of *LNCS*, pages 389–405. Springer, Berlin, Heidelberg, June 2008.

[BBK16] Nina Bindel, Johannes Buchmann, and Juliane Krämer. Lattice-based signature schemes and their sensitivity to fault attacks. Cryptology ePrint Archive, Report 2016/415, 2016.

[BCL17] Daniel J. Bernstein, Chitchanok Chuengsatiansup, and Tanja Lange. Double-base scalar multiplication revisited. Cryptology ePrint Archive, Report 2017/037, 2017.

[BDL+11] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. In Bart Preneel and Tsuyoshi Takagi, editors, *CHES 2011*, volume 6917 of *LNCS*, pages 124–142. Springer, Berlin, Heidelberg, September / October 2011.

[BGV11] Josep Balasch, Benedikt Gierlichs, and Ingrid Verbauwhede. An in-depth and black-box characterization of the effects of clock glitches on 8-bit mcus. In *2011 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 105–114, 2011.

[BH22] Jakub Breier and Xiaolu Hou. How practical are fault injection attacks, really? *IEEE Access*, 10:113122–113130, 2022.

[BL07] Daniel J. Bernstein and Tanja Lange. Faster addition and doubling on elliptic curves. In Kaoru Kurosawa, editor, *ASIACRYPT 2007*, volume 4833 of *LNCS*, pages 29–50. Springer, Berlin, Heidelberg, December 2007.

[BL13] Daniel Bernstein and Tanja Lange. Two grumpy giants and a baby. *The Open Book Series*, 1, 2013.

[BMM00] Ingrid Biehl, Bernd Meyer, and Volker Müller. Differential fault attacks on elliptic curve cryptosystems. In Mihir Bellare, editor, *CRYPTO 2000*, volume 1880 of *LNCS*, pages 131–146. Springer, Berlin, Heidelberg, August 2000.

[BN06] Mihir Bellare and Gregory Neven. Multi-signatures in the plain public-key model and a general forking lemma. In Ari Juels, Rebecca N. Wright, and Sabrina De Capitani di Vimercati, editors, *ACM CCS 2006*, pages 390–399. ACM Press, October / November 2006.

[BR23] Davide Baroffio and Federico Reghenzani. Compiler-injected sihft for embedded operating systems. In *Proceedings of the 20th ACM International Conference on Computing Frontiers*, CF '23, page 337–343, New York, NY, USA, 2023. Association for Computing Machinery.

[BTD+13] R. Possamai Bastos, F. Sill Torres, J.-M. Dutertre, M.-L. Flottes, G. Di Natale, and B. Rouzeyre. A bulk built-in sensor for detection of fault attacks. In *IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, pages 51–54. IEEE, 2013.

[CDG+14] Jean-Michel Cioranesco, Jean-Luc Danger, Tarik Graba, Sylvain Guilley, Yves Mathieu, David Naccache, and Xuan Thuy Ngo. Cryptographically secure shields. In *2014 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, pages 25–31, 2014.

[CH17] Ang Cui and Rick Housley. Badfet: Defeating modern secure boot using second-order pulsed electromagnetic fault injection. In *Workshop on Offensive Technologies*, 2017.

[CMO98] Henri Cohen, Atsuko Miyaji, and Takatoshi Ono. Efficient elliptic curve exponentiation using mixed coordinates. In Kazuo Ohta and Dingyi Pei, editors, *ASIACRYPT'98*, volume 1514 of *LNCS*, pages 51–65. Springer, Berlin, Heidelberg, October 1998.

[CP95] J.A. Clark and D.K. Pradhan. Fault injection: a method for validating computer-system dependability. *Computer*, 28(6):47–56, 1995.

[CS18] Craig Costello and Benjamin Smith. Montgomery curves and their arithmetic - the case of large characteristic fields. *Journal of Cryptographic Engineering*, 8(3):227–240, September 2018.

[CSC+21] Weiqiong Cao, Hongsong Shi, Hua Chen, Wei Wei, and Jiazhe Chen. Lattice-Based Weak Curve Fault Attack on ECDSA. In Audun Jøsang, Lynn Futcher, and Janne Hagen, editors, *ICT Systems Security and Privacy Protection*, volume 625, pages 146–161. Springer International Publishing, 2021.

[DBC+18] Jean-Max Dutertre, Vincent Beroulle, Philippe Candelier, Stephan De Castro, Louis-Barthelemy Faber, Marie-Lise Flottes, Philippe Gendrier, David Hely, Regis Leveugle, Paolo Maistri, Giorgio Di Natale, Athanasios Papadimitriou, and Bruno Rouzeyre. Laser Fault Injection at the CMOS 28 nm Technology Node: An Analysis of the Fault Model. In *2018 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 1–6. IEEE, 2018.

[DDRT12] Amine Dehbaoui, Jean-Max Dutertre, Bruno Robisson, and Assia Tria. Electromagnetic transient faults injection on a hardware and a software implementations of aes. In *2012 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 7–15, 2012.

