A Fault-Based Secret Key Retrieval Method for ECDSA: Analysis and Countermeasure

ALESSANDRO BARENGHI, Politecnico di Milano
GUIDO M. BERTONI, STMicroelectronics
LUCA BREVEGLIERI, GERARDO PELOSI, and STEFANO SANFILIPPO, Politecnico di Milano
RUGGERO SUSELLA, STMicroelectronics

Elliptic curve cryptosystems proved to be well suited for securing systems with constrained resources like embedded and portable devices. In a fault-based attack, errors are induced during the computation of a cryptographic primitive, and the results are collected to derive information about the secret key safely stored in the device. We introduce a novel attack methodology to recover the secret key employed in implementations of the Elliptic Curve Digital Signature Algorithm. Our attack exploits the information leakage induced when altering the execution of the modular arithmetic operations used in the signature primitive and does not rely on the underlying elliptic curve mathematical structure, thus being applicable to all standardized curves. We provide both a validation of the feasibility of the attack, even employing common off-the-shelf hardware to perform the required computations, and a low-cost countermeasure to counteract it.

CCS Concepts: • **Security and privacy → Cryptography; Security in hardware; Hardware attacks and countermeasures; Side-channel analysis and countermeasures; Embedded systems security;**

Additional Key Words and Phrases: Digital signatures, cryptography, embedded systems security, fault attacks, elliptic curve digital signature algorithm, ECDSA

1. INTRODUCTION

In the last few years, there has been a rapidly growing interest for digital signature frameworks from both public institutions and private enterprises to facilitate the adoption of large-scale IT applications. Digital signature schemes allow the detection of forgery or tampering of transmitted data through providing data integrity, data origin

This manuscript is an improved and extended version of previous conference publications that appeared in Barenghi et al. [2011a, 2011b]. Sections 2 through 4 have been revised and now report all the important mathematical and implementation details. Section 5 provides a new and detailed description of a portable OpenCL implementation of the attack, together with a performance evaluation on up-to-date platforms. Section 6 provides a review of the current state of the art of the attacks and countermeasures against ECDSA, while Section 7 provides a novel fault-secure ECDSA signature generation algorithm.
Authors' addresses: A. Barenghi, L. Breveglieri, G. Pelosi (corresponding author), and S. Sanfilippo, Department of Electronics, Information and Bioengineering (DEIB), Politecnico di Milano, P.zza Leonardo da Vinci, 32–20133 Milano, Italy; email: gerardo.pelosi@polimi.it; G. M. Bertoni and R. Susella, STMicrolectronics s.r.l., Via Camillo Olivetti, 2–220864 Agrate Brianza, Italy.

authentication, and nonrepudiation assurances of previous actions or commitments. Indeed, digital signature schemes represent an essential building block of many cryptographic protocols that provide security services including entity authentication and authenticated key agreement.

Currently, a widely adopted scheme for digital signatures is the Elliptic Curve Digital Signature Algorithm (ECDSA), as standardized in Hall and Keller [2014], NIST [2013], and ANSI [2005]. The same standards certify also the ECDSA key sizes advised by the U.S. National Security Agency in its public cryptographic suite for SECRET- and TOP SECRET-grade security documents [NSA-CSS 2010].

The basis for the security of ECDSA is the intractability of the Elliptic Curve Discrete Logarithm Problem (ECDLP), which appears to be harder than both the discrete logarithm problem in finite fields and the problem of factoring a composite integer [Blake and Seroussi 1999]. Assuming a predetermined security level, the parameters and operands involved in the ECDSA are smaller than the ones employed in other systems, with the important consequence of obtaining resource-saving and low-power consumption implementations while keeping high-security margins.

In principle, the only option for potential attackers should be tackling the underlying mathematically hard problem through an extensive computational effort, without any extra information obtained from either the observation or the manipulation of the inputs/outputs of the device. However, most signature creation and signature verification primitives are implemented on embedded and portable devices, which keep all necessary private information (e.g., keys and certificates) in a nonvolatile storage either to prove their authenticity to other embedded systems or to accept only firmware/software updates from valid issuers. This, in turn, provides a practical way to compromise or recover the private information manipulated through the cryptographic primitive in secure devices, either by means of a passive observation of the device behavior [Kocher et al. 2011; Kocher 1996] or through taking an active stance and disturbing its regular functioning [Joye and Tunstall 2012]. The latter strategy encompasses the so-called *fault attacks*, where the adversary is able to infer information on the secret parameter held securely in the device through comparing the erroneous results with the correct ones (differential fault analysis) or through deductions on the absence of errors even in case of a fault (safe error attacks). There is a conspicuous amount of literature on fault attacks targeting both dedicated hardware implementation of ciphers and software libraries running on general-purpose CPUs, both proving their actual feasibility on real-world targets and proposing effective and efficient countermeasures [Joye and Tunstall 2012]. These attacks are effectively able to undermine the confidence in hardware and firmware support for trusted platforms, especially since it is possible to target digital signature primitives, a key component in providing authenticity guarantees on code and data.

In this work, we present a novel fault attack against ECDSA aimed at recovering the secret key through inducing multibit faults during the computation of the signature generation primitive done by means of either a dedicated hardware implementation or a software library. Our attack relies on the fact that multiprecision multiplications are implemented either in an *operand scanning* or a *product scanning* fashion [Koren 2002]. This is common either in a dedicated ASIC implementation, where the datapath width represents a design parameter, or in software, where the word length of the multiprecision multiplication algorithm is determined by the underlying CPU architecture. The most common hardware implementation for high-speed multiprecision multipliers, the "Coarsely Integrated Operand Scanning" [Walter 1993], relies on an operand scanning methodology. A notable example on the software libraries side is the latest OpenSSL implementation [Cox et al. 2014], conforming to ANSI X9.62 [ANSI 2005], which is used on a wide variety of computing platforms. We consider architecture data-path widths ranging from 8 to 32 bits to cover most current implementations of ECDSA for embedded platforms. To enhance the practicality of our attack, we also consider the product scanning strategy proposed by Comba [1990], where a rescheduling of the single-precision multiplications and storage operations is made to the end of reducing the number of memory accesses.

The attack workflow relies on collecting the erroneous results and recognizing if they are exploitable or if they are to be discarded immediately. We provide a complexity analysis detailing the running time of the secret key retrieval algorithm and the average number of faults required to retrieve the secret key. To provide a concrete evaluation of the attacker resources required, we implemented an optimized version of our key extraction algorithm, employing OpenCL to provide a multiplatform, parallel implementation of our attack, able to tap into the computational power of common off-the-shelf hardware. Finally, we provide a novel, low-computational-cost algorithmic countermeasure able to counteract the described attack, together with guidelines for a fault-attack-resistant implementation of ECDSA and a survey of the current state of the art on fault attacks.

Organization of the article. Section 2 provides the mathematical background on elliptic curve cryptography and the ECDSA algorithm, while Section 3 introduces the fault model assumed by the proposed attack. Section 4 describes the novel secret key retrieval algorithm designed to operate on the faulty outputs of the ECDSA signature generation primitive, of which Section 5 describes the efficient implementation and reports the performance results. Section 6 describes related work through surveying the main results on fault attacks aimed at Elliptic Curve Cryptosystems (ECCs) and the ones at the current state of the art that specifically target the ECDSA primitive. Section 7 proposes our novel countermeasure against the attack and describes a fully protected ECDSA implementation. Finally, Section 8 concludes our work.

2. PRELIMINARIES

Let $\mathbb{F}_q$ denote a finite field with a number of elements equal to $q = p^m$, where $p \geq 2$ is a prime number (called the *characteristic* of the field), and m is a positive integer.

We denote as $\mathbb{E}(\mathbb{F}_q)$ the elliptic curve represented by the set of points $\mathrm{P}(x_\mathrm{P},\ y_\mathrm{P}) \in \mathbb{E}$, $x_\mathrm{P}, y_\mathrm{P} \in \mathbb{F}_q$ satisfying Equation (1), plus the point at infinity $\mathcal{O}$ that represents the directions parallel to the y-axis in the projective plane:

$$y^2 + a_1\,xy + a_3\,y = x^3 + a_2\,x^2 + a_4\,x + a_6 \quad \text{with} \;\; a_1, a_2, a_3, a_4, a_6 \in \mathbb{F}_q. \tag{1}$$

The coefficients in Equation (1) define a smooth absolutely irreducible curve (i.e., without any self-intersections, cusps, or isolated points) in the projective plane with at most $|\mathbb{F}_q|^2$ points plus the unique point at infinity $\mathcal{O}$. Let $\mathrm{P},\ \mathrm{Q}$ be two points of $\mathbb{E}(\mathbb{F}_q)$, and let $\mathrm{R} \in \mathbb{E}(\mathbb{F}_q)$ be the third point of intersection of $\mathbb{E}$ with the straight line joining P and Q (or with the tangent line at P if $\mathrm{P} = \mathrm{Q}$). The point S derived as the third point of intersection between $\mathbb{E}$ and the vertical line joining R and $\mathcal{O}$ is defined to be the outcome of a commutative internal composition law (a.k.a. "secant-&-tangent rule") between P, Q and denoted as $\mathrm{S} = \mathrm{P} + \mathrm{Q}$. The set of points of an elliptic curve with the previous internal composition law constitutes an algebraic *commutative group* $(\mathbb{E}(\mathbb{F}_q),\ +)$, where $\mathcal{O}$ is the neutral element (i.e., $\forall \mathrm{P} \in \mathbb{E}(\mathbb{F}_q)$, $\mathrm{P} + \mathcal{O} = \mathrm{P}$) [Washington 2008].

The group of points of an elliptic curve defined over a finite field is commutative and with a finite number of elements $N = |(\mathbb{E}(\mathbb{F}_q), +)|$; therefore, it also has the algebraic structure of a *cyclic group*, with as many subgroups as the number of prime power factors of N (each of which also includes the point at infinity $\mathcal{O}$ as the neutral element). We denote as G the *generator* of the largest cyclic subgroup $\langle \mathrm{G} \rangle \subseteq (\mathbb{E}(\mathbb{F}_q), +)$ with a prime number of elements and denote as $n = |\langle \mathrm{G} \rangle|$, $n > 1$, its cardinality (a.k.a., *order* of the subgroup or *order* of G). Given an integer k in $\mathbb{Z}_n$ (the set of canonical representatives of residue classes modulo n) and a point $\mathrm{P} \in \langle \mathrm{G} \rangle$, the "scalar-point multiplication" operation is defined as the iterated sum: $\mathrm{P} + \cdots + \mathrm{P} = [k]\mathrm{P}$, with $[k]\mathrm{P} = \mathcal{O}$ if $k = 0$.

In the following, we introduce a lemma about the properties of the coordinates of elliptic curve points, which will be employed to design the attack methodology described in the next sections.

LEMMA 1. *Let* $(\mathbb{E}(\mathbb{F}_q), +)$ *be the group of points of an elliptic curve defined over the finite field* $\mathbb{F}_q$*, with* $q = p^m$*,* $p \geq 2$ *a prime integer, and* $m \geq 1$ *a positive integer. Denote as* G *the generator of the largest cyclic subgroup* $\langle \mathrm{G} \rangle \subseteq (\mathbb{E}(\mathbb{F}_q), +)$*, with prime order* $n = |\langle \mathrm{G} \rangle|$*, and denote as* $\mathrm{P}(x_\mathrm{P}, y_\mathrm{P}) \in \langle \mathrm{G} \rangle$ *a generic elliptic curve point in the chosen subgroup.*

Let $\overline{x_\mathrm{P}}$, $\overline{y_\mathrm{P}}$ *be the values obtained from the bijective mapping of the finite field representation of* x_P, $y_\mathrm{P} \in \mathbb{F}_q$ *into integer numbers, and let* $r = \overline{x_\mathrm{P}} \bmod n$ *be the residue (modulo the order of the subgroup) of the integer number mapped from the* x*-coordinate of* P.

There are at most three points, other than P*, belonging to* $\langle \mathrm{G} \rangle \subseteq \mathbb{E}(\mathbb{F}_q)$ *having the integer representation of their* x*-coordinate in the same equivalence class modulo* n *of* r.

PROOF. Hasse's theorem is a well-known result in elliptic curve theory that bounds the number of points lying on an elliptic curve defined over a finite field. In particular, if $N = |\mathbb{E}(\mathbb{F}_q)|$ denotes the cardinality of the set of points of the chosen curve, it is true that $q + 1 - 2\sqrt{q} \leq N \leq q + 1 + 2\sqrt{q}$ [Washington 2008]. Making use of simple algebraic equivalences, it is easy to see how the previous relation implies $N + 1 - 2\sqrt{N} \leq q \leq N + 1 + 2\sqrt{N}$, and then also $q < 2N$.

