Studies and implementation of hardware countermeasures for ECDSA cryptosystems

Jérémy Dubeuf

▶ To cite this version:

Jérémy Dubeuf. Studies and implementation of hardware countermeasures for ECDSA cryptosystems. Micro and nanotechnologies/Microelectronics. Université Grenoble Alpes, 2018. English. NNT : 2018GREAT048 . tel-03644277

HAL Id: tel-03644277
https://theses.hal.science/tel-03644277v1

Submitted on 19 Apr 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Communauté
UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE LA COMMUNAUTÉ UNIVERSITÉ GRENOBLE ALPES

Spécialité : NANO ELECTRONIQUE ET NANO TECHNOLOGIES

Arrêté ministériel : 25 mai 2016

Présentée par

Jérémy DUBEUF

Thèse dirigée par **Vincent BEROULLE**
et codirigée par **David HELY**

préparée au sein du **Laboratoire Laboratoire de conception et d'intégration des systèmes**
dans **l'École Doctorale Electronique, Electrotechnique, Automatique, Traitement du Signal (EEATS)**

Etude et implémentation de contre-mesures matérielles pour la protection de dispositifs de cryptographie ECDSA

Studies and implementation of hardware countermeasures for ECDSA cryptosystems

Thèse soutenue publiquement le **3 mai 2018**,
devant le jury composé de :

Monsieur VINCENT BEROULLE
MAITRE DE CONFERENCES, GRENOBLE INP, Directeur de thèse
Monsieur SYLVAIN GUILLEY
PROFESSEUR, TELECOM PARISTECH, Rapporteur
Monsieur LILIAN BOSSUET
PROFESSEUR, UNIVERSITE JEAN MONNET - SAINT-ETIENNE, Rapporteur
Madame MARIE-LISE FLOTTES
CHARGE DE RECHERCHE, CNRS DELEGATION LANGUEDOC-ROUSSILLON, Examinateur
Monsieur PHILIPPE ELBAZ-VINCENT
PROFESSEUR, UNIVERSITE GRENOBLE ALPES, Président
Monsieur DAVID HELY
MAITRE DE CONFERENCES, GRENOBLE INP, Examinateur

Studies and Implementation of Hardware Countermeasures for ECDSA Cryptosystems

Contents

List of Figures 7

List of Tables 9

List of Algorithms 11

List of Acronyms 13

1 Introduction 15

1.1 Information Security, a necessity 15
1.2 Integrated Circuit 17
1.3 Side channel attacks 20
1.4 Fault injection attacks 22
1.5 Ph.D. interests 23

2 Elliptic Curve Cryptography 27

2.1 Group, ring and field 28
2.2 Generalities on ECC 30
2.2.1 Computation of the ECC scalar operation 32
2.3 ECDLP and Security level of ECC 33
2.4 Point representation 35
2.5 Elliptic Curve Digital Signature Algorithm 36
2.5.1 Key generation 37
2.5.2 Signature 37
2.5.3 Verification 37

2.6 Lattice attacks on ECDSA 37
2.6.1 Gathering signatures and information 38
2.6.2 Computing a lattices attack 38
2.6.3 Basic numerical example of a lattice attack 40
2.6.4 Lattice attack results on NIST P256 42
2.7 Summary 43

3 Side Channel Against ECC and ECDSA 45
3.1 Scalar algorithms and leakages sources 46
3.1.1 Double-and-Add 47
3.1.2 Fixed-base windowing 49
3.1.3 Fixed-base comb method 51
3.1.4 Double-and-Add countermeasures 53
3.1.5 Global timing 56
3.2 Leakages on underlying algorithms 56
3.3 Leakages usage against the ECDSA 60
Example on NIST P-256 60
3.4 Summary 64

4 Fault Attacks Against ECC and ECDSA 67
4.1 Safe-error attack against the scalar algorithm 69
4.1.1 Local dummy operations, C safe-error against the Montgomery ladder 71
4.1.2 Unused memory values 73
4.1.3 The infinity point and dummy operands 74
4.1.4 Safe-error on underlying algorithms 76
4.2 Fault attacks against the ECDSA signature 78
4.2.1 Faulted secret key 78
4.2.2 Faulted nonce 81
4.2.3 Faulted intermediate values 85
4.2.4 Example of architecture and fault opportunities 86
Architecture description 86
Signatures with shared nonces 88
Signature nonces with mostly shared bits 89
Nonces with some known or shared bits 89
Signature generations with faulted secret key 90
Signature generations with faulted nonces 91
4.3 Summary 92

5 Side Channel and Fault Countermeasures 95
5.1 Our secure scalar point multiplication 96
5.1.1 Scalar operation 97
5.1.2 Scalar operation with pre-calculation 99
5.1.3 The non-standard $2k \cdot P$ result and exceptional cases 100
5.2 Secure implementation strategy 101
5.2.1 Scalar with a fixed base point: $k \cdot P$ 101
5.2.2 Sum of two scalars: $k \cdot P + v \cdot G$ 103
5.2.3 Scalar with any base point: $k \cdot G$ 104
5.3 Preventing attacks against ECDSA nonce and private key 106
5.3.1 ECDSA signature, s part private key countermeasures 106
5.3.2 ECDSA signature, s part nonce countermeasures 109
Countermeasure against signatures with faulted nonce 109
Countermeasure against nonce updating tampering 115
5.4 Algorithms Summary 116

6 Conclusion 119

References 125

Résumé substantiel 137

List of Figures

1.1 Integrated Circuit with and opened package 17
1.2 $NMOS$ transistor, left represents the physical implementation, right is the equivalent schematic representation. 18
1.3 $CMOS$ inverter, left represents the transistor level representation, right is the gate level representation. 19

2.1 Hierarchy of $ECDSA$ operations. 27
2.2 Left: Elliptic curve point addition operation over the field of real numbers. Right: Elliptic curve point doubling operation over the field of real numbers. 31
2.3 Security level of ECC vs RSA, [1]. 34
2.4 Lattice illustration in 2 dimensions 39
2.5 Lattice attack result against $NIST$ P256 42

3.1 Double-and-Add left to right power trace
axe: Time y-axe: Power consumption. 47
3.2 SPA result of left to right
axe: Time y-axe: Power consumption. 48
3.3 Fixed-base windowing power trace
axe: Time y-axe: Power consumption. 50
3.4 SPA result of the fixed-base windowing implemented with Jacobian representation.
axe: Time y-axe: Number of peaks 50
3.5 Patterns extracted from a power trace of Algorithm 3.3 execution.
axe: Time y-axe: Number of peaks. 52

3.6 Key used with Algorithm 3.3 and Figure 3.5. Arrows represent the parsed secret columns when the 2nd orphan point addition appears 53
3.7 SPA results of Algorithm 3.4.
axe: Time y-axe: Correlation value. 54
3.8 Comparison of two different scalar operations based on Algorithm 3.4.
x-axe: Time y-axe: Correlation value. 56
3.9 Comparison between genuine scalar and attacker scalar.
genuine, grey: attacker, $0xE00 \cdots 0000$
axe: Time y-axe: Power consumption. 58
3.10 2nd comparison between genuine scalar and attacker scalar.
genuine, grey: attacker, $0xC00 \cdots 0000$
axe: Time y-axe: Power consumption. 58
3.11 3rd comparison between genuine scalar and attacker scalar.
genuine, grey: attacker, $0xA00 \cdots 0000$
axe: Time y-axe: Power consumption. 58
3.12 4th comparison between genuine scalar and attacker scalar.
genuine, grey: attacker, $0x800 \cdots 0000$
axe: Time y-axe: Power consumption. 59
3.13 Superposition of power traces from different signatures.
axe: Time y-axe: Power consumption. 61

4.1 Illustration of a laser-based fault injection setup 68
4.2 Illustration of the number of faults required to recover an 8 bits value arbitrary set to 0x84 80
4.3 Average of the number of fault required to recover an 8 bits value . . . 80
4.4 Error e distribution on a random 8 bits register 82
4.5 Total number of required signatures to recover the private key thanks to a lattice attack and random fault on MSB bits on NIST-P256 83
4.6 Number of occurrences of error e on a random $m = 8$ bits register that is forced to a constant value $v = 54$ 84
4.7 Error e distribution on a random $m = 8$ bits register that lead to two different values $v_1 = 54$, $v_2 = 168$ 85
4.8 Overview of the considered architecture. Grey blocks containing some side channel and fault countermeasures 87
4.9 Chronograph of a $ECDSA$ signature generation 87
4.10 Illustration of an 8 bits to 32 bits interface 91

List of Tables

2.1 Security Level of ECC vs RSA, [1]. 33
2.2 Operation counts for EC double and EC addition. 36
2.3 Lattice example, known information summary 40

3.1 Detailed operation flow of Algorithm 3.4 depending on the MSB . . . 55
3.2 Figure 3.9 to 3.12 results summary 59

4.1 Representation of the neutral element in different systems 76
4.2 AES bits accumulation through $SRAM$ read/write iteration 92

5.1 Algorithms summary 117

List of Algorithms

2.1 Left-to-right Double-and-Add 33

3.1 Left-to-right Double-and-Add 47
3.2 Fixed-base windowing EC scalar multiplication algorithm 49
3.3 Fixed-base comb method with two tables 52
3.4 Left to right Double-and-Always-Add 54
3.5 Coron always Double-and-add 57

4.1 Coron always Double-and-add 70
4.2 Montgomery scalar operation 71
4.3 Montgomery scalar operation with a specific scalar value 72
4.4 Binary Expansion with RIP (BRIP) 74
4.5 Scalar operation with pre-computed points 75
4.6 Right-to-left double-and-add always with coherency checking 76
4.7 ECC Jacobian point addition 77

5.1 Scalar operation 97
5.2 ECC Jacobian-affine point addition/subtraction 98
5.3 Modified Comb method 100
5.4 $2kP$ operation optimized with two precomputed points 102
5.5 $2k \cdot P + 2v \cdot G$ operation 103
5.6 kG operation computed as $2k \cdot G = k_1 G + k_2(r \cdot G)$ 105
5.7 Field scalar operation 110
5.8 Side channel and fault resistant field scalar operation 111
5.9 EC and field scalar operation with a CFI 112

5.10 ECC Jