POLITECNICO DI MILANO

Facoltà di Ingegneria dell'Informazione
Corso di Laurea Specialistica in Ingegneria Informatica
Anno Accademico 2009-2010

A Novel Differential Fault Attack to Digital Signature Algorithm on Elliptic Curves: Theory, Simulation and Experimentation Aspects

Relatore: Prof. Luca BREVEGLIERI
Correlatore: Ing. Guido BERTONI
Ing. Alessandro BARENGHI

Andrea PALOMBA – 707731

A Novel Differential Fault Attack to Digital Signature Algorithm on Elliptic Curves: Theory, Simulation and Experimentation Aspects

Andrea Palomba

22 October 2010

This thesis has been developed at the *Advanced System Technology* laboratory of ST Microelectronics, Agrate Brianza.

Contents

List of Algorithms

List of Figures

Acknowledgements

Thanks to Guido sensei and prof. Breveglieri, for being valuable guides and for their share in driving my life towards the direction it has now.

Thanks to Alessandro, for his irreplaceable help during the development of this work.

Thanks to all my friends and colleagues, for playing the main role in making my experience at ST one of the most joyful of my life: Amit, Beatrice, Diego, Emanuele, Fabrizio, Filippo, Gabriela, Licio, Lilli, Massimo, Mauro, Mirco, Nicola, Ruggero, Tommaso.

Thanks to all my friends from present, past and everywhere, for having shared some piece of their lives with me, thus actually building mine: Alessandro B. B., Alessio, Alex, Andrea S., Anna, Annalisa P., Annalisa R., Arietta, Chiara, Claudia C., Daniele, Dario, Davide Q., Eros, Federica, Francesca Ga., Francesca Gh., Francesco F., Francesco R., Fulvio, Ilaria, Irina, Laura A., Laura G., Laura P., Lucia, Manuel, Mara, Marco, Mario, Matteo C., Matteo D., Matteo L., Mattia M., Mattia V., Michele, Niccolò Z., Nicolò B., Paolo G., Roberta, Sara, Silvia, Stefania, Stefano A., Stefano F., Stefano G., Stefano P., Vittorio, and all those I may have forgotten.

Thanks to my skarbuś, for appearing like a lightning to fill my heart and take my breath away.

Thanks to my parents, for having always supported and encouraged me to follow my wishes and for having given me a chance to study what I love.

A Costanza, Ewa e Salvatore.

Chapter 1

Introduction

Cryptography is the discipline of securing a communication by employing different kinds of strategies to protect against undesired behaviors. Nowadays, it is present ubiquitously in everyday life, being used in personal computers, smartphones, set-top boxes and a number of other devices. In a digital communication scenario with many interconnected devices, cryptography is used to prevent unauthorized users to access confidential information and to provide secure authentication of communicating users.

On the other side, someone may want to break the security of a communication system, typically by accessing confidential information or gaining authorizations he was not supposed to have. This is commonly called *attacking* a system.

The devices willing to communicate securely through cryptographic means have to run specific algorithms aimed at providing the necessary security constraints. For simplicity, we can consider these algorithms as accepting two inputs, called the *plaintext* and the *key*, and producing an output called the *ciphertext*. The key is the secret element that makes it possible to compute the other data and therefore it is the main target of an attacker. Originally, an attacker was supposed to consider a possible access to plaintext and ciphertext and to proceed by discovering weaknesses in the underlying mathematical structure to relate the data in his possession and the key, thus potentially discovering it.

In the present times, with the great improvement of cryptographic algorithms and the ubiquitous presence of cryptographic devices, a new possibility has to be taken into consideration. The attacker can indeed consider, if in possession of a target device, different outputs and inputs. Namely, he can monitor some physical quantity as an additional output and he can manipulate the physical interface of the device as an additional input for his purposes. This way it becomes possible to analyze and exploit the internals of the algorithms, and the spectrum of information that the attacker can use to relate with the key greatly extends. Because of the nature of this kind of attacks, they are referred to as *side channel attacks*.

The commonly measured physical quantities are the *power consumption* and the *electromagnetic radiation* of the target device. Attacks that use this kind of information are called *passive attacks*, because the attacker only monitors the device. If, conversely, the attack comprises physically interfering with the

computations, we speak of an *active attack*. Common ways to actively attacking a device include, as an example, forcing variations on the power supply line causing clock glitches or other kinds of errors, or shooting with a laser on specific zones of the chip. Active attacks are commonly called also *fault attacks* because the computations done by the device are subject to errors due to the effects of the attacking procedure.

Cryptographic algorithms vary among different kinds of applications. One of these is known as *digital signature*. Digital signature is a way to authenticate the source of digital data to allow a secure verification that the entity on one end of the communication channel is really who he pretends to be. A widespread algorithm for signatures is DSA (Digital Signature Algorithm), which is a standard by NIST [NIS09]. There exists a variant of it as well, called ECDSA (Elliptic Curve Digital Signature Algorithm), which is based on the same principles while using a different algebraic structure to perform computations, namely the elliptic curves.

In this thesis we present a new method to develop a fault attack against the mentioned ECDSA algorithm. Our attack belongs to the aforementioned family of fault attacks. Some manipulation of the device under attack has to be performed in order to cause it to produce wrong results, which will be subsequently analyzed to extract leaked secret information.

The considered fault model is one of the most commonly studied in the literature, namely the *bit flip*. An example of this kind of fault can be found in [BBPP09]. We suppose that the device uses some wrong value during its computations, differing from the correct one in a single bit. The error appearing as a changing flip then propagates through subsequent computations and finally displays as an invalid signature. Collecting a number of this kind of faulty signatures allows to analyze them by means of our attacking procedures, and eventually discovering the sought secret key.

The proposed procedures are an evolution of some ideas taken from known attacks. Nonetheless, they are a novelty in the particular way they are developed and applied. The major distinguishing points are the position in the computations considered for the fault, and the main search procedure used to analyze the results.

About the position of the fault, we developed an attack to the final operation performed during the signature computation. This operation is usually considered less interesting than the rest of the algorithm, and therefore also leaved unprotected. Our attack shows how it is possible to exploit this kind of vulnerability.

The employed analysis procedure then allows to reveal a-posteriori whether or not the error introduced is exploitable through the attack. This is important since the fault injection procedure is usually unable to induce the error in such a precise way. Our procedure exploits a special version of the *elliptic curve discrete logarithm problem*, which we will discuss in deeper detail in subsequent sections.

The attack is discussed with great detail in its theoretical aspects. Then, simulations of the attack are described as well, to show its practical feasibility and performance. Some experimentation is also performed, to the extent permitted by a very cheap equipment.

The subsequent chapters are structured as follows. In chapter 2 we present an overall description of the considered cryptosystem and the commonly employed

algorithms. Chapter 3 is dedicated to a deeper presentation of side channel attacks and fault injection techniques. In chapter 4 we describe the known scientific literature about fault attacks against elliptic curve based cryptosystems, to better frame our work. In chapter 5 the new proposed attack is detailed and all possible issues are explained and analyzed. We also present some ways to extend the attack ideas to different, more general, fault models. Chapter 6 describes some performance evaluations and the results that were obtained applying the attack in both simulated and physical environments. In chapter 7 a very cheap countermeasure to protect against the attack is proposed and analyzed. Finally, in chapter 8 we draw conclusions and sketch the guidelines for a future expansion of the work.

Chapter 2

Elliptic Curve Cryptography

We now introduce the basics of cryptography as it is studied today. Especially, we focus on the mathematical structures and algorithms that will be considered in following.

At first, we give a brief overview of the different purposes of cryptography, the methods employed to achieve them, and the evolution lines of the algorithms followed in the past times. We then start exploring in detail the mathematics behind a relatively recent family of cryptographic functions, namely *elliptic curves*. In doing this we will introduce the structure, describe it in a precise mathematical way and present the reasons why it is considered a suitable basis for the development of cryptographic primitives, namely the *elliptic curve discrete logarithm problem*. A resume of some elliptic curve based algorithms is also presented, together with a complete analysis of the *ECDSA* algorithm. At last, we will present also the most used algorithms for a kind of *arithmetic operations* that will be the main object of the subequent work.

2.1 Introduction to Modern Cryptography

The purpose of cryptography is to provide strong security properties to means of communication. We can think of two main classes of security problems the communications have to deal with, and for which cryptography offers a valuable help. Such problems are *confidentiality* and *authentication*.

When referring to confidentiality we are thinking of some parties who want to communicate a message that should not be read or interpreted by external entities. The important point is to deny the ability to *read* or *interpret* the given messsage.

A different problem is represented by authentication. We do not mind who can access a content, instead we are interested in the ablity to *generate* a specific message. It is possible to perform authentication if we know that only a given subject can produce some information. An example from the non-digital world is represented by signatures. The signature authenticates the subject writing it, since no one else is (hopefully) able to generate the same strokes. Said differently, authentication opposes the falsification of information.

Over the centuries, a number of algorithms have been developed to meet the cited security needs. The main reason to develop a new algorithm being a successful attempt to break an older one, i.e. to access confidential information or to forge authentication messages.

A cardinal concept has however remained constant in the history of cryptography. A *key* is the object that allows to perform the secure operations. The various algorithms are developed in order to allow the owner of a key to do secure operations, while everyone else cannot do the same.

Let's introduce some useful and wide used terminology. We already saw the *key*, which is a value needed to be able to compute cryptographic functions. The operation of making a message unreadable is called *encrypting* or *enciphering*, while the converse is *decrypting* or *deciphering*. When dealing with authentication, the message created by the involved subject is called, for the analogy seen above, *digital signature*. The operation performed by the other party is called *signature verification*, and consists in verifying wether a given signature was generated by the known subject rather than someone else.

When considering the security of a system a number of properties have to be taken into account. We give an informal description of the properties which should be satisfied by a system to be considered secure.

Confidentiality An operation provides confidentiality if only a set of authorized subjects can access the considered informations.

Integrity Information enjoys integrity if it is not possible to manipulate it without explicit authorization.

Authenticity It is related to identification, an authenticated message is guaranteed to have been generated by a clearly identified subject.

Non-Repudiation It should not be possible to deny a previosly granted authorization.

Traditional encryption algorithms have recently also been called *symmetric key* or, less frequently, *private key* cryptography. This is because such algorithms are designed to allow the use of only one key, which has to be kept *secret* in order to be useful. The key is used for example to encrypt a message as well as to decrypt it, or to sign a message and to verify the signature. This has been practical for a long time, but today's digital communication infrastructure needs a more flexible tool. Actually a number of issues arise from the use of symmetric key protocols. We sketch the most important ones:

- A different key is needed for each communication channel. Two subjects need one key, but if n subjects are involved, they all need a key shared with everyone else, thus making the number of keys explode, as $n(n-1)/2$ keys must exist.
- An owner of the key is able to do every computation allowed in the cryptosystem, i.e. both encryption and decryption, or the verification of a signature as well as its generation. This imposes problems for example on non-repudiation properties.
- The key must be exchanged by the parties before they are able to use it, meaning the entities have to meet in a secure way to guarantee they can securely communicate in the future.

The last of these is particularly famous and known as the *key distribution problem*. Indeed, this has been the main reason for developing a new kind of cryptography.

When considering a world-wide system, with constant addition of new users, it is clear that the mentioned problems are really restrictive. Fortunately, a totally new species of algorithms has been invented to bypass all those issues. The first of these algorithms is the famous RSA, proposed by [RSA78]. The new style is known as *public key* cryptography, as it proposes the use of different keys, of which one has to be *published*. More precisely, each user has a pair of keys, a secret one and a public one. He then has access to the public keys of everyone else. The secret key is used to decrypt messages and to generate signatures, while the public key is needed to encrypt data and to verify signatures.

2.2 Elliptic Curve Basics

In 1985, Koblitz and Miller independently proposed the use of a mathematical object as a basis for building new cryptographic algorithms. Such an object is elliptic curves.

Elliptic curves began their history as an object of studies more than one century ago, but were confined to pure mathematics. Namely, their main field of application has been represented by number theory. Elliptic curves have been an important tool used in the proof of Fermat's Last Theorem [Wil95].

Precisely speaking, elliptic curves are geometric objects endowed of an algebraic structure, which is the actual support to cryptographic applications. Elliptic curves are defined over some kind of field. This means they can be defined over $\mathbb{R}$ or $\mathbb{C}$, and the first studies were indeed on curves defined over such fields. For number theory and cryptography in particular it is however more advantageous to consider finite fields.

Cryptographic curves base fields are partitioned into two main classes, namely so-called binary fields and prime fields. As binary field is intended a field of characteristic 2. Conversely, a prime field is meant to be a field with characteristic p, which of course has to be prime and it is chosen to be a large number. The used fields must have a high order and, if binary fields are considered, they have to be extension fields, i.e. $\mathbb{F}_{2^n}$. Prime fields are instead usually base fields, such as $\mathbb{Z}/p\mathbb{Z}$, for some large prime p. We will describe here only prime field curves, as these are the type which we will focus on in the whole subsequent work. Binary field curves are defined in an analogous way, but lead to different arithmetic to be considered for both implementation and attacks. Our choice is however not arbitrary, since prime field curves are the more used class.

2.2.1 Formal Description

Consider the base field, with large prime characteristic $p > 3$, $\mathbb{Z}/p\mathbb{Z}$. In the following we will steal the notation usually reserved to p-adic numbers and denote this field as $\mathbb{Z}_p$ for readability purposes. We will also continue to use the $\mathbb{Z}$ symbol instead of the generic $\mathbb{F}$ to remember the isomorphism between the field and the set of residues modulo p.

An elliptic curve over $\mathbb{Z}_p$ is an algebraic variety in $\mathbb{Z}_p \times \mathbb{Z}_p$ defined by the

polynomial given in equation (2.1)

$$y^2 = x^3 + a_4x + a_6 \tag{2.1}$$

and it is denoted by $\mathbb{E}$, or by $\mathbb{E}/\mathbb{Z}_p$ to emphasize its defining field. The coefficients a_4 and a_6 are named this way to maintain consistency with different expressions also used for elliptic curve defining polynomials. Briefly, this means considering the points $P = (x, y) \in \mathbb{Z}_p \times \mathbb{Z}_p$ that satisfy the equation $y^2 = x^3 + a_4x + a_6$ together with a point at infinity, usually denoted as $\mathcal{O}$. For the curve to be regular it must also hold $4a_4^3 + 27a_6^2 \neq 0$.

The order of the curve, denoted as $|\mathbb{E}|$, is defined as the number of points lying on it, including $\mathcal{O}$. An important result states that the order $n = |\mathbb{E}|$ of a curve satisfies

$$p + 1 - 2\sqrt{p} \leq n \leq p + 1 + 2\sqrt{p} \tag{2.2}$$

We will see in section 5.6 that these tight bounds on the order n are fundamental for our attack feasibility.

An endowed group structure is the foundation of computations and cryptographic properties of the curve. The point at infinity $\mathcal{O}$ is taken as the group identity. Given two points $P = (x_P, y_P)$ and $Q = (x_Q, y_Q)$ an associative and commutative operation, usually called *point addition* and denoted by $+$, yielding $R = (x_R, y_R)$, is defined in the following way:

$$\begin{aligned} x_R &= \lambda^2 - x_P - x_Q \\ y_R &= \lambda(x_P - x_R) - y_P \end{aligned}$$

where

$$\lambda = \begin{cases} \frac{y_Q - y_P}{x_Q - x_P} & \text{if } P \neq Q \\ \\ \frac{3x_P + a_4}{2y_P} & \text{if } P = Q \end{cases}$$

The inverse of P is denoted by $-P$. It is easy to see, from the defining polynomial, that $P = (x_P, y_P) \Rightarrow -P = (x_P, -y_P)$.

The order n of the group is typically chosen to be prime. This implies that the obtained group is cyclic, and a point called the *generator* is usually taken and standardized together with other curve parameters for protocol definitions. We will denote such a point with G.

The group addition law is used to build another operation, called *scalar multiplication*. A scalar is some integer $k \in \mathbb{Z}$, and a point P is called the *base point* for the scalar multiplication defined as

$$[k]P = \underbrace{P + P + \ldots + P}_{k \text{ times}} \tag{2.3}$$

It is taken $[0]P = \mathcal{O}$, and $[-k]P = -[k]P$.

Some useful properties of the scalar multiplications follow from the associativity and commutativity of the group operation and from the finiteness of the order n of the curve:

1. $[k_1 + k_2]P = [k_1]P + [k_2]P$
2. $[k](P + Q) = [k]P + [k]Q$
3. $[k]P = [k + \lambda n]P = [k \bmod n]P \qquad n = |\mathbb{E}|, \lambda \in \mathbb{Z}$

It follows that an elliptic curve $\mathbb{E}/\mathbb{Z}_p$ with prime order n is isomorphic to the group $\mathbb{Z}_n$ through the map $k \leftrightarrow [k]P$.

2.2.2 Discrete Logarithm Problem

Considering the operation $Q = [k]P$, we refer to P as the *base point* and to k as the *scalar factor*. We will consider also the inverse operation consisting in finding a scalar k satisfying $Q = [k]P$ given the points P and Q. This is known as an *elliptic curve discrete logarithm* computation, because of the structural similarity with modular exponentiation operations.

Just as in modular exponentations, it happens that computing a logarithm is far more difficult than computing a point scalar multiplication. Indeed, there are no known algorithms for computing discrete logarithms in polynomial time. For this reason the scalar multiplication plays the role of one-way function in elliptic curve based cryptosystems, providing their public key capabilities. The problem of computing this kind of logarithm is known as the *elliptic curve discrete logarithm problem*, or ECDLP. Practical applications use some secret, like a key or a nonce, as scalar factor in multiplications. This way it is possible to publish the resulting point, as it is not possible for an attacker to compute the employed secret.

2.2.3 Baby Step Giant Step

Even though discrete logarithms take exponential time to be computed, it is of interest to examine known algorithms for the task. As it can be guessed, many of the ideas are taken from the discrete logarithm in modular arithmetic. The main exception being the impossibility to employ the famous *index calculus* algorithm, because of significant requirements missing in the elliptic curve version of the problem. This is one of the reasons that let elliptic curve systems supersede classical ones in expected security margin.

One of the most used algorithms for discrete logarithm computation is the *baby-step-giant-step*. We describe it because, as it will be showed, it can be used to greatly improve some attack performances. We are not interested in the details of other algorithms, though they can be used too. The main reasons why we chose the baby-step-giant-step algorithm are the ease to adapt it from the general case to the particular scenario we will have to deal with, and the great possibilities offered in terms of space/time trade-off flexibility.

The algorithm is based on the following rewriting of the sought logarithm k, for some suitable values of the other variables,

$$k = \alpha m + \beta \tag{2.4}$$

This then leads to the scalar multiplication

$$Q = [k]P = [\alpha m]P + [\beta]P$$

$$[\beta]P = Q - [\alpha][m]P$$

The algorithm proceeds by precomputing the values $[\beta]P$ for every $0 \leq \beta < m$. The successive step sets an accumulator starting from Q and checks whether one of the precomputed values is found when subsequently subtracting $[m]P$ from it. It is easy to see that this procedure eventually finds a suitable value for the logarithm if it exists. The value m can be used to tune the algorithm's memory and time requirements. It is moreover possible to compute many logarithms

Algorithm 1 Baby Step Giant Step

Input: point P, point Q, precomputed values $t_\beta = [\beta]G \; \forall \beta \in [0, m-1]$
Output: logarithm $k : Q = [k]P$

```
A ← [m]P
α ← 0
found ← false
while found = false do
    if ∃β : Q = t_β then
        k ← αm + β
        found ← true
    end if
    Q ← Q − A
    α ← α + 1
end while
return k
```

with the same base in shorter time, because the precomputations must be done only once.

We sketch the details of the procedure in algorithm 1.

2.3 Elliptic Curve Based Protocols

A number of algorithms exists based on elliptic curve primitives for all the major public key tasks. Namely, there are protocols for encryption of confidential information, for authentication and for key exchange. We present here the most significant ones, to show how elliptic curves are practically used to build various kinds of cryptographic algorithms. Please refer to [HMV04] for more details.

An explanation of how a cryptographic protocol typically works is shown in figure 2.1.

A typical elliptic curve key generation procedure works in two steps. A private key d is generated at random as a number satisfying $0 < d < n$, where n is the order of the curve used. The corresponding public key Y is then obtained by scalar multiplication, namely as $Y = [d]G$ where G is the chosen generator of the curve.

2.3.1 ECIES

A protocol for data encryption, originally proposed by Bellare and Rogaway, is Elliptic Curve Integrated Encryption Scheme. It is a variant of the ElGamal encryption scheme and is presented in algorithms 2 and 3.

A representation of this algorithm is shown in figure 2.2.

In the algorithm description the following functions are used:

KDF A key derivation function based on a hash function (or some other cryptographic function) depending on the considered standard.

ENC A symmetric key encryption function.

DEC The decryption function corresponding to ENC.