Given a point $\mathrm{P}(x_\mathrm{P}, y_\mathrm{P})$ over a generic elliptic curve $\mathbb{E}(\mathbb{F}_q)$, the finite field representation of its coordinates x_P, y_P depends on the characteristic p of the field $\mathbb{F}_q$, $q = p^m$, $p \geq 2$, $m \geq 1$. In case of a prime field (i.e., $p > 2$ and $m = 1$), the coordinates are represented as integer numbers in the range $\{0, \ldots, p - 1\}$. Consequentially, the bijective mapping between the field domain and the integer domain is easily given by the identity map: $x_\mathrm{P} = \overline{x_\mathrm{P}}$, $y_\mathrm{P} = \overline{y_\mathrm{P}}$. The following constraints on the range of values bound to the coordinates of an elliptic curve point over a prime finite field hold: $0 \leq x_\mathrm{P}, y_\mathrm{P} < q < 2N$. In case of a composite field (i.e., $p \geq 2$ and $m > 1$), the most common choice for representing the finite field elements $x_\mathrm{P}, y_\mathrm{P} \in \mathbb{F}_q$ is a *polynomial form* with either a *canonical* basis or a *normal* basis [Lidl and Niederreiter 2008]. The one-to-one correspondence between each polynomial and an integer value maps the binary encoding of each polynomial coefficient as a digit of a multiprecision integer, keeping a correspondence between the leading coefficient of the polynomial and the most significant digit of the integer, the second-leading coefficient of the polynomial with the second most significant digit of the integer, and so on, until the correspondence between the constant term of the polynomial and the least significant digit of the integer is considered [NIST 2013; ANSI 2005]. Denoting as $\overline{x_\mathrm{P}}$, $\overline{y_\mathrm{P}}$ the integer values corresponding to the original coordinates, the relations $0 \leq \overline{x_\mathrm{P}}, \overline{y_\mathrm{P}} < q < 2N$ hold in the same fashion.

Considering a generic point $\mathrm{P} \in \mathbb{E}(\mathbb{F}_q)$ and its opposite, $-\mathrm{P}$, it is worth noting that they belong to the same cyclic subgroup (due to existence in each subgroup of the neutral element, i.e., $\mathrm{P} + (-\mathrm{P}) = \mathcal{O}$) and they also have the same x-coordinate (as Equation (1) describes a curve symmetric with respect to the x-axis).

Given a generic point $\mathrm{P}(x_\mathrm{P}, y_\mathrm{P}) \in \mathbb{E}(\mathbb{F}_q)$, $x_\mathrm{P}, y_\mathrm{P} \in \mathbb{F}_q$, the residue of the integer representation of its x-coordinate ($0 \leq \overline{x_\mathrm{P}} < q < 2N$) modulo N can be expressed (in the range $\{0, \ldots, N - 1\}$) as $\overline{x_\mathrm{P}} \bmod N = \overline{x_\mathrm{P}} - \lambda\, N$, where $\lambda = \lfloor \frac{\overline{x_\mathrm{P}}}{N} \rfloor$, and $\lambda \in \{0, 1\}$. This means that, given P, it may exist at most another point $\mathrm{Q} \neq \pm\mathrm{P}$ over $\mathbb{E}(\mathbb{F}_q)$ with the value of the residue of its x-coordinate modulo N such that $\overline{x_\mathrm{Q}} = \overline{x_\mathrm{P}} + 1 \cdot N$. Thus, there are at most three points different from P (i.e., $-\mathrm{P}$, Q, $-\mathrm{Q}$) on the elliptic curve having

ALGORITHM 1: ECDSA Signature Generation

```
Globals: $\langle G \rangle = (\mathbb{E}(\mathbb{F}_q), +)$, $n = |\langle G \rangle|$, $\mathcal{H}$: hash function
Input: message, msg; secret key, $d \in \mathbb{Z}_n \backslash \{0\}$
Output: signature token, $(r, s)$ with $r, s \in \mathbb{Z}_n \backslash \{0\}$
1 begin
2     repeat
3         $e \leftarrow \mathcal{H}(\texttt{msg}), k \overset{rand}{\leftarrow} \{1, \ldots, n-1\}$        /* $e, k \in \mathbb{Z}_n$ */
4         $r \leftarrow \texttt{x-coord}([k]G) \bmod n$
5         $s \leftarrow (e + r\,d)\,k^{-1} \bmod n$
6     until $r \neq 0$ AND $s \neq 0$
7 end
8 return $(r, s)$
```

the x-coordinate in the same residue class modulo N as the one of P. There can only be two other points if $P = -P$, $Q \neq -Q$ or just another one in case $P = -P$, $Q = -Q$. Since we are considering a point P belonging to the largest prime cyclic subgroup $\langle G \rangle$ of $(\mathbb{E}(\mathbb{F}_q), +)$, and its order is a factor of the order of the curve, that is, $1 < n \leq N$ and $n|N$, the residue class modulo n of the integer representation of the x-coordinate of P, $r = \overline{x_P} \bmod n$, will still include at most three residues derived from points in $\langle G \rangle$ other than r. □

2.1. Digital Signature

Standards on ECDSA [Hall and Keller 2014; NIST 2013; ANSI 2005] provide a list of recommended elliptic curves $\mathbb{E}(\mathbb{F}_q)$, chosen for optimal security and implementation efficiency, each of which has a specified equation, group generator G, group order $n = |\langle G \rangle|$, and finite field structure. Specifically, they report five prime fields $\mathbb{F}_p$ for certain primes p with sizes 192, 224, 256, 384, or 521 bits, and five binary fields $\mathbb{F}_{2^m}$ with m equal to 163, 233, 283, 409, or 571, respectively. For each binary field, one elliptic curve and one Koblitz curve are selected.

The ECDSA specification defines three algorithms for the *key generation*, the *signature generation,* and the *signature verification*, respectively.

The *key generation* algorithm selects a cryptographically strong random integer $d \in \mathbb{Z}_n \backslash \{0\}$ as a private key and computes a public key $(\mathbb{E}(\mathbb{F}_q), G, n, Y)$, where $Y = [d]G$.

The *signature generation* algorithm (see Algorithm 1) takes as input the private key and a message msg, and produces a signature token (r, s), with $r, s \in \mathbb{Z}_n \backslash \{0\}$. From now on, we will denote a value assignment in the algorithms with the $\leftarrow$ operator. The algorithm first obtains a hashed version $e \in \mathbb{Z}_n$ of message msg and a cryptographically strong random number $k \in \mathbb{Z}_n \backslash \{0\}$ that must be different in every run of the primitive (note that a reuse of the same random in multiple signatures allows an attacker to retrieve d solving a set of two simultaneous equations). Subsequently, the scalar-point multiplication $[k]G$ is performed and the x-coordinate of the resulting point (in $\mathbb{F}_q$) is interpreted as an integer prior to reduce it modulo the order of the group n to obtain the first part of the signature token, r (line **4**). The interpretation of the x-coordinate as an integer is done even when the curve is defined over a characteristic two field. In such a case, the standards adopt the convention of simply evaluating the corresponding binary polynomial over the integers [NIST 2013; ANSI 2005], as recalled in the previous section. The second part of the signature token, s, is computed through combining together the hash of the message e, the value r, and the extracted random number k through computing one modular inversion, one modular addition, and two modular multiplications (line **5**). In case either $r = 0$ or $s = 0$, the procedure is rerun with a different k until an admissible signature is obtained.

The *verification algorithm* takes as input the message $\texttt{msg}$, the signature token (r, s), and the public key $(\mathbb{E}(\mathbb{F}_p),\ \mathrm{G},\ n,\ \mathrm{Y}), \mathrm{Y} = [d]\mathrm{G}$. It first verifies that $r, s \in \mathbb{Z}_n \backslash \{0\}$, then computes $u_1 \leftarrow \mathcal{H}(\texttt{msg})\, s^{-1} \bmod n$, $u_2 \leftarrow r\, s^{-1} \bmod n$, and $v \leftarrow \texttt{x-coord}([u_1]\mathrm{G} + [u_2]\mathrm{Y}) \bmod n$ to return a positive validation of the signature token if and only if $v = r$.

2.2. Discrete Logarithm Problem

The mathematical security of the ECDSA signature generation algorithm is based on the hardness of the underlying ECDLP. The complexity of the logarithm problem largely depends on the considered algebraic group structure. Indeed, the best methods to solve the DLP in the multiplicative group of a finite field, $(\mathbb{F}_q^*, \cdot)$, $q = p^m$, $p \geq 2$, $m \geq 1$, are inspired by the so-called "index calculus" algorithm. Namely, the Number Field Sieve (NFS), the Function Field Sieve (FFS), and the NFS in High Degree (NFSHD) are the best-known algorithms for computing a discrete logarithm when the size of the characteristic of the field, p, compared to the size of the multiplicative group, $q - 1$, is large, small, and midsized, respectively [Galbraith 2012]. Informally, all these techniques find a relatively small *factor base* to express most of the group elements as products of elements in the factor base. The group of points on a carefully chosen elliptic curve $\mathbb{E}(\mathbb{F}_q)$ does not have the same "smoothness" of $\mathbb{F}_q^*$; thus, the aforementioned factor base strategy cannot be applied efficiently. In addition, as a further security measure (to avoid the attack in Pohlig and Hellman [1978]), the ECDSA primitives were defined, taking care of performing the computation in the largest cyclic subgroup of the points of a standardized elliptic curve, with the order of the subgroup n having roughly the same size as the order of the curve N (i.e., $1 \leq \frac{N}{n} \leq 4$).

The best algorithms to solve the discrete logarithm problem in a generic finite cyclic subgroup with prime order are the Baby-Step/Giant-Step (BSGS) method [Shanks 1971] and Pollard's methods [van Oorschot and Wiener 1999; Pollard 1974] (a.k.a. Pollard's ρ-algorithm and Pollard's λ-algorithm). Informally, Pollard's algorithms involve computing a random sequence of powers of the logarithm base until two of them have the same value, while the BSGS method precomputes an ordered list of powers and compares the value of another ordered sequence of powers against it to find a match. The spatial complexity of the BSGS method ($O(\sqrt{n})$) makes this technique inconvenient when compared with Pollard's ρ-algorithm, assuming no further information regarding the expected value of the logarithm is available. Specific for the ECDLP, there are two other attack techniques known as MOV [Menezes et al. 1993] and anomalous attacks [Smart 1999], respectively, which are avoided through a careful choice of the elliptic curve order. In the next sections, we will employ a DLP extraction routine to find a logarithm value that ranges in a predetermined interval of values. It is worth noting that this a priori knowledge about the range limits of the discrete logarithm is of no use with Pollard's ρ-method (since its *random walk* among the powers of the logarithm base is uniformly spread over the entire set of group elements). By contrast, Pollard's λ-algorithm can be employed to perform a bounded random walk aimed at computing the (possible) discrete log in the predetermined interval, while the table lookup scheme of a BSGS strategy can be easily tailored to sweep a bounded range of values. For the purposes of the attack described in the next sections, we chose to employ a BSGS strategy, as it profitably allows us to reuse the precomputed table for the computation of two ECDLPs.

Assuming to solve the ECDLP: $\mathrm{Q} = [\delta]\mathrm{B}$, $0 \leq \delta \leq M$, where $\mathrm{Q}, \mathrm{B} \in \mathbb{E}(\mathbb{F}_q)$, $0 < M \ll \sqrt{n}$, $n = |(\mathbb{E}(\mathbb{F}_q), +)|$, the BSGS strategy considers the discrete logarithm δ as

$\delta = a\lceil\sqrt{M}\rceil + b$, $0 \leq a, b \leq \lceil\sqrt{M}\rceil$ and reformulates the problem as:

$$\underbrace{\mathrm{Q} - [b]\,\mathrm{B}}_{\text{Baby}-\text{Step}} = \underbrace{[a]\left(\left[\lceil\sqrt{M}\rceil\right]\mathrm{B}\right)}_{\text{Giant}-\text{Step}}. \quad (2)$$

A list of Baby-Steps is first computed and stored in a hash table. Then, the Giant-Steps are computed for each value $a \in \{0, \ldots, \lceil\sqrt{M}\rceil\}$ and checked against the table of Baby-Steps. If a match occurs, then the logarithm does exist and the values of a, b (and δ) are easily recovered, with an overall cost bounded by $O(\sqrt{M})$ group operations.

2.3. Modular Arithmetic

In an elliptic curve cryptosystem, the modular multiplication operations among the values of the point coordinates account for the majority of the total execution time. Therefore, the performances of any implementation of this scheme heavily depend on the underlying speed of the finite field arithmetic operations.

To achieve an efficient modular multiplication, the ECDSA standards [NIST 2013; ANSI 2005] specify the finite field parameters to be used for each recommended curve. Considering the prime curves (e.g., P$-$192), the finite field characteristic p has a specific binary form allowing a fast modular reduction procedure, that is, $p = p_{t-1}(2^w)^{t-1} \pm \cdots \pm p_0(2^w)^0$, $p_i \in \{0, 1\}$, $0 \leq i \leq t-1$, where t is the number of w-bit processor words composing the multiprecision integer, p. Indeed, the reduction operation is implemented as a few single-precision additions among the words of the input operand. Also, for the binary curves (e.g., B $-$ 163) and the Koblitz curves (e.g., K $-$ 163), the standards report the field parameters to specify the polynomial and normal basis to be employed.