[Ela20] Elaine Barker. Recommendation for Key Management: Part 1 – General (Revision 5). NIST Special Publication 800-57 Part 1 Rev. 5, National Institute of Standards and Technology, May 2020.

[Fay14] Björn Fay. Double-and-add with relative Jacobian coordinates. Cryptology ePrint Archive, Report 2014/1014, 2014.

[FBCP10] John J. Leon Franco, Eduardo Boemo, Encarnacion Castillo, and Luis Parrilla. Ring oscillators as thermal sensors in FPGAs: Experiments in low voltage. In *2010 VI Southern Programmable Logic Conference (SPL)*, pages 133–137. IEEE, 2010.

[FKP16] Manuel Fersch, Eike Kiltz, and Bertram Poettering. On the provable security of (ec)dsa signatures. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 1651–1662, New York, NY, USA, 2016. Association for Computing Machinery.

[Gau09] Pierrick Gaudry. Index calculus for abelian varieties of small dimension and the elliptic curve discrete logarithm problem. *Journal of Symbolic Computation*, 44(12):1690–1702, 2009.

[GG14] Steven D. Galbraith and Shishay W. Gebregiyorgis. Summation polynomial algorithms for elliptic curves in characteristic two. In Willi Meier and Debdeep Mukhopadhyay, editors, *INDOCRYPT 2014*, volume 8885 of *LNCS*, pages 409–427. Springer, Cham, December 2014.

[GG16] Steven D. Galbraith and Pierrick Gaudry. Recent progress on the elliptic curve discrete logarithm problem. *DCC*, 78(1):51–72, 2016.

[GHS02] P. Gaudry, F. Hess, and N. P. Smart. Constructive and destructive facets of Weil descent on elliptic curves. *Journal of Cryptology*, 15(1):19–46, 2002.

[GRRV06] Olga Goloubeva, Maurizio Rebaudengo, Matteo Sonza Reorda, and Massimo Violante. *Software-Implemented Hardware Fault Tolerance*. Springer, 2006.

[Hua03] Andrew Bunnie Huang. *Hacking the Xbox: an introduction to reverse engineering*. No Starch Press, 2003.

[IHH+20] Md Mainul Islam, Md Selim Hossain, Moh Khalid Hasan, Md Shahjalal, and Yeong Min Jang. Design and Implementation of High-Performance ECC Processor with Unified Point Addition on Twisted Edwards Curve. *Sensors*, 20(18):5148, 2020.

[ISO18] IT Security techniques — Digital signatures with appendixPart 3: Discrete logarithm based mechanisms. International Standard ISO/IEC 14888-3:2018, International Organization for Standardization, 2018.

[JMV01] Don Johnson, Alfred Menezes, and Scott Vanstone. The Elliptic Curve Digital Signature Algorithm (ECDSA). *International Journal of Information Security*, 1(1):36–63, 2001.

[KG20] Chelsea Komlo and Ian Goldberg. FROST: flexible round-optimized schnorr threshold signatures. In Orr Dunkelman, Michael J. Jacobson Jr., and Colin O'Flynn, editors, *Selected Areas in Cryptography - SAC 2020 - 27th International Conference, Halifax, NS, Canada (Virtual Event), October 21-23, 2020, Revised Selected Papers*, volume 12804 of *Lecture Notes in Computer Science*, pages 34–65. Springer, 2020.

[KH14] Thomas Korak and Michael Hoefler. On the effects of clock and power supply tampering on two microcontroller platforms. In *2014 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 8–17, 2014.

[KJP14] Raghavan Kumar, Philipp Jovanovic, and Ilia Polian. Precise fault-injections using voltage and temperature manipulation for differential cryptanalysis. In *2014 IEEE 20th International On-Line Testing Symposium (IOLTS)*, pages 43–48. IEEE, 2014.

[KSV13] Duško Karaklajić, Jörn-Marc Schmidt, and Ingrid Verbauwhede. Hardware Designer's Guide to Fault Attacks. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 21(12):2295–2306, 2013.

[LBC+15] Marc Lacruche, Nicolas Borrel, Clement Champeix, Cyril Roscian, Alexandre Sarafianos, Jean-Baptiste Rigaud, Jean-Max Dutertre, and Edith Kussener. Laser fault injection into SRAM cells: Picosecond versus nanosecond pulses. In *2015 IEEE 21st International On-Line Testing Symposium (IOLTS)*, pages 13–18. IEEE, 2015.

[LG19] Haohao Liao and Catherine Gebotys. Methodology for em fault injection: Charge-based fault model. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 256–259, 03 2019.

[low24] lowRISC CIC. OpenTitan: Open-Source Silicon Root of Trust. https://opentitan.org/, 2024.

[MDP+20] Alexandre Menu, Jean-Max Dutertre, Olivier Potin, Jean-Baptiste Rigaud, and Jean-Luc Danger. Experimental Analysis of the Electromagnetic Instruction Skip Fault Model. In *15th Design & Technology of Integrated Systems in Nanoscale Era (DTIS)*, pages 1–7, 2020.