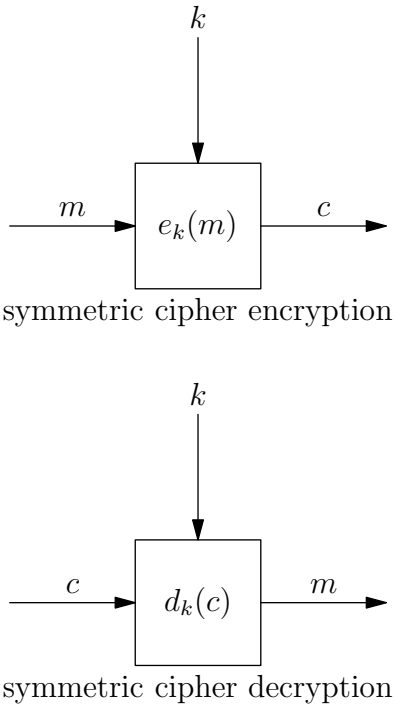

Figure 2.1: Cryptographic Functions

Algorithm 2 ECIES Encryption

Input: curve parameters $(\mathbb{E}, G)$, plain data m, public key $Y = [d]G$
Output: encrypted data (R, C, t)

1: $k \leftarrow$ random $\in [1, \ldots, n-1]$
2: $R \leftarrow [k]G$
3: $Z \leftarrow [k]Y$
4: **if** $Z = \mathcal{O}$ **then**
5: goto 1
6: **end if**
7: $(k_1, k_2) \leftarrow \mathrm{KDF}(x_Z, R)$
8: $C \leftarrow \mathrm{ENC}_{k_1}(m)$
9: $t \leftarrow \mathrm{MAC}_{k_2}(C)$
10: **return** (R, C, t)

MAC A message authentication code.

The decryption algorithm works because $Z = [k]Y = [kd]G = [d]R$.

2.3.2 ECDH

Simply derived from the classical Diffie-Hellman key agreement protocol ([DH75]) is Elliptic Curve Diffie-Hellman. It is based on computing two scalar multiplications performed in a different order but leading to the same result, and therefore

Algorithm 3 ECIES Decryption

Input: curve parameters $(\mathbb{E}, G)$, encrypted data (R, C, t), private key d
Output: plain data m
1: $Z \leftarrow [d]R$
2: $(k_1, k_2) \leftarrow \mathrm{KDF}(x_Z, R)$
3: check $t = \mathrm{MAC}_{k_2}(C)$ or reject the ciphertext
4: $m \leftarrow \mathrm{DEC}_{k_1}(C)$
5: **return** m

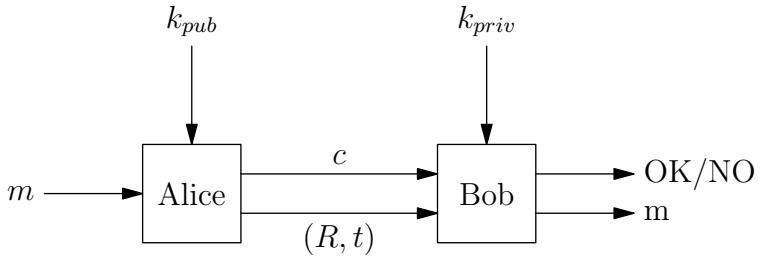

Figure 2.2: ECIES

used as a key exchange protocol. We show the one party's computations in algorithm 4. The only secret employed is each party's secret key. Curve parameters and obviously public keys are publicly available.

We sketch this algorithm too in figure 2.3.

Algorithm 4 ECDH

Input: Alice's private key a, curve parameters $(\mathbb{E}, G)$
Output: Shared secret k
1: $A \leftarrow [a]G$
2: **send** A to Bob
3: **receive** $B = [b]G$ from Bob (b being Bob's secret key)
4: $P \leftarrow [a]B$
5: **return** x_P

Note that $P = [a]B = [a]([b]G) = [ab]G$. Bob computes the same way so that both parties obtain the point $P = [ab]G$, which is otherwise computable only knowing both secret keys. A third player is then not able to compute P unless she can solve ECDLP. The x-coordinate of the point is then typically used as shared secret for subsequent computations.

2.3.3 ECDSA

An algorithm for digital signatures is derived from the classic DSA and ported to the elliptic curve domain. Elliptic Curve Digital Signature Algorithm, or ECDSA, is used to authenticate the origin of a message. It works as described in the algorithms 5 and 6. We will discuss the algorithm in deep details in the next section, since the rest of the work is developed for this signature algorithm.

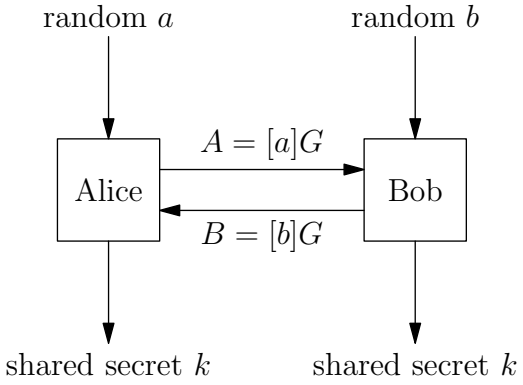

Figure 2.3: ECDH

The ECDSA algorithm is standardized by NIST in [NIS09] and it is considered to meet high security requirements, being its use allowed for top secret information [NSA10].

2.4 Signature Algorithm Details

In this section, we will discuss in detail the major points in the ECDSA computations: a secure parameters generation, the actual signature generation, and a description of the verification procedure.

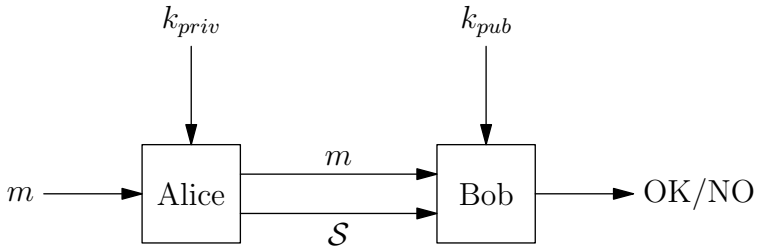

Figure 2.4: Digital Signature Protocol

The ECDSA protocol is a digital signature protocol, and therefore it is aimed at communication authentication. It can be used for example in protocols of the type *challenge-response*. In this scenario, a subject willing to authenticate to another will be requested to *sign* a *message*. This operation is shown in figure 2.4. A *private key* is used to generate a *signature* of a given message. The algorithms then allow a different subject to verify whether the signature was generated using the given private key. The verification proceeds without actually knowing the private key that was used, but using a different one, called the *public key*, that is mathematically related to the private one.

We also give graphical representations of the algorithms in the ECDSA family in figures 2.5, 2.6 and 2.7.

2.4.1 Parameters Generation

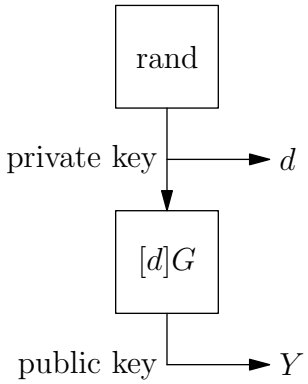

Figure 2.5: Key Pair Generation

The choice of the curve parameters is of course an important issue to guarantee the security of the system. For this reason there are a number of standardized curves, allowing the use of either different base fields and different key lengths, to meet different needs and security levels.

The NIST curves are referred to as P-l for the prime field and as B-l for the binary field, where l denotes the length of the key to be employed.

A different option is to generate a random curve for a particular need. A procedure exists to generate a curve at random allowing for a subsequent verification of it to be truly random. This algorithm is standardized in [ANS05]. Anyway the use of a random curve is not a common option, due to the great algorithm complexity when one wants to ensure the meeting of security constraints.

Once a curve has been chosen, either standard or random, a key pair is needed to use the algorithms. Such a pair is of course made up of a secret and a public key.

Consider a curve $\mathbb{E}/\mathbb{Z}_p$, we denote by n the order $|\mathbb{E}|$ of the curve. For security reasons, the order n must not be a smooth number (i.e. with small prime factors). The reason for this constraint is the Pohlig-Hellman attack, see [PH78]. The order is considered secure if it is of the form $n = hq$, where q is a large prime, and h, called the *co-factor*, must satisfy $1 \leq h \leq 4$. The NIST curves have prime orders.

As for most elliptic curve protocols, the private key is a random scalar. In other words an integer between 1 and $n-1$, i.e. a random element of $\mathbb{Z}_n^*$. Let's denote with l the length in bits of n, $l = \lceil \log_2 n \rceil$. Therefore, the overall length of the secret key is l bits too.

The public key corresponding to a secret $d \in \mathbb{Z}_n^*$ is computed as $Y = [d]G$. Retrieving the secret from the public key requires solving an ECDLP, and is therefore considered not feasible.

It is important to note that the keys in the pair are very different objects: the private key lies in $\mathbb{Z}_n^*$, while the public key is a point on the curve $\mathbb{E}/\mathbb{Z}_p$. In signature algorithms, some computations are done as integer computations

modulo n, in other words, being n prime, in the field $\mathbb{Z}_n$. This field should not be confused with $\mathbb{Z}_p$, which is the field for the curve construction and which the point coordinates belong to.

2.4.2 Signature Generation

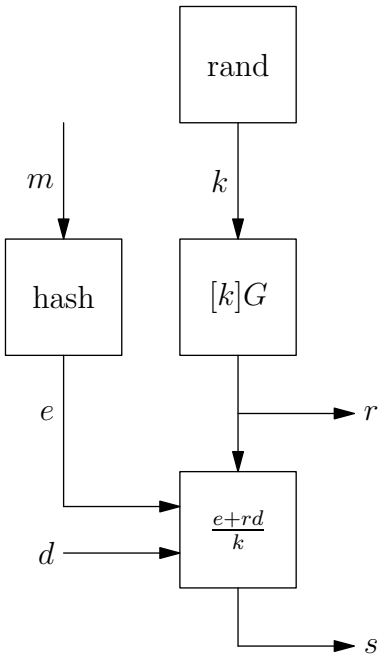

Figure 2.6: ECDSA Signature Generation

Signature computation is done by means of algorithm 5.

Algorithm 5 ECDSA Signature Generation

Input: private key d, curve parameters $(\mathbb{E}, G)$, message m
Output: signature $\mathcal{S}$
1: $e \leftarrow \text{hash}(m)$
2: $k \leftarrow \text{random} \in [1, \ldots, n-1]$
3: $P \leftarrow [k]G$
4: $r \leftarrow x_P \bmod n$
5: $s \leftarrow (e + rd)/k \bmod n$
6: **return** $\mathcal{S} = (r, s)$

In the unlikely case either r or s is 0 the algorithm is run again with a different k to obtain valid values.

The hash funcion employed in the step 1 of the algorithm must return an l-bit value and it should obviously satisfy all the requirements usually requested for a cryptographic hash function.

The random value k generated in step 2 is called the *nonce* and it is important it is truly random and never reused. If two signatures are generated

with the same nonce $k = k_1 = k_2$, it is trivial to retrieve the secret key. In this case it holds $r_1 = x_{[k]G} = r_2$. This makes it also trivial to identify such signatures. From $s_1 = (e_1 + rd)/k$ and $s_2 = (e_2 + rd)/k$ we can write $(e_1 + rd)/s_1 = k = (e_2 + rd)/s_2$. It follows

$$\frac{e_1}{s_1} - \frac{e_2}{s_2} = \left(\frac{r}{s_2} - \frac{r}{s_1}\right) d$$

from which the secret key can be obtained since the message hashes and all the involved values except d are known.

Also if the random generation function has any weaknesses it can be easy to exploit it to recover the secret key from the resulting signatures, as in the case presented in [NNTW05].

The point $P = [k]G$ computed in step 3 and more specifically its x-coordinate is the *ephemeral key* used to mask the other values. Being the nonce discarded at the end of execution, it is not possible to compute again the same ephemeral key, which therefore is responsible for carrying information about the nonce in a non-invertible way.

Eventually, s is the actual signature, being computed mixing the message hash with the secret key.

The output of the algorithm is the whole pair (r, s). Without the value r there would be no information about k.

2.4.3 Signature Verification

The steps to verify an ECDSA digital signature are presented in algorithm 6.

Algorithm 6 ECDSA Signature Verification

Input: public key $Y = [d]G$, curve parameters $(\mathbb{E}, G)$, message m, signature $\mathcal{S}$
Output: signature validity

1: $e \leftarrow \text{hash}(m)$
2: $u_1 \leftarrow e/s \bmod n$
3: $u_2 \leftarrow r/s \bmod n$
4: $T \leftarrow [u_1]G + [u_2]Y$
5: $w \leftarrow x_T \bmod n$
6: **if** $w = r$ **then**
7: **return** valid signature
8: **else**
9: **return** not valid signature
10: **end if**

We now write the verification algorithm in a more compact form to better analyze it and understand its correctness. This is important to understand some attacks shown in the sequel.

For readability we intend all the operations implicitly in $\mathbb{Z}_n$, i.e. modular additions, products and inversions. Starting from the generation algorithm the following holds

$$s = \frac{e + rd}{k}$$

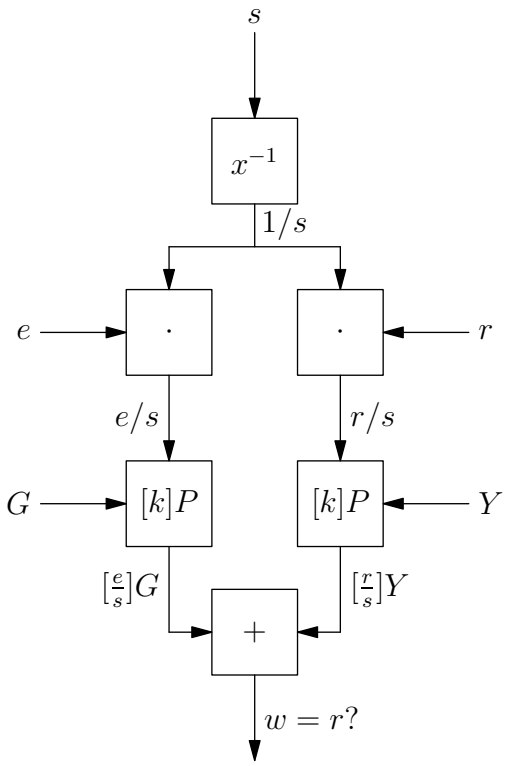

Figure 2.7: ECDSA Signature Verification

From the field structure of $\mathbb{Z}_n$, we can write it as

$$k = \frac{e + rd}{s}$$

Using these values as scalar factors to multiply the curve generator we obtain

$$[k]G = \left[\frac{e}{s}\right] G + \left[\frac{rd}{s}\right] G$$

By substituting the public key $Y = [d]G$, the equation becomes

$$[k]G = \left[\frac{e}{s}\right] G + \left[\frac{r}{s}\right] Y$$

Note that the right part is exactly the point T computed by the verification algorithm, since $u_1 = e/s$ and $u_2 = r/s$. From the signature generation equations, it holds $r = x_{[k]G} \bmod n$, thus

$$r = x_{[k]G} \bmod n = x_T \bmod n = w$$

2.5 Scalar Multiplication Algorithms

The heaviest computation in the signature algorithm is represented by the elliptic curve scalar multiplication. We examine the main algorithms used to

implement it in ECDSA context. Note that these are just portings of the *square and multiply* algorithm normally employed for modular exponentiation.

2.5.1 Double and Add

The basic method is described in algorithm 7.

Algorithm 7 Binary Left-to-Right Double-and-Add

Input: base point G, scalar factor $k = (k_{l-1} \cdots k_1 k_0)_2$
Output: point multiplication $P = [k]G$
1: $P \leftarrow \mathcal{O}$
2: **for** i in $l-1$ downto 0 **do**
3: $\quad P \leftarrow 2P$
4: $\quad$ **if** $k_i = 1$ **then**
5: $\quad\quad P \leftarrow P + G$
6: $\quad$ **end if**
7: **end for**
8: **return** P

The algorithm can be understood considering the binary representation of the scalar k. The following recursion is then exploited

$$(k_{l-1} \cdots k_\mu k_i)_2 = 2(k_{l-1} \cdots k_\mu)_2 + k_{\mu-1}$$

Multiplying the generator by these values leads to

$$[(k_{l-1} \cdots k_\mu k_{\mu-1})_2]\, G = [2]\, [(k_{l-1} \cdots k_\mu)_2]\, G + [k_{\mu-1}]G$$

There exists also a version of the algorithm proceeding in the opposite direction when scanning the scalar k, known as right-to-left. It has an analogous overall structure, and we don't need the details here.

2.5.2 Double and Always Add

A vulnerability of algorithm 7 is due to the conditional branch of step 4. This leads to different and recognizable power consumption patterns depending on the key bits, which are exploitable in so called SPA attacks. To address this problem, an algorithm is developed. We present it in algorithm 8. As it can be noted it always performs point addition. The absence of branches makes the power consumption resulting from different bits of the key undistinguishable.

2.5.3 Montgomery Ladder

The Montgomery Powering Ladder algorithm, shown in algorithm 9, was originally proposed by [Mon87] and analyzed by [JY02]. In this case too, the power consumption does not depend upon the scalar factor bits.

2.6 Integer Multiplication Algorithms

Many attacks exist that target the scalar multiplication operation, and it actually seems natural to attack this operation. We will see many examples of this

Algorithm 8 Binary Left-to-Right Double-and-Always-Add

Input: base point G, scalar factor $k = (k_{l-1} \cdots k_1 k_0)_2$
Output: point multiplication $P = [k]G$
1: $P_0 \leftarrow \mathcal{O}$
2: **for** i in $l-1$ downto 0 **do**
3: $\quad P_0 \leftarrow 2P_0$
4: $\quad P_1 \leftarrow P_0 + G$
5: $\quad P_0 \leftarrow P_{k_i}$
6: **end for**
7: **return** P_0

Algorithm 9 Montgomery Ladder

Input: base point G, scalar factor $k = (k_{l-1} \cdots k_1 k_0)_2$
Output: point multiplication $P = [k]G$
1: $P_0 \leftarrow \mathcal{O}$
2: $P_1 \leftarrow G$
3: **for** i in $l-1$ downto 0 **do**
4: $\quad P_{1-k_i} \leftarrow P_{1-k_i} + P_{k_i}$
5: $\quad P_{k_i} \leftarrow 2P_{k_i}$
6: **end for**
7: **return** P_0

in chapter 4. It is anyway important to consider also another operation involved in the signature computation, i.e. modular multiplication. Since our attack will target the multiplications in prime modular fields, we present in this section the most used algorithms for integer multiplications. We will see in the sequel how these relate to modular multiplications.

In the cryptographic applications, the involved numbers are often large integers that do not fit in a computer register. For this reason, algorithms to manage multiple precision integer arithmetic must be developed. We will therefore show such multiple precision algorithms, assuming that the computer is able to handle the basic single precision operations, such as addition and multiplication of w-bit numbers.

2.6.1 Operand Scanning

The most natural procedure, in fact also known as *schoolbook multiplication*, is the *operand scanning*, which we describe in algorithm 10. We consider w-bits words and σ-words operands. We show the algorithm working on operands of the same length, because this is the case we will deal with. Anyway, it can be easily adapted to operands of different lengths.

In the algorithm we use the notation (uv) to denote a $2w$-bits value which is the concatenation of u and v, i.e. $u2^w + v$.

2.6.2 Product Scanning

Algorithm 11, known as *product scanning*, is basically a restructuring of the scanning cycles.

Algorithm 10 Operand Scanning Integer Multiplication

Input: factors $a = (a_{\sigma-1} \cdots a_1 a_0)_{2^w}$ and $b = (b_{\sigma-1} \cdots b_1 b_0)_{2^w}$
Output: product $c = ab = (c_{2\sigma-2} \cdots c_1 c_0)_{2^w}$

```
1: c_i ← 0 ∀i ∈ [0, ..., 2σ − 2]
2: for i in 0 to σ − 1 do
3:     u ← 0
4:     for j in 0 to σ − 1 do
5:         (uv) ← c_{i+j} + a_j b_i + u
6:         c_{i+j} ← v
7:     end for
8:     c_{i+σ} ← u
9: end for
10: return c
```

Algorithm 11 Product Scanning Integer Multiplication

Input: factors $a = (a_{\sigma-1} \cdots a_1 a_0)_{2^w}$ and $b = (b_{\sigma-1} \cdots b_1 b_0)_{2^w}$
Output: product $c = ab = (c_{2\sigma-2} \cdots c_1 c_0)_{2^w}$

```
1: R_0 ← 0
2: R_1 ← 0
3: R_2 ← 0
4: for k in 0 to 2σ − 2 do
5:     for {(i, j) : i + j = k, 0 ≤ i, j ≤ σ − 1} do
6:         (uv) ← a_j b_i
7:         (ε, R_0) ← R_0 + v
8:         (ε, R_1) ← R_1 + u + ε
9:         R_2 ← R_2 + ε
10:    end for
11:    c_k ← R_0
12:    R_0 ← R_1
13:    R_1 ← R_2
14:    R_2 ← 0
15: end for
16: c_{2σ−1} ← R_0
17: return c
```

The operands are anyway used in the same way: a step of the algorithm is dedicated to combine each pair (a_i, b_j) of words through a single precision multiplication. As we will see, this means that the developed attacks work on both algorithms without modifications.

We use the same notation (uv) as above for multiplication results. Plus, we denote with (ε, R) the addition results, meaning the sum yields the single precision word R and a possible carry out ε, which can assume the values 0 and 1.

Chapter 3

Side Channel Attacks

In the history of cryptography, encryption algorithms have been menaced by constantly improving attacks, ranging from the early *known ciphertext* attacks, in which the cryptanalyst is able to find secret information knowing only a ciphertext, to modern *chosen plaintext* attacks, when the attacker can choose a plaintext and get it encrypted by a remote device. All these attacks share an important common feature: they are aimed at attacking the structure of the cryptographic *algorihtm*.

In the recent times, cryptographic algorithms have become very strong against such kinds of mathematical attacks, and the match between cryptographers and cryptanalysists seems greatly unbalanced in favor of the former. But attacks of a very different kind, known as *side channel attacks*, are arising today. Instead of attacking the mathematics underlying an algorithm, the attacker targets the actual *implementation*. This leads to a number of different ideas and threats to be taken into account. The main disadvantage of such a kind of attack is the need for physical access to a cryptographic device, but this is becoming a less important point with the advent of modern widespread use of digital hardware in every application field. It indeed often happens that the owner of the device and the owner of the secret stored inside it do not coincide. This happens for example with mobile sim's, credit cards, pay tv cards, and other such mobile devices.

It seems therefore important to study the techniques employed for such attacks, and the possible ways to defend against them.

3.1 Classification of Attacks

A first class of side channel attacks is represented by *passive attacks*. These are based on the passive observation of information related to the cryptographic hardware, that should not be normally taken into consideration. The attacker does not modify the hardware he has access to, but monitors a measure of some quantity, including for example the power consumption or the electromagnetic radiation. These quantities are supposed to be related to the secrets inside the device in a way that depends on the considered algorithms. After collecting sets of data measurement, the attack proceeds by analyzing such datas by correlations or pattern matching techniques. This analysis, together with suitable

hypotheses, allows to retrieve the sought secret information.

A different class of side channel attacks is made up by *active attacks*, or *fault attacks*. In this case, an attacker aims at somehow influence the behaviour of a device to make it perform in a wrong way. To reach this goal, a number of techniques have been developed and present different properties. We will sketch them in the following section.

A fault attack proceeds by provoking an unexpected behaviour of the device, collecting resulting wrong data, and eventually analyzing them together with some hypotheses. If suitable conditions are met, this again leads to the discovery of secret information. Usually, it is necessary to compare correct and wrong results corresponding to the same encryption to recover secret information. For this reason, fault attacks are commonly refered to as *differetianl fault attacks*, or DFA. We will see examples of details of this kind of technique in chapter 4.

3.2 Fault Injection Techniques

A classification of fault attacks is as follows:

Non-invasive attacks The attacker limits his action on normally accessible interfaces of the device, without inducing physical modifications to the system. This class is typically populated by passive attacks, such as power analysis, timing analysis, but some fault attacks also belong here, i.e. those aimed at disturbing the clock or power signals.

Semi-invasive attacks The attacker is expected to unpack the chip but there is no electrical contact with the surface. Fault attacks belonging to this class are evolving very quickly for their relative cheapness. Typical examples of this class are optical and laser attacks.

Invasive attacks The maximum level of interaction with the chip is reached, in that depackaging and depassivation of it are employed. This way very powerful attacks become possible, like direct probing or forcing of electrical signals. These techniques are anyway related to very high costs to be performed.

Various techniques exist to induce a faulty behaviour during the computation of a cryptographic device. A summary is as follows:

Cosmic rays It is well known, from space travel application, that cosmic rays interacting with electronic hardware cause mulfunctioning. Typically, the consequence is a *bit flip* in memory areas, as it is described for example in [GA03]. Anyway, it is not an easy task to produce and direct cosmic rays to the wanted surface or area, and waiting for a cosmic ray to hit the desired location by chance is too time consuming.

High energy radiation Other kinds of radiation, like α-, β-rays can induce faulty behaviour in hardware. Having less energy than cosmic rays, these radiations cannot penetrate the packaging of chips, and thus require semi-invasive operation. With X-rays the situation is slightly better, because the more energy of these radiations let them to penetrate the packaging, see [BECN$^+$06] and [GA03]. The main problems for such techniques reside again in the difficulty to target the specific location.

Heat Electronic devices have a well defined range of temperatures allowing them to work correctly. If the chip is overheaten it starts showing a faulty behaviour. The idea is to tune the temperature to such a value that allows certain operations to be carried out correctly while others are faulted, e.g. to obtain working writes but faulty reads. For more details, see again [BECN+06] and [GA03]. The equipment needed for such attacks is not too expensive, a strobe lamp being enough, but the difficulties are related to the reachable precision, besides the fine-tuning needs. The induced faults typically hit many locations and are not narrow in time.

Light attacks It is possible to induce currents, and thus faults, in the device because of photoelectric effect. To exploit this, it is needed to focus light on the target chip area. For this reason, lasers are good tools in that they offer great directionality of the beam, and a description of their use can be found in [MRL+06] and [LAM+07]. A cheaper mean of fault injection is white light. It is possible to induce faults also by simply focusing an intense light for a short time, as presented in [SA03].

Ion beams Focused ion beams can interfere with the chip too. They can be used to obtain a number of effects, ranging from the destruction of the passivation layer to allow electrical probing of the inner circuit, to local ionization of the silicon that is interpreted as a signal by the circuit.

Power line alteration Since physical access to the device is assumed, it is possible to consider altering the power supply of the chip. Two ways are mainly employed. Namely, injecting spikes on the power line or keeping the power at a different level throughout the whole computations. Both these techniques clearly lead to possibly modifying the data being processed by the device, in a way that tightly depends on the specific hardware. Typically, bit flips in read and written data appear, as well as errors on the program flow caused by wrong instructions being decoded by the logic. Practical results obtained with power spikes are presented in [ABF+03], [SH08] and [KQ07]. Results with underfeeding the power line are showed in [BBPP09].

Clock line alteration Modifications of the supplied clock, usually called glitches, can also be considered, as are described in [AK96]. The effects of clock alteration are due to the program counter logic. Being this much simpler than the rest of the computer, it is possible to make the program counter register to increment while the rest of the computer has not yet completed its execution. This then leads to an instruction execution being skipped, and thus to modify the behaviour of the program.

External electrical field Applying a controlled electrical field near the device is another possible method of fault injection. Again, the problem is targeting specific areas on the chip, moreover, if the chip is not unpackaged, some knowledge of the internal layout is needed. In [QS02] it is presented a way to induce eddy currents near the circuits by use of magnetic fields. This method of fault injection can be very precise and does not need the chip to be unpackaged.

3.3 Fault and Error Models

To effectively develop a fault attack, it becomes necessary to have a formal description of the physical alteration that happen in the circuit. For this reason two models are introduced to better describe the system and abstract from the physical layer. Namely, a *fault model* is used to characterize the way followed to induce the fault and its properties. An *error model* is instead constructed to describe the logical alterations that affect the algorithm exectuion. Once an attacker has access to such characterization of his operations, it becomes possible for him to analyze the behavior of the wrong computations and, making hypotheses and checking results against them, obtain some knowledge about secret information.

A fault model has a couple of directions to be classified into.

It is possible to characterize the fault basing on the control on the fault location:

no control it is not possible to control the location of the fault,

loose control it is possible to choose a target variable,

complete control a particular bit can be targeted by the fault.

The possibilities in timing control are also interesting, and are classified as:

no control it is not possible to control the location of the fault,

loose control a block of few operations is targeted,

precise control the exact time is chosen and a certain operation is therefore hit.

An error model specifies in which way the error affects the algorithms, and it is possible to differentiate among the following:

The error, seen at the algorithm level, can behave in different ways:

- change the value of a variable,
- modify the flow control of the program.

In case a variable is modified, the number of affected bits can be:

- single faulty bit,
- few faulty bits, e.g. a byte,
- random amount of faulty bits.

The fault type can be one of:

set/reset the bit or variable is set to 1 or 0

bit flip the bit has its value flipped

random the bit or variable is changed in a random way

In case the program flow is altred, typically we observe a conditional computation taking the wrong branch.

For the construction of the fault model, it can be useful to take into account also the probability distributions of the various characterization dimensions. Usually, no control over a location means that one among a certain number N of locations can be hit with a probability $1/N$.

3.4 DFA Example

We present here an example of fault attack, to help the reader in better understanding the mechanisms that lead an attacker to exploit a computation error to discover some secret.

A very simple yet effective attack is considered, namely the so-called *safe-error* attack. Let us consider a digital signature function $s_d(m)$ using the secret key d. A corresponding signature verification function $v_y(m)$ using the public key y will be used to analyze the results.

Suppose that the attacker is able to manipulate the device in such a way that it computes the function $s_{d'}(m)$ where d' is obtained by forcing one of the bits of d to assume the value 0. The actual manipulation technique is not important here. It suffices that the attacker can consider this effect among his hypotheses. The attacker can then apply the verification function, since he is in possession of the public key. We see how the output of this verification will lead to a secret leakage.

If the verification fails the attacker can tell that the bit of the private key at the position of the bit that was changed has value 1. If, conversely, the verification succeeds, then the attacker can tell that the bit was not actually changed, and therefore has value 0.

If the attacker has the possibility to repeat this attack applying the technique to force a bit for every position within the secret key, he can then recover the whole key by hitting one bit of it at a time.

Of course this attack is very simple, and its major weakness is the need for a very precise fault injection technique, that can imply very high costs. A challenge for a fault attack is to be effective with the most flexible hypotheses as possible, thus to possibly reduce the cost of the whole procedures, both in terms of laboratory equipment and of computational effort to analyze the results. We will see in the next chapter several other examples, specifically targeting elliptic curve based protocols, and their characteristics in terms of performance and feasibility will be discussed.

Chapter 4

Known Attacks Against Elliptic Curve Based Protocols

We present the most important known results on fault attacks to elliptic curve based cryptosystems, with a particular focus on ECDSA.

We start considering the basic elliptic curve operation: the scalar multiplication. This is usually implemented with the double and add algorithm, which is just a different name for the square and multiply algorithm commonly used for exponentiation in the RSA cryptosystem (see chapter 2 for more details on the algorithm). Because of this inherent analogy between the scalar multiplication and the modular exponentiation operations, it is possible to port some attacks, originally targeting RSA, to attack elliptic curve cryptosystems too.

We then analyze attacks thought specifically for the elliptic curve scalar multiplication. In this section we show some vulnerabilities that can arise from a too simplistic implementation of an elliptic curve based cryptosystem and the ways to protect from these menaces.

Eventually we consider a full fledged cryptosystem such as ECDSA and the known methods to attack it by means of fault injection.

In the following we adopt the usual notation, concentrating on the scalar multiplication $P = [k]G$ on the curve defined by $\mathbb{E}/\mathbb{F}_p : y^2 = x^3 + a_4x + a_6$.

4.1 Attacks Mutuated from RSA

The following are attacks originally thought to target the RSA crpytosystem. Because of the inherent similarity between modular exponentation and elliptic curve scalar multiplication, the same ideas can be used. However, typical elliptic curve based cryptosystems involve more operations than the bare scalar multiplication, and this is the cause of the main disadvantages of the shown attacks.

4.1.1 Faults on Intermediate Values During the Multiplication

An attack on the RSA cryptosystem was described in [BDL97] which exploits faults on intermediate operands during the square and mutliply algorithm execution. Such attack has been ported to elliptic curve scalar multiplication in [BMM00] since it uses the same overall algorithm, i.e. the double and add. It is possible to adapt this kind of attack to both the left-to-right and the right-to-left versions of the double and add algorithm.

The attack needs a fault model of single-bit flips happening at random moments during the scalar multiplication. By collecting a sufficient amount of faulty results it becomes possible to retrieve all the bits of the secret scalar k in blocks.

As stated by the authors, in the general case, given a curve $\mathbb{E}/\mathbb{F}_p$, said $m = o(\log\log\log p)$, l the length in bit of k, the number of needed faulty results is $O((l/m)\log l)$. The attack proceeds by simulating the device computation starting from suitable hypotheses, checking whether a fault on one bit might have produced some of the collected output.

The authors sketch a way to perform the attack to ElGamal decryption as well. It does not seem obvious how to apply it to ECDSA because of the nonce.

4.1.2 Sign Change Attack

A different idea, derived from an RSA equivalent, comes from [BOS04].

The fault model assumed requires the attacker to be able to change the sign of an intermediate point used during the computation, i.e. changing only the y-coordinate of a target point Q_i, leading to $\tilde{Q}_i = -Q_i$ to be used instead. A faulty multiplication result $\tilde{Q}$ is then output in place of the correct $Q = [k]P$. Computing $Q + \tilde{Q}$ can lead to bits of k to be revealed in blocks as described in the article.

While most of the attacks rely on modifying in some way the curve over which the computations are made, the sign change attack does not require to change such structure. This has the important (dis)advantage of being less practically detectable by the device, since a simple verification of the involved points is no more effective. The attack is possible on various double and add algorithms, though, according to the authors, it is easier to realize on NAF-based implementations.

4.1.3 Faults on the Scalar Factor

In [BDH+98] it is presented an attack to the RSA cryptosystem in which a fault on the secret exponent is exploited. The idea can be easily ported to elliptic curve scalar multiplication. Suppose a single bit flip in the representation of k is injectable, producing $\tilde{k} = k \pm 2^i$ for some i. An attacker then uses both a correct result $Q = [k]P$ and a faulty one $\tilde{Q} = [\tilde{k}]P$ to recover the i-th position bit of k by simply computing $Q - \tilde{Q} = \mp 2^i P$ and checking this value against all possible candidates. It is also possible to use the attack even with multiple bit errors, given a heavier computation effort: said l the length of k, the attack runs in $O(l^t)$ time instead of $O(l)$ in the case of t-bit errors.

The authors also show how to extend this attack to DL-based systems, in particular they describe a way to attack classical DSA using the same faults and ideas shown above.

To adapt this attack to ECDSA very little effort is necessary. Let $d = \sum_{i=0}^{l-1} d_i 2^i$, and there be a bit-flip fault yielding $d_j \to \tilde{d}_j$ for some j. Denote the faulty key by $\tilde{d}$. A faulty signature is produced such that $\tilde{\mathcal{S}} = (r, \tilde{s})$ where $\tilde{s} = (e + \tilde{d}r)/k \bmod p$ and $\tilde{d} = d + (\tilde{d}_j - d_j)2^j$. Let $\tilde{u}_1 = e/\tilde{s} \bmod p$, $\tilde{u}_2 = r/\tilde{s} \bmod p$. Computing $\tilde{T} = [\tilde{u}_1]P + [\tilde{u}_2]Y$, where Y is the public key $[d]P$, makes the attack possible.

For each $j \in 0, \ldots, l-1$ and $\sigma \in \{-1, 1\}$, check whether $\tilde{T} + [\frac{\sigma 2^j r}{\tilde{s}}]P = r$. If that is the case then $\tilde{d}_j - d_j = \sigma$ and we can recover the whole secret d, faulting one bit of it at a time.

4.1.4 Safe-Error Attack

Consider an implementation of the double-and-always-add algorithm as a protection against SPA. If a precise timing control fault model is assumed it is possible to inject a fault during the addition step in a given iteration i. If an invalid output is detected then the fault is effective, $k_i = 1$. If, conversely, the output is correct, then the addition is dummy, hence $k_i = 0$. The secret k has been completely broken.

4.2 Elliptic Curve Scalar Multiplication Attacks

The attacks in this section are thought specifically for the elliptic curve scalar multiplication. They typically involve the use of some modification of the curve structure. This way it is possible to make a discrete logarithm problem arise which is different from the original one and likely easier to compute.

4.2.1 Chosen Input Point

The first attack proposed by [BMM00] performs an injection of a malicious base point directly as input to the device. In the case it does not perform a check to establish whether or not the point lies on the curve then the computation takes place on the shifted curve $\tilde{\mathbb{E}}$ as previously described. The attack consists in carefully choosing a weak curve and a point on it in such a way that it has a small or smooth order n'. Feeding the device with this point will result in an output point for which the discrete logarithm is solvable, thus obtaining some c for which $k \equiv_{n'} c$ holds. Repeating the process for other input points and related orders ${n'}_i$ we obtain more equations that can be used to recover the secret k by means of the Chinese Remainder Theorem.

The previously described attack does not actually use any fault and is not to be considered feasible in practice, since it relies only on a somehow trivial implementation error. It is however useful to understand the general mechanism of more sophisticated attacks.

4.2.2 Faults on the Base Point

A second attack in [BMM00] considers a less naïve device, which in turn checks for the input point to be on the expected curve. The way to make the device compute on a shifted curve is now an actual fault injection on the register containing the point coordinates. The device then computes $k\tilde{P} \in \tilde{\mathbb{E}}$ instead of $kP \in \mathbb{E}$ and the logarithm computation is feasible again. A minor disadvantage of this procedure is the missed possibility to preventively choose an insecure shifted curve, having to rely on chance instead.

The curve over which the computation has been done on can be recovered from the output point (x_Q, y_Q), since the following holds: $\tilde{a}_6 = y_Q^2 - x_Q^3 - a_4 x_Q$. In case the coefficients define an elliptic curve (i.e. the discriminant does not nullify when $\tilde{a}_6$ is put in the curve equation), we have reduced the original logarithm problem to a new one.

The problem now is we do not know the base point $\tilde{P}$ for the logarithm computation. We have therefore to rely on knowing a precise fault model, e.g. suppose that $\tilde{P}$ differs in only one bit from the input point P. We can then test all candidates for $\tilde{P}$ and verify which one belongs to the curve $\tilde{\mathbb{E}}$, and be able to compute the discrete logarithm with respect to that base point.

Considering a more general fault model, in the case only one coordinate of P is changed (assume x for simplicity, faulty y case is analogous), the point can still be retrieved. [CJ05] notes that $\tilde{x}$ is a root of $x^3 + a_4 x + \tilde{a}_6 - y^2$, and can therefore be guessed among at most three choices.

The authors in [BMM00] propose a way to apply this attack to the ElGamal decryption as well. Slight modification must be taken into account since we do not have access to the y-coordinate of the output point. They claim that the logarithm may still be computable if again $\tilde{P}$ differs in only one bit from P.

The attack can also be adapted to ECDSA, again with slight modifications to meet the cryptosystem properties. Since the base point is now fixed and publicly known, our problem is only to retrieve the $\tilde{\mathbb{E}}$ curve parameters, which again cannot be done since we do not have the y-coordinate of the output point. Moreover, also the x-coordinate now belongs to a small set of possible values which we have to explore. Given an appropriate fault model we can however recover the nonce k and thus the private key in subexponential time.

4.2.3 Faults on the Underlying Field

A fault can be injected to affect the representation of the field K on which the curve is defined. Such idea has been presented by [CJ05]. Let $\mathbb{F}_p$ be a field of prime characteristic p and $\mathbb{E}/\mathbb{F}_p : y^2 = x^3 + a_4 x + a_6$ an elliptic curve. If we manage to modify the internal representation of p we can attack the system to recover the secret k even with an only pair of correct and faulty results.

Let $\tilde{p}$ be the faulty value which the system relies on for its computations. The performed operation then becomes $\tilde{Q} = [k]\tilde{P}$ where $\tilde{P} = (\tilde{x}_P, \tilde{y}_P)$, $\tilde{x}_P \equiv_{\tilde{p}} x_P$ and $\tilde{y}_P \equiv_{\tilde{p}} y_P$, and can be viewed as to happen over a curve $\tilde{\mathbb{E}}/\mathbb{Z}_{\tilde{p}} : y^2 = x^3 + \tilde{a}_4 x + \tilde{a}_6$ with $\tilde{a}_4 \equiv_{\tilde{p}} a_4$ and $\tilde{a}_6 \equiv_{\tilde{p}} \tilde{y}_Q^2 - \tilde{x}_Q^3 - a_4 \tilde{x}_Q \equiv_{\tilde{p}} \tilde{y}_P^2 - \tilde{x}_P^3 - a_4 \tilde{x}_P$.

Knowing the points $\tilde{P}$ and $\tilde{Q}$ it is possible to recover $\tilde{p}$ since it divides $(\tilde{y}_Q^2 - \tilde{x}_Q^3 - a_4 \tilde{x}_Q) - (\tilde{y}_P^2 - \tilde{x}_P^3 - a_4 \tilde{x}_P)$. This is made possible by the fact that today's curves use small primes for p with respect to modern factorization algorithms possibilities, thus making the discover of $\tilde{p}$ feasible.

It is now possible to solve the discrete logarithm on smaller curves and reconstruct the value of k through the CRT ([CJ05]).

If ECDSA is considered, we don't have access to the complete output point Q but only to a value r for which $r = \tilde{x}_Q \bmod n$ holds where n is the order of P in the original curve $\mathbb{E}$, so that we cannot recover $\tilde{p}$. Anyway, if we have a somehow stronger fault model such that we know that only one bit of p is flipped, we can try all possible candidates ξ_i for $\tilde{p}$. For each ξ_i we can search for an $\tilde{x}_Q$ in a small set (since n is roughly the same size as p) and consider the curve $\tilde{\mathbb{E}}_i/\mathbb{Z}_{\xi_i}$, then proceed as described above by solving the discrete logarithm on $\tilde{\mathbb{E}}_i$. The problem of this method resides in the low probability for $\tilde{\mathbb{E}}$ to have smooth order and give rise to an easily solvable DL problem [BMM00].

4.2.4 Faults on the Curve Parameters

A fault on the a_4 coefficient leads to a similar situation. The device output $\tilde{Q}$ lies on $\tilde{\mathbb{E}} : y^2 = x^3 + \tilde{a}_4 x + \tilde{a}_6$. Knowing P and $\tilde{Q}$ we can compute $\tilde{a}_4$ and $\tilde{a}_6$ as the solutions to $y_P^2 = x_P^3 + \tilde{a}_4 x_P + \tilde{a}_6$ and $y + Q^2 = x_Q^3 + \tilde{a}_4 x_Q + \tilde{a}_6$, then solve the DL.

4.2.5 Attacking the Montgomery Ladder

The Montgomery Ladder algorithm for double and add computation has the important property of not using the y-coordinate of the points during the computation. This has been exploited by [FLRV08] to conduce an attack by feeding the device with a malicious point. The main idea is that about half of the possible x-coordinate values lead to a point on the given curve, while the other half leads to points on the twist of the curve, which is typically cryptographically less secure.

It is therefore straightforward how to conduce the attack by asking the device for a computation based on a malicious x-coordinate, corresponding to a point on the twist, and then solving the DL on the twist.

The obvious countermeasure is to validate the input or output (or both) coordinate, by simply verifying that $x^3 + a_4 x + a_6$ be a quadratic residue in $\mathbb{F}_p$. In the case the device carries out this verification we can however try to modify the value injected faults to bypass the check. Given a model in which only one register is affected and yields a faulty value, the authors show a technique to recover the secret k with only one or two faults (in case of large registers).

The attack can be furthermore adapted to multiplications with a fixed base point, thus obviously belonging to the curve and not to the twist. In this case the authors propose to inject faults both at the beginning and at the end of the computation. Now, having more than only two faulty results leads to a more efficient attack, the overall attacking procedure remaining the same.

4.3 Attacks to ECDSA

The attacks in this section target the whole ECDSA cryptosystem, and are yet the most effective. The attack we will propose in chapter 5 belongs to this type.

4.3.1 Faults on the Group Order Parameter

A fault model for which we inject a fault during the loading of the group order n is used by [KIIK08] to fully recover the secret key d from a target ECDSA implementation. Suppose we obtain from the device a bunch of faulty results $(\tilde{n}_i, \tilde{r}_i, \tilde{s}_i)$, assuming also that we don't know the $\tilde{n}_i$'s.

Suppose a small prime ξ exists such that $\xi \mid \tilde{n}_i$ and $\xi \nmid d$. Suppose also that $\tilde{s}_i \equiv_\xi 0$ and $\tilde{r}_i \not\equiv_\xi 0$. Then $(e + rd)/k \equiv_\xi 0$ and, with high probability, $(e + rd) \equiv_\xi 0$ and hence $d \equiv_\xi -e/\tilde{r}_i$. For a given ξ, we can recover $d \bmod \xi$ by computing the various $-e/\tilde{r}_i$ as candidates for $d \bmod \xi$ for each admissible signature, i.e. a signature for which $\tilde{s}_i \equiv_\xi 0$ and $\tilde{r}_i \not\equiv_\xi 0$ holds.

Analyzing the frequencies of the candidates occurrences leads to discovering of $d \bmod \xi$ as described by the authors. It is then possible to reconstruct d by CRT if sufficiently many ξ's were used. The article notes that, for a common 160-bit key d, using ξ's among all primes less than 131 suffice and 250000 faulty signatures are required to recover d with high probability.

4.3.2 Nonce Recovering

In [SM09] is presented an attack which exploits instruction skips to recover some bits of the nonces k used in signature computation. These can be later used to perform lattice attacks (see [HGS01] and [NS03]) to recover the secret key d. The presented attack can be adapted to every variation of the double and add algorithm. We will now consider a left-to-right implementation.

Assume a doubling operation is skipped while computing $[k]P$, say $\tilde{Q}$ the faulty result and Q the corresponding correct one. Let l be the length in bits of the nonce k, and $l - i$ be the faulty iteration. Let also $\tilde{k} = (k_i, \ldots, k_0)_2$. It can be shown that $[2]\tilde{Q} - Q = [\tilde{k}]P$.

We now try to find $\tilde{k}$, i.e. some bits of the used nonce k.

$\tilde{Q}$ can be easily recovered from the faulty signature $(\tilde{r}, \tilde{s})$ since $x_{\tilde{Q}} \equiv_n \tilde{r}$. Being both $x_{\tilde{Q}}$ and $\tilde{r}$ less than p, and being n roughly the same size as p we can consider all the (few) possibilities for $x_{\tilde{Q}}$, and hence for $\tilde{Q}$.

To retrieve the correct point Q another way has to be followed. Let $\tilde{u}_1 = e/\tilde{s} \bmod p$, $\tilde{u}_2 = r/\tilde{s} \bmod p$. It can be shown that $Q = [\tilde{u}_1]P + [\tilde{u}_2]Y$ holds, where Y is the public key $[d]P$.

Our only unknown value is now $\tilde{k}$, which indeed is only i bits in length. For sufficiently small i we can explore the whole space and find the $\tilde{k}$ actually used in the signing process. This eventually leads to feasibility of lattice attacks to recover the whole secret key d.

The authors state that 50 signatures are sufficient to recover a common 160-bit key.

4.3.3 Known Nonce Bits

An attack based on the same underlying ideas is presented in [NNTW05]. Here the authors discuss a way to inject a fault during the random choice of k. Because of the fault, the value k is not truly random, but it is known to have some bits always set to 0 in the lowest positions.

An important characteristic of the attack is the maintained validity of the output signature, even in faulty conditions. This happens because the same

nonce k, being it really random or faulty, is used throughout the whole algorithm. This is both an advantage and a disadvantage from the attacker's point of view.

It is advantageous because the device cannot simply discard invalid signatures to counter this attack. It is anyway also a disadvantage because the attacker must develop some procedure to identify the signatures that were produced after an occured fault. This is not simply achievable by using only the resulting signatures. The authors indeed propose to monitor the power consumption of the device to evaluate the time passed before cryptographic operations start. When this time differs from the average the corresponding signature is taken to be derived from a faulty nonce.

If a number of signatures are collected and correct estimations of the faulty nonces are made, it is possible again to employ the so-called lattice attack to find the secret key, as already showed in the previous section.

4.4 Final Considerations

The presented attacks cover a wide range of possiblities and performance, as well as ideas to break a given algorithm or cryptosystem. We can sketch here a resume of what we can understand by studying these attacks.

Great computational effort Attacks are often feasible theoretically, needing only subexponential time or amount of faulty signatures. However, this does not always correponds to practical feasibility.

Check of parameters If a device does not check for the parameters it is given to be valid, then it becomes easy to cheat the device and make it perform computations that can lead to secret leakage. See for example the attack in section 4.2.1. Anyway, this method is not much suited to every elliptic curve based protocol, because it wants the algorithm to use points or some variable recevied from exteranl world. This is not the case for example in ECDSA, where the algorithm receives only the hash to be signed and generates all the other values at random or has it memorized internally. Moreover, the countermeasure is very cheap and indeed it is very unlikely to be find an device unprotected with respect to this kind of attacks.

Scalar multiplication Some attacks are not difficult to perform on the primitive scalar multiplication. It can happen that attacks lead to move an original ECDLP problem to a smaller or weaker curve, thus letting it to be solved more easily. It can however become unclear how to adapt this to attack more complicated algorithms, like ECDSA, that employ more operations to produce their output. Anyway, the scalar multiplication is the most characterizing operation of elliptic curve based algorithms, and therefore it is the main target of all the presented attacks.

Very precise fault If the fault is injectable with a high level of precision, e.g. a given variable can be altered in a known way for sure, then it is quite easy to recover some secret, for example by cheking whether the injected fault was effective or not, or whether the resulting signature was valid or not. Attacks of this type are described in the sections 4.1.3 and the 4.1.4. This

however has implications of large costs to generate the fault accordingly to the hypotheses, since it is necessary to repeatedly fault every bit or portion of the secret that is sought for. This is a quite wide concept, the more precise knowledge of the fault and more precise the control on space and time is, the more easy an attack becomes.

The most interesting attacks to ECDSA are the ones reported in sections 4.3.1, 4.3.2 and 4.3.3, since they target the whole signature scheme. However they also have some drawbacks. Namely we either need a large amount of faulty signatures (attack 4.3.1) or a fault model quite difficult to obtain in practice (attack 4.3.3). Attack 4.3.2 is instead quite practical and not much greedy of resources.

In the subsequent chapters we will show how to develop a different attack, with some similar aspects to the one in section 4.3.2, to obtain the possiblity to exploit faults in a different operation. Instead of attacking the scalar multiplication, normally considered in the presented attacks, we will move to a commonly less considered and protected part of the algorithm. The performance of our attack remains of the same order of magnitude of attack 4.3.2. Hoewever, the attack target is different and the associated fault and error model is not difficult to obtain.

Chapter 5

Proposed Fault Attack

This chapter introduces the actual attack procedure. We start describing the funding ideas that lead us to develop the attack, considering its greatest innovation point, i.e. the possibility to perform an attack by inducing a fault into a different position of the algorithm than the scalar multiplication. Then we show an *overview* of the whole attacking procedure, from the analysis of the collected results to the discovery of the secret key used by the attacked device.

A deeper insight to the *fault* analysis is then shown. We start considering a suitable hypothesis for the induced fault and show how this impacts on the final results. Each possibility is taken into account, leading to an overall algebraic description of the faulty results, which will be the ones collected from the real device computations.

The possible faults are then again considered to explain how to recover some *intermediate values*. Each faulty result is indeed carrying information about some variable not directly accessible from the outside of device. Getting to know these variables is a starting point to reconstruct the whole *secret key*.

One more point is the possibility to use the *logarithm search procedure*. This is an important factor to achieve reasonable and feasible performance of the mentioned analysis. It is derived from a rewriting of the signature equations that, through adapting an algorithm to compute discrete logarithms, exploits a favorable property of the variables to effectively speed up the analysis.

Once these hidden variables are discovered it becomes possible, through the use of more procedures, to use them to reconstruct the whole secret key. Such procedures are then detailed with a deep analysis of every issue that can show up.

It must be kept in mind that all the procedures described are a general idea of development for the attack and have to be tuned to the specific conditions and results to obtain the maximum performance. The factors that influence such performance are highlighted and commented in the next sections.

Eventually, we also consider some ways to possibly extend the work to more complicated fault models than the one considered, since we focused only on a very simple model. Two ideas are sketched together with a few results related to them.

5.1 Notation and Conventions

We present a brief summary of conventions used in the chapter. The purpose of naming conventions is to keep a certain level of consistency between different sections to help reading. Each variable will be explained in detail in the text, when it is introduced. This list should serve only as a quick reference for the large amount of variables used in the sequel.

Sets	
$\mathbb{Z}$	the integer ring
$\mathbb{Q}$	the rational field
$\mathbb{Z}_n$	the quotient $\mathbb{Z}/n\mathbb{Z}$
$\mathbb{Z}_p$	basic finite field of characteristic p
$\mathbb{Z}_p^* = \mathbb{Z}_p \setminus \{0\}$	the multiplicative subgroup of $\mathbb{Z}_p$
$\mathbb{E}/\mathbb{Z}_p$	an elliptic curve over the field $\mathbb{Z}_p$
Operator Notation	
$1/k$	finite field multiplicative inverse k^{-1}
$[k]P$	elliptic curve scalar multiplication
Elliptic Curves	
$\mathcal{O}$	point at infinity of an elliptic curve
$P = (x_P, y_P)$	a point on a curve $\mathbb{E}$
$\lvert\mathbb{E}\rvert$	order of the curve
$\lvert P\rvert$	order of a point on the curve
$G : \langle G\rangle = \mathbb{E}$	the generator of the curve $\mathbb{E}$
Integers	
p	a large prime
$n = \lvert G\rvert = \lvert\mathbb{E}\rvert$	prime order of (the generator of) a curve
ECDSA	
d	private key
$Y = [d]G$	public key
e	hash of the signed message
k	nonce
$r = x_{[k]G}$	first signature value
$s = (e + rd)/k$	second signature value
$u_1 = e/s$	first verification scalar
$u_2 = r/s$	second verification scalar
$T = u_1G + u_2Y$	verification point
Faults	
$\tilde{z}$	faulty value of variable z
$\varepsilon = \tilde{z} - z$	error characterizing a faulty variable in additive form
Indexes and Lengths	
l	a variable's length in bits
σ	a variable's length in words
w	the device's word length in bits
i, j	loop iteration indexes
$\bar{\imath}, \bar{\jmath}$	iteration index of the faulty computation
$\lambda = \bar{\imath} + \bar{\jmath}$	fault exponent
μ	word-wise faulty bit index

5.2 Attack Rationale

Almost every known attack against elliptic curve cryptosystems mainly targets the scalar multiplication, commonly refered to as the $[k]P$ operation. It actually seems quite natural to try modifying in some extent the points or the curve during the computation, since this is the heaviest and most characterizing operation. It has to be noted however that in ECDSA the scalar multiplication does not involve the secret key.

When a fault is targeting the scalar multiplication, the resulting signature is typically not valid because the nonce is used again during the computation of s. The only way to have in output a valid signature is to modify the nonce k in a way that affects both the scalar multiplication and the computation of s. This is actually done in the attack proposed in [NNTW05], but the fault model they propose is not easy to realize in practice.

Moreover, by faulting the scalar multiplication it is not easy to understand the kind of fault or where it happened, since a faulty $\tilde{r}$ represents some sort of random point on the curve. In other words, a strong a-priori knowledge of the injected fault must exist. The common way to find some relation binding the faulty $\tilde{r}$ to the real one, $[k]G$, is to know to some extent how a faulty $\tilde{k}$ can be used to represent the fault. To understand this, consider for example a fault that can be represented as $\tilde{k} = k - 2^j$. In this case, the output $\tilde{r}$ will be related to the expected correct value as it is the x-coordinate of the point $\tilde{P} = [\tilde{k}]G = [k - 2^j]G = [k]G - [2^j]G = P - [2^j]G$. The output of faulty computations can usually be checked against this kind of relations to help finding the occured fault. It is clear that a random fault during the scalar multiplication can not be easily recognized and identified.

These observations lead to think of a fundamentally different type of attack. We do not affect the scalar multiplication, willing to hit the computation of s instead. This way, we obtain an r value which is still correct w.r.t. the nonce k. A problem has to be faced up to, since every time a signature is generated, a different nonce is used and the resulting signature is different. This makes impossible to study the comparison of corresponding correct and faulty computation. Therefore, we will rely on r as the only source of information. In some occasions we will even manage to find the correct s starting from a faulty one thanks to the knowledge of r. The correctness of r allows us to relax the hypotheses about the knowledge of the fault. We will indeed be able to determine whether the fault was of the foregone type or not, and even to localize it with some precision, doing computations starting from the output values.

The operations involved in the considered computation are modular multiplications, inversions and additions. We focus our attention on the multiplication operations because those are the operations involving the most interesting values, i.e. d and k.

5.3 Attack Overview

Let's now introduce an overall description of the proposed attack. The fault model we consider assumes the fault happening during one of the two multiplications constituting the final stage, i.e. the computation of the value s, reported in step 5 of algorithm 5. We will show in detail the differences arising from the

two cases in the following sections. The fault is also supposed to hit only one word of one of the factors. In other words it hits one of the variables being multiplied at step 5 of algorithm 10, or at step 6 of algorithm 11.

The last condition we impose is the fault being a *bit flip*. This means the operand gets one bit of its representation changed from 0 to 1 or vice-versa. This error model is quite realistic, see for example [BBPP09]. We can express such an error as $a \rightarrow a \oplus 2^{\mu}$, however, we will use a different representation to make it possible to analyze it algebrically. Given these hypotheses, we derive a representation of the output faulty s value and analyze it to get back informations about the multiplication involved during the fault.

We have to go through two main steps in order to get the whole secret key. Namely, our attacks can be split in

1. word retrieval,

2. key reconstruction.

While the fault hits a factor of one multiplication step, we will be able to retrieve information about the other factor being multiplied by the faulty one. This factor is actually a word of one of the multiple precision values involved. In the best case we are able to retrieve the whole word. We will see two techniques to perform this retrieval:

- exhaustive search,

- logarithm extraction.

The exhaustive search is a natural and simple method. The logarithm computation is made possible by some rewritings of the signature equations, and provides a great performance improvement. Moreover, this kind of search will give important information on whether the introduced fault was of the foregone type or not, by simply failing to find a solution.

Once we recover a word, we will be able to use it to reconstruct the secret key. In this case too, we have different ways to perform the operation, depending on which multiplication and operand were affected by the fault:

- direct key words permutation,

- lattice attack.

This last stage of the attack has quite good performance, and it is possible to tune the algorithms depending on the actual data.

5.4 Faulty Computation Analysis

Let us now consider the evolution of the algorithm when it is affected by the fault. We will obtain a description of the outputs of the device resulting from the given error.

5.4.1 Bit Flip

As described in section 5.3, we suppose a single bit flip fault model. Introducing a bit flip means changing one bit of a given value. A formal description of the fault is

$$a \rightarrow a \oplus 2^{\mu}$$

It is however too complicated to handle such a representation of the error, since we are going to deal with operations in integer ring arithmetic. We therefore use a different equation to represent the same fault. The notation we will use in the following to denote a faulty value is:

$$\tilde{a} = a + \varepsilon$$

This notation allows us to handle the arithmetic of the fault, at the cost of a sign indetermination. We use the variable ε to generically denote an additive error value.

$$\varepsilon = \pm 2^{\mu}$$

Such a value must be interpreted as follows:

- $\mu \in [0, w-1]$ is the flipped bit position in the w-bit word,
- the $\pm$ takes care of the fault direction, i.e. whether it is a flip-up ($0 \rightarrow 1$, sign $+$) or a flip-down ($1 \rightarrow 0$, sign $-$).

This kind of fault can happen for example as a consequence of a faulty load operation, or for some modification in a register.

5.4.2 Fault Propagation in the Integer Multiplication

We now show how an injected fault propagates through the multiplication and analyze the resulting values. Note that the operands are referenced only in step 5 of algorithm 10 or in step 6 of algorithm 11. We suppose that the fault happens when accessing one of those values during *one* iteration of the algorithm.

The algorithm can be rewritten for better understanding in compact form as

$$c = \sum_{i=0}^{\sigma-1} \sum_{j=0}^{\sigma-1} a_j b_i 2^{(i+j)w}$$

We refer to the multiplications as $t_{ij} = a_j b_i$. Thus obtaining

$$c = \sum_{i=0}^{\sigma-1} \sum_{j=0}^{\sigma-1} t_{ij} 2^{(i+j)w}$$

Consider the iteration $(\bar{\imath}, \bar{\jmath})$ of the algorithm, thus specifically the multiplication $a_{\bar{\jmath}} b_{\bar{\imath}}$ and a bit flip fault on $a_{\bar{\jmath}}$, yielding $\tilde{a}_{\bar{\jmath}} = a_{\bar{\jmath}} + \varepsilon$. This is without loss of generality, for the simmetry of the algorithm w.r.t. the two operands. The result of the faulty multiplication is

$$\tilde{t}_{\bar{\imath}j} = \tilde{a}_{\bar{\jmath}} b_{\bar{\imath}}$$

$$\tilde{t}_{\bar{\imath}j} = (a_{\bar{\jmath}} + \varepsilon) b_{\bar{\imath}}$$

$$\tilde{t}_{\bar{\imath}j} = a_{\bar{\jmath}} b_{\bar{\imath}} + \varepsilon b_{\bar{\imath}}$$

and, refering to the correct result,

$$\tilde{t}_{\bar{\imath}j} = t_{\bar{\imath}j} + \varepsilon b_{\bar{\imath}}$$

The whole multiplication thus runs as

$$c = \sum_{i=0}^{\sigma-1} \sum_{j=0}^{\sigma-1} t'_{ij} 2^{(i+j)w}$$

where

$$t'_{ij} = \begin{cases} \tilde{t}_{ij} = t_{ij} + \varepsilon b_i & \text{if } (i,j) = (\bar{\imath}, \bar{\jmath}) \\ t_{ij} & \text{otherwise} \end{cases}$$

The faulty iteration $(\bar{\imath}, \bar{\jmath})$ intervenes in that it influences which word $b_{\bar{\imath}}$ gets propagated and also the alignment of the fault in the final result. In effect, we can write the final result as

$$\tilde{c} = \sum_{i} \sum_{j} t'_{ij} 2^{(i+j)w}$$

$$\tilde{c} = \sum_{i \neq \bar{\imath}} \sum_{j \neq \bar{\jmath}} t_{ij} 2^{(i+j)w} + \tilde{t}_{\bar{\imath}\bar{\jmath}} 2^{(\bar{\imath}+\bar{\jmath})w}$$

$$\tilde{c} = \sum_{i \neq \bar{\imath}} \sum_{j \neq \bar{\jmath}} t_{ij} 2^{(i+j)w} + t_{\bar{\imath}\bar{\jmath}} 2^{(\bar{\imath}+\bar{\jmath})w} + \varepsilon b_{\bar{\imath}} 2^{(\bar{\imath}+\bar{\jmath})w}$$

$$\tilde{c} = \sum_{i} \sum_{j} t_{ij} 2^{(i+j)w} + \varepsilon b_{\bar{\imath}} 2^{(\bar{\imath}+\bar{\jmath})w}$$

$$\tilde{c} = c + \varepsilon b_{\bar{\imath}} 2^{(\bar{\imath}+\bar{\jmath})w}$$

Remembering the definition of the bit flip $\varepsilon = \pm 2^{\mu}$, we can write the final expression for the faulty product

$$\tilde{c} = c \pm b_{\bar{\imath}} 2^{(\bar{\imath}+\bar{\jmath})w+\mu} \tag{5.1}$$

We can see from (5.1) that the result of the faulty product is again writable as the correct value plus an additive error. Such an error is composed of the word $b_{\bar{\imath}}$ together with a shifting factor $2^{(\bar{\imath}+\bar{\jmath})w+\mu}$. As we will see in the sequel, we will be able to retrieve information about the interested word $b_{\bar{\imath}}$ and about $\lambda = \bar{\imath} + \bar{\jmath}$.