After performing the scalar-point multiplication (line **4** in Algorithm 1), the ECDSA signature primitive performs all the subsequent computations modulo the order of the curve n, which is a generic prime without any particular form. However, there is only a small number of operations to be performed modulo n, namely, two multiplications, one addition, and one inversion (line **5**, Algorithm 1). The field inversion is performed via Euclid's extended algorithm [Knuth 1981], thus avoiding the need to employ the Montgomery representation [Montgomery 1985] to compute it via exponentiation. This, in turn, results in the modular multiplications being done via a common multiprecision multiplication followed by a reduction made via schoolbook division algorithm. The division algorithm is optimized for each specific value of the group order, n, depending on the choice of the elliptic curve recommended by the standard.

The operand scanning method [Koren 2002; Walter 1993] reported in Algorithm 2 is one out of the two common multiprecision multiplication strategies employed in the most adopted software libraries.[1] The algorithm outputs the value of the product from the least significant word to the most significant word, one at each outer iteration via summing the outcomes of equal-order single-precision products, $(\mathtt{hi}, \mathtt{lo})_{2^w}$ (see line **6**), and properly propagating the single-precision carry values. The second most common multiprecision multiplication algorithm implements the so-called Comba's multiplication technique [Koren 2002; Comba 1990] as shown in Algorithm 3.

The Comba's algorithm is a rearrangement of the operations performed by the Operand Scanning Method with the aim to minimize the number of memory operations (namely, the `store` instructions) at the cost of an increased number of registers (i.e., local variables in the pseudo-code). The extent of the performance gains offered by Comba's method depends on the features of the memory subsystem of the target

[1]In the following sections, for the sake of clarity, we will denote a single-precision multiplication between factors with w-bit size as $\times$, as reported in line **6** of Algorithm 2.

ALGORITHM 2: Operand Scanning Multiplication

```
Input: $a = (a_{t-1}, \ldots, a_0)_{2^w}$,
       $b = (b_{t-1}, \ldots, b_0)_{2^w}$
Output: $c = ab = (c_{2t-1}, \ldots, c_0)_{2^w}$
begin
    $(c_{2t-1}, \ldots c_0)_{2^w} \leftarrow (0, \ldots, 0)_{2^w}$
    for $j \leftarrow 0$ to $t-1$ do
        carry ← 0
        for $i \leftarrow 0$ to $t-1$ do
            $(\mathtt{hi}, \mathtt{lo})_{2^w} \leftarrow a_i \times b_j$
            lo ← lo + carry
            hi ← hi + (lo < carry)
            lo ← lo + $c_{i+j}$
            hi ← hi + (lo < $c_{i+j}$)
            $c_{i+j}$ ← lo /* store    */
            carry ← hi
        end
        $c_{j+t}$ ← carry /* store       */
    end
    return $c$
end
```

ALGORITHM 3: Comba's Multiplication

```
    Input: $a = (a_{t-1}, \ldots, a_0)_{2^w}$, $b = (b_{t-1}, \ldots, b_0)_{2^w}$
    Output: $c = ab = (c_{2t-1}, \ldots, c_0)_{2^w}$
 1  begin
 2      $(\mathtt{msd}, \mathtt{ind}, \mathtt{lsd})_{2^w} \leftarrow (0, 0, 0)_{2^w}$
 3      for $i \leftarrow 0$ to $t-1$ do
 4          for $j \leftarrow 0$ to $i$ do
                /* (msd, ind, lsd) ← (msd, ind, lsd) + $a_j \times b_{i-j}$  */
 5              $(\mathtt{hi}, \mathtt{lo})_{2^w} \leftarrow a_j \times b_{i-j}$
 6              lsd ← lsd + lo
 7              carry ← (lsd < lo)
 8              ind ← ind + hi
 9              carry ← carry + (ind < hi)
10              msd ← msd + carry
11          end
12          $c_i$ ← lsd                            /* memory store */
13          (msd, ind, lsd) ← (0, msd, ind)
14      end
15      for $i \leftarrow t$ to $2t-2$ do
16          for $j \leftarrow i-t+1$ to $t-1$ do
                /* (msd, ind, lsd) ← (msd, ind, lsd) + $a_j \times b_{i-j}$  */
17              $(\mathtt{hi}, \mathtt{lo})_{2^w} \leftarrow a_j \times b_{i-j}$
18              lsd ← lsd + lo
19              carry ← (lsd < lo)
20              ind ← ind + hi
21              carry ← carry + (ind < hi)
22              msd ← msd + carry
23          end
24          $c_i$ ← lsd                            /* memory store */
25          (msd, ind, lsd) ← (0, msd, ind)
26      end
27      $c_{2t-1}$ ← lsd                           /* memory store */
28      return $c$
29  end
```

platform (i.e., caches, buses, or RAM chips). However, it is quite common to implement the multiplication between multiprecision integer operands (when the amount of such operations is significant) as the speedup is around 30% with respect to the traditional method, on most computing platforms.

Algorithm 3 consists of two outer loops and two rather simple inner loops that perform the bulk of computation. In each iteration of the inner loop, a multiply-&-accumulate operation is carried out; that is, two w-bit words are multiplied and the $2w$-bit product is added to a cumulative sum with $3w$-bit precision stored in three distinct variables (registers). Algorithm 3 denotes the cumulative sum by the triple $(\mathtt{msd}, \mathtt{ind}, \mathtt{lsd})_{2^w}$, which represents the integer value with radix 2^w: $\mathtt{msd} \cdot 2^{2w} + \mathtt{ind} \cdot 2^w + \mathtt{lsd}$. The operation carried out at lines **12 and 13** and lines **24 and 25** is just a w-bit right-shift of the cumulative sum $(\mathtt{msd}, \mathtt{ind}, \mathtt{lsd})$. The algorithm performs exactly t^2 multiplication operations when the two operands a and b consist of t words. The product $c = a \backslash b$ is obtained one word at a time, starting with the least significant word c_0. The first outer loop (lines **3–14**) calculates the t least significant words of the product c (i.e., the words c_0 to c_{t-1}), while the second outer loop (lines **15–26**) calculates the upper half of the product (i.e., the t words c_t to c_{2t-1}).

3. FAULT MODEL FOR MULTIPRECISION MULTIPLICATIONS

A fault induction technique not spatially precise enough to limit the impact of the alteration in the computation will most likely cause a multiple-bit flip in one of the intermediate values of an operation. Targeting an embedded device, this kind of hazard is commonly attainable through a number of technical means, and even employing low-cost equipments as the ones described in Korak and Höfler [2014], Barenghi et al. [2013], and Schmidt and Herbst [2008], where either clock glitches or power-supply underfeeding are exploited to intentionally inject a fault. Reportedly, single, precise, instruction skips and repetitions, leading to a word-sized fault in the computation, are easily achieved. In particular, by employing timed clock glitching [Korak and Höfler 2014], it is possible to hit exactly a chosen single instruction, with a probability close to 1, while this can only be achieved with a lower probability (which depends on the number of instructions of the code) in case the hazard is not timed [Barenghi et al. 2013]. We consider the effects of faults injected into the ECDSA signature generation primitive (see Algorithm 1) targeting the multiprecision multiplication executed during the computation of the second part, s, of the signature token (see line **5** in Algorithm 1) before the subsequent modular reduction modulo the order of the curve n takes place. The considered faults are modeled as a random change in one of the two single-precision operands employed either during the execution of the operand scanning multiplication strategy, within a single iteration of the nested-loop structure, or in one of the two loop nests in Comba's multiplication algorithm. For the sake of clarity, we will present the definition of faulted multiplication outcome and the related multiplication error only for the operand scanning multiplication algorithm (Algorithm 2). In case Comba's method (Algorithm 3) is employed, the definitions can be adapted through a straightforward change of variables in the indexes of the single-digit operands involved.

Definition 3.1 (*Faulted Multiplication*). Let $a,\ b$ be two multiprecision integers composed by t processor words with w-bit size each: $a = (a_{t-1}, \dots, a_0)_{2^w}$, $b = (b_{t-1}, \dots, b_0)_{2^w}$, and let $c = a\,b = (c_{2t-1}, \dots, c_0)_{2^w}$ be the result of a multiprecision multiplication computed by Algorithm 2. A faulted multiplication is the result of Algorithm 2, when a change is induced in one word of an input factor during a single iteration $(i,\ j)$ of the loop nest structure, with $i, j \in \{0, \dots, t-1\}$, just before the execution of the single-precision multiplication operation (see line **6** in Algorithm 2).

The knowledge of the loop indexes of the nested-loop structure where the fault is injected enables the attacker to deduce a precise characterization of the multiplication error as formalized in the following definition.

Definition 3.2 (*Multiplication Error*). Let $a,\ b$ be two multiprecision integers composed by t processor words with w-bit size each: $a = (a_{t-1}, \dots, a_0)_{2^w}$, $b = (b_{t-1}, \dots, b_0)_{2^w}$, and let $c = a\,b = (c_{2t-1}, \dots, c_0)_{2^w}$ be the result of a multiprecision multiplication computed following Algorithm 2.

A multiplication error is defined as the integer value given by the difference between the faulty ($\tilde{c}$) and faulty-free (c) multiprecision multiplication outcomes:

$$\tilde{c} = c \pm \texttt{MulError}.$$

If the multiprecision multiplication algorithm is faulted during a specific (i,j) loop nest iteration, with $i, j \in \{0, \dots, t-1\}$, the multiplication error is expressed as:

$$\texttt{MulError} = \begin{cases} (\texttt{emf} \times a_i)(2^w)^{i+j}, & \text{when } b_j \text{ is altered} \\ (\texttt{emf} \times b_j)(2^w)^{i+j}, & \text{when } a_i \text{ is altered,} \end{cases}$$

where $\texttt{emf} \in \{1, \dots, 2^w - 1\}$ is an (unknown) erroneous multiplication factor.

In our attack scenario, the ECDSA signature generation routine is considered. In particular, we will refer to the multiprecision multiplication employed to compose the second part of the signature (line **5** in Algorithm 1) combining $r = (r_{t-1}, \dots, r_0)_{2^w}$ with the secret key $d = (d_{t-1}, \dots, d_0)_{2^w}$.

The faulty signature obtained when the operation $r\,d$ is affected by a hazard on an operand of a specific single-precision multiplication can be expressed as the pair $(r, \tilde{s})$, where $\tilde{s} = s \pm \texttt{MulError}\, k^{-1} \bmod n$. In particular:

$$\tilde{s} = s \pm ((\texttt{emf} \times d_i)(2^w)^{i+j})\, k^{-1} \bmod n, \quad i, j \in \{0, \dots, t-1\} \tag{3}$$

$$\tilde{s} = s \pm ((\texttt{emf} \times r_i)(2^w)^{i+j})\, k^{-1} \bmod n, \quad i, j \in \{0, \dots, t-1\}, \tag{4}$$

depending on whether the fault has damaged either r (Equation (3)) or d (Equation (4)).

The faulty results needed for the secret key retrieval process described in the next section are at least t different faulty outcomes of the ECDSA signature generation primitive (i.e., $(r', \tilde{s}'), (r'', \tilde{s}''), \dots$). Each one of the faulty signatures is characterized by a single fault injected in a specific word of the operand $r = (r_{t-1}, \dots, r_0)_{2^w}$ as described by Equation (3). The fault injections should be performed in such a fashion that there is at least a faulty signature for each one of the words of r. From an operational point of view, a low-cost fault injection technique does not allow a faulty signature value to be precisely ascribed to a fault induced on either d_l or r_l, for any $l \in \{0, \dots, t-1\}$. Therefore, the exploitable faulty signature values described by Equation (3) will be distinguished from the ones in Equation (4), observing that the whole value of r is known to the attacker, since it is a portion of the resulting signature value.

We note that in case any word between d_i and r_i in the former equations is equal to zero, the output of the faulty signature generation routine will be correct instead of erroneous, that is, $\tilde{s} = s$. However, neither of the two possibilities poses an issue for recovering the whole secret key d. On one hand, the fact that the value of r randomly changes at each signature generation avoids the possibility of having the ith word of the operand r (i.e., r_i) being always zero. On the other hand, the possible zero values taken by the ith word of the secret key d (i.e., d_i) can be dealt with via initializing the whole guessed secret key to zero and checking, at the retrieval of each of its words, whether the value d, obtained by replacing the zeroes wherever a word of the key has been retrieved, generates a valid public key Y: $\mathrm{Y} = [d]\mathrm{G}$ (see Section 2.1).