[MKS12] Benjamin Michéle, Juliane Krämer, and Jean-Pierre Seifert. Structure-Based RSA Fault Attacks. In Mark D. Ryan, Ben Smyth, and Guilin Wang, editors, *Information Security Practice and Experience*, pages 301–318. Springer, 2012.

[MM18] G. McGuire and D. Mueller. A New Index Calculus Algorithm for the Elliptic Curve Discrete Logarithm Problem and Summation Polynomial Evaluation. 2018.

[MOV93] A.J. Menezes, T. Okamoto, and S.A. Vanstone. Reducing elliptic curve logarithms to logarithms in a finite field. *IEEE Transactions on Information Theory*, 39(5):1639–1646, 1993.

[MPSW19] Gregory Maxwell, Andrew Poelstra, Yannick Seurin, and Pieter Wuille. Simple schnorr multi-signatures with applications to bitcoin. *DCC*, 87(9):2139–2164, 2019.

[Mui06] James A. Muir. Seifert's RSA Fault Attack: Simplified Analysis and Generalizations. In Peng Ning, Sihan Qing, and Ninghui Li, editors, *Information and Communications Security*, volume 4307, pages 420–434. Springer Berlin Heidelberg, 2006.

[NRS21] Jonas Nick, Tim Ruffing, and Yannick Seurin. MuSig2: Simple two-round Schnorr multi-signatures. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part I*, volume 12825 of *LNCS*, pages 189–221, Virtual Event, August 2021. Springer, Cham.

[OGSM16] Sébastien Ordas, Ludovic Guillaume-Sage, and Philippe Maurine. Electromagnetic fault injection: the curse of flip-flops. *Journal of Cryptographic Engineering*, 7:183 – 197, 2016.

[OGST+15] S. Ordas, L. Guillaume-Sage, K. Tobich, J.-M. Dutertre, and P. Maurine. Evidence of a larger em-induced fault model. In Marc Joye and Amir Moradi, editors, *Smart Card Research and Advanced Applications*, pages 245–259, Cham, 2015. Springer International Publishing.

[PH78] S. Pohlig and M. Hellman. An improved algorithm for computing logarithms over GF(p) and its cryptographic significance (Corresp.). *IEEE Transactions on Information Theory*, 24(1):106–110, 1978.

[Pol78] J. M. Pollard. Monte Carlo methods for index computation (mod p). *Mathematics of Computation*, 32(143):918–924, 1978.

[PS00] David Pointcheval and Jacques Stern. Security arguments for digital signatures and blind signatures. *Journal of Cryptology*, 13(3):361–396, June 2000.

[RCB16] Joost Renes, Craig Costello, and Lejla Batina. Complete addition formulas for prime order elliptic curves. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part I*, volume 9665 of *LNCS*, pages 403–428. Springer, Berlin, Heidelberg, May 2016.

[RSDT13] Cyril Roscian, Alexandre Sarafianos, Jean-Max Dutertre, and Assia Tria. Fault Model Analysis of Laser-Induced Faults in SRAM Memory Cells. In *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 89–98, 2013.

[Sch91] C. P. Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, 4(3):161–174, 1991.

[Sec24] Security Target BSI-DSZ-CC-1216-2024, 2024.

[Sei05] Jean-Pierre Seifert. On authenticated computing and RSA-based authentication. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, pages 122–127. Association for Computing Machinery, 2005.

[Sem04] Igor Semaev. Summation polynomials and the discrete logarithm problem on elliptic curves. Cryptology ePrint Archive, Report 2004/031, 2004.

[Sha71] Daniel Shanks. Class Number, a Theory of Factorization, and Genera. *Proc. Symp. Math. Soc., 1971*, 20:415–440, 1971.

[Sil00] Joseph H. Silverman. The Xedni Calculus and the Elliptic Curve Discrete Logarithm Problem. *Designs, Codes and Cryptography*, 20(1):5–40, 2000.

[SM09] Jörn-Marc Schmidt and Marcel Medwed. A Fault Attack on ECDSA. In *2009 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 93–99, 2009.

[Tec18] BSI TR-03111 Elliptic Curve Cryptography (ECC). Technical guideline, Bundesamt für Sicherheit in der Informationstechnik, 2018.

[TMS+13] Nikolaus Theissing, Dominik Merli, Michael Smola, Frederic Stumpf, and Georg Sigl. Comprehensive Analysis of Software Countermeasures against Fault Attacks. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2013*, pages 404–409. IEEE Conference Publications, 2013.

[Tru24] TrustedFirmware.org. MCUboot: Secure Boot for Microcontrollers. https://mcuboot.com/, 2024.

[VBH23] Andersson Calle Viera, Alexandre Berzati, and Karine Heydemann. Fault attacks sensitivity of public parameters in the dilithium verification. Cryptology ePrint Archive, Report 2023/1796, 2023.

[Was08] Lawrence C Washington. *Elliptic Curves: Number Theory and Cryptography*. Chapman and Hall/CRC, 2008.

[wol24] wolfSSL Inc. wolfBoot: Secure Bootloader for Embedded Systems. https://www.wolfssl.com/products/wolfboot/, 2024.