5.4.3 Fault Propagation in the Whole Signature

We saw how an integer multiplication gets modified if the supposed fault happens. Although, the actual operations used in ECDSA computations are modular multiplications. The algorithm usually employed for modular multiplications in the final stage of ECDSA is quite simple, it barely consists of an integer multiplication followed by the appropriate modular reduction. We cite the well known library OpenSSL [Ope] as an example of use of this algorithm. In this

scenario, we don't need special care to handle our description of the fault. If $c = ab$ and $z = ab \bmod n$, it in effect holds

$$\tilde{z} = \tilde{c} \bmod n$$

$$\tilde{z} = c \pm b_{\bar{\imath}} 2^{(\bar{\imath}+\bar{\jmath})w+\mu} \bmod n$$

Let's consider the actual multiplications involved in ECDSA. Namely, in step 5 of algorithm 5. Such operations are rd and $k^{-1}(e + rd)$.

Suppose the fault hits the multiplication $c = rd$, and specifically a word of r. Then, denoting by λ the sum $\bar{\imath} + \bar{\jmath}$, the result can be written as

$$\tilde{c} = rd \pm d_{\bar{\imath}} 2^{\lambda w+\mu} \bmod n$$

This leads to the output

$$\tilde{s} = k^{-1}(e + \tilde{c}) \bmod n$$

$$\tilde{s} = k^{-1}e + k^{-1}\tilde{c} \bmod n$$

$$\tilde{s} = k^{-1}e + k^{-1}(rd \pm d_{\bar{\imath}} 2^{\lambda w+\mu}) \bmod n$$

$$\tilde{s} = s \pm k^{-1} d_{\bar{\imath}} 2^{\lambda w+\mu} \bmod n \tag{5.2}$$

Suppose the fault hits the multiplication $s = k^{-1}(e + rd)$ instead, and in particular a word of k^{-1}. It then holds

$$\tilde{s} = s \pm b_{\bar{\imath}} 2^{\lambda w+\mu} \bmod n \tag{5.3}$$

where $b = e + rd \bmod n$.

5.5 Exhaustive Search

We show how to recover the word $b_{\bar{\imath}}$ involved in the error affecting the output value s. The notable point is that the admissible errors are few in number, compared with the key space size. While the key is typically a few hundreds bits long, the error we are looking for is only w bits long. Common devices manage words of 32 bits, but other devices could use words as short as 16 or even 8 bits. This allows to make hypotheses on the fault and subsequently test every possible value, according to the hypotheses, thus recovering the sought data.

Let's consider the case, shown in section 5.4, of the fault happening during the last multiplication. We saw it holds

$$\tilde{s} = s \pm b_{\bar{\imath}} 2^{\lambda w+\mu} \bmod n$$

where we again denote with b the computed value $e + rd \bmod n$. Obviously, $\tilde{s}$ makes the signature verification fail. We can therefore guess the error part ε and verify the correctness of our guess by applying the verification algorithm to the value $\hat{s}$ computed as

$$\hat{s} = \tilde{s} - \varepsilon \bmod n = \tilde{s} \mp b_{\bar{\imath}} 2^{\lambda w+\mu} \bmod n$$

There is only one correct s, and therefore only one ε that leads to the verification succeeding.

The error ε is made up by the following elements:

- the faulty multiplication loop iteration $\bar{\imath}$,
- the faulty word $\bar{\jmath}$,
- the faulty bit position within the word μ,
- the direction of the bit flip,
- the operand word $b_{\bar{\imath}}$.

The complexity of the search process can therefore be expressed as the number of signature verifications to be made to find the error ε. Any a-priori knowledge of one or more of the previous elements reduces the search space for ε, thus improving the search process performance.

In the worst case, i.e. if no information is available, the information needed for the error retrieval is equivalent to

- $\lambda = \bar{\imath} + \bar{\jmath}$: $\log(2\sigma - 1)$ bits,
- μ: $\log w$ bits,
- sign: 1 bit,
- $b_{\bar{\imath}}$: w bits.

For example, for a common 256-bits key, and 32-bits device, we have

$$\sigma = l/w = 8$$

and a number N of needed verifications

$$N = 15 \cdot 32 \cdot 2 \cdot 2^3 2 \approx 2^{4+5+1+32} = 2^{42}$$

Suppose we have some informations about the fault instead, like knowing precisely the hit iteration $(\bar{\imath}, \bar{\jmath})$ and the flipped bit position μ, in this case we have to test only $N = 2^{32}$ signatures.

As can be seen, the heaviest contribute is given by the sought word $b_{\bar{\imath}}$. In either case, the procedure is fully parallelizable, since each signature verification can be carried out by a single processor.

Let's now analyze the case the fault hit the first multiplication instead, i.e. rd. It is not easy as before to guess the error affecting the final $\tilde{s}$, but we show how to use the verification algorithm to retrieve the error in this case too. The idea is based on the attack decribed in [BDH+98].

Consider the fault $r_{\bar{\jmath}} \rightarrow r_{\bar{\jmath}} \pm 2^{\mu}$ during the iteration $\bar{\imath}$. As seen before (equation (5.2)) this time the error is made up by the inverse of the nonce too, k^{-1}.

$$\tilde{s} = s \pm k^{-1} d_{\bar{\imath}} 2^{\lambda w + \mu} \bmod n$$

Remembering the signature verification algorithm presented in section 2.4, we can write the following, refering to the faulty $\tilde{s}$

$$\tilde{s} = s + \varepsilon$$

$$\tilde{s} = \frac{e + rd}{k} \pm \frac{d_{\bar{\imath}} 2^{\lambda w + \mu}}{k}$$

$$k = \frac{e + rd}{\tilde{s}} \pm \frac{d_{\bar{\imath}} 2^{\lambda w + \mu}}{\tilde{s}}$$

$$k = \frac{e}{\tilde{s}} + \frac{rd}{\tilde{s}} \pm \frac{d_{\bar{\imath}} 2^{\lambda w + \mu}}{\tilde{s}}$$

multiplying by the generator G of the curve we obtain

$$[k]G = \left[\frac{e}{\tilde{s}}\right] G + \left[\frac{r}{\tilde{s}}\right] Y \pm \left[\frac{d_{\bar{\imath}} 2^{\lambda w + \mu}}{\tilde{s}}\right] G \tag{5.4}$$

We note that in equation (5.4) the only unknown value is $\pm d_{\bar{\imath}} 2^{(\bar{\imath}+\bar{\jmath})w+\mu}$. This is because, although k is not known, we have access to r, which carries information about the x-coordinate of the point $[k]G$, being $r = x_{[k]G} \bmod n$. From this consideration we derive the possibility of an exhaustive search over the unkonwn variables. Guessing the correct values lets equation 5.4 to hold. It can be noted that the degrees of freedom for the search are the same as the case seen above.

For the nature of the verification step, which does not involve directly the scalar s but performs a comparison of point coordinates instead, it happens that the values leading to a correct verification may be not unique. Anyway, empirical tests made evidence that, for a curve of size suited for cryptographic applications, false positives never happen.

In both cases (either multiplication affected), the search procedure reveals the value $b_{\bar{\imath}} 2^{\lambda w+\mu}$ or $d_{\bar{\imath}} 2^{\lambda w+\mu}$. From now on, we will refer only to $b_{\bar{\imath}}$ for brevity, remembering that the same holds for $d_{\bar{\imath}}$ too. They remain two kinds of uncertainty:

- we do not know the alignment of the word, since we can not tell which power of two is a factor of $b_{\bar{\imath}}$ and which contributes in μ,
- we know only the sum $\lambda = \bar{\imath} + \bar{\jmath}$ and can not tell the separate values $\bar{\imath}$ and $\bar{\jmath}$.

We show how to deal with these issues in the next sections.

A minor point has also to be discussed. Namely, in the very unlikely case ($P \approx 2^{-w}$) that the word $b_{\bar{\imath}}$ is 0, it can be noted that there is no error on the output, thus making it not possible to retrieve this word.

5.5.1 Word Alignment

As stated above, given the retrieved value $b_{\bar{\imath}} 2^{\lambda w+\mu}$, we can not a-priori know which power of two is a factor of the value $b_{\bar{\imath}}$ and which amount have to be considered in μ.

Consider the example below.

- $w = 8$
- $b_{\bar{\imath}} = (01101010)_2$
- $\mu = 3$

The recovered data is therefore $(01101010)_2 \cdot 2^{\lambda 8} \cdot 2^3$. We can not distinguish among these cases:

- $(01101010)_2 \cdot 2^{\lambda 8} \cdot 2^3$

- $(11010100)_2 \cdot 2^{\lambda 8} \cdot 2^2$
- $(00110101)_2 \cdot 2^{\lambda 8} \cdot 2^4$

The information recovered from the search process is equivalent to an odd integer b' and a power of two $2^{\lambda w+\mu+h}$, such that it holds

$$b_{\bar{\imath}} 2^{\lambda w+\mu} = b' 2^{\lambda w+\mu+h}$$

It is not possible to distinguish between different admissible values of h. From the equation above we can write

$$b_{\bar{\imath}} = b' 2^h$$

We know that $b_{\bar{\imath}}$ is at most w bits in length, i.e. it holds $b_{\bar{\imath}} < 2^w$. Therefore it is possible to put the bound $b'2^h < 2^w$. Besides this we can not discern the real value of h.

An interesting value for the subsequently described attack is the expected value of the uncertainty we have over the shift of the word. We want to give a measure of how much a word can be shifted, i.e. how many possible retrieved words correspond to a certain value. We consider that a word can be shifted at most k positions to the right if it has k many trailing 0's, and k positions to the left if k many leading 0's are present. As an example, if the word $b_{\bar{\imath}} = (z_{w-1} \cdots z_0)_2 = (001 \cdots 10)_2$ is being used in computations, it can be retrieved as 4 different possible shifted values. Supposing the word is evenly distributed in $[0, 2^w - 1]$, said k the number of possible shifts of the word, the expected value of k is $E[k] \approx 3$.

5.5.2 Word Position

The second mentioned problem is the uncertainty about the retrieved word index $\bar{\imath}$. We in fact may know only the sum $\lambda = \bar{\imath} + \bar{\jmath}$. The indexes $\bar{\imath}$ and $\bar{\jmath}$ can assume values in the set $[0, \sigma - 1]$, then $\lambda \in [0, \ldots, 2\sigma - 2]$.

Table 5.2: Fault on given iteration

$\bar{\imath}$ (fault)	λ (possible result)
0	$0, \ldots, \sigma - 1$
1	$1, \ldots, \sigma$
$\vdots$	$\vdots$
$\sigma - 2$	$\sigma - 2, \ldots, 2\sigma - 3$
$\sigma - 1$	$\sigma - 1, \ldots, 2\sigma - 2$

We show in table 5.2 how the sought index $\bar{\imath}$ is related to the retrieved λ. Briefly, if the fault happens on iteration $(\bar{\imath}, \bar{\jmath})$, it follows that $\lambda \in [\bar{\imath}, \ldots, \bar{\imath} + \sigma - 1]$. This set is only half as big as the whole domain of λ, i.e. $[0, \ldots, 2\sigma - 2]$.

In table 5.3 we show the same information, in a reversed form. In other words, we show the possible values for $\bar{\imath}$ which could have generated a given retrieved λ. We note that the case $\lambda = \sigma - 1$ is associated with the maximum uncertainty, while for $\lambda = 0$ and $\lambda = 2\sigma - 2$ there is no uncertainty at all, since we know that $\bar{\imath}$ is respectively 0 or $\sigma - 1$.

Table 5.3: Faulty iteration retrieval

λ (retrieved)	$\bar{\imath}$ (possible source)
0	0
1	0, 1
2	0, 1, 2
⋮	⋮
$\sigma-2$	$0, 1, 2, \ldots, \sigma-2$
$\sigma-1$	$0, 1, 2, \ldots, \sigma-2, \sigma-1$
σ	$1, 2, \ldots, \sigma-2, \sigma-1$
$\sigma+1$	$2, \ldots, \sigma-2, \sigma-1$
⋮	⋮
$2\sigma-3$	$\sigma-2, \sigma-1$
$2\sigma-2$	$\sigma-1$

5.6 Logarithm Search

We now propose a way to achieve a great performance improvement in the search procedure. Namely, we see how it is possible to compute the sought word as an elliptic curve discrete logarithm.

The presented ideas are based on the observation that, even if the nonce k is clearly not known outside the device, it is possible to have access to informations about it through the value r. This is where the importance of having a correct computation of this value becomes evident.

The value r is bound to the x-coordinate of the point $[k]G$ through a modular reduction: $r = x_{[k]G} \bmod n$. We can anyway make a guess on $[k]G$. For equation 2.2, we know that the possible values for $x_{[k]G}$ are at most 2. Admissibles values for the coordinate are r and possibly $r+n$. For each possible x-coordinate, we have 0 or 2 corresponding points, because of the symmetry of the elliptic curve defining polynomial 2.1 w.r.t. the y-coordinate. Moreover, said x the guessed coordinate, it must be checked also whether the value $x^3+a_4x+x_6$ is a quadratic residue modulo n. In case the resulting value is not a quadratic residue, then the coordinate has to be discarded.

Summarizing, the possible elliptic curve points arising from a given r are as follows:

$$\begin{array}{cc} (r, y_1) & (r, -y_1) \\ (r+n, y_2) & (r+n, -y_2) \end{array}$$

where y_1 satisfies $y_1^2 = r^3 + a_4 r + a_6 \bmod n$ and $y2$ satisfies $y_2^2 = (r+n)^3 + a_4(r+n) + a_6 \bmod n$. Clearly, if r is coming from a correct computation $[k]G$, at least one of the above pairs of points must exist, i.e. at least one of r and $r+n$ must yield a quadratic residue. There is no way of determining which of these 2 or 4 points is the correct $[k]G$, therefore we will have to repeat the following attack procedure up to 2 or 4 times, until a solution is found. We will refer to the considered hypothesis as the point R, and we intend the following to be implicitly repeated for every candidate R.

Let's analyze separetely the two possible situations arising from a fault on the two different multiplications that can be affected by the fault.

Suppose the fault occurs on the multiplication rd. We already saw the expression for the output $\tilde{s}$ value

$$\tilde{s} = s \pm k^{-1} d_{\bar{\imath}} 2^{\lambda w + \mu}$$

We know also how to rewrite the verification algorithm to express a way of checking hypotheses on $d_{\bar{\imath}}$, from equation (5.4)

$$[k]G = \left[\frac{e}{\tilde{s}}\right] G + \left[\frac{r}{\tilde{s}}\right] Y \pm \left[\frac{d_{\bar{\imath}} 2^{\lambda w + \mu}}{\tilde{s}}\right] G$$

It is possible to further manipulate this equation to obtain

$$\pm \left[\frac{d_{\bar{\imath}} 2^{\lambda w + \mu}}{\tilde{s}}\right] G = [k]G - \left[\frac{e}{\tilde{s}}\right] G - \left[\frac{r}{\tilde{s}}\right] Y$$

$$[d_{\bar{\imath}}] \left[\frac{2^{\lambda w + \mu}}{\tilde{s}}\right] G = \pm \left([k]G - \left[\frac{e}{\tilde{s}}\right] G - \left[\frac{r}{\tilde{s}}\right] Y\right)$$

If we substitute to the unknown $[k]G$ a guess R we obtain

$$[d_{\bar{\imath}}] \left[\frac{2^{\lambda w + \mu}}{\tilde{s}}\right] G = \pm \left(R - \left[\frac{e}{\tilde{s}}\right] G - \left[\frac{r}{\tilde{s}}\right] Y\right)$$

In this form, we have evidence of a scalar multiplication involving the key word $d_{\bar{\imath}}$ as scalar, some known data on the right side, and unknown data $\lambda w + \mu$ that can assume only a small range of values. We can now compute the word $d_{\bar{\imath}}$ as logarithm using a guess for $\left[\tilde{s}^{-1} 2^{\lambda w + \mu}\right] G$ as base point, and $\pm \left(R - \left[\tilde{s}^{-1} e\right] G - \left[\tilde{s}^{-1} r\right] Y\right)$ as target point.

In case the fault occurs during the second multiplication, the equations are only slightly different. The output from the device is

$$\tilde{s} = s \pm b_{\bar{\imath}} 2^{\lambda w + \mu}$$

We can again consider the verification algorithm and write

$$\tilde{s} = \frac{e + rd}{k} \pm b_{\bar{\imath}} 2^{\lambda w + \mu}$$

$$\tilde{s} = \frac{e + rd}{k} \pm \frac{k b_{\bar{\imath}} 2^{\lambda w + \mu}}{k}$$

$$k = \frac{e}{\tilde{s}} + \frac{rd}{\tilde{s}} \pm \frac{k b_{\bar{\imath}} 2^{\lambda w + \mu}}{\tilde{s}}$$

Performing the point multiplication, we obtain

$$[k]G = \left[\frac{e}{\tilde{s}}\right] G + \left[\frac{r}{\tilde{s}}\right] Y \pm \left[\frac{b_{\bar{\imath}} 2^{\lambda w + \mu}}{\tilde{s}}\right] [k]G$$

$$[b_{\bar{\imath}}] \left[\frac{2^{\lambda w + \mu}}{\tilde{s}}\right] [k]G = \pm \left([k]G - \left[\frac{e}{\tilde{s}}\right] G - \left[\frac{r}{\tilde{s}}\right] Y\right)$$

We again can substitute the R point, leading to

$$[b_{\bar{\imath}}] \left[\frac{2^{\lambda w + \mu}}{\tilde{s}}\right] R = \pm \left(R - \left[\frac{e}{\tilde{s}}\right] G - \left[\frac{r}{\tilde{s}}\right] Y\right)$$

In this case too, it appears clear how to perform a logarithm computation to obtain the sought word $b_{\bar{\imath}}$.

In both the cases shown above we wrote our problem in the form of an ECDLP. Our writings anyway expose a special case of the problem, since there is a-priori information about the scalar factor. Namely, we know in advance that such factor is very small if compared with the order n. Actually, our sought value is only w bits long. This makes it possible to perform the logarithm computation in tractable time. We can in effect adapt the baby-step-giant-step algorithm saw in section 2.2 to this special case. Recall equation (2.4)

$$k = \alpha m + \beta$$

Since we know that $k < 2^w$, we just have to tune m and the bounds on α and β to restrict the search phase to interesting values. This makes the logarithms computable, see section 6.1 for details.

5.7 Key Reconstruction

It is now possible to use the words retrieved through the search process described above to compute the whole secret key. It happens again that the fault position influences the procedure needed to recontruct the key. We now show how to proceed in the two situations.

5.7.1 Direct Key Words

We analyze the easier case first, i.e. if the fault occured on the first multiplication rd. In this case we know the recovered word is directly one word of the key, $d_{\bar{\imath}}$. Clearly, the attack should be repeated a number of times to gain access to all the various words. The questions to face up to are the ones mentioned in section 5.5. If no hypotheses are made, we would have to discover the key only by trying every value and check whether the guess $\hat{d}$ satisfies $[\hat{d}]G = Y$. This means trying all the permutations of the σ retrieved words, possibly considering a different alignment for each word, as described above.

The number of scalar multiplications needed to make this verification can quickly explode and become unfeasible.

Anyway, it is possible to use the information from the retrieved λ to sort the words in the correct way without the need for scalar multiplications.

Remember from what was shown in table 5.3 that there are cases for which we have no uncertainty, i.e. if $\lambda = 0$ or $\lambda = 2\sigma - 2$. These can be used to put apart the words d_0 and $d_{2\sigma-2}$. After that it is possible to proceed with the same consideration but on a problem of smaller size. Indeed, once the words d_0 and $d_{2\sigma-2}$ are known, the values $\lambda = 1$ and $\lambda = 2\sigma - 3$ carries no more uncertainty too and it becomes possible to know also which words belong to the 1-st and $(\sigma - 2)$-th position. It is possible to continue this way until the full ordering of the key words is known.

5.7.2 Lattice Attack

In case the second multiplication is affected by the fault, the method to obtain the key is slightly more complicated. Since we came to know a word $b_{\bar{\imath}}$, the

immediate consideration is indeed the following

$$e + rd \bmod n = \alpha_1 2^{(\bar{\imath}+1)w} + b_{\bar{\imath}} 2^{\bar{\imath}w} + \alpha_0$$

where the $\alpha_{0,1}$ represent the unknown part of $e + rd$.

It is not possible to derive some bits of the key from the relation above. Informally, this is because of the modular reduction $\bmod n$ involved, which causes the high bits coming from the product rd to scramble the rest of the number. What is needed is a number of equations of this form. Having more datas it will become possible to compute the unknown d involved in the equations.

Since r changes at every execution of the signature algorithm, it is easy to have the needed datas. Moreover, it should be pointed out that for the following attack to be more efficient, it would be useful that the equations come from the case $\bar{\imath} = 0$. For more considerations on this, please see [NS03]. This is not an issue anyway, since we can suppose to hit the wanted word with probability $P = 1/\sigma$, and we are also able to identify the good cases by looking at the λ's. In other words, it can be necessary to repeat the attack σ times, and to keep only the results for which it holds $\lambda = 0$.