4. ATTACK DESCRIPTION

The attack is formulated with an *online* strategy that extracts information about one word d_i of the secret key $d = (d_{t-1}, \dots, d_0)_{2^w}$ at a time and collecting faults until the whole value is revealed. The actual physical collection of the faulty signatures may be easily decoupled from the secret key retrieval procedure, thus reducing the time during which the opponent needs to seize the signing device. For the sake of clarity, the attack will be described as collecting faulty signatures from the same message. Since this hypothesis is not used in the secret key retrieval procedure, it is possible to employ signatures coming from different messages without any penalty. Moreover, the key recovery procedure does not rely on knowing the value of a correct signature of the message m. This is particularly appropriate since the ECDSA signature generation algorithm mandates the use of a random nonce k for every run of the primitive, thus effectively yielding a different signature every time. Note that despite the value of k changing at each signature generation, this has no effect on the proposed attack.

4.1. Secret Key Retrieval Algorithm

Algorithm 4 acts by recovering secret-key-related information through injecting faults during the execution of one of the iterations of the loop nest structures of the multiprecision multiplication operation, $r\,d$, involved in the signature generation process (see line **5** of Algorithm 1). Information related to one word of the secret key is then extracted from the analysis of any faulty signature result, $\tilde{s}$. Through collecting a number of faults related to each word, it is eventually possible to reveal the whole value d. The end is reached when the guessed d correctly yields the known associated public key, that is, when $[d]\mathrm{G} = \mathrm{Y}$. Through injecting a fault during a single-precision multiplication within the nested-loop iteration (i, j) with $i = j = \mathtt{ind}$, $\mathtt{ind} \in \{0, \ldots, t-1\}$, it is possible to obtain a faulty signature result, $\tilde{s}$, carrying information about either the word $d_{\mathtt{ind}}$ or $r_{\mathtt{ind}}$, depending on the actual position where the fault occurred, as explained in Section 3. Since a signature carrying information on $r_{\mathtt{ind}}$ is not useful to the attacker, he or she can perform an aposteriori check to distinguish the two cases and keep the signature $\tilde{s}$ only when it is recognized as carrying useful information to derive the word $d_{\mathtt{ind}}$ of the secret key via Equation (3). Considering a correct signature $s = (e + rd)k^{-1} \bmod n$ (see Algorithm 1) and assuming that a change of value in the word $r_{\mathtt{ind}}$ has been caused, a careful rewriting of Equation (3) allows a "reduced" ECDLP to be formulated.

Indeed, starting from the following equation:

$$((\mathtt{emf} \times d_{\mathtt{ind}})\,(2^w)^{2\,\mathtt{ind}})\,\tilde{s}^{-1} = \pm(k - e\tilde{s}^{-1} - rd\tilde{s}^{-1}) \bmod n, \tag{5}$$

and considering both members as coefficients in a scalar-point multiplication by the curve generator G, we obtain a reduced ECDLP instance: $\delta = \mathrm{dlog}_{\mathrm{B}}\mathrm{Q}$, with a proper definition of both the base point B and the logarithm argument Q:

$$\underbrace{[\mathtt{emf} \times d_{\mathtt{ind}}]}_{\text{discrete log: } \delta}\,\underbrace{[(2^w)^{2\,\mathtt{ind}}\,\tilde{s}^{-1}]\,\mathrm{G}}_{\text{log base: point B}} = \underbrace{\pm(\hat{\mathrm{P}} - [e\tilde{s}^{-1}]\mathrm{G} - [r\tilde{s}^{-1}]\mathrm{Y})}_{\text{discrete log argument: point Q}}, \tag{6}$$

where $\hat{\mathrm{P}}$ is one of the possible curve points having the same x-coordinate of the unknown point $[k]\mathrm{G}$ (see line **4** in Algorithm 1), as described by Lemma 1.

The reduced ECDLPs formulated previously (one for each value of the right-hand side of the relation) can be efficiently solved by employing a BSGS strategy, thanks to the observation that the discrete logarithm value δ is an integer in the range $\{0, \ldots, M-1\}$, with $M = 2^{2w}$, since $M-1$ is the maximum value that a single-precision multiplication may yield (see Section 2).

An optimized implementation of the attack can solve the two ECDLP instances shown by Equation (6) (depending on the sign of Q) through coupling them together. In particular, starting from the basic description of the BSGS in Section 2.2, the reduced ECDLP instance $\delta = \mathrm{dlog}_{\mathrm{B}}(\pm\mathrm{Q}) \Leftrightarrow [\delta]\mathrm{B} = \pm\mathrm{Q}$, $0 \le \delta < 2^{2w}$ can be rewritten as follows:

$$\underbrace{\pm\mathrm{Q} - [b]\,\mathrm{B}}_{\text{Baby-Steps}} = \underbrace{[a]\,([2^w]\,\mathrm{B})}_{\text{Giant-Steps}}, \quad 0 \le a, b \le 2^w, \quad \delta = a\,2^w + b. \tag{7}$$

The computational cost for finding the discrete logarithm value $\delta = a\,2^w + b$ is bounded by the number of elliptic curve operations needed to compute both the Baby-Steps and Giant-Steps for every possible value of the coefficients $a, b \in \{0, \ldots, 2^w\}$ and to check the constraint held by Equation (7). Applying a *double-&-add* strategy for the computation of each scalar-point multiplication at the level of elliptic curve (EC) arithmetics, the average cost of each of them is $O(\frac{3}{2}\,w)$ EC operations. The reuse of the values of the Giant-Steps for checking both values $\pm\mathrm{Q} - [b]\,\mathrm{B}$ allows one to easily derive a computational cost of $O\,(3\,w\,2^w)$ EC operations.

After obtaining a discrete logarithm δ, the computation of $d_{\mathtt{ind}}$ is carried out exploiting multiple faulty signatures, where the fault was injected targeting the same single-precision multiplication in the multiplication algorithm ($i = j = \mathtt{ind}$). Specifically, the derivation of the correct value of $d_{\mathtt{ind}}$ is carried out through a sequence of *greatest common divisor* (GCD) operations among a set of values of the form $\delta = \mathtt{emf} \times d_{\mathtt{ind}}$,

ALGORITHM 4: Secret-Key Retrieval

```
   Globals: $n$: order of the group, $n = \langle G \rangle = |\big(\mathbb{E}(\mathbb{F}_q), +\big)|$;
            $w$: processor word size;
            $t$: number of words to represent $\mathbb{Z}_n$ elements, $t = \lceil \frac{\lceil \lg_2 n \rceil}{w} \rceil$
   Input:   public key, $Y = [d]G \in \mathbb{E}(\mathbb{F}_q)$
   Output:  value of the private key, $d \in \mathbb{Z}_n d = (d_{t-1}, \ldots, d_0)_{2^w}$, $0 \le d_{\mathtt{ind}} \le 2^w - 1$, $0 \le \mathtt{ind} \le$
            $t - 1$
1  begin
2      $d = (d_{t-1}, \ldots, d_0)_{2^w} \leftarrow (0, \ldots, 0)_{2^w}$
3      $\mathtt{msg} \overset{rand}{\leftarrow} \{0, 1\}^*$                             /* random inputmessage */
4      $e \leftarrow \mathcal{H}(\mathtt{msg})$      /* output of the ECDSA hashfunction $\mathcal{H}(\cdot)$: $e \in \mathbb{Z}_n$ */
5      $\mathtt{ind} \leftarrow 0$
6      while $[d]G \neq Y$ do
7          $(r, \tilde{s}) \leftarrow$ FAULTED_SIGN$(\mathtt{msg}, \mathtt{ind})$   /* Faulty sign., $r \equiv_n$ x-coord$([k]G)$, Section 3
           */
8          foreach $\hat{P} \in \{(x, y) \in \langle G \rangle : x \bmod n = r\}$ do
                                        /* Obs: $\hat{P}$ is derived following Lemma 1 */
9              $Q \leftarrow \hat{P} - [e\,\tilde{s}^{-1}]G - [r\,\tilde{s}^{-1}]Y$          /* ref. Equation (6) in Section 4 */
10             $B \leftarrow [(2^w)^{2\,\mathtt{ind}} \tilde{s}^{-1}]G$
11             $\delta \leftarrow$ OPTIMIZED_BSGS$(B, Q)$                  /* $0 \le \delta < 2^{2w}$, $\delta = \mathtt{emf} \times d_{\mathtt{ind}}$ */
12             if $\delta \neq \perp$ AND $r_{\mathtt{ind}} \nmid \delta$ then
13                 $d_{\mathtt{ind}} \leftarrow$ GCD$(\delta, d_{\mathtt{ind}})$   /* GCD between the current guess of $\delta$ and $d_{\mathtt{ind}}$ */
14                 break
15             end
16         end
17         $\mathtt{ind} \leftarrow (\mathtt{ind} + 1) \bmod t$
18     end
19     return $d$
20 end
```

with emf being a random w-bit value. The secret key recovery procedure is detailed in Algorithm 4, which takes as inputs the public key Y and the public parameters of the employed elliptic curve, and outputs the value of the secret key d.

As a first initialization step, the algorithm sets the value of all the words $d_{t-1}, \ldots, d_0$ of the key hypothesis d to zero (line **2**). Subsequently, it draws a random message msg from the acceptable message space (line **3**) and computes its hash e (line **4**). The algorithm will recover every word of the secret key through injecting a fault in the single-precision multiplication between two words indexed by the same value, ind, which will take all the values from 0 to $t - 1$ (line **5**). While the value of the key hypothesis is not correct (line **6**), the algorithm gathers new information about the secret key via obtaining a flawed signature respecting the fault model defined in Section 3. The FAULTED_SIGN primitive (line **7**) takes as inputs the message msg and the index ind to inject a fault in the single-precision multiplication $r_{\mathtt{ind}} \times d_{\mathtt{ind}}$, computed during the execution of the ECDSA signature generation procedure (Algorithm 1) when the multiplication $r\,d$, implemented following either Algorithm 2 or Algorithm 3, occurs. Given a faulty signature, $(r, \tilde{s})$, the attack algorithm seeks the value of the multiplication error that occurred in the computation (lines **8–16**) through evaluating each possible value for the point $\hat{P}$ (line **8**). The first step is to compute the value of both the logarithm argument Q (line **9**) and the base point B (line **10**) from the collected signature and the elliptic curve public parameters. Employing such points, the OPTIMIZED_BSGS subroutine computes the discrete logarithm value δ, which contains the useful information regarding the secret keyword $d_{\mathtt{ind}}$. If the logarithm exists (i.e., $\delta \neq \perp$), to distinguish whether the hazard caused a multiplication error ($\mathtt{MulError} = \delta\,(2^w)^{\mathtt{ind}+\mathtt{ind}}$) with $d_{\mathtt{ind}}$ as a factor or not, it is sufficient to check the divisibility of δ by $r_{\mathtt{ind}}$ (line **12**). Note that $r_{\mathtt{ind}}$ is a known value output by the signature primitive (line **7**). Subsequently, the hypothesis for the $d_{\mathtt{ind}}$ word is updated with the greatest common divisor between the current $d_{\mathtt{ind}}$ value stored in the multiprecision variable d and δ (line **12**). We will demonstrate in the following section that the algorithm is able to recover, with high probability, the actual value of the secret key d, through collecting only three or four exploitable faults for each value of ind (i.e., for each secret keyword). Before checking if the updated hypothesis for the secret key d is correct (see loop condition at line **6**), the procedure increments the index of the targeted single-precision multiplication (line **17**), preparing the state for the retrieval of another secret keyword. When the public key is correctly derived from the current hypothesis of the secret key d (line **6**), the algorithm ends (line **19**).

4.2. Complexity Analysis

The computational cost required to lead the fault attack previously described is formally expressed by the following propositions.

PROPOSITION 4.1 (KEYWORD RECOVERY). *Given a fault injection technique able to fit the fault model described in Section 3 into effect, and given $\eta_{\mathtt{ind}}$ as the number of different faults injected on the w-bit word with index* ind *of the targeted single-precision multiplication performed during the execution of the last step of the ECDSA signature generation primitive (Algorithm 1, line* **5***), the recovery of the correct value of the secret keyword is obtained through Algorithm 4 with a probability of* 99% *and $\eta_{\mathtt{ind}} = 4$ faults.*

PROOF. To obtain the correct value of the secret keyword $d_{\mathtt{ind}}$, Algorithm 4 computes the GCD among different discrete logarithms $\delta = \mathtt{emf} \times d_{\mathtt{ind}} \in \{0, \ldots, (2^w)^2 - 1\}$, thus eliminating the random value emf. The correct value of d_{ind} will be therefore recovered when at least two values of emf are coprime.