[WWO+23] Alexander Wagner, Vera Wesselkamp, Felix Oberhansl, Marc Schink, and Emanuele Strieder. Faulting Winternitz One-Time Signatures to Forge LMS, XMSS, or SPHINCS+ Signatures. In Thomas Johansson and Daniel Smith-Tone, editors, *Post-Quantum Cryptography*, pages 658–687. Springer Nature Switzerland, 2023.

[YPA08] Asier Goikoetxea Yanci, Stephen Pickles, and Tughrul Arslan. Detecting Voltage Glitch Attacks on Secure Devices. In *2008 Bio-inspired, Learning and Intelligent Systems for Security*, pages 75–80. IEEE, 2008.

A Appendix

A.1 Random Curve Order Smoothness

The experimental results mentioned in Section 4.2 were found by generating 10,000 random 256-bit curves and factorizing their curve orders. The curves were generated by choosing a random value for the curve parameter $b \in \{1, \ldots, p-1\}$ with a fixed value for the parameter a. Only the parameter b was randomized in order to more closely resemble the scenario of introducing bit flip faults into an elliptic curve point. The results discussed in Section 4.2 suggest that adding more randomness through the curve parameter a will not make the resulting curve order any more random as the smoothness distribution already closely resembles the expected results for fully random 256-bit integers. Figure 3 shows the results of this experiment. Accumulating them and dividing by the total number of data points yields the curve shown in Fig. 1.

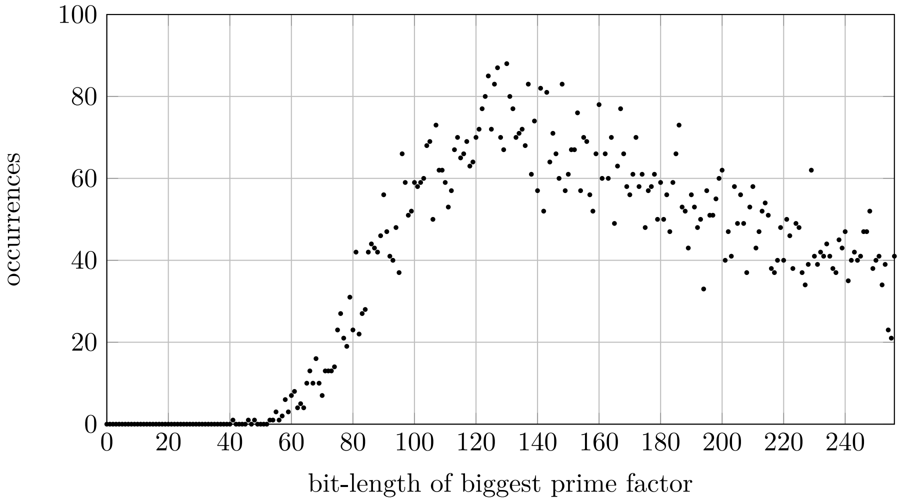

Figure 3: Smoothness of curve orders from randomly generated 256-bit curves.

A.2 Attempts to Fix Signature Values for ECDSA

For this attempt to leverage faulty points to attack ECDSA verification, it is crucial to expand the FM to encompass all potential scenarios. Biehl et al. [BMM00] initially proposed to fault an elliptic curve point with the goal of performing the calculations on another, possibly weaker curve. Expanding on this idea, we can also target the curve parameter a on the target device. By introducing a bit-flip to a, we can change the curve on which the calculations are performed, not only for a single point but for all of them with a single fault, which follows directly from the proof on pseudo addition given by Biehl et al. For the purpose of the attack on ECDSA, this means that a single bit flip fault on the curve parameter a can lead to both G and P ending up on weak (but almost certainly distinct) curves on which the ECDLP can be more easily solved.

When trying to apply the same attack as with ECGDSA, we can see that a fixed value of r is not enough to fix the value for any of the two summands, since both factors of the scalar point multiplications still depend on s. The only free variables in this equation are now the summands of the point addition and s. Assigning a value to any of these variables, however, already strongly restricts the possible values of the remaining ones to a few. Choosing a sensible value here is the crucial step to finding a solution for the signature.

The rationale behind introducing faults to the elliptic curve points or curve parameters is to leverage the Pohlig-Hellman algorithm or its principles for the efficient resolution of the ECDLP. We are, however, certain that it is impossible to reduce the problem to a variation of the ECDLP that only includes points on the same curve. We will try to elucidate this somewhat abstract statement through two examples where we set s and r to 1.

Fixing s When setting s to 1, we can simplify the witness calculation to:

$$\begin{aligned} R' =& [h \cdot 1]G + [r \cdot 1]P \\ R' =& [h]G + [r]P \qquad\qquad |X = [h]G \\ r =& (X + [r]P).x. \end{aligned}$$

This simplifies the problem to finding a value r so that $r = (X + [r]P).x$. Being able to efficiently find a solution for this problem is a prerequisite for solving the more complex problem above. We have already discussed that a fault in the curve parameter a may result in G and P ending up on weaker but distinct curves. Applying the Pohlig-Hellman algorithm to this problem would, however, require all involved group elements, the elliptic curve points, to be located in a common group.

Fixing r Alternatively, we can also choose a value for r before finding a fitting s to complete the signature. After doing this, we are left with $1 = [h \cdot s_{\text{inv}}]G + [s_{\text{inv}}]P$. With a well-chosen fault, we are also able to drastically reduce the smoothness of both G and P. While this enables us to efficiently solve the ECDLP for both points individually, it is not helpful when trying to find a solution for our equation since the scalar factors of the point multiplications depend on each other through s. When trying to apply the Pohlig-Hellman algorithm to this more complex problem, one quickly realizes that the weak order of both points is of no use.

In both cases, we will end up with two points that are extremely unlikely to be on the same curve after the fault. Consequently, it is safe to assume that the points will not generate the same group or even share a subgroup. Hence, solving the problem or its parts by computing a solution for a subgroup is not possible due to the absence of a common subgroup to work with.

A.3 Attempt to Exploit Coprime ECDSA Curve Orders

We start by formulating the equivalence the signature needs to fulfill as

$$r = ([h \cdot s_1]G' + [r \cdot s_2]P').x.$$

The first step of the attack is to fix arbitrary values for r, h, and s_1. Fixing a value for r this early restricts us later on but is necessary to decouple the elliptic curve operations from the integer calculations on the polynomial ring of the signature scheme by allowing us to calculate the required points in the final addition right away. Now, we can calculate the point $[h \cdot s_1]G$ and find the missing point X in the group generated by P', so that

$$r = ([h \cdot s_1]G' + X).x.$$

It is now possible to calculate the value of s_2 by solving the ECDLP over the weak curve of P' using the Pohlig-Hellman algorithm. The orders of the groups generated by G' and P' will be referred to as n_g and n_p, respectively, from now on.

With $n_g \neq n_p$, we need to find an s so that

$$\begin{aligned} s_1 &= s \bmod n_g \\ s_2 &= s \bmod n_p. \end{aligned}$$

A solution for this system of equations is easily found using the CRT under the condition that n_g and n_p are coprime (i.e., their greatest common divisor (GCD) is 1). In that case, the CRT states that there exists exactly one solution for s in the range $\{0, \ldots, n_g \cdot n_p - 1\}$. Even if n_g and n_p were not coprime, it might still be possible to find a solution if the GCD is sufficiently small by brute-forcing the message hash until s_1 and s_2 have the same remainders regarding their shared divisors. We will, however, assume that every pair of randomly generated pairs of group orders is coprime to show that, even with this simplification, a solution for this problem is still harder to find than solving the ECDLP for the public key on the unfaulted curve.

Considering our solution for s falls within the range $\{0, \ldots, n_g \cdot n_p - 1\}$, it is highly probable that it exceeds n, the order of the original curve. As the verification algorithm is terminated if a value larger than $n-1$ is provided for s, we are somewhat restricted in our solution space. By taking a closer look at the calculation of the factors for the scalar point multiplications,

$$u_1 = h \cdot s^{-1} \bmod n$$
$$u_2 = r \cdot s^{-1} \bmod n,$$

the set of possible solutions is restricted even more. The result of the multiplications is taken modulo n. Until now, we got away with using the group orders of the faulted points since they are implicitly respected when performing scalar point multiplications. During the calculation of u_1 and u_2, however, the algorithm explicitly performs a modulo operation using the order of the original curve that is completely unrelated to the orders n_g and n_p, resulting in:

$$u_1 = ((h \cdot s^{-1}) \bmod n) \bmod n_g \tag{3}$$
$$u_2 = ((r \cdot s^{-1}) \bmod n) \bmod n_p. \tag{4}$$

Since r can be freely chosen in the first step, we are going to set $r = 1$ to simplify the equation, which is also the smallest possible value. Again, we need to fix the value of r before we can calculate u_1 and u_2, giving us less freedom when trying to find a valid signature in this step. We consequently end up with

$$u_2 = (s^{-1} \bmod n) \bmod n_p.$$

The inverse s^{-1} is calculated modulo n in an earlier step of the verification process. It is thus necessarily in the range $[1; n[$, meaning that the first modulo operation of the equation above can be removed:

$$u_2 = s^{-1} \bmod n_p.$$

It follows that

$$s^{-1} = u_2 + i \cdot n_p \qquad \mid i \in \mathbb{N}_0; i < \lfloor \frac{n - u_2}{n_p} \rfloor.$$

Similarly, we describe candidates for s in the first equation:

$$u_1 = x \bmod n_g$$
$$x = (h \cdot s^{-1}) \bmod n.$$

We can describe the possible values of x using the first equation:

$$x = u_1 + j \cdot n_g \qquad \mid j \in \mathbb{N}_0; j < \lfloor \frac{n - u_1}{n_g} \rfloor.$$

And the value of s by deriving a congruence from the second one:

$$s^{-1} \equiv h^{-1} \cdot x \pmod{n}.$$

Combining the two equations, we can describe all possible values for s in Eq. (3). This results in a system of equations over a finite field of size n:

$$s^{-1} = u_2 + i \cdot n_p \qquad \mid i \in \mathbb{N}_0; i < \lfloor \frac{n - u_2}{n_p} \rfloor, \quad (5)$$

$$s^{-1} = h^{-1} \cdot (u_1 + j \cdot n_g) \qquad \mid j \in \mathbb{N}_0; j < \lfloor \frac{n - u_1}{n_g} \rfloor. \quad (6)$$

Since n is prime on a well-chosen curve, with n_g and n_p coprime to n, it is apparent that both Eqs. (5) and (6) can generate any value of the finite field of size n if the upper bound of i and j is n. Nevertheless, with the existence of the upper bound, the existence of a solution is not guaranteed. We still have the option of using different messages and corresponding hash values. Because of the hash function's preimage resistance, we have to brute-force messages until the system of equations has a solution instead of carefully choosing a value for h.

Instead of trying to find a solution for the system of equations above, we will take a look at the likelihood of generating a problem for which a solution exists. Except for the unknown variables s, i, and j, each variable contains a random value we cannot directly choose. Each value for s for which a solution exists is described by exactly one pair of values $\{i, j\}$. The reason for this, again, lies in the primality of the curve order n. As mentioned earlier, i and j could generate the entire group of the finite field of size n if they were unrestricted. With the bound in mind, this means that no value will be generated more than once by the Eqs. (5) and (6). Vice versa, it is easy to see that for every set $\{i, j\}$ that is a solution to the problem, only a single possible value for s exists. Consequently, each solution can be uniquely defined by the set $\{i, j\}$.

By looking at the bounds of i and j, we can calculate the total possible amount m of distinct sets $\{i, j\}$ through

$$m = \lfloor \frac{n - u_2}{n_p} \rfloor \cdot \lfloor \frac{n - u_1}{n_g} \rfloor.$$

Since we cannot freely choose the value of the already defined values in the Eqs. (5) and (6), we can assume the resulting values for s^{-1} to be randomly distributed in the range $[0, n[$. Consequently, $1/n$ gives the likelihood that a random set $\{i, j\}$ is a solution. We can combine this probability with the total number of possible sets $\{i, j\}$ to calculate the probability that there exists at least one solution for any given problem statement:

$$Pr(X) = 1 - (1 - \frac{1}{n})^m = 1 - (1 - \frac{1}{n})^{\lfloor \frac{n - u_2}{n_p} \rfloor \cdot \lfloor \frac{n - u_1}{n_g} \rfloor}.$$

Given the replaceability of the message hash value, we can systematically experiment with various values of h until a solution is found for the problem. The decrease in the runtime complexity of identifying an appropriate message hash h through brute-force is directly correlated with the reduction in the product of n_g and n_p.

Finding small orders n_g and n_p is, however, also not trivial, as it is achieved through a fault injection on the curve parameter a. This limitation arises from the restricted predictability and reproducibility of faults, which limits the number of faults that can be introduced. We will thus grant the attacker powerful fault capabilities by assuming they are able to fault an unlimited number of distinct and random values for the curve parameter a to assess the feasibility of this attack under a strong fault model. The complexity of this attack consists of the two steps of finding points with small group orders and finding a fitting message hash, which stand in direct correlation with each other. The smaller the group orders are, the faster we can find a message hash h that results in a solvable system of equations, and vice versa. To find the optimal amount of time spent on each step, we need to analyze this trade-off.

The runtime complexity to find $n_g \cdot n_p \leq x$ is given by n^2/x when n_g and n_p are repeatedly chosen arbitrarily from the range $[0; n[$. To describe the runtime complexity of finding a fitting message hash that results in a solvable problem, we can use $Pr(X)$. In this case, we are interested in the expected value of the message hashes needed to generate a solvable problem, which is found with $E(X) = Pr(X)^{-1}$. As this expression is not easy to work with, we will approximate it with $E(X) \approx x/n$. We can now calculate the minimum number of steps required to find a solvable system of equations by adding both runtime complexities and finding the global minimum through the derivative:

$$\begin{aligned} C(x) &= \frac{n^2}{x} + \frac{x}{n} \\ C'(x) &= -\frac{n^2}{x^2} + \frac{1}{n} \\ \\ 0 &= -\frac{n^2}{x^2} + \frac{1}{n} \\ x &= n^{1.5}. \end{aligned}$$

Consequently, the fastest way to find a solution is to brute-force the group orders n_g and n_p until $n_g \cdot n_p < n^{1.5}$. The total runtime complexity is then described by $C(n^{1.5})$ where

$$C(n^{1.5}) = \frac{n^2}{n^{1.5}} + \frac{n^{1.5}}{n} = 2\sqrt{n}.$$

Now, even if the remaining problem of finding a solution to Eqs. (5) and (6) is solvable in constant time, this approach already has the same asymptotic runtime as the generic Pollard-Rho method.