In this scenario, the equation above becomes

$$e + rd \bmod n = \alpha 2^w + b_{\bar{\imath}}$$

The important fact to note is that α is smaller than n, actually being only $l - w$ bits long. We can rewrite this, as

$$\frac{e + r}{2^w} d \bmod n = \frac{b_{\bar{\imath}}}{2^w} + \alpha \bmod n$$

$$\frac{r}{2^w} d \bmod n = \frac{b_{\bar{\imath}} - e}{2^w} + \alpha \bmod n \tag{5.5}$$

The problem, put in this form, is known in mathematics literature as *hidden number problem*, and its relationship with the DSA family of algorithms is deeply analyzed in [HGS01] and [NS03].

We limit ourselves to a brief description of the problem. The right side of equation 5.5 can be considered an *approximation* of the left side because, as we saw above, α is small w.r.t. to the rest of the numbers involved. Thus, a system of equations like 5.5 is a linear system with approximated known terms. This can be reconduced to a *closest vector problem* in an integer *lattice*, whence the attack name. The solution of the system is simply the unknown d. We managed to retrieve the secret key in this case too.

To address the problem of word alignment (see section 5.5), we have two main ways to proceed. The actual choice should be a combination of the two to obtain the best performance, considering the cost of the word retrieval.

It is possible to try the lattice attack more than once, using every possible alignment of the words. Or it is possible to discard some values, keeping only the certain ones, i.e. the words of the type $(1 \cdots 1)_2$. We know that r and thus $e + rd$ change at every execution, and the retrieved word $b_{\bar{\imath}}$ can be considered to be evenly distributed in $[1, 2^w - 1]$, therefore the discussed type of word appears with probability $P \approx 1/4$.

5.8 More Fault Models

Let's now consider a few different fault models, to see how the attack ideas can be extended or adapted in different scenarios.

Because of the nature of the integer multiplication, if the fault gets more complicated than the base case of single bit flip, the error increases in size and becomes dependent on more data, thus increasing also the computational effort needed to perform the search procedures. The cases leading to a still simple description are shown below.

In the following sections we sketch the ideas that can be used to perform an attack, without describing in deep detail the problems that can arise.

5.8.1 Word Set to 0

Consider a word of a factor being set to 0 before intervening in the multiplication. Said a and b the two factors,

$$c = ab = \sum_{i=0}^{\sigma-1}\sum_{j=0}^{\sigma-1} a_j b_i 2^{(i+j)w}$$

we obtain, as a consequence of the fault $\tilde{t}_{\bar{i}\bar{j}} = 0$,

$$\tilde{c} = \sum_{i=0}^{\sigma-1}\sum_{j=0}^{\sigma-1} t_{ij} 2^{(i+j)w} - a_{\bar{j}} b_{\bar{i}} 2^{(\bar{i}+\bar{j})w}$$

$$\tilde{c} = c - a_{\bar{j}} b_{\bar{i}} 2^{(\bar{i}+\bar{j})w}$$

If this fault happens during the multiplication rd, we can apply the same search procedure seen above to find the value $t_{\bar{i}\bar{j}} = r_{\bar{j}} d_{\bar{i}}$. This is of course $2w$ bits long, so the logarithm extraction must be adapted to the case. The point here is that we know r, we can therefore try the integer division $t_{\bar{i}\bar{j}}/r_k$ for every $k \in [0, \sigma - 1]$. For $k = \bar{j}$ we obtain the key word $d_{\bar{i}}$. It is moreover of negligible probability that $t_{\bar{i}\bar{j}}$ is divisible by words of r other than the correct one. We can repeat this for many signatures as it has been done in the attack described in the previous sections to reveal all the words. Then it is possible to apply the same technique to discover the ordering of the words exploiting the retrieved λ's.

5.8.2 Few Changed Bits

Let's now consider a more general fault type. The only constraint we impose is again that the fault hits one iteration of the multiplication. We anyway allow a variable to be modified in an almost random way.

Suppose the fault is on a word of r, $r_{\bar{j}}$, and suppose it is modified in a completely unknown way. The computed value is

$$\tilde{c} = c + \Delta d_{\bar{i}} 2^{\lambda w}$$

where we denote with Δ the error affecting the faulty word of r:

$$\tilde{r}_{\bar{j}} = r_{\bar{j}} \pm \Delta$$

Again, we know a way to recover $\Delta d_{\bar{\imath}}$. Since the key is always the same, we can collect many signatures affected by this kind of fault, and then, for each word position $\bar{\imath}$, compute the corresponding word $d_{\bar{\imath}}$ as the greatest common divisor of the retrieved $\Delta d_{\bar{\imath}}$'s.

Suppose now the fault hits a word of d instead of r. The faulty result is then

$$\tilde{c} = c \pm \Delta r_{\bar{\jmath}} 2^{\lambda w}$$

It is no more so simple to retrieve the key, but we anyway found it is possible to get to full knowledge of the key if we know in advance that the fault modifies less than half the number of bits in the word, i.e. the number of faulty bits N satisfies $N < w/2$.

We know that we can recover, through the logarithm, the value $\Delta r_{\bar{\jmath}}$. The words $r_{\bar{\jmath}}$ are not constant, but we can exploit the fact we know r. We therefore can get to know the fault Δ that was affecting the key word $d_{\bar{\imath}}$. A Cellection of many results of this kind can be used to guess in which way the bits of the word have been modified.

The idea is to find a way to express the value Δ as a set of bit flips. If it is possible to know whether a bit flipped up or down, then it is possible to reconstruct the key. A way to reach such a goal is to write the error Δ in NAF form, i.e. as a signed digit number $(z_{w-1} \cdots z_0)$ with $z_k \in \{-1, 0, 1\}$. The NAF form is chosen because it is the signed digit representation with the lowest number of non-zero digits for a given Δ, and thus there is a good chance that this is the actual occured fault. Each digit carries the information codified as

$z_k = 0$	unmodified bit
$z_k = 1$	bit flip up
$z_k = -1$	bit flip down

This estimation of the bit flips is of course not always correct, and the limit for this idea to work is, as said above, the number of bit flips occured. If they are always less than $w/2$, simulated experiments show that a number of faulty signatures can lead to discover the involved key word. To do this, we just have to collect many informations for evey key word bit k. If, for given bit position k, a majority appears of digits $z_k = 1$ the key bit is 0, while if, conversely, there is a majority of $z_k = -1$ then the key bit is taken to be 1. Remember that it is also possible to check the recovered key by comparing $[d]G$ with the public key Y. In case it is not correct, we can modify the key hypothesis starting from the more ambiguous bits, since the difference in the amount of positive and negative digits is an index for the hypothesis quality.

The described idea leads also to consider that a less uncertain fault can be used in a similar way. In case we know in advance that the modified bits are always flipped in a known direction, i.e. always up or always down, then we can use faults on d also if they are truly random, since we no more need the NAF conversion.

It seems more difficult to apply procedures similar to the ones described in the section to faults on the multiplication $k^{-1}(e + rd)$, because in this case the operands enjoy no special properties, i.e. there is no constant or known operand.

5.9 Final Remarks

We have presented a novel method to develop a complete fault attack against the ECDSA digital signature algorithm.

To achieve this purpose we took ideas from some known attacks, namely [], [], and adapted them to ours. A very common and easy to obtain fault model has been considered, with the purpose of letting the attack remain at a quite general level. A theoretical line of attack has been completely developed, with every step well defined from the fault injection to the final recovery of the secret key. Once it is possible to apply the considered fault model and some results are collected, a precise way to derive the secret key is shown to be effective.

A minor disadvantage of this attack is the possibility of a trivial countermeasure, i.e. if the device checks for the validity of a signature to be output then it immediately will refuse to output every signature interesting for our attack. This is due to the fact that the r value is correct with respect to the generated nonce k, while the s value is wrong. Anyway, this is quite common to a number of attacks and the full signature verification is considered to be a quite complex countermeasure.

The main advantages of the proposed attack are to be found among the following:

- The fault needs only to appear during the multiplication iterations we have described. The actual positions within the word or loop iteration do not matter for the attack effectiveness. Anyway, if the fault hits a specific position known a-priori, then the attack performance improves.

- The procedures employed to recover the intermediate values used in the computations can identify a-posteriori whether or not the fault was of the foregone type. In other words, it is possible to set apart the faulty signatures derived from an interesting fault from the ones that have been corrupted in some other way and are therefore not to be considered. This way it is possible to proceed with the key reconstruction.

- The position of the fault is on an operation that is typically considered less interesting than the scalar multiplication and therefore probably less protected. With this we want to demonstrate that also such operation has to be protected in order to have a secure implementation of an overall very secure algorithm like ECDSA.

In the next chapters we show a more practical point of view, considering simulations and an actual physical realization of the developed procedures, as well as an idea for a very cheap countermeasure to protect against this new attack.

Chapter 6

Analysis of the Attack

We now present the practical results that have been obtained by applying the attack in a real world scenario.

Initially, software simulations of the various parts of the attack have been performed. We report here the main results obtained and a description of the test methodology. Simulations were meant to test both the attacck correctness and performance. A comparison of the performance and characteristics of the proposed attack with other known attacks is then presented to highlight the obtained results.

Finally we present also the methodology and technique used to attack a real device. We describe the equipment and the reasons why its low cost can be sufficient for this attack. In the last sections we show the results obtained when testing the fault injection at various levels, from the use of a program consisting in only simple multiplication operations to a whole ECDSA signature.

6.1 Software Simulations

Prior to the actual fault injection on the physical device, software simulations have been made to test feasibility and performance of the attack.

The main tool adopted for the task is Sage [SAG], a Python library for algebra and other mathematics. Different kinds of simulations have been made to test the various parts and versions of the attack.

The exhaustive search procedure (see section 5.5) has been initially tested for faults happening on the final multiplication $k^{-1}(e+rd)$. Being the signature verification algorithm quite slow, the procedure has been tested considering devices using small words, at most $w = 16$ bits, and key sizes of at most $l = 256$ bits. This is because the procedure needs to perform the verification procedure for every candidate $\hat{s} = \tilde{s} - \hat{b}_{\bar{\imath}} 2^{\hat{\lambda} w+\mu}$, and the value $\hat{b}_{\bar{\imath}}$ satisfies $0 < \hat{b}_{\bar{\imath}} < 2^w$. Employing a small word length thus heavily reduces the number of trials needed. Moreover, the algorithm is quite slow because for every candidate $\hat{s}$ it is necessary to compute its inverse $\hat{s}^{-1} \bmod n$ for the signature verification. This leads to the need of running the whole verification algorithm for every different candidate.

The exhaustive search has been also employed for the word retrieval when the fault hits the first multiplication, i.e. rd. In this case the procedure can be

sped up a little, because there is no need for repeated signature verifications. It is possible to improve the search so that the needed operations are limited to a few scalar multiplications at setup time, followed by subsequent point additions. Recalling equation (5.4), we can rewrite it as

$$[k]G = \left[\frac{e}{\tilde{s}}\right] G + \left[\frac{r}{\tilde{s}}\right] Y \pm \hat{d}_{\bar{i}} \left[\frac{2^{\hat{\lambda}w+\hat{\mu}}}{\tilde{s}}\right] G$$

We can therefore precompute the values

$$T_0 = \left[\frac{e}{\tilde{s}}\right] G + \left[\frac{r}{\tilde{s}}\right] Y$$

$$P = \left[\frac{1}{\tilde{s}}\right] G$$

and subsequently add and subtract $[2^{\hat{\lambda}w+\hat{\mu}}]P$ to an accumulator starting from T_0. The needed operations are therefore only point doublings and additions, thus saving the time needed for scalar multiplications, which are the heaviest computation to be performed in the verification algorithm.

Logarithm search procedures have been tested to check the correctness of the retrieved result. Recall that in this case it is possible to have a collision in the computed points, since the point $[k]G$ is only partially known, i.e. it is possible that different scalar factors exist leading to the same sought x-coordinate value. This can lead to the logarithm finding a scalar factor which is not correct with respect to the sought word. The procedure actually often finds wrong values when tested on a system involving small numbers, for example a curve defined on a prime number 16 or 32 bits long. This is however not a real problem, since such sizes are obviously not suited for cryptographic applications. When dealing with curves of suitable size, e.g. 192 or 256 bits, a point collision never happens and the scalar factor extracted by the logarithm is the correct one.

The logarithm search lets us obtain a considerable speed up against the exhaustive search. As an example, consider the following scenario

- $l = 256 \rightarrow 8$ bits
- $w = 32 \rightarrow 32$ bits
- $\sigma = l/w = 8 \rightarrow 3$ bits

where we see the contribute, expressed in bits, of each variable to the search space size. We also have 1 bit for the sign indetermination An exhaustive search would therefore require nearly $2^{32+8+3+1}/2 = 2^{43}$ signature verifications to be computed, where the halving is because we want an expected value estimation. For a verification time of about 8.5 ms (see [Cry]), this results in approximately $7.5 \cdot 10^{10}$ s $\approx 2.4 \cdot 10^3$ years. A logarithm extraction for the example above has been computed in approximately 26 s. Being both the exhaustive search and the logarithm extraction parallelizable to the same extent, the speed up factor is approximately $3 \cdot 10^9$.

Another good property of the proposed search procedures, important for the attack performance, is the possibility to check whether the fault was of the foregone type or not. It is indeed possible to tell that the fault does not satisfy

the hypotheses if the search fails when computed against such hypotheses, since it always finds a solution if the fault was correctly put forward. This way it becomes possible to throw away the faulty results that do not allow a valid word retrieval and consider only the ones which actually are useful for the remainder of the attack.

Simulations of the lattice attack have then been done, to make estimations of its performance. Such performance is bound to the employed lattice dimension, which is $\nu = N + 1$, where N is the number of faulty signatures contributing in the attack. A higher dimension lattice leads to the attack finding the correct value with higher probability, but it needs more computational effort.

Table 6.1: Lattice attack performance, probability of success

	p-192		p-256		p-384		p-521	
	30	*1*	**42**	*1*	**72**	*1*		
8	26	*0.57*	36	*0.57*	60	*0.56*		
	23	*0*	31	*0*	50	*0*		
	13	*1*	**18**	*1*	**28**	*1*	**37**	*1*
16	12	*0.33*	16	*0.19*	25	*0.75*	34	*0.43*
	11	*0*	15	*0*	23	*0*	32	*0*
	7	*1*	**9**	*1*	**13**	*1*	**17**	*1*
32	6	*0.48*	8	*0.43*	12	*0.25*	16	*0*
	5	*0*	7	*0*	11	*0*		

The values reported in table 6.1 have been experimented for the lattice attack. The table reports lattice dimensions and corresponding success probability for various word size and key length combinations. It is possible to note that the success probability quickly diminishes when using a smaller amount of faulty signatures, especially when large word sizes are considered.

These simulations showed that the overall procedure of the attack can be used starting from the faulty values hypotheses. With the use of logarithm search the most heavy computation becomes feasible, requiring only few seconds if appropriately tuned. A whole attack on real world numbers has not been performed but we can give estimation of its performance. We can consider the curve p-256 and a 32-bit machine, so read from table 6.1 that we need 9 words. The involved numbers are composed by 8 words, and therefore we expect needing $9 \cdot 8 \cdot 3 \approx 200$ faulty signatures if we suppose to have an even distribution of the faults among the words. The factor 3 is considered to take into account what we called the word alignment issue, see section 5.5.1. It is possible to consider for the extraction of 200 logarithms a required time of about $200 \cdot 10 = 2000$ seconds, if no parallelization is employed. The lattice attack time is not relevant now, being less than 10 seconds. We can therefore estimate an effort of less than 4 hours to break a 256-bit key.

6.2 Comparison with Known Attacks

As sketched in section 5.2, the proposed attack does not involve the scalar multiplication. This has some effects on the power of the attack, inducing some advantages together with some disadvantages. We outline here the most

significant differences with other attacks, particularly those targeting ECDSA, as listed in section 4.3.

Main weaknesses include:

Operation execution time The target multiplication has a very short execution time w.r.t. the scalar multiplication. A measured value is a factor of about 15 between the time needed for modular multiplication (computation of s) and the time required for scalar multiplication (computation of r).

Need for quick fault The fault must hit only one iteration of the operation, otherwise the faulty value gets more complicated and needs more (exponential) effort to let the retrieval procedure succeed.

Use of the faulty variable A faulty value must be used from the beginning of the multiplication in place of a correct one. A fault targeting for example the internals of the ALU or a write of a product value is not exploitable through the presented attack.

Conversely, the following properties make our attack powerful:

Admissible random fault The fault needs not be too much precise in time nor in location. An even distribution of the fault during the multiplication is admissible and even useful.

Fault check possibility Given a faulty signature pair (r, s) it is possible to try the retrieval procedure and subsequently tell whether the fault was or was not obeying the hypotheses oberving the retrieval success.

Small fault requirement Our attack needs very few faulty signatures when compared with other known attacks. Even in the worst cases a few tenths of faulty signatures are sufficient to retrieve the whole key.

Not tight dependency from the underlying group The attack is not tightly bound to elliptic curve based groups. It is simply portable to other DSA-based schemes defined over any DLP-secure group, since it hits the modular multiplication, which is independent of the employed group.

Fault hypotheses greatly improve performance Any a-priori knowledge over the fault properties greatly reduce uncertainty. The attack performances thus greatly improve, because the retriaval procedures has to explore smaller search spaces. Moreover, some kinds of knowledge can reduce the importance of the otherwise unavoidable issues described in section 5.5. For example, knowing the position of the bit being modified by the fault completely erases the word alignment issue.

6.3 Towards Experimentation

We now describe how our attack can be practically tested in physical environment. The major point is the very low cost of the employed apparatus.

As a fault injection mean, we used low supply voltage. As previously sketched in chapter 3, the kind of faults injectable through a low voltage attack enjoy some distinguishing features.

It is not possible to control the location nor the timing of the fault. The fault model has therefore to rely on a probabilistic description of location and timing. Moreover, the particular nature of the injected fault is strongly dependent upon the actual considered hardware. It is not immediate to estimate which kind of behaviour will a certain device expose under low voltage conditions.

A different characterization dimension is about the life span of the fault. In the case of low voltage we deal with transient faults. The hardware is not physically altered, and the fault effects cease after a short period of time, i.e. it is not necessary to wait until a system reset to experiment a correct behaviour again.

For our tests we used the ST SpearHEAD development board. The board has a well studied faulty behaviour when forced in low supply voltage conditions, see [BBPP09] for details.

The fault model of this board is greatly compatible with the one needed for the proposeed attack. Such a fault model is indeed a single bit flip during the loading of data from memory. Specifically, when underfeeding the power line it can happen that a memory read leads to a wrong value to be transfered to the core. Such a value is of a very well defined form: the bit in a given position is flipped from 1 to 0. No flip in the opposite direction occurs, and the position of the wrong bit is a characteristic of the board remaining constant between the executions.

It is clear that this fault model is compatible with the presented attack. It also has some advantages because of the fixed position of the flipped bit, which makes us know the value μ used to improve the attack performance. We expect that a fraction of the loads during the integer multiplication algorithms execution is affected by the fault, leading to our attack feasibility. The attacker's job thus consists in letting the system run its ECDSA algorithm tuning the supply voltage to reach the desired fault rate. Collection of the output signatures should then lead to detection and exploitation of the faulty ones.

6.4 Equipment

Let us now describe the actual apparatus used to perform the tests. The board, as previously said, is a development and testing board produced by ST Microelectronics. It is named SpearHEAD and comprises an ARM-based SoC. The processor is specifically an ARM 926 EJ-S, featuring all kinds of features expected from a modern computer. The board is also equipped with various I/O interfaces, including serial lines, ethernet, USB, and GPIO, among others. We used a Pentium 3 machine to drive the board and collect the results connecting to the board through a serial line, as well as to analyze the results.

The board specifications tell the SoC has to be fed from a 1.2 V line. This is normally carried out by proper circuitry on the board. To attack it we have to bypass this circuitry and impose our own. We must operate this way because underfeeding the whole board from the power supply pins would not lead to the desired results. First, the SoC would not get underfeeded if the original circuitry is left as is, and second we would experiment great mulfunctioning of the whole board, for example on I/O parts, probably before than on the chip. Clearly this is not wanted, in that we have to manipulate only the SoC behaviour, for example to keep the ability to communicate with the board at least to collect

results data.

For the actual attacking operation, we used the power supplier 6633B from Agilent. This is anyway not directly suitable for the attack because of too coarse resolution of the output voltage. The supplier indeed has a resolution of about 12 mV, but this is too high a step for the SoC, in that it causes sudden changes between normal work and system hang. To face this issue, we set up a breadboard to mount an op-amp used to downscale the voltage provided by the supplier. Being G the gain of the set up op-amp circuit, we obtain

$$V_{in} = GV_{supply}$$

meaning that we have to set the power supplier at a higher voltage than the required one to get the desired resolution

$$\Delta V_{in} = G\Delta V_{supply}$$

With suitable gain, we obtained a resolution at the board pins of about 1 mV. This step is acceptable for our purposes.

The injection of the faults was done by slowly decreasing the voltage supplied to the SoC. The system requires more energy at reset time, so we let it do this step in normal conditions. Moreover, before starting the main program, we lower the system's clock rate to reach more stability while injecting the faults, as described in [BBPP09]. After the system starts running the program code, the voltage is actually descreased, until the results start showing faulty behaviour. This happens somewhere between 800 and 850 mV, depending on environmental conditions. The system is then left running in this condition to collect a large amount of data. The fault rate is kept low, thus having a lot of correct computations interleaved with a few faulty ones, so that we can expect each faulty computation to be the consequence of one only error.

The ECDSA implementation we ran our attack against is based on CryptoLib, a proprietary software library for various cryptographic operations from ST Microelectronics, similar to OpenSSL [Ope]. This choice has been made to have full control on the software to tune it to the desired features.

The actual code run on the board is varied from time to time to experiment and test the behaviour of different operations. Namely, we can summarize our tests in the following:

- finding the rough point where the faults start happening and where the board exhibits hang up,
- checking whether the theoretical fault model of wrong memory read corresponds to the board's behaviour,
- checking whether the integer multiplication exhibits the expected behaviour,
- performing the attack on the whole signature algorithm.

In the next sections we will see the results obtained in these tests. We anticipate that unfortunately our setup was unable to show a complete success of the attack on the ECDSA signature, but we anyway attacked the basic multiplication operation to a satisfying extent.

6.5 Integer Multiplication

After verifying that the observed fault is of the type described in previous studies and tuning up the fault injection apparatus, we started testing the integer multiplication. This is because it is the core operation we want to target, and the experiment is aimed at verifying the level at which the board exhibits our needed fault.

We experimented the following key factors to determine whether the multiplication leads to the expected results.

Instruction cache If the instruction cache is enabled the system executes as expected, while if the instruction cache is disabled the system becomes very unstable and hangs up after very few executions, thus preventing us to collect data.

Data cache The data cache influences the ability to hit the desired operators. Indeed, with data cache disabled the fault appears in different places in the program, and the multiplication operands are hit very rarely among others, non-identified, variables. Moreover it is easy for the board to early start emitting values which do not back up to correct ones.

Compilation options If the code is compiled turning off every code optimization then the resulting program has a much larger number of otherwise avoidable loads from memory, thus in this case too we saw almost no faults on the multiplication operands. A normal optimized compilation leads to foregone faults intead, since the multiplication becomes the most load greedy operation.

Supply voltage The system is quite susceptible to little voltage changes of as low as few, $3 \div 5$, mV. A small variation quickly takes the system from correct behaviour to hang up, so we have to tune the voltage every time to a good point that depends, amongst all, on the physical environment.

Clock rate The voltage at which the faults appear changes with the clock setting. Also the voltage interval from correct behaviour and hang up depends on the clock, and we found that at the nominal clock rate the board does not tolerate our voltage variations. Thus we run the system at a slower clock to be able to tune the voltage in an acceptable range.

Once these points have been correctly set up, we experiment the following basic statistics.

- The faulty computations are about 1/100 of the total.
- The faults of the wanted kind are always about 1/3 of all faults.

The faulty computations rate is quite variable, being it dependent among an always varying voltage level. The order of magnitude of the rate is anyway managed to be kept between 10^{-3} and 10^{-1}.

The rate of the faults of the good kind is instead very stable. It is always approximately 1/3, no matter how other conditions change. The majority of the remaining 2/3 of the faults seem to hit the values when they are loaded for

being output. A small fraction, about 1/30 of total, is instead a combination of more than one fault of the preceeding kinds.

We ran the tests especially on multiplications of 256-bit integers, which, on a 32-bit device, are composed of 8 words. The resulting statistics do anyway not change when different types of operands are used. They are constant when changing the operands' length as well as the operands' values, being these random values or particular forged ones.

Statistics on the faulty iteration $(\bar{\imath}, \bar{\jmath})$ have also been made. Remembering that j is the inner cycle variable and i the outer one, we saw that the distribution on $\bar{\imath}$ is almost uniform, while $\bar{\jmath}$ shows a different distribution, which changes greatly between runs with different operands. The cause of this very non-even distribution was not deeply investigated.

These results show that the multiplication is targetable by the faults injectable through low voltage technique and the fault distribution is suitable for our attack against the multiplication.

6.6 Complex Operations

We now show the problems encountered when attacking more complex operations than simple integer multiplication. In other words, we complicated the program to make it compute modular multiplications and even a full ECDSA signature.

Recall that the modular multiplication is simply an integer multiplication followed by a modular reduction. When attacking this operation, a different behaviour emerged. The faulty output values are not as expected, and early hang up of the system also appeared. By analyzing the power traces associated to the computations, it emerged that the integer division operation employed during the modular reduction is associated with an abnormal power consumption. The reason should be bound to the fact that such an operation is a 64-bit division, which is not implemented in the ARM hardware. The fault appeared is of a completely different type from the bit flip observed when hitting memory loads. Namely, the fault affecting the division is causing a quotient being set to 0 instead of its correct value.

In these conditions, it was discovered that the faults on the division appear at a higher voltage than the faults on the multiplication, thus making the attack not feasible anymore. Faults on the multiplication indeed appear only if the voltage is quickly dropped down, but this causes also the system to hang.

Even with this problematic behaviour, we tried attacking the whole ECDSA algorithm. We aimed at finding foreseen faults on the s value together with correct r. One more problem, yet easily predictable, arises here, i.e. the scalar multiplication taking longer time than the final stage, thus being subject to attract the majority of faults. The ratio between the two operation durations has been computed in about 15. The rate of faults on s w.r.t. the ones on r is anyway much different. We actually can't even tell a precise estimation, because such a kind of fault appeared only 2 times among hundreds of computations. Moreover, it was not possible to recognize these faults among the expected ones.

We can conclude that our setup was probably too cheap to attack ECDSA. More clever fault injection techniques such as glitches would probably lead to better results.

6.7 Results Resume

The simulations showed that the attack is correctly outlined in its parts. Once an error of the correct type can be injected through some fault, the studied procedures manage to recover the expected values. Full success has been obtained in both parts of the attack with good performance.

The word recovery has been practiced with real cryptographic curve parameters, but considering a device with a small word size (namely, 8 bits), because of the large time needed with real world values (32 bits). The procedure retrieves correct values, but needs a quite large amount of time. Anyway, the logarithm search procedure has been subsequently developed, and, since the great performance improvement of about 10^9 that comes with it, it has become possible to consider also more realistic numbers, i.e. words 32 bits in length. The retrieved values are still correct, and the time taken by the logarithm search is about 26 seconds on a Pentium 3 machine. This procedure has not anyway been optimized to the maximum extent, as it would be possible as explained in section 2.2.3. Full parallelism and optimization would allow for even better results.

The second part of the attack, specifically consisting in the application of a so-called lattice attack to solve the hidden number problem deriving from the involved values, has been also run with real world parameters. Here, the main determining factor for the performance of the attack is bound to the dimension of the lattice, or in other words the number of collected signatures that take part in the attack. We found that the lattice attack is carried out in less than 10 seconds on an i7 processor if the lattice dimension does not exceed 30, which is a value suitable for our attack as shown in table 6.1. Greater dimensions take anyway a still admissible time.

The practical experimentation showed that the cheap equipment used is able to produce the fault needed for the attack. Moreover, we experiment the foregone type of fault, suitable for subsequent computations. This happens while the device is computing integer multiplications, and the wanted type of fault appears at a rate of about 1/3 among the overall faulty results. The good faults are also easily distinguishable from the rest.

When switching to more complex operations than the integer multiplication, however, the faults change their type of distribution and it seems more difficult to obtain the sought values. The cause of this is related to a different power absorbtion by the device when computing divisions that are not implemented in hardware. Two solutions are possible to face this problem.

- It is possible to change the software to try adapting it in avoiding the use of such high precision divisions, replacing them with normal precision ones. This way it may happen that the observed fault is again of the known type. Anyway this option is clearly suitable only for a testing environment and cannot be applied in a real attacking scenario.

- More expensive equipment can be used, to produce faults through more sophisticated methods like glitches or spikes (as presented in section 3.2) in place of a simple constant underfeeding of the power supply line.

Thus in general we conclude the attack is well designed and there are reasonable suggestions on how to put it into full practice.

Chapter 7

Countermeasures

In this chapter we discuss a possible countermeasure aimed at protecting from the presented attack. A brief introduction to generic countermeasures is first given. We introduce physical layer countermeasure techniques. These are methods that aim to protect against any kind of attack, considering the fault at hardware level and trying to recognize it. These countermeasures are characterized by a very high cost, and the need to completely redesign a hardware piece to allow for their introduction in the system. Software countermeasures are a cheaper and more flexible solution, but they can only protect against known attacks.

Here we focus on the latter type and propose a countermeasure for the specific attack discussed in the rest of the work. The fundamental advantage of the proposed countermeasure is its very low cost, as we will see below.

7.1 Generic Countermeasures

We show some typically employed countermeasures that can be applied to any cryptosystem or any device, being very generic. The drawback of these methods is their inherent high realization and design cost.

7.1.1 Physical Countermeasures

A first way to protect cryptographic hardware is of course represented by physical layer countermeasures. It is possible to embed sensors in the hardware to make it aware of unwanted conditions, as well as to introduce systems that make the device not usable if put in abnormal conditions.

Let's describe a list of possible ideas.

Supply voltage detectors To prevent low or high voltage attacks.

Power spikes detectors React to abrupt variations on the power line.

Frequency detectors Check whether the clock frequency is inside a safe interval.

Light detectors Using photodiodes it becomes possible to prevent light attacks.

Active shields Metal shields covering the whole chip can react by making the hardware not usable if a disconnection or modification of this layer happens. It also mkes more difficult to locate specific circuit areas.

Temperature sensors Check for environmental temperature.

Hardware redundancy Hardware can be made redundant to check whether a computation leads to different results in different locations.

Dummy random cycles Attacks that need synchronization can become difficult if dummy instruction cycles are inserted at random during execution.

Unstable frequency generator Another protection against attacks requiring synchronization, because events occur at different times during different executions.

7.1.2 Logic Level Countermeasures

The use of *dual rail* logic has been proposed to counteract fault attacks. In dual rail logic each bit is encoded by using two wires. This way, 2 of the 4 possible combinations ($< 0, 1 >$ and $< 1, 0 >$) are made correspond to actual bits, while the other 2 configurations ($< 0, 0 >$ and $< 1, 1 >$) represent errors.

It therefore becomes possible to detect faults that affect only one wire directly in the logic level.

7.1.3 Software Countermeasures

A very different kind of countermeasure is given by software modifications. The underlying hardware is left as is, perhaps unprotected. The cost of the device therefore does not increase, and it is possible to apply these countermeasures on already existing pieces of hardware too. The software is modified in an attempt to make the data obtainted through faults unusable for attacks. Anyway, software countermeasures usually impact on algorithms complexity, thus increasing computation time.

Some possibilities are the following.

Checksums Usually employed to check data integrity.

Redundancy As in hardware, it is possible to introduce redundancy in software algorithms too, and subsequently check whether a computation yields different, thus faulty, results.

7.2 Proposed Countermeasure

The countermeasures described above are generic, and applicable to any device or algorithm. Anyway, such methods introduce a not so little additional cost. Indeed, it is clear that ideas like hardware duplication are quite expensive in terms of chip area. Software countermeasures are also typically expensive, this time in terms of execution time. We propose here a different idea, applicable only to the kind of signatures saw in the rest of the work, and aiming to protect especially against the new attack presented in chapter 5.

7.2.1 Algorithm

The proposed countermeasure belongs to the software class. We modify the signature algorithm to make it secure against the attack to the final stage multiplication saw earlier.

Algorithm 12 ECDSA Signature Generation

Input: signature intermediate values e, k, r, secret key d
Output: signature value s
1: $k^{-1} \leftarrow 1/k \bmod n$
2: $t_1 \leftarrow rd \bmod n$
3: $t_2 \leftarrow t_1 + e \bmod n$
4: $s \leftarrow t_2 k^{-1} \bmod n$
5: **return** s

The trivial way to compute s is presented in algorithm 12, while the proposed modification is referenced in algorithm 13.

Algorithm 13 ECDSA Signature Generation

Input: signature intermediate values e, k, r, secret key d
Output: signature value s
1: $k^{-1} \leftarrow 1/k \bmod n$
2: $t_1 \leftarrow k^{-1}k^{-1} \bmod n$
3: $t_2 \leftarrow t_1 d \bmod n$
4: $t_3 \leftarrow kr \bmod n$
5: $t_4 \leftarrow k^{-1}e \bmod n$
6: $t_5 \leftarrow t_2 t_3 \bmod n$
7: $s \leftarrow t_4 + t_5 \bmod n$
8: **return** s

Algorithm 13 outputs exactly the same result as algorithm 12. Anyway, it performs slightly different operations in the final stage, i.e. the computation of s. This makes the attack no more applicable, because the new operations lead to faulty values propagate in a different way, which is more difficult to eploit in order to recover the secret key.

For clarity, we express the two algorithms in a compact form:

$$s = k^{-1}(e + rd) \bmod n$$

$$s = (k^{-2}d)(kr) + k^{-1}e \bmod n$$

It is easy to see that the two computations lead to the same result, being the second just a complicated version of the first.

Let's see some example computation.

An attacker can inject a fault in step 2, obtaining a faulty $\tilde{t}_2 = t_2 \pm d_{\bar{\imath}} 2^{\lambda w+\mu}$. This yields an output value $\tilde{s} = s \pm d_{\bar{\imath}} kr 2^{\lambda w+\mu}$, which can be interesting because it contains the key word $d_{\bar{\imath}}$. It is however not possible to retrieve the error by exhaustive search, because it contains also the nonce k, which is not known and large. The logarithm search is not possible either:

$$[k]G = \left[\frac{e}{\tilde{s}}\right] G + \left[\frac{r}{\tilde{s}}\right] Y \pm \left[\frac{d_{\bar{\imath}} k^2 r 2^{\lambda w+\mu}}{\tilde{s}}\right] G$$

The problem here is the factor k^2, which makes it not possible to know the point it is involved in.

A different fault injectable by an attacker is in step 5. He can then obtain $\tilde{s} = s \pm t_{\bar{i}} 2^{\lambda w+\mu}$, where t denotes the computed $t_2 = k^{-2}d$. This would make it possible to perform a search procedure and obtain the word $t_{\bar{i}}$. Anyway, this would not be sufficient to mount the lattice attack, because of the presence of the unknown and not constant k^{-2}.

It can be seen that other locations for the fault would lead to useless datas too.

7.2.2 Cost Analysis

Being this countermeasure specific to a certain attack to the DSA class of algorithms, it must not impact on cost as much as a generic countermeasure, which aims at protecting against many more kinds of attack. It furthermore is a software countermeasure, thus its influence is to be expressed in terms of execution time penalty.

We express execution time in terms of the basic modular arithmetic operations, denoting with M a modular multiplication, I a modular inversion and S a modular squaring.

The trivial algorithm 12 has a complexity of $2\,\mathrm{M}+1\,\mathrm{I}$. Adopting the countermeasure in algorithm 13 makes the cost of the computation increase to $5\,\mathrm{M}+1\,\mathrm{I}$. Although being almost double the original cost, this is not the actual complexity to be considered, because the heaviest computational effort is needed for the elliptic curve scalar multiplication during the first stage of the signature generation.

Let's take as an example the elliptic curve p-256 from NIST [NIS09]. The complexity of the scalar multiplication takes $256\,\mathrm{D} + 128\,\mathrm{A}$, where D is the elliptic curve point doubling, and A the elliptic curve point addition. A typical implementation entails a point doubling take $4\,\mathrm{M} + 4\,\mathrm{S}$ and a point addition $12\,\mathrm{M} + 4\,\mathrm{S}$. This translates in a complexity of $2562\,\mathrm{M} + 1536\,\mathrm{S} + 1\,\mathrm{I}$ for the trivial algorithm and $2565\,\mathrm{M} + 1536\,\mathrm{S} + 1\,\mathrm{I}$ for the countermeasure.

It is now clear that the countermeasure has a neglactable impact on the software implementation cost.

7.3 Considerations

The attack presented in the previous chapters is easily counterable. The countermeasure we have introduced is strictly countering the discussed attack. Anyway, its funding idea, i.e. to infect the computations with the nonce, can be useful in preventing from new, active or passive, attacks too since an attacker has to deal with this large, unknown and always changing value whenver he tries to study the computation of the second element of the signature.

The countermeasure is yet very cheap, as we saw in section 7.2.2. It not only requires very few additional operations, but it also uses the already available nonce thus saving the time for another random value to be computed.

We showed that also the second stage of the ECDSA signature generation is vulnerable and should therefore be taken into consideration for protection.

However, a cheap method exists to defend against possible attacks on this part of the algorithm.

Chapter 8

Conclusions and Future Work

This thesis presents a novel fault attack against the ECDSA signature algorithm. Major target points that are obtained concern the reduced cost of both realizing the required fault model and the computational effort needed for the attack procedures.

The developed attack relies on a simple fault model obtainable with cheap equipment. The bit-flip is indeed one of the most commonly considered fault models in the literature. It is practical to describe mathematically and leads to effective attacks. Moreover, the possibility exists to induce bit flips at a very low cost, needing only a suitable power supplier to feed the chip with low power.

From the actual attack point of view, we have a very straightforward procedure we can summarize as the following:

1. set up a working environment for fault injection,
2. collect a number of results from the perturbed computations,
3. analyze the results and keep the useful ones,
4. performing the final attack procedure over the chosen results.

This workflow can be employed because some conditions are met:

- typically we can set up the environment to make the faults appear at a low rate, therefore it is possible to easily discard the great amount of valid signatures and keep a few faulty ones for subsequent analysis,
- the logarithm analysis procedure is able to identify and tell apart the signatures resulting from an occured fault matching the foregone type.

Our attack is also powerful in that it does not need a large amount of signatures, even if long keys are employed. Once it is known that a given set of faulty signatures come from the foregone kind of fault, it is possible to proceed with the attack on such set. A typical order of magnitude for the number of needed signatures is around 100, but this can vary slightly depending on some decisions of the attacker. In very lucky scenarios it is possible to complete the secret key recovery with as few as some tenths of faulty signatures.

The needed computing power for the attack is also not big a deal. Simulations showed that it is possible to perform the attack on a typical desktop computer in a few hours or even minutes.

The real novelty of the proposed attack is the discovery of a potential weakness in a step of the algorithm which is typically not considered to be interesting and therefore underprotected. The attack then proceeds by using some ideas taken from other known attacks as well as one new procedure to efficiently analyze the results collected from the device, i.e. what we called the logarithm search procedure.

In this work we also presented a possible countermeasure to protect an implementation of the cryptosystem from the analyzed attack. The countermeasure we suggest is very cheap because it targets the precise point where the weakness was identified. This anyway means that it cannot protect against future attacks if they will be based on different ideas.

Further studies are possible starting from the idea presented here.

Indeed, we did not analyze in great detail the possibilities that arise when considering different kinds of fault models. This is because when the fault gets more complex than a single bit flip the modifications to the output values get complicated too, and the computational effort needed for the attack increases very quickly if a naïve approach is employed. We sketched some ideas on how to deal with some still simple kinds of faults. Obviously, it turns out that the more the attacker knows about the fault in advance, the fewer trials and the less computational complexity are needed. That said, it can be interesting to further develop and study methods for dealing with these more complex kinds of fault as well.

Another advancement can be possibly studied. We dealt only with modular multiplications performed through trivial, yet actually employed, algorithms: an integer multiplication followed by modular reduction. It may be interesting to analyze the results obtainable when different strategies are employed, like the Montgomery modular multiplication algorithm.

Finally, we propose a more practical development line too. It may be possible to analyze the different type of fault observed on our equipment when dealing with high precision divisions. The fault type is very precise and it may be exploitable to build an attack somehow related to the one that has been presented in the previous sections.

Appendice A

Estratto in Italiano

A.1 Introduzione

La crittografia è la disciplina che si occupa di sviluppare algoritmi per garantire la sicurezza delle comunicazioni. Storicamente gli algoritmi crittografici sono stati continuamente oggetto di *attacchi*, ovvero metodi studiati per violarne la sicurezza. Al giorno d'oggi esistono algoritmi noti particolarmente robusti e difficilmente gli attacchi hanno prestazioni in grado di minacciare realmente la sicurezza delle applicazioni che li utilizzano.

Un nuovo scenario si è però aperto da poche decine di anni. Invece delle comuni tecniche di attacco basate esclusivamente su proprietà matematiche degli algoritmi, si considerano nuove tipologie di attacco, cosiddette *side-channel*. Un attacco side-channel non si limita allo studio della struttura matematica dell'algoritmo, bensì si concentra su una specifica *implementazione*. In altre parole un attacco side-channel richiede di avere un accesso fisico ad un dispositivo crittografico in modo da utilizzare canali di comunicazione con esso che siano non-standard e non previsti in fase di progettazione. Tali canali sono costituiti dall'interfaccia fisica del dispositivo sotto attacco.