A well-known result in number theory [Hardy et al. 2008] asserts that the probability $p_{\mathtt{co}}$ that two positive integers (≥ 2), chosen uniformly at random, are coprime ranges in the interval $[\frac{1}{2}, \frac{6}{\pi^2})$. This, in turn, implies that the probability of obtaining at least one pair of coprime values, after $\eta_{\mathtt{ind}}$ faults have been collected, amounts to $p_{\mathtt{ok}} = 1 - (1 - p_{\mathtt{co}})^{\binom{\eta_{\mathtt{ind}}}{2}}$. Willing to make a conservative assumption, choosing $p_{\mathtt{co}} = \frac{1}{2}$, the original value of $d_{\mathtt{ind}}$ can be obtained with only $\eta_{\mathtt{ind}} = 4$ faults with a probability $p_{\mathtt{ok}} = 0.99$. For numbers greater than 15 (i.e., $w > 4$) the probability that two of them have no common factors quickly increases up to $p_{\mathtt{co}} \geq 0.6$. Thus, keeping $p_{\mathtt{co}} = \frac{6}{\pi^2}$ gives a fault number $\eta_{\mathtt{ind}} = 3$, which is better suited for any realistic architecture word size. $\sqcup$

PROPOSITION 4.2 (COMPLEXITY). *Given a fault injection technique able to correctly put the fault model in Section 3 into effect, the secret key retrieval procedure described in Algorithm 4 recovers the t w-bit words of the private key used in an ECDSA signature primitive employing η different faults for each word and an average computational complexity of $O\,(3\,w\,2^w\,t\,\eta)$ elliptic curve operations.*

PROOF. Each call to the OPTIMIZED_BSGS subroutine (line **10**) has an average case complexity of $O(3\,w\,2^w)$ elliptic curve point operations (see Section 4.1). Indeed, the OPTIMIZED_BSGS subroutine is called as many times as the number of values for the candidate points $\hat{\mathrm{P}}$ having their reduced x-coordinate $x_{\hat{\mathrm{P}}}$ equal to the first part of the faulty signature, $(r, \tilde{s})$ (line **7**), that is, $x_{\hat{\mathrm{P}}} = r \bmod n$. Specifically, it is called four, three, or two times according to Lemma 1. The number of possible values for the x-coordinate ranges in the set of the finite field values $|\mathbb{F}_q| = q$. Hasse's theorem [Washington 2008] bounds the number of points n on an elliptic curve $\mathbb{E}(\mathbb{F}_q)$ as $-2\sqrt{n} \leq q - n - 1 \leq 2\sqrt{n}$. Thus, the number of elliptic curve points with the reminder modulo n of their x-coordinate equal to the abscissa of another point is at most $2\sqrt{n}$. Being an elliptic curve equation symmetric with respect to the x-axis, each x-coordinate satisfying the equation corresponds to either two different points or a single point with double multiplicity. Thus, the probability that more than two points have an x-coordinate equivalent to $r \bmod n$ is less than or equal to $\frac{2\sqrt{n}}{n} = \frac{2}{\sqrt{n}}$. The number of points n on $\mathbb{E}(\mathbb{F}_q)$ recommended in the ECDSA standard [NIST 2013; ANSI 2005] is greater than 2^{163}; thus, the probability to call the Optimized_BSGS routine more than two times is less than 2^{-82}. Therefore, it is safe to consider the number of candidate points to be always 2, and the probability of choosing the correct $\hat{\mathrm{P}}$ at first to be $\frac{1}{2}$. As stated in Proposition 4.1, the algorithm collects η values for each of the t words of the secret key prior to finishing; thus, the computational complexity of the whole procedure is, on average, $O\left(3\, w\, 2^{w}\, t\, \eta\right)$. The complexity of the calls to the gcd subroutine is ignored as these are negligible with respect to a single run of the BSGS. $\square$

5. ATTACK IMPLEMENTATION AND EVALUATION

The secret key retrieval procedure employed in Algorithm 4 was implemented in order to demonstrate the feasibility of the attack on commodity hardware.[2]

We chose to evaluate the computation times for the elliptic curves defined over fields with a large prime characteristic (i.e., the P$-$ series from NIST) as their arithmetic operations are more computationally demanding than the ones over characteristic-2 fields, thus providing an effective upper bound for the computation effort required to break the standard binary curves (K$-$ and B$-$ series). The algorithms described in this section are general and can be applied to any of the ECDSA standard curves, with the only exception of the modular reduction adopted for the chosen finite field, which is specifically optimized for the value of the field characteristic provided by the standard.

As already noted, the calls to the Optimized_BSGS routine account for almost the total computation of the key retrieval procedure. It is thus useful to exploit the significant amount of coarse-grained data parallelism of this routine since the computations of both the Baby- and Giant-Steps can be split into a large amount of independent computations. Moreover, a further fine-grained level of parallelism stems from the multiprecision arithmetic procedures required to perform the scalar-point multiplication.

To the end of gaining an advantage from both forms of parallelism, OpenCL was selected as the development platform, as it both allows one to exploit fine-grained parallelism through the use of native vector types and was designed to be the best-fitting programming model for data parallelism. Additionally, OpenCL allows one to write code portable across a wide range of computing devices, from multicore CPUs to computation accelerators such as Intel Xeon Phi and modern GPUs, in turn allowing the exploitation of common off-the-shelf parallel platforms [Agosta et al. 2010].

Coarse-grained data parallelism. The coarse-grained parallelism of the BSGS is exploited by implementing it as two separate OpenCL kernels: the first one precomputes the Giant-Steps and stores them in a hash map, while the second one takes care of computing the Baby-Steps and of performing table lookups to solve the reduced ECDLP. Kernels in OpenCL provide the programming abstraction required to define

[2]For reproducibility, the source is available at the author's `GitHub` page: https://github.com/esseks/debacl.

a computation, commonly called *work item*, which should be replicated and executed in parallel over different data. The work items can be clustered into *work groups* to enhance the efficiency of the computation exploiting the underlying memory hierarchy of the computing platforms. Since all the work items of the aforementioned kernels are independent, no constraint was imposed on the work group sizes. Consequently, each OpenCL runtime is free to select the most suitable value, given the features of the underlying hardware.

Both kernels are characterized by a total number of work items equal to the optimal number of scalar-point multiplications of the Giant-Step ($2^{w-\lambda}$) and of the Baby-Step (2^{λ}), respectively. We note that, despite the Baby-Step needing to compute an extra point subtraction as per Equation (7) in Section 4, thus providing a small imbalance in the computation done by the two kernels, trying to rebalance the two steps choosing $\lambda = \frac{w}{2} - 1$ actually makes things worse. This is due to the fact that the Baby-Step will effectively shift an amount of computation from the Baby-Step to the Giant-Step, which is actually far larger than the former imbalance. As a consequence, the most efficient choice for λ is still $\lambda = \frac{w}{2}$.

The kernel computing the Giant-Step stores the results in a hash map together with the corresponding scalar multiplier. Points are stored as the affine x-coordinate plus the sign (parity) of the affine y-coordinate. In order to guarantee a good insertion and search performance, the hash table is built with a load factor $\alpha \leq 0.5$, that is, twice as large as the number of points computed by the Giant-Step. Points are hashed through the "DJBX33A" hash function [Torek 1990] (a.k.a., Daniel J. Bernstein, times 33 with addition) applied to the normalized homogeneous representation (with $Z = 1$), interpreted as a string of bytes, and collision resolution is handled in open addressing with linear probing. All the insertions in the hash table are performed before any lookup takes place; hence, no synchronization is needed on lookup operations. Concerning insertions, data is loaded in parallel as soon as a bucket has been marked "in use," while allocation is performed using atomic compare-and-exchange OpenCL built-ins to test the status of a bucket and possibly mark it as used in a single, noninterruptible operation. To this end, we exploit the newly introduced OpenCL 2.0 memory model, namely, `memory_order_acq_rel` (Khronos [2014], Section 3.3.4), to ensure that whenever a work item locks a bucket in the hash map, work items belonging to different work groups see the update. Where OpenCL 2.0 was not available, we had to rely on platform-specific behavior. The hash map is stored in global memory and retained between the execution of the two kernels, avoiding the need to transfer the map from the device to the host, thus overcoming possible bottlenecks due to slow buses, such as PCI for most GPUs.

In contrast with the need for locks of the Giant-Step computation kernel, all the work items composing the baby step computation do not need to be synchronized at all. This is true as the only apparent synchronization point, that is, the write action of the solution of the reduced ECDLP, has only one work item acting since the logarithm is guaranteed to be unique on all ECDSA standardized curves.

Fine-grained arithmetic level parallelism. The fine-grained parallelism in the computation is exploited through a proper use of OpenCL vector types. To this end, the multiple precision arithmetic routines need to act on arrays of digits, which can be conveniently represented via OpenCL native vectors, and thus lowered by the compiler efficiently whenever vector units are available. To this end, the choice of the digit size, also known as limb size, and the number of digits to represent each integer value over $\mathbb{F}_p$ should meet two criteria. The first one is that the limb size *should* match or exceed the word size for which optimized modulo reduction routines are available. In fact, the routines to perform reductions modulo p are tailored to the particular form of the primes chosen by the standards [NIST 2013; Hall and Keller 2014]. These

Table I. Multiprecision Type Size and Composition

Curve	Type Size	Limb Size	No. of Limbs
P−192	192-bit	64-bit	3
P−224 P−256	256-bit	32-bit	8
P−384	512-bit	32-bit	16
P−521	1, 024-bit	64-bit	16

primes, known as generalized Mersenne Primes [Solinas 2011; Brown et al. 2001], can be expressed as a summation of a few powers of two, thus providing the possibility of performing a modular reduction with only a few subtractions instead of a full division algorithm. The value of the powers of two composing the influences the choice of the limb size and suggests it to be at least 32 bits wide for curves P−224, P − 256, and P−384, while 64-bit wide limbs are needed for curve P−192. The modular reduction of curve P−521 is well fit for any limb size greater than or equal to 8 bits. The second criterion restricts the number of limbs to the ones contained in an OpenCL vector type with properly sized cells. The number of limbs thus *must* be equal to the number of components of an OpenCL vector type, as the language does not allow arbitrary-sized vector types: the available sizes are 2, 3, 4, 8, or 16.

Table I reports the choices made to fit the multiple precision integers for each curve into OpenCL native vector types. It is worth noting that fitting the needs of curves P−384 and P−521 requires one to employ significantly larger vector types, as no better fit matching the aforementioned criteria was possible.

Exploiting native vector types, the limb-by-limb operations of multiprecision addition and subtraction, and the respective underflows (overflows, respectively) are computed as vector operations. Subsequent carry and borrow propagations are performed sequentially for each digit: since the carry/borrow propagation loop has a small and fixed number of iterations, the OpenCL compiler was instructed to unroll it, preventing control-flow divergence. Modular addition and subtraction are achieved through computing the nonmodular operation at first and bringing the result back in the $[0, p-1]$ range by respectively subtracting or adding p up to one time.

Multiprecision multiplication is implemented through the schoolbook method, as the ability to perform the digit-number multiplication through vector units allows us to achieve faster performances with respect to the possible gain of applying a Karatsuba strategy [Koren 2002] for the multiplication.

In particular, the schoolbook multiplication algorithm, as reported in Section 2, Algorithm 2, was adapted to profit from OpenCL vector types in the inner loop. The digit-number multiplication is in fact performed in a single cycle employing two vector-typed temporary variables for its result, taking proper care of adding them to the multiprecision multiplication result accumulator at each cycle. Modular multiplication is obtained by reducing the result of a nonmodular operation through the optimized mod p routine specified by ECDSA standards.

The scalar-point multiplication over elliptic curves has been implemented according to the recommendations provided by ECDSA standards (NIST [2010], Section 2.2)) employing the aforementioned multiprecision arithmetic operations. Following the guidelines provided by ECDSA standards, the curve points are represented in Jacobian projective coordinates, that is, as triples (X, Y, Z), from which the corresponding affine coordinates (x, y) can be obtained as $x = \frac{X}{Z^2}$ and $y = \frac{Y}{Z^3}$ [Blake and Seroussi 1999].