A.4 Fault Mbed TLS Signature Verification Return Value

In attack 4.3.2.1, we outlined an attack on the final comparison of an ECDSA signature, for which we describe a proof of concept in this section. It is sufficient to skip a single instruction in Mbed TLS to mistakenly accept any provided signature during verification, as long as r and s are in the expected bound of $\{1, \ldots, n\}$. One possibility of achieving this is to skip the instruction that moves the error constant into the return value of the signature verification function, which signifies an invalid signature. Doing this, the function will return the default return value instead of the error, effectively accepting the signature. Figure 4 shows the source code in Mbed TLS that performs the final comparison of the value r of the signature and the x-coordinate of the witness R.

```c
if (mbedtls_mpi_cmp_mpi(&R.X, r) != 0) {
    ret = MBEDTLS_ERR_ECP_VERIFY_FAILED;
    goto cleanup;
}

cleanup:
```

Figure 4: Last step of signature verification in Mbed TLS. Snippet taken from file `library/ecdsa.c` at version 3.5.2.

A.5 Fault Mbed TLS Point Addition

In this section, a practical attack on the point addition step in ECDSA based on the attack described in attack 4.3.2.3 is shown. Figure 5 shows two shortcuts of the point addition function in Mbed TLS. If either summand point is zero, the other one is immediately returned without performing any arithmetic operations. This extra step may save time, but poses an easy target for a FA allowing an attacker to return either point by skipping a single jump instruction.

```c
/*
 * Trivial cases: P == 0 or Q == 0 (case 1)
 */
if (MPI_ECP_CMP_INT(&P->Z, 0) == 0) {
    return mbedtls_ecp_copy(R, Q);
}

if (MPI_ECP_CMP_INT(&Q->Z, 0) == 0) {
    return mbedtls_ecp_copy(R, P);
}
```

Figure 5: Shortcuts in point addition function in Mbed TLS. Snippet taken from file `library/ecp.c` at version 3.5.2.

Below is an example for both of the attacks based on the faults described above. These examples use the 256-bit curve secp256r1, together with SHA-256, the message *"i swear i know the secret"* encoded in ASCII, and the public key

$$P = (\text{0xbf222c8670af189348a5fe54bf3dcc83dab676bcaee05a6c425590edd8fb637b}, \\ \text{0x5c21de060a7e929d4e764f3f3677d31474ac1c65bda89df2fe6cba4252982497}).$$

When forging a signature for the fault that targets the final addition such that it only returns the summand $[u_1]G$, we start by choosing an arbitrary value for s:

$$s = \text{0x537}.$$

After hashing the message we want to sign, we can calculate $u_1 = h \cdot s^{-1}$ and finally the value of $r = ([u_1]G).x$ giving us:

$$r = \text{0x98406802d2aa1ac6218bb78ac141b03d39973e2674cca9ff6fccfb2f9e708d84}.$$

Forging a signature for the second FA that targets the final addition such that only the summand $[u_2]P$ is returned, we first choose an arbitrary value for u_2:

$$u_2 = \text{0x537}.$$

Now we can calculate

$$\begin{aligned} r =& ([u_2]P).x \\ =& \text{0x8726c5c6ba2870e4301effaadddfc3de13e0cd70e90dab436e781f9369f3a2a5} \end{aligned}$$

and use r and u_2 to calculate the value of

$$\begin{aligned} s =& r^{-1} \cdot u_2 \\ =& \text{0x9e42165b3f224dd337c1384afe715402e053b3b91b0194fe52aefeb9dd7d00f7}. \end{aligned}$$

A.6 Fault Mbed TLS Curve Order

This section describes an attack on the curve order during the verification of an ECDSA signature, as described in attack 4.3.2.2. The same curve, hashing algorithm, message, and public key as used in Appendix A.5 are used for this example. We simulated this FA, by manually reducing the limb count of the curve order *bigint* to 1, ending up with our new curve order

$$n' = 0\text{xfc632551}.$$

First, we arbitrarily select a value for s from $\{1, \ldots, n' - 1\}$:

$$s = 0\text{x1337}.$$

We then verified the signature for all values of r in the range $\{1, \ldots, n' - 1\}$ until the verification succeeded. This task is easily parallelized by splitting the searched range into chunks. Doing this we were able to find a fitting value for r (0x6d7dd) in 5 days on a 64-core AMD EPYC 7763 processor, after checking around 3.1 billion signatures.

A.7 Fault wolfSSL Final Compare

In wolfSSL a fault in the if-condition of the final comparison can be used to accept an invalid signature. The relevant code is shown in Fig. 6.

```
/* comparison of R created to R in sig */
ret = ConstantCompare(rcheck, sig, ED25519_SIG_SIZE/2);
if (ret != 0) {
    ret = SIG_VERIFY_E;
} else {
    /* set the verification status */
    *res = 1;
}
```

Figure 6: Last step of Ed25519 signature verification in wolfSSL. Snippet taken from file `wolfcrypt/src/ed25519.c` at version v5.7.0.

An effective fault for this attack could either prevent loading the return value of the *ConstantCompare* function or target the branch instructions of the if-then-else structure. Whether it is possible to perform this fault, however, depends on the final assembly generated by the compiler.

A.8 Fault wolfSSL Witness Calculation

Figure 7 shows the part of the function in wolfSSL that performs the addition and scalar multiplications in the third step of the verification. The actual calculation is performed by the loop starting on line 9948. Skipping it with a single fault will result in variable r being returned as the result, which has been freshly initialized with 0, the identity element, in line 9442.

A.9 Fault wolfSSL Scalar Multiplication with Hash

Figure 8 shows a snippet of the source code used for verification in wolfSSL. Skipping the function call in line 528 gives us the desired effect of removing the dependency on the hash. With this fault, the witness point will be calculated as $[s]G - A$. Note that the public key point is already stored in the variable A here.

```
ge_p2_0(r); // set r to 0

for (i = 255;i >= 0;--i) {
    if (aslide[i] || bslide[i]) break;
}

for (;i >= 0;--i) { // calculation using double and add
    ge_p2_dbl(t,r);

    if (aslide[i] > 0) {
        ...
```

Figure 7: Snippet of double scalar multiply function in wolfSSL, taken from file `wolfcrypt/src/ge_operations.c` at version 5.7.0.

```
/* find SB */
ed25519_smult(&p, &ed25519_base, sig); // calculate p = [s]G

/* find H(R,A,M) * -A */
ed25519_smult(&A, &A, h); // calculate [h]A and store in A

/* SB + -H(R,A,M)A */
ed25519_add(&A, &p, &A); // calculate p + A and store in A
```

Figure 8: Snippet of low-resource double scalar multiply function in wolfSSL, taken from file `wolfcrypt/src/ge_low_mem.c` at version 5.7.0.

A.10 Fault wolfSSL Hash Calculation

We can exclude the signature point R from the hash in wolfSSL, by faulting a shortcut in the hash update function. Figure 9 shows the SHA-512 update function that immediately returns in line 535 if the length of the buffer is zero. Faulting this check can be used to effectively skip the inclusion of point R in the hash. The signature is forged by calculating the value of the faulted hash $h' = H(P||m)$, choosing a valid value for s, and calculating $R = [s]G - [h']P$.

A.11 Protections against Extended Fault Model

In Section 4.3.1, the FM was extended with faults that can set variables to constants unrelated to their original values or terminate arithmetic operations early. Defending against this FM using only lightweight boundary checks, as done in the previous sections, is not possible. The additional faults included in this model can result in values within the expected range for a variable while being completely unrelated to its original value.

We can find many more similar attacks using this extended FM. For instance, in attack 4.3.3.4, a function call is skipped to prevent the multiplication of the hash with the public key point. A fault targeting the data pointer of the message hash h could be injected to make it point to some non-zero constant in memory. Now that h is not dependent on R anymore, a signature is easily forged by calculating $R = [s]G - [h]P$ using the faulted value of h. Since the faulted h value is not some special value, like 0 or the identity element, it is not possible to detect the fault in the calculation with a simple check.

Additionally, an attacker may also be able to terminate arithmetic operations early instead of completely skipping them. Using the same example, the multiplication $[h]P$

```
int Sha512Update(wc_Sha512* sha512, const byte* data, word32 len)
{
    int ret = 0;
    /* do block size increments */
    byte* local = (byte*)sha512->buffer;

    /* check that internal buffLen is valid */
    if (sha512->buffLen >= WC_SHA512_BLOCK_SIZE)
        return BUFFER_E;

    if (len == 0)
        return 0;
```

Figure 9: Snippet of SHA-512 update function in wolfSSL, taken from file `wolfcrypt/src/sha512.c` at version 5.7.0.

could be terminated early so that the resulting value only depends on a few bits of h. Whether this is possible strongly depends on the implementation and will still require a small brute-force attack to find a solution.

It is thus necessary to calculate some steps of the verification routine redundantly. The results are then checked for equality to confirm the correctness of the calculations. In the same example of the fault on the scalar multiplication, the check is replaced with redundant computation $Q2' = [h]P$ and a check that verifies that $Q2$ is equal to $Q2'$ and aborts the verification process otherwise.

Luckily, the costly double scalar multiplication in ECDSA and EdDSA does not necessarily need to be computed twice to realize this hardening. Instead, it suffices to introduce a new value b that is derived by hashing the signature and incorporating it into the calculation. This technique is similar to blinding; in the context of signature verification, we are, however, only dealing with public values. We can multiply each scalar factor of the double scalar multiplication with b and the resulting witness point Q with the modular inverse of b. This hardening renders any faults in the double scalar multiplication or their factors ineffective while saving one scalar multiplication compared to computing the witness point twice.

All of this is still insufficient to protect against curve-order faults as outlined in attack 4.3.2.2. To prevent this attack, the data layer can be protected with generic methods like ECC memory or data duplication.

The scenarios depicted here require an attacker with a lot of knowledge about the system and resources. The hardenings presented in Sections 5.2.1 and 5.2.2 already significantly improve the fault resistance of the algorithm by covering easily exploited faults that neither require expensive equipment nor huge amounts of resources.