Esistono due tipologie principali di attacchi side-channel, gli attacchi *passivi* e quelli *attivi*. Un attacco passivo procede monitorando una qualche quantità fisica relativa al dispositivo, come la corrente assorbita o la radiazione elettromagnetica, durante le computazioni. Attraverso l'analisi della quantità osservata è poi possibile risalire alla chiave segreta utilizzata. Durante un attacco di tipo attivo invece si manipola direttamente l'interfaccia del dispositivo, in modo da indurlo ad eseguire computazioni sbagliate. Il tipo di manipolazione può variare significativamente dall'applicazione di un'alimentazione troppo bassa, a dei glitch sull'alimentazione o sul clock, al bombardamento con laser o altre particelle. Una volta che il dispositivo abbia eseguito le computazioni in questo stato disturbato ed emesso i suoi risultati, sarà possibile, attraverso opportune ipotesi ed analisi dei risultati, arrivare a scoprire la chiave segreta. I due metodi di attacco, attivo e passivo, possono essere adeguatamente miscelati per ottenere eventuali prestazioni migliori.

Gli algoritmi crittografici si dividono anch'essi in due grandi famiglie, ovvero quelli detti a *chiave privata* e a *chiave pubblica*. Le principali differenze tra le due famiglie si trovano nelle applicazioni e nelle prestazioni degli algoritmi. Gli

algoritmi a chiave privata vengono tipicamente usati per fornire proprietà di confidenzialità ad una comunicazione, cifrandone il contenuto in modo che questo non sia intelligibile da soggetti diversi da quelli autorizzati. Questi algoritmi hanno anche tipicamente le prestazioni maggiori. Gli algoritmi a chiave pubblica sono usati principalmente per le applicazioni di autenticazione, ovvero per garantire in modo sicuro che i soggetti partecipanti alla comunicazione siano chi pretendono di essere. In altre parole ci si concentra sulla possibilità di impedire la falsificazione dei dati. Per questo motivo gli algoritmi usati a questo scopo vengono anche detti di *firma digitale*.

In questa tesi si presenta un nuovo metodo di attacco basato su iniezione di guasti, dunque di tipo attivo, mirato all'algoritmo di firma digitale ECDSA, standardizzato dal NIST in [NIS09].

I punti di novità riguardando principalmente la posizione del guasto iniettato e la procedura di ricerca dell'errore. Per quanto riguarda la posizione, l'idea è di colpire un punto nella computazione della firma che viene di solito ritenuto meno importante e per questo probablmente meno protetto. La procedura di ricerca sfrutta una particolare caratteristica dei risultati che permette di risolvere efficacemente il problema del logaritmo discreto e così ricostruire la chiave.

A.2 Crittografia Basata su Curve Ellittiche

L'algoritmo crittografico oggetto dello studio è ECDSA, un algoritmo sviluppato a partire da DSA utilizzando come struttura algebrica per le primitive le curve ellittiche.

Le curve ellittiche sono oggetti geometrici studiati in matematica per una varietà di applicazioni. In crittografia esse sono usate in quanto è possibile costruire su esse strutture algebriche adatte alla creazione di una primitiva crittografica sicura.

Precisamente, i punti di una curva ellittica vengono dotati di un'operazione di gruppo. Il gruppo così ottenuto è utilizzato per la creazione di una funzione *one-way*, v. equazione 2.3. Una funzione di questo tipo è facilmente calcolabile, mentre la sua inversa, pur essendo ben definita, non è calcolabile in tempi rapidi. La difficoltà dell'inversione di questa funzione viene sfruttata come base per la costruzione di algoritmi crittografici, e su di essa si basa la sicurezza di tali algoritmi.

La famiglia ECDSA è costituita dagli algoritmi di generazione delle chiavi, generazione della firma, verifica della firma, ed eventualmente generazione della curva nel caso non venga scelto di usare una curva standard. La generazione della chiave consiste nella semplice scelta di un numero casuale. I dettagli di generazione e verifica della firma sono riportati negli algoritmi 5 e 6. Per gli scopi dell'attacco descritto nel seguito è importante tenere presente anche l'algoritmo di moltiplicazione modulare, riportato nell'algoritmo 10.

A.3 Attacchi Side-Channel

Gli algoritmi crittografici vengono continuamente sottoposti a tentativi di *rottura*, ovvero si cerca di scoprire metodi per violare la sicurezza degli stessi. Un metodo di questo tipo, in grado per esempio di decifrare informazioni confiden-

ziali o che consenta di guadagnare l'accesso ad una chiave segreta, viene detto *attacco*.

Fino a pochi decenni fa, gli attacchi erano ideati e indirizzati alla sola struttura matematica sottostante gli algoritmi. Si classificano ad esempio attacchi in grado di recuperare una chiave segreta dati una o più coppie di testo cifrato con il corrispondente testo in chiaro, oppure ipotizzando di avere la possibilità di ottenere il cifrato di qualunque testo in chiaro, e di seguito con altre ipotesi di questo tipo.

Più recentemente, un altro tipo di attacco sta guadagnando successo. Si tratta di attacchi che ipotizzano un accesso fisico ad un dispositivo che esegua operazioni crittografiche, contenendo al suo interno i segreti necessari. Si considera di non essere in grado di avere un accesso diretto a tali segreti. Gli attacchi di tipo side-channel non considerano solo la forma matematica di un algoritmo crittografico, ma anche la sua specifica *implementazione*, e per raggiungere lo scopo vengono usati *canali* di comunicazione con il dispositivo diversi da quelli ufficialmente pensati in fase di progettazione.

Ad esempio, si considerano canali di accesso secondari ad un dispositivo il suo consumo di potenza o la radiazione da esso emessa. Un attacco può svilupparsi dal monitoraraggio di alcune di queste quantità interessanti.

Un'altra via percorribile è quella di manipolare in modo anomalo l'interfaccia fisica del dispositivo. Per esempio inviando glitch sul segnale di clock, o spike sul segnale di alimentazione. Altri metodi di questo tipo sono riportati nel capitolo 3. L'obiettivo di queste tecniche è di ottenere risultati sbagliati a causa delle condizioni anomale forzate sul dispositivo. L'analisi di questi risultati può portare a scoprire la chiave segreta. Un semplice esempio è riportato in sezione 3.4.

Per la realizzazione di attacchi basati sui guasti si utilizzano modelli per astrarre dal livello fisico del dispositivo in esame. Una spiegazione dei tipi di questi modelli si trova in sezione 3.3.

Inoltre, una lista di attacchi noti a sistemi basati su curve ellittiche è riportata nel capitolo 4.

A.4 Attacco Proposto

L'attacco che proponiamo appartiene alla categoria degli attacchi basati sui guasti. Considereremo un modello di guasto semplice ed efficace, molto studiato in letteratura: il *bit flip*. Supporremo cioè che durante lo svolgimento delle computazioni, il dispositivo sotto attacco utilizzi una variabile con un bit cambiato rispetto al valore corretto.

L'attacco procede dall'iniezione del guasto e si articola successivamente in alcune fasi di realizzazione. Si può delineare la seguente struttura per la procedura di attacco:

- immersione del dispositivo in ambiente controllato al fine di costringerlo ad operare in condizioni di funzionamento anomale,
- esecuzione dell'algoritmo di firma digitale da parte del dispositivo in condizioni di guasto,
- raccolta dei risultati delle computazioni guaste e non,

- identificazione all'interno della collezione dei risultati derivanti dal tipo di guasto previsto,

- taratura e utilizzo di una procedura di recupero della chiave segreta a partire dai risultati ottenuti al passo precedente.

La presente descrizione si concentra sugli ultimi due passi esposti. I metodi per iniettare guasti sono infatti ampiamente studiati in letteratura e, finchè valgono le ipotesi di bit flip, i dettagli del metodo utilizzato non sono importanti al fine delle procedure seguenti.

Sono state identificate due posizioni, all'interno dell'algoritmo di generazione di una firma digitale ECDSA, per le quali un guasto provoca la riuscita dell'attacco. Precisamente, è necessario che il guasto colpisca una delle due moltiplicazioni modulari che compongono il calcolo del valore s della firma digitale, come descritto in sezione 5.3.

Data la natura piuttosto semplice del guasto, è possibile analizzare come l'errore nella computazione si propaga durante l'esecuzione dell'algoritmo. Si ottiene che il bit sbagliato provoca l'addizione di un termine di tipo $2^{\bar{\imath}}b_{\bar{\imath}}$ al risultato s correttamente corrispondente al valore r precedentemente calcolato. In questa scrittura, $b_{\bar{\imath}}$ rappresenta una parola di uno degli operandi partecipanti alla moltiplicazione modulare. Maggiori dettagli sulla propoagazione dell'errore sono esposti in sezione 5.4. A seconda della posizione in cui il guasto avviene, questa parola potrebbe essere una parola del valore intermedio $(e + rd)$ oppure una parola della chiave segreta d. La firma così ottenuta è quindi costituita dalla coppia $(r, \tilde{s})$ e ovviamente non è valida, ovvero non supera il test dell'algoritmo di verifica.

Se il guasto è avvenuto in questo modo, è possibile recuperare la parola $b_{\bar{\imath}}$ attraverso una procedura di ricerca esaustiva, oppure attraverso una procedura adattata dall'algoritmo per il calcolo del logaritmo discreto su curve ellittiche. Si vedano a questo scopo le sezioni 5.5 e 5.6.

Le procedure di ricerca appena citate si applicano ad una firma guastata e consentono di stabilire se il guasto è avvenuto nel modo cercato oppure no, oltre a trovare la parola $b_{\bar{\imath}}$ in caso positivo. Per portare l'attacco a successo, questa ricerca va ripetuta per una quantità di firme maggiore di uno. L'ordine di grandezza di tale valore dipende dalla lunghezza della chiave segreta, dalla lunghezza delle parole del dispositivo utilizzato. Per sistemi tipici possiamo comunque indicare un numero di firme necessario tra 10 e 100. Il valore preciso dovrebbe invece essere trovato durante l'esecuzione delle ricerche dipendentemente dai risultati trovati, se si intende ottimizzare le prestazioni.

La successiva parte dell'attacco consiste nell'ottenere la chiave segreta a partire dalle singole parole ritrovate per ogni firma guasta. Per questo la procedura è diversa a seconda che il guasto abbia colpito l'una o l'altra moltiplicazione modulare. Comunque la ricerca precedente è in grado di evidenziare a posteriori in quale caso ci si trovi.

Nel caso le parole trovate corrispondano alle parole della chiave segreta sarà necessario soltanto trovare la permutazione giusta di tali parole per ricostruire la chiave. Ciò può essere effettuato per tentativi esaustivi, ma il costo dell'operazione è piuttosto elevato se la chiave è composta di più di 16 parole. Abbiamo perciò trovato il modo di ottenere degli indizi dai risultati della procedura di ricerca al fine di limitare notevolmente il numero di tentativi da effettuare. In

casi fortunati, ma realistici, questo può ridursi anche ad un tentativo solo, ovvero è nota in partenza la posizione di ogni parola della chiave. Questa procedura è dettagliata nelle sezioni 5.5.2 e 5.7.1.

A.5 Analisi dell'Attacco

L'attacco descritto è stato verificato con sia con simulazioni software che su un dispositivo reale.

Le simulazioni hanno evidenziato la correttezza delle analisi e delle procedure proposte. È stato inoltre possibile valutare le prestazioni di dette procedure su versioni ridotte del crittosistema. Una stima per un caso reale dice che è possibile recuperare una chiave da 256 bit in meno di 4 ore.

Per i dettagli dei risultati ottenuti con le simulazioni, si veda il capitolo 6.

L'attacco è stato verificato anche su un dispositivo fisico, utilizzando un metodo di iniezione di guasto che sfrutta la sottoalimentazione del chip. Il modello di errore della board usata è noto e particolarmente adatto per l'attacco da testare, consistendo nel cambiamento di un bit di cui è nota a priori la posizione all'interno della parola e la direzione del cambiamento, ovvero da 1 a 0. I guasti sono stati effettivamente trovati e le moltiplicazioni hanno subito i cambiamenti predetti. Un problema si è però verificato durante il test su operazioni più complesse della moltiplicazione intera. Il modello di errore del dispositivo si è infatti rivelato essere diverso quando vengono eseguite divisioni intere a precisione doppia, non implementate in hardware.

I dettagli dei test e alcune idee per fronteggiare questo problema si trovano ancora nel capitolo 6.

A.6 Contromisura

Una contromisura indirizzata particolarmente all'attacco descritto è stata inoltre realizzata. Si tratta di una modifica all'algoritmo di calcolo del secondo elemento della firma, tale da rendere impossibile l'analisi dei risultati descritta nelle precedenti sezioni. La contromisura è dedicata a questo specifico attacco, ma verosimilimente l'idea è abbastanza generale da consentire di proteggere anche contro eventuali attacchi futuri indirizzati allo stesso frammento dell'algoritmo di firma.

I dettagli della modifica da apportare all'algoritmo sono riportati nel capitolo 7. Caratteristica positiva della contromisura sviluppata è il costo estremamente basso in termini di operazioni aggiunte all'algoritmo.

A.7 Conclusioni

Abbiamo presentato un contro ECDSA, basato sui guasti, e di tipo innovativo, in quanto viene attaccata una parte dell'algoritmo di generazione della firma tipicamente ritenuta meno interessante. Il modello di guasto richiesto per l'attacco, ovvero il bit flip, è semplice ed è tra i più studiati in letteratura. Inoltre esso è ottenibile anche con strumentazione molto economica, come quella utilizzata per i nostri esperimenti.

L'attacco richiede poche firme errate, e può essere facilmente adattato a casi specifici migliorandone le prestazioni. Anche la possibilità di parallelizzare al massimo ogni procedura consente di raggiungere buone prestazioni.

Una contromisura efficace è stata sviluppata, ed ha un costo molto basso in termini di tempo di esecuzione.

Possibili sviluppi futuri di questo lavoro si possono trovare tra lo studio di modelli di errore più complicati rispetto al cambiamento di un singolo bit e lo studio di algoritmi diversi per l'esecuzione della moltiplicazione modulare, v. capitolo 8. Semplici esempi per quanto riguarda diversi modelli di errore sono riportati in sezione 5.8.

Bibliography

[ABF+03] C. Aumüller, P. Bier, W. Fischer, P. Hofreiter, and J.-P. Seifert. Fault attacks on rsa with crt: Concrete results and practical countermeasures. In Burton Kaliski, çetin Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2002*, volume 2523 of *Lecture Notes in Computer Science*, pages 81–95. Springer Berlin / Heidelberg, 2003.

[AK96] Ross Anderson and Markus Kuhn. Tamper resistance: a cautionary note. In *WOEC'96: Proceedings of the 2nd conference on Proceedings of the Second USENIX Workshop on Electronic Commerce*, number 2, pages 1–11, Berkeley, CA, USA, 1996. USENIX Association.

[ANS05] ANSI. *ANSI X9.62*. American National Standards Institute, 2005.

[BBPP09] Alessandro Barenghi, Guido Bertoni, Emanuele Parrinello, and Gerardo Pelosi. Low voltage fault attacks on the RSA cryptosystem. In *Fault Diagnosis and Tolerance in Cryptography*, pages 23–31. IEEE Computer Society, 2009.

[BDH+98] F. Bao, R. H. Deng, Y. Han, A. Jeng, A. D. Narasimhalu, and T. Ngair. Breaking public key cryptosystems on tamper resistant devices in the presence of transient faults. In *Security Protocols*, pages 115–124. Springer-Verlag, 1998.

[BDL97] Dan Boneh, Richard A. Demillo, and Richard J. Lipton. On the importance of checking cryptographic protocols for faults. pages 37–51. Springer-Verlag, 1997.

[BECN+06] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan. The sorcerer's apprentice guide to fault attacks. volume 94 of *Proceedings of the IEEE*, pages 370–382. IEEE, 2006.

[BMM00] Ingrid Biehl, Bernd Meyer, and Volker Müller. Differential fault attacks on elliptic curve cryptosystems. In *CRYPTO '00: Proceedings of the 20th Annual International Cryptology Conference on Advances in Cryptology*, pages 131–146, London, UK, 2000. Springer-Verlag.

[BOS04] Johannes Blömer, Martin Otto, and Jean-Pierre Seifert. Sign change fault attacks on elliptic curve cryptosystems. cryptology

eprint archive, report 2004/227. In *Fault Diagnosis and Tolerance in Cryptography 2006 (FDTC '06), volume 4236 of Lecture Notes in Computer Science*, pages 36–52. Prentice Hall, 2004.

[CJ05] Ciet and Joye. Elliptic curve cryptosystems in the presence of permanent and transient faults. In *Designs, Codes and Cryptography*, pages 33–43. Springer Netherlands, 2005.

[Cry] Crypto++. *Crypto++ benchmark.* http://www.cryptopp.com/benchmarks.html.

[DH75] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. In *IEEE Information Theory Workshop*, 1975.

[FLRV08] Pierre-Alain Fouque, Reynald Lercier, Denis Réal, and Frédéric Valette. Fault attack on elliptic curve montgomery ladder implementation. In *FDTC '08: Proceedings of the 2008 5th Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 92–98, Washington, DC, USA, 2008. IEEE Computer Society.

[GA03] Sudhakar Govindavajhala and Andrew W. Appel. Using memory errors to attack a virtual machine. In *SP '03: Proceedings of the 2003 IEEE Symposium on Security and Privacy*, page 154, Washington, DC, USA, 2003. IEEE Computer Society.

[HGS01] N. A. Howgrave-Graham and N. P. Smart. Lattice attacks on digital signature schemes. *Des. Codes Cryptography*, 23(3):283–290, 2001.

[HMV04] Darrel Hankerson, Alfred J. Menezes, and Scott Vanstone. *Guide to Elliptic Curve Cryptography.* Springer-Verlag, 2004.

[JY02] Marc Joye and Sung-Ming Yen. The montgomery powering ladder. In *CHES 2002, LNCS*, pages 291–302. Springer-Verlag, 2002.

[KIIK08] M. Kara-Ivanov, E. Iceland, and A. Kipnis. Attacks on authentication and signature schemes involving corruption of public key (modulus). In *FDTC '08: Proceedings of the 2008 5th Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 108–115. IEEE Computer Society, 2008.

[KQ07] Chong Kim and Jean-Jacques Quisquater. Fault attacks for crt based rsa: New attacks, new results, and new countermeasures. In Damien Sauveron, Konstantinos Markantonakis, Angelos Bilas, and Jean-Jacques Quisquater, editors, *Information Security Theory and Practices. Smart Cards, Mobile and Ubiquitous Computing Systems*, volume 4462 of *Lecture Notes in Computer Science*, pages 215–228. Springer Berlin / Heidelberg, 2007.

[LAM+07] R. Leveugle, A. Ammari, V. Maingot, E. Teyssou, P. Moitrel, C. Mourtel, N. Feyt, J.-B. Rigaud, and A. Tria. Experimental evaluation of protections against laser-induced faults and consequences on fault modeling. In *DATE '07: Proceedings of the conference on Design, automation and test in Europe*, pages 1587–1592, San Jose, CA, USA, 2007. EDA Consortium.

[Mon87] Peter L. Montgomery. Speeding the pollard and elliptic curve mthods of factorization. *Mathematics of Computation*, 48:243–264, 1987.

[MRL+06] Y. Monnet, M. Renaudin, R. Leveugle, N. Feyt, P. Moitrel, and F. M'Buwa Nzenguet. Practical evaluation of fault countermeasures on an asynchronous des crypto processor. In *IOLTS '06: Proceedings of the 12th IEEE International Symposium on On-Line Testing*, pages 125–130, Washington, DC, USA, 2006. IEEE Computer Society.

[NIS09] NIST. *FIPS PUB 186-3: Digital Signature Standard*. National Institute of Standards and Technology, 2009.

[NNTW05] David Naccache, Phong Q. Nguyen, Michael Tunstall, and Claire Whelan. Experimenting with faults, lattices and the dsa. In *Public Key Cryptography*, pages 13–24. Springer-Verlag, 2005.

[NS03] Phong Q. Nguyen and Igor E. Shparlinski. The insecurity of the elliptic curve digital signature algorithm with partially known nonces. *Des. Codes Cryptography*, 30(2):201–217, 2003.

[NSA10] NSA. *NSA Suite B Cryptography*. National Security Agency, 2010.

[Ope] OpenSSL. *OpenSSL*. The OpenSSL Project http://www.sagemath.org.

[PH78] Stephen C. Pohlig and Martin E. Hellman. An improved algorithm for computing logarithms over GF(p) and its cryptographic significance. In *IEEE Transactions on Information Theory*, pages 106–110. IEEE Computer Society, 1978.

[QS02] Jean-Jacques Quisquater and David Samyde. Eddy current for magnetic analysis with active sensor. In *Esmart 2002, Nice, France*, 2002.

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.

[SA03] Sergei P. Skorobogatov and Ross J. Anderson. Optical fault induction attacks. In *CHES '02: Revised Papers from the 4th International Workshop on Cryptographic Hardware and Embedded Systems*, pages 2–12, London, UK, 2003. Springer-Verlag.

[SAG] SAGE. *Sage Math*. Sage Math http://www.sagemath.org.

[SH08] J?rn-Marc Schmidt and Christoph Herbst. A practical fault attack on square and multiply. *Fault Diagnosis and Tolerance in Cryptography, Workshop on*, 0:53–58, 2008.

[SM09] Jörn-Marc Schmidt and Marcel Medwed. A fault attack on ecdsa. In David Naccache and Elisabeth Oswald, editors, *6th Workshop on Fault Diagnosis and Tolerance in Cryptography - FDTC 2009, Proceedings*, pages 93 – 99. Verlag IEEE-CS Press, 2009.

[Wil95] Andrew Wiles. Modular forms, elliptic curves, and fermat's last theorem. In *The Annals of Mathematics. Second Series*, volume 142, pages 443–551. Princeton University Press, 1995.