Performance evaluation. The described BSGS implementation was run on a common desktop endowed with an AMD A8-6600K APU and GiB of DDR3-1866 DRAM, running Debian GNU/Linux Sid x86_64 and employing the AMD OpenCL runtime

Table II. Total Computation Time for the Full Secret Key Retrieval. For All the Curves Examined, $\eta_{ind} = 3$ Faults Were Considered Enough for Each Limb of the Secret Key Pointed Out in Section 4. The Table Reports the Number of Calls to the OPTIMIZED_BSGS Procedure (Line **11**, Algorithm 4), Together with the Total Computation Time, for Each Curve-Limb Size Pair. All the Computation Times Are Obtained as the Average of 25 Trials, and the Average Sample Mean Square Error Is Reported for Each Column

	8-Bit Limb Size		16-Bit Limb Size		32-Bit Limb Size	
Curve	Number of BSGS Calls	Time [ms] ±5.5%	Number of BSGS Calls	Time [s] ±2.0%	Number of BSGS Calls	Time [min] ±0.2%
P−192	72	57.91	36	0.624	18	2.32
P−224	84	256.3	42	3.487	21	14.4
P−256	96	627.9	48	9.085	24	38.2
P−384	144	3672	72	56.09	36	236.6
P−521	196	1894	98	25.35	49	105.6

version 14.9+ga14.201-2 to perform the computations on the host quad-core CPU. Table II reports the running times of the complete attack considering architecture word sizes of 8, 16, and 32 bits, which are the most common in embedded and low-power devices where ECDSA is used. All the reported figures are obtained as an average of 25 runs of the algorithm for each examined combination of curve and word size, and the corresponding average sample mean square error for each architecture word size is reported. All benchmarks have been run on an otherwise idle machine over randomly chosen inputs, and measurements have been taken using OpenCL profiling events. Random input was generated by reading an appropriate number of bytes from the OS provided random number generator `/dev/urandom`. Note that the average sample mean square error decreases on larger-architecture words, as the variation in the timings for the attack is caused by the setup overhead of the OpenCL framework, and thus becomes negligible with respect to larger ECDLP computations. The peak in memory fingerprint, reached in the case of an attack against the P−521 curve on a 32-bit architecture, is around 64MiB of allocated memory, well within the capabilities of any off-the-shelf available desktop. The reported times show how the attack is feasible on a commercially available CPU, as it requires a negligible amount of computation to complete the attack against an architecture with a processor word size 8 bits and takes less than 1 minute against a 16-bit architecture. The worst-case timing, which is the case of the P−384 curve on a platform with an architecture word size of 32 bits, is of 4 hours of computation, still entirely reasonable. We note that the counterintuitive results represented by the P−384 curve taking more computational effort to be broken with respect to its larger counterpart, the P−521, are to be ascribed to a less efficient modular reduction due to the form of the prime integer selected by the ECDSA standard as a finite field characteristic for the P−384. It is interesting to note that the size of the finite field $\mathbb{F}_p$ has a comparatively small effect on the computational requirements of the attack with respect to a change in the architecture word size. This matches the fact that the computational complexity of this attack grows exponentially in the architecture word size, while it grows only linearly in the bit size of the prime p.

6. RELATED WORK

In this section, we provide a survey of the state of the art about the fault attacks against the implementations of the ECDSA, together with the techniques proposed to counteract them.

The distinctive feature of each attack method is its choice of either the variable or the high-level operation to be faulted. For example, alterations of the elliptic curve coefficients [Ciet and Joye 2005; Biehl et al. 2000] allow one to consider the results of the elliptic curve operations lying on a nonstandard "weak" curve, where the ECDLP can be solved with limited computational effort. Other methods address the possibility of introducing faults in the value of the base point [Fouque et al. 2008], rather than managing the erroneous values of either the random nonce k [Schmidt and Medwed 2009; Naccache et al. 2005; Joye and Yen 2002] or the modulus n [Kara-Ivaniov et al. 2008] of a multiprecision integer multiplication (see Algorithm 1 at line **3** and line **5**, respectively). Different from other works, the secret key recovery technique described in the previous sections points out a novel vulnerability of the ECDSA implementations.

Fault-based attacks against ECDSA implementations. The first fault-based attack against ECDSA is proposed in Biehl et al. [2000], where the authors observe that, in the formulas employed to compute both a point addition and a point doubling over an elliptic curve $\mathbb{E}_1(\mathbb{F}_q) : y^2 + a_1 xy + a_3 y = x^3 + a_2 x^2 + a_4 x + a_6$, the value of the coefficient a_6 is never used. Therefore, the ECDSA algorithm computes a correct result even when the base point G of the group $\langle \mathrm{G} \rangle = \mathbb{E}_1(\mathbb{F}_q)$ is replaced by another point G′ lying on any curve of the form $\mathbb{E}_{\overline{a}_6}(\mathbb{F}_q) : y^2 + a_1 xy + a_3 y = x^3 + a_2 x^2 + a_4 x + \overline{a}_6$, for all values $\overline{a}_6 \in \mathbb{F}_q$. Observing that $\mathbb{E}_{\overline{a}_6}(\mathbb{F}_q)$ is not necessarily a cryptographically strong curve, an adversary can use a point G′ lying on it and solve properly the ECDLP on the (potentially) smaller subgroup $\langle \mathrm{G}' \rangle$ of points over $\mathbb{E}_{\overline{a}_6}(\mathbb{F}_q)$. This method was further extended in Ciet and Joye [2005], where the authors showed that any fault injected in any of the coefficients $a_1, a_2, a_3, a_4 \in \mathbb{F}_q$, or in the field representation parameters (e.g., $q = p^m$, $p \geq 2$, m, or the generating polynomial), effectively allows the retrieval of the nonce k employed in the scalar-point multiplication via a reduced computational effort.

Another attack, directly targeted at the elliptic curve algebraic structure, is described in Fouque et al. [2008], where the authors notice that the state-of-the-art point multiplication algorithm (Montgomery laddering without the y-coordinate [Montgomery 1987; Joye and Yen 2002]) yields correct results for both the elliptic curve it is working on and its *quadratic twist*. The authors pointed out that the quadratic twists of cryptographically strong curves are not necessarily strong curves (this includes also the P−224 standard curve). Since one of every two points having coordinates over $\mathbb{F}_q \times \mathbb{F}_q$ lies on the curve twist, the attack strategy involves altering at random the base point G in input to the scalar-point multiplication algorithm and consequentially checking if the resulting point is on the curve twist.

The aforementioned methodologies effectively reduce the security margin in the computation of the first portion, r, of the output of the ECDSA signature: (r, s), as the ECDLP problem protecting it is in fact formulated on a (possibly) much smaller curve than the one it was intended to. Consequentially, if the attacker is able to know, thanks to a precise fault injection technique, which is the actual value of the point G′ obtained through a fault injection in the value of G, he or she may be able to solve the weaker ECDLP associated to it. Exploiting the newfound value of k, he or she will be able to retrieve the value of the secret signature key by solving for d the signature recombination equation $s = (e + r\,d)\,k^{-1} \bmod n$, $n = |\langle \mathrm{G} \rangle|$.

We note that the open literature provides a number of attacks to the scalar-point multiplication primitive; however, some of them are not applicable to effectively break ECDSA, as they require the repeated execution of the same scalar-point multiplication without changing the value of the scalar, which in turn never happens in standard ECDSA [Fan and Verbauwhede 2012]. By contrast, in case of the deterministic variant of the ECDSA, as specified in the IETF RFC 6979 [Thomas Pornin 2013], where the value of k is derived in a deterministic fashion from a combination of the secret signature key and the input message, it is crucial to take into account also the class of attacks relying on repeated fault induction with a fixed k. These attacks exploit the insertion of repeated faults into the scalar-point multiplication primitive, to the end of damaging the operations concerning the contribution of a single bit of k to the result, and require that k is constant in all the trials. Notable examples of such attacks are the ones relying on errors affecting the control flow of the double-and-add point multiplication [Joye and Yen 2002] or targeting the dummy registers employed to provide resistance against passive side-channel attacks [Yen and Joye 2000]. We will not report the details of such attacks as they do not concern directly the security of ECDSA, and we refer the interested reader to the specific survey in Fan and Verbauwhede [2012].

An alternate ECDSA attack methodology has been proposed in Kara-Ivaniov et al. [2008] and exploits the injection of a fault in the modulus n of the last computations of the signature generation algorithm (Algorithm 1, line **5**). Exploiting the statistical distribution of faulty results, the authors describe a method to obtain a set of candidates for the secret value d. The method recovers the correct secret value d with around 250k faults for a 131-bit-long modulus, and around 600k faults for a 256-bit-long one.

A different strategy to recover the value of d was proposed in Naccache et al. [2005]. The authors exploit a glitch-based fault injection technique to set to zero some of the least significant bytes of the random nonce k for a set of DSA signatures. Once a sufficient number of faulty signatures is collected, it is possible to exploit a *lattice reduction* technique (based on the Hidden Number Problem introduced in Boneh and Venkatesan [1996]) to recover the signature key: the authors report that 27 faulty signatures generated by a nonce value k with the least significant bytes reset to zero are sufficient. The attack is applicable in a straightforward fashion to ECDSA: in Schmidt and Medwed [2009], an instruction skip fault, instead of a memory blanking, is used to recover some bits of the random nonce k, employed by the ECDSA signature generation routine. The authors report that 50 faults are enough to successfully break the security provided by the smallest curve recommended by the ECDSA standard.

State-of-the-art of the countermeasures to ECDSA fault-based attacks. Open literature provides efficient and effective countermeasures for each of the aforementioned attacks. However, it is crucial to take into account the effect of combining different countermeasures, as some of them have been reported to increase the efficiency of an attack while hindering another one. An instance of this is represented by the *ladder algorithm* [Montgomery 1987] to compute the ECDSA scalar-point multiplication in its variant that does not employ the y-coordinate of the elliptic curve points.

This method was reported in Fouque et al. [2008] to allow one to move any point on the ladder onto the quadratic twist of the curve (which may not be cryptographically secure) and subsequently find the nonce value k and the secret key d. Nevertheless, the ladder algorithm variant without the y-coordinate plays also as an effective countermeasure against the so-called *sign change* attacks [Blömer et al. 2006; Fan and Verbauwhede 2012] aimed at flipping the sign of the y-coordinate of an elliptic curve point during the computation of $[k]$G (Algorithm 1, line **4**) to recover the nonce value k and subsequently the secret key d.

In the following, we summarize the set of countermeasures that should be applied to ECDSA in such a way that all the known attacks are thwarted, without any negative effects, and retaining the least possible overhead.

We consider the scalar-point multiplication, $[k]$G, to be performed by means of the Montgomery ladder algorithm [Montgomery 1987]. First of all, we note that the attacks in both Biehl et al. [2000] and Ciet and Joye [2005] (i.e., providing a base point G that is not on the correct curve, or altering the curve coefficients) are effectively thwarted by checking, before the computation of the ECDSA begins, that the curve coefficients are correct and that the point actually lies on the curve.

In order to ensure the correctness of the curve parameters, the solution pointed out by Ciet and Joye [2005] suggests the use of common error correction codes on them, while it is possible to check if the point belongs to the curve through evaluating the curve equation in its coordinates. The aforementioned check on the point coordinates also prevents the attacks that try to move the point on a quadratic twist of the curve, effectively preventing the attack in Fouque et al. [2008]. We note that the use of a Montgomery ladder algorithm to compute the scalar-point multiplication, together with checking at each of its iterations that the ladder invariant holds true, also provides protection against the attacks involving the need for a repetition in the value of *k* [Fan and Verbauwhede 2012], thus protecting also the deterministic version of the ECDSA.

To prevent the partial zeroing of the nonce *k* in the scalar-point multiplication, two strategies should be applied. The first, which generates *k* from the xor-combination of different random values while introducing random delays in the combination, was suggested by Naccache et al. [2005] to prevent memory-based blanking of *k* portions.

The best-known strategy to prevent instruction skipping (i.e., control-flow-based fault induction) with the same effect of blanking a portion of *k* was suggested in Schmidt and Medwed [2009], and involves performing the Montgomery ladder algorithm employing Jacobian coordinates, at the cost of a 30% increase in the execution time of the scalar-point multiplication. Such a choice for the coordinates opens the implementation to the possibility of performing a sign-change attack [Blömer et al. 2006] on the [*k*]G computation. However, we note that for the sign-change attack to be able to extract information on *k*, it is necessary that a significant number of faulty signatures created with the same nonce *k* are collected. This is not applicable for a standard implementation of the ECDSA signature generation primitive as the nonce *k* should change at every run of the algorithm. Vice versa, if a deterministic (nonstandard) ECDSA implementation [Thomas Pornin 2013] is considered, sign-change attacks can be countered by a redundant computation on a smaller curve, which can be added without compromising other countermeasures [Blömer et al. 2006].

7. COUNTERMEASURES FOR MULTIPRECISION MULTIPLICATION FAULTS

In this section, we describe the countermeasure to prevent the novel attack based on multiprecision multiplication faults proposed in Section 4. We point out that the devised countermeasure, besides being effective, has a negligible computational overhead. We also report a description of an ECDSA algorithm with a set of countermeasures to prevent all known fault-based attacks, including ours.

7.1. A Novel Signature Recombination Protection

The idea for securing the ECDSA implementations against attacks based on the injection of faults during the execution of the multiprecision multiplication operations relies on changing the way the nonce k is combined in the last step of the ECDSA primitive (Algorithm 1, line **5**), thus preventing the attacker from inferring the secret key d word-wise. Specifically, the last step of the algorithm should execute the integer multiprecision multiplications shown in the following equation:

$$s = k^{-1} e \ + \ (k^{-2} r)(k d). \tag{8}$$

The parentheses in Equation (8) indicate a strictly enforced precedence in the computation of the signature value s. Clearly, the obtained signature is still correct with respect to the definition of ECDSA; nonetheless, an attacker is not able to perform the kind of analysis shown in Section 4.

To validate this claim, we illustrate the results of two different fault injections and the corresponding analyses, showing that they do not allow the attacker to retrieve the secret key. From now on, we will assume the attack is led against an ECDSA instance based on an elliptic curve $\mathbb{E}(\mathbb{F}_q)$ with base point G (i.e., $\langle \mathrm{G} \rangle = \mathbb{E}(\mathbb{F}_q)$ and correct signature (s, r).

First case. Consider a fault hitting the jth, w-bit-wide word of the factor k during the execution of an iteration (i, j) of the nested loops describing the multiplication $(k\,d)$ (see either Algorithm 2 or Algorithm 3), where $i, j \in \{0, \dots, t-1\}$, and t denotes the number of limbs of each factor. The faulty signature is expressed as follows:

$$\tilde{s} = k^{-1}e \; + \; (k^{-2}r)\,(k\,d \; + \; \varepsilon\, d_i),$$

where ε describes the value responsible for the multiplication error according to the same convention employed by Definition 3.2 in Section 3, that is, $\varepsilon = (\texttt{emf}(2^w)^{i+j})d_i$, with $\texttt{emf}$ being a random single-precision value, while d_i is the word of the secret key targeted by the attacker.

Following the same derivation described in Section 4.1, we can solve the previous equality for k and employ both sides of it as scalar coefficients of a point-scalar multiplication by the base point G:

$$[\varepsilon\, d_i]\,([k^{-1}]([\tilde{s}^{-1}]\mathrm{G})) = [k]\mathrm{G} - [e\,\tilde{s}^{-1}]\mathrm{G} - [\tilde{s}^{-1}r]\mathrm{Y}.$$

From this derivation, it is possible to note how the use of Equation (8) prevents the attacker from computing the first member of the previous equality (refer to Equation (6) in Section 4.1 for comparison). Indeed, the attacker only knows the public parameters and the public key, $\mathrm{Y} = [d]\mathrm{G}$, and thus cannot compute a reduced ECDLP instance as not knowing the value of k^{-1} prevents him or her from setting up a reduced ECDLP instance to compute the value $\varepsilon\, d_i$ as the discrete log of $[k]\mathrm{G} - [e\,\tilde{s}^{-1}]\mathrm{G} - [\tilde{s}^{-1}r]\mathrm{Y}$, with respect to an unknown point: $[k^{-1}]([\tilde{s}^{-1}]\mathrm{G})$.

Since the target of the ECDLP cannot be computed by the attacker, the retrieval of the word d_i of the signature key is effectively prevented by the countermeasure. It is worth noting that trying to guess the value of k^{-1}, which has roughly the same size of the curve order, implies an effort comparable to an effective brute force of the signature key d; thus, the countermeasure effectively preserves the original security margin of the ECDSA scheme.

Second case. Let us suppose that the attacker targets the multiprecision multiplication $(k^{-2}\,r)\,(k\,d)$. Let us assume that a fault hits the jth w-bit word of the factor $(k^{-2}\,r)$ during the execution of an iteration (i, j) of the nested loops followed by the control flow of the aforementioned multiplication, where $i, j \in \{0, \dots, t-1\}$, and t denotes the number of limbs of each factor. The faulty signature is expressed as follows:

$$\tilde{s} = k^{-1}e \; + \; (k^{-2}r)\,(k\,d) \; + \; \varepsilon(k\,d)_i,$$

where ε describes the value responsible for the multiplication error according to the same convention employed by Definition 3.2 in Section 3, that is, $\varepsilon = (\texttt{emf}\,(2^w)^{i+j})\,(k\,d)_i$, with $\texttt{emf}$ a random single-precision value, while $(k\,d)_i$ is the candidate single-precision value for the leakage.

Following the same derivation sketched previously and described in Section 4.1, the attacker is now able to recover the value $\texttt{emf} \times (k\,d)_i$ through the extraction of the discrete logarithm from the following instance of reduced ECDLP:

$$[\texttt{emf} \times (k\,d)_i]([(2^w)^{i+j}\,\tilde{s}^{-1}]\mathrm{G}) = [k]\mathrm{G} - [e\,\tilde{s}^{-1}]\mathrm{G} - [\tilde{s}^{-1}r]\mathrm{Y},$$

as the base-point value can be derived from the faulty signature and the public parameters, $[(2^w)^{i+j}\,\tilde{s}^{-1}]\mathrm{G}$, while the second member of the previous equality can be guessed executing the attack in Algorithm 4.

Nonetheless, even the knowledge of the values $\texttt{emf}_\texttt{i} \times (k\,d)_i$, for all $i \in \{0, \dots, t-1\}$, would not be sufficient to recover the secret key d. This is a consequence of the fact that in the expression of the retrieved discrete log there are two unpredictable factors, $\texttt{emf}$ and k, and that the latter changes at the run of the ECDSA signature generation in an unpredictable way. Therefore, the collection of more faulty signatures, even with faults injected in the same position, would not allow one to exploit the GCD computation technique (Algorithm 4, lines **12–15**) to remove the obfuscation factor from any secret key word d_i. A similar analysis can be performed for the remaining operations of the proposed signature recombination (e.g., $k^{-1}e$ in Equation (8)), showing that it is impossible to recover information about either the key d or the nonce k.

Interactions with other countermeasures. Considering the issue of Differential Power Attack (DPA) vulnerabilities, Hutter et al. [2011] authors report a countermeasure similar to the one we propose for preventing the fault attack introduced in Section 4. In particular, to protect the integer multiplication in the last step of the ECDSA primitive (Algorithm 1, line **5**), Hutter et al. [2011] suggest replacing the computation reported in the standard specification with the following one:

$$s = k^{-1}e \; + \; (k^{-1}r)d. \tag{9}$$

Equation (9) is different from computing the straightforward expression $s = (e + rd)k^{-1}$, as it uses one extra integer multiprecision multiplication to obtain the same value s. The key point of the countermeasure in Hutter et al. [2011] is to avoid the direct combination of d with r, as the knowledge of r (from the first half of the signature primitive output) is the Achilles' heel to execute a successful DPA attack. The countermeasure described by Equation (9), while being particularly inexpensive and actually defending against DPA, does not protect against the fault attack discussed in this work. Indeed, applying the same derivation strategy previously described, it is easy to point out the same vulnerabilities as in the unprotected scheme. By contrast, the signature recombination formula proposed in Equation (8) is able to prevent both the DPA and the fault-based attack described in Section 4, at the cost of two more multiplications w.r.t. the ECDSA standard. However, we point out that our countermeasure, despite being effective against a simple DPA attack (focused on the last ECDSA operation), cannot prevent more sophisticated power analyses based on *Template* strategies [Medwed and Oswald 2009; Herbst and Medwed 2008].

7.2. Fault-Secure ECDSA Algorithm

In the following, we summarize the changes advised for the implementation of an ECDSA signature primitive secure against fault attacks. The protected algorithm employs countermeasures for fault-based attacks known in the current state of the art, as well as the one proposed in this article.

Algorithm 5 reports the way a secure ECDSA is computed and takes the same inputs as the unprotected ECDSA reported in Section 2.

The algorithm starts by zeroing the signature token values (line **2**) and proceeds to check the integrity of the parameters of both the elliptic curve and the underlying finite field, employing an adequate error correction code for each of their uses during the algorithm execution (line **3**). We note that, for the sake of clarity, the checks for the curve parameters' integrity (i.e., a_i, q) are made only at the beginning of the algorithm. However, if the kind of platform and the fault induction technique employed by the attacker allow him or her to induce a fault after the parameters have been loaded, such integrity checks should be repeated before each one of their uses. After checking the integrity of the curve parameters, the base point G is checked to be residing on the chosen elliptic curve $\mathbb{E}(\mathbb{F}_q)$ (line **4**).

In case either the system parameters are corrupt or the coordinates of the base point are incorrect, the device executing the algorithm is supposed to take an appropriate action to counteract the presence of malicious faults, for example, zeroing out the signature key d. If the checks on the curve and base point integrity are passed, the protected

ALGORITHM 5: Fault-secure ECDSA Signature Generation

```
   Globals: elliptic curve, $\mathbb{E}(\mathbb{F}_q)$: $y^2 + a_1 xy + a_3 y = x^3 + a_2 x^2 + a_4 x + a_6$, with
            system parameters: $a_1, a_2, a_3, a_4, a_6 \in \mathbb{F}_q$
            $\langle G \rangle = (\mathbb{E}(\mathbb{F}_q), +)$, $n = |\langle G \rangle|$, $\mathcal{H}$: hash function
   Input: message, msg; secret key, $d \in \mathbb{Z}_n \backslash \{0\}$
   Output: signature token, $(r, s)$ with $r, s \in \mathbb{Z}_n \backslash \{0\}$
1  begin
2      $(r, s) \leftarrow (0, 0)$        /* error correction codes are employed for checking system
                                             parameters whenever at risk of alteration */
3      ok $\leftarrow$ checkParameters($\mathbb{E}(\mathbb{F}_q)$)
4      ok $\leftarrow$ checkPoint(G) $\wedge$ ok        /* ok=true if $G \in \mathbb{E}(\mathbb{F}_q)$; ok=false, otherwise */
5      while ($r = 0$ or $s = 0$) and (ok=true) do
6          $e \leftarrow \mathcal{H}$(msg)            /* output of the ECDSA hash function $\mathcal{H}(\cdot)$: $e \in \mathbb{Z}_n$ */
7          $k \leftarrow$ Rand$(1, n-1) \oplus$ Rand$(1, n-1) \oplus \ldots \oplus$ Rand$(1, n-1)$
                               /* $k \in \mathbb{Z}_n$ is computed as the repeated xor of nonce values */
8          T $\leftarrow$ securedScalarPointMul($k$, G)
           /* scalar-point multiplication $[k]$G is performed with an invariant-
           checked Montgomery ladder with Jacobian coordinates (ref. Section 6)   */
9          ok $\leftarrow$ checkPoint(T)        /* ok=true if $T \in \mathbb{E}(\mathbb{F}_q)$; ok=false, otherwise */
10         $r \leftarrow$ x-coord(T) mod $n$
11         $s_1 \leftarrow k^{-1} e$                                      /* ref. Section 7.1 */
12         $s_2 \leftarrow k^{-2} r$
13         $s_3 \leftarrow k d$
14         $s \leftarrow s_1 + s_2 s_3$ mod $n$
15     end
16     if ok=false then abort
17
18     return $(r, s)$
19 end
```

ECDSA algorithm iterates the following operations until nonzero values of both parts of the signature token (r, s) are computed (line **5**). The first steps are the computation of the input message digest (line **6**) and the generation of the random nonce k. Such generation should be performed drawing several random values from the *random number generator* of the device and combining them via bitwise-xor operations to hinder the possibility of blanking the least significant bits of the final value of k (line **7**) as described in Naccache et al. [2005]. We note that the technical effort to execute such an attack raises with the number of random values employed to compute the nonce k. Subsequently, the signature generation proceeds to compute a secure scalar-point multiplication through the use of a Montgomery ladder on Jacobian coordinates, checking at each step of the ladder whether the algorithmic invariant holds true (see Section 6), to prevent *sign-change* attacks (line **8**). Once the scalar-point multiplication is computed, the coordinates of the resulting elliptic curve point T are checked to be on the correct curve (line **9**), and then the first part of the output value r is computed (lines **10**). Finally, the second part of the signature token is composed employing our countermeasure (see Section 7.1) against attacks based on faulty multiprecision multiplications (lines **11–14**). We note that, in case any error check fails, the algorithm should report as soon as possible the failing behavior and undertake appropriate actions, given the platform it is running on. The proposed algorithm refrains from indicating the calls to the error-handling functions for the sake of description clarity.

Taking into account the computational complexity of the mentioned countermeasures, we note that the most significant overhead is introduced by the secure scalar-point multiplication primitive (SECUREDSCALARPOINTMUL$(\cdot,\cdot)$), which, according to Schmidt and Medwed [2009], increases by 30% the total running time of the signature operation. Besides the overhead for the computation of the error correction codes, the execution of the remaining CHECKPOINT$(\cdot)$ primitives and of the arithmetic operations added to put into effect the protection proposed in this paper (see Equation (8) in Section 7.1) accounts for a total of 11 multiprecision multiplications and 20 multiprecision additions over $\mathbb{F}_q$. We note that the computational overhead, with respect to an unprotected implementation, depends on the structure of the finite field $\mathbb{F}_q$ only in terms of increasing the cost of the multiple-precision additions and multiplications, but does not increase their number. As a consequence, the overhead of our countermeasure takes up less than 1% the overall computation effort of the fault-secure ECDSA implementation, which is of around $6{,}200$ multiprecision multiplications and $2{,}080$ multiprecision additions for the smallest elliptic curve recommended by the standard (i.e., the one where our overhead is most noticeable). We also note that, since our countermeasure uses the same nonce generated by the standard ECDSA algorithm, no extra effort on the platform random number generator is imposed.

8. CONCLUDING REMARKS

In this work, we presented a novel attack effectively exploiting the faulty computation of ECDSA signatures to reveal their secret signature keys. The attack allows the underlying ECDLP to be mapped onto a couple of discrete logarithms defined on a much smaller domain. The experimental campaign reports how it is possible to breach the security of the cryptosystem implemented on CPU architectures with word sizes of 8, 16, and 32 bits. We also provided a survey of the other currently known attacks against the ECDSA cryptosystem, together with their countermeasures, and supplied a negligible-cost one for our proposed attack. We described a protected ECDSA primitive able to ward off all currently known fault-based attacks while retaining its computational overhead around 30% w.r.t. the unprotected primitive.

REFERENCES

Giovanni Agosta, Alessandro Barenghi, Fabrizio De Santis, and Gerardo Pelosi. 2010. Record setting software implementation of DES using CUDA. In *Proceedings of the 7th International Conference on Information Technology: New Generations (ITNG'10),* Shahram Latifi (Ed.). IEEE Computer Society, 748–755. DOI:http://dx.doi.org/10.1109/ITNG.2010.43

ANSI. 2005. Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA). American National Standard: ANS X9.62-2005.

Alessandro Barenghi, Guido Bertoni, Andrea Palomba, and Ruggero Susella. 2011b. A novel fault attack against ECDSA. In *Proceedings of the 2011 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST'11)*. IEEE, 161–166. DOI:http://dx.doi.org/10.1109/HST.2011.5955015

Alessandro Barenghi, Guido Marco Bertoni, Luca Breveglieri, and Gerardo Pelosi. 2013. A fault induction technique based on voltage underfeeding with application to attacks against AES and RSA. *Journal of Systems and Software* 86, 7 (2013), 1864–1878. DOI:http://dx.doi.org/10.1016/j.jss.2013.02.021

Alessandro Barenghi, Guido Marco Bertoni, Luca Breveglieri, Gerardo Pelosi, and Andrea Palomba. 2011a. Fault attack to the elliptic curve digital signature algorithm with multiple bit faults. In *Proceedings of the 4th International Conference on Security of Information and Networks (SIN'11)*, Mehmet A. Orgun et al. (Ed.). ACM, 63–72. DOI:http://dx.doi.org/10.1145/2070425.2070438

Guido Bertoni, Luca Breveglieri, Liqun Chen, Pasqualina Fragneto, Keith A. Harrison, and Gerardo Pelosi. 2008. A pairing SW implementation for smart-cards. *Journal of Systems and Software* 81, 7 (2008), 1240–1247. DOI:http://dx.doi.org/10.1016/j.jss.2007.09.022

Ingrid Biehl, Bernd Meyer, and Volker Müller. 2000. Differential fault attacks on elliptic curve cryptosystems. In *Proceedings of CRYPTO*, 131–146.

Ian F. Blake and Gadiel Seroussi. 1999. *Elliptic Curves in Cryptography*. Cambridge University Press.

Johannes Blömer, Martin Otto, and Jean-Pierre Seifert. 2006. Sign change fault attacks on elliptic curve cryptosystems. In *Proceedings of FDTC'06 (LNCS)*, Vol. 4236. Springer, 36–52. DOI:http://dx.doi.org/10.1007/11889700_4

Dan Boneh and Ramarathnam Venkatesan. 1996. Hardness of computing the most significant bits of secret keys in Diffie-Hellman and related schemes. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*. 129–142. DOI:http://dx.doi.org/10.1007/3-540-68697-5_11

Michael Brown, Darrel Hankerson, Julio López, and Alfred Menezes. 2001. Software implementation of the NIST elliptic curves over prime fields. In *Proceedings of Topics in Cryptology (CT-RSA'01), (LNCS)*, David Naccache (Ed.), Vol. 2020. Springer, 250–265. DOI:http://dx.doi.org/10.1007/3-540-45353-9_19

Mathieu Ciet and Marc Joye. 2005. Elliptic curve cryptosystems in the presence of permanent and transient faults. *Design Codes Cryptography* 36, 1 (2005), 33–43. DOI:http://dx.doi.org/10.1007/s10623-003-1160-8

Paul G. Comba. 1990. Exponentiation cryptosystems on the IBM PC. *IBM Systems Journal* 29, 4 (1990), 526–538.

Mark J. Cox et al. 2014. The OpenSSL Project, ver.1.0.1j. Retrieved from http://www.openssl.org/.

Junfeng Fan and Ingrid Verbauwhede. 2012. An updated survey on secure ECC implementations: Attacks, countermeasures and cost. In *Cryptography and Security: From Theory to Applications (LNCS)*, David Naccache (Ed.), Vol. 6805. Springer, 265–282. DOI:http://dx.doi.org/10.1007/978-3-642-28368-0_18

Pierre-Alain Fouque, Reynald Lercier, Denis Réal, and Frédéric Valette. 2008. Fault attack on elliptic curve montgomery ladder implementation. In *Proceedings of FDTC'08*. IEEE CS, 92–98. DOI:http://dx.doi.org/10.1109/FDTC.2008.15

Steven D. Galbraith. 2012. *Mathematics of Public Key Cryptography*. Cambridge University Press.

Timothy A. Hall and Sharon S. Keller. 2014. The FIPS 186-4 Elliptic Curve Digital Signature Algorithm Validation System. NIST. Retrieved from http://csrc.nist.gov/groups/STM/cavp/documents/dss2/ecdsa2vs.pdf.

Godfrey H. Hardy, Edward M. Wright, and Andrew Wiles. 2008. *An Introduction to the Theory of Numbers* (6th ed.). Oxford Mathematics Press.

Christoph Herbst and Marcel Medwed. 2008. Using templates to attack masked montgomery ladder implementations of modular exponentiation. In *Proceedings of WISA'08 (LNCS)*, Vol. 5379. Springer, 1–13. DOI:http://dx.doi.org/10.1007/978-3-642-00306-6_1

Michael Hutter, Martin Feldhofer, and Johannes Wolkerstorfer. 2011. A cryptographic processor for low-resource devices: Canning ECDSA and AES like sardines. In *Proceedings of WISTP'11 (LNCS)*, Vol. 6633. Springer, 144–159. DOI:http://dx.doi.org/10.1007/978-3-642-21040-2_10

Marc Joye and Michael Tunstall (Eds.). 2012. *Fault Analysis in Cryptography*. Springer. DOI:http://dx.doi.org/10.1007/978-3-642-29656-7

Marc Joye and Sung-Ming Yen. 2002. The montgomery powering ladder. In *Proceedings of the 4th International Workshop on Cryptographic Hardware and Embedded Systems (CHES'02), Revised Papers (LNCS)*, Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar (Eds.), Vol. 2523. Springer, 291–302. DOI:http://dx.doi.org/10.1007/3-540-36400-5_22

Michael Kara-Ivaniov, Eran Iceland, and Aviad Kipnis. 2008. Attacks on authentication and signature schemes involving corruption of public key (modulus). In *Proceedings of FDTC'08*. IEEE CS, 108–115. DOI:http://dx.doi.org/10.1109/FDTC.2008.20

Khronos. 2014. The OpenCL Specification, Version: 2.0, Document Revision: 22. Retrieved from https://www.khronos.org/registry/cl/specs/opencl-2.0.pdf.

Donald E. Knuth. 1981. *The Art of Computer Programming: Seminumerical Algorithms*. Addison-Wesley.

Neal Koblitz (Ed.). 1996. *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO'96) (LNCS*), Vol. 1109. Springer.

Paul C. Kocher. 1996. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology.* 104–113. DOI:http://dx.doi.org/10.1007/3-540-68697-5_9

Paul C. Kocher, Joshua Jaffe, Benjamin Jun, and Pankaj Rohatgi. 2011. Introduction to differential power analysis. *Journal of Cryptographic Engineering* 1, 1 (2011), 5–27. DOI:http://dx.doi.org/10.1007/s13389-011-0006-y

Thomas Korak and Michael Höfler. 2014. On the effects of clock and power supply tampering on two microcontroller platforms. In *Proceedings of the 11th International Workshop Fault Diagnosis and Tolerance in Cryptography, (FDTC'14).* Luca Breveglieri et. al. (Ed.). IEEE CS, 36–52.

Israel Koren. 2002. *Computer Arithmetic Algorithms*. A. K. Peters.

Rudolf Lidl and Harald Niederreiter. 2008. *Finite Fields*. Cambridge University Press.

Marcel Medwed and Elisabeth Oswald. 2009. Template attacks on ECDSA. In *Proceedings of WISA'09 (LNCS)*, Vol. 5379. Springer, 14–27. DOI:http://dx.doi.org/10.1007/978-3-642-00306-6_2

Alfred Menezes, Tatsuaki Okamoto, and Scott A. Vanstone. 1993. Reducing elliptic curve logarithms to logarithms in a finite field. *IEEE Transactions on Information Theory* 39, 5 (1993), 1639–1646. DOI:http://dx.doi.org/10.1109/18.259647

Peter L. Montgomery. 1985. Modular multiplication w/o trial division. *Mathematics of Computation* 44 (1985), 519–521.

Peter L. Montgomery. 1987. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation* 48, 177 (1987), 243–264. DOI:http://dx.doi.org/10.2307/2007888

David Naccache, Phong Nguyen, Michael Tunstall, and Claire Whelan. 2005. Experimenting with faults, lattices and the DSA. In *Public Key Cryptography (PKC'05)*, Serge Vaudenay (Ed.). LNCS, Vol. 3386. Springer, 16–28. http://dx.doi.org/10.1007/978-3-540-30580-4_3

NIST. 2010. Mathematical Routines for the NIST Prime Elliptic Curves. Retrieved from https://www.nsa.gov/ia/_files/nist-routines.pdf.

NIST. 2013. Digital Signature Standard (DSS). Federal Information Processing Standards Publication (FIPS) 186-4 - National Institute of Standards and Technology (NIST) - U.S. Department of Commerce. http://dx.doi.org/10.6028/NIST.FIPS.186-4. (2013).

NSA-CSS. 2010. Suite B Implementers' Guide to FIPS 186-3 (ECDSA). National Security Agency/Central Security Service (NSA/CSS). Retrieved from http://www.nsa.gov/ia/_files/ecdsa.pdf.

Stephen C. Pohlig and Martin E. Hellman. 1978. An improved algorithm for computing logarithms over GF(p) and its cryptographic significance (corresp.). *IEEE Transactions on Information Theory* 24, 1 (1978), 106–110. DOI:http://dx.doi.org/10.1109/TIT.1978.1055817

John M. Pollard. 1974. Theorems on factorization and primality testing. *Proc. of the Cambridge Philosophical Society* 76 (1974), 521–528.

Jörn-Marc Schmidt and Christoph Herbst. 2008. A practical fault attack on square and multiply. In *Proceedings of FDTC'08*. IEEE CS, 53–58. DOI:http://dx.doi.org/10.1109/FDTC.2008.10

Jörn-Marc Schmidt and Marcel Medwed. 2009. A fault attack on ECDSA. In *Proceedings of FDTC'09*. 93–99. DOI:http://dx.doi.org/10.1109/FDTC.2009.38

Donald Shanks. 1971. Class number, a theory of factorization and genera. *Proceedings of Symposia on Pure Mathematics, American Mathematical Society* 20 (1971), 415–440.

Nigel P. Smart. 1999. The discrete logarithm problem on elliptic curves of trace one. *Journal of Cryptology* 12 (1999), 193–196.

Jerome A. Solinas. 2011. Generalized Mersenne prime. In *Encyclopedia of Cryptography and Security* (2nd Ed.), Henk C. A. van Tilborg and Sushil Jajodia (Eds.). Springer, 509–510. DOI:http://dx.doi.org/10.1007/978-1-4419-5906-5_32

Thomas Pornin. 2013. Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA). IETF RFC 6979. Retrieved from http://tools.ietf.org/html/rfc6979.

Chris Torek. 1990. Hash Function for Text in C. Usenetmessage < 27038mimsy.umd.edu> in *comp.lang.c*. (Oct. 1990).

Paul C. van Oorschot and Michael J. Wiener. 1999. Parallel collision search with cryptanalytic applications. *Journal of Cryptology* 12, 1 (1999), 1–28. DOI:http://dx.doi.org/10.1007/PL00003816

Colin D. Walter. 1993. Systolic modular multiplication. *IEEE Transactions on Computers* 42, 3 (1993), 376–378.

Lawrence C. Washington. 2008. *Elliptic Curves: Number Theory and Cryptography, Second Edition* (2nd ed.). Chapman & Hall/CRC.

Sung-Ming Yen and Marc Joye. 2000. Checking before output may not be enough against fault-based cryptanalysis. *IEEE Transactions on Computers* 49, 9 (2000), 967–970. DOI:http://dx.doi.org/10.1109/12.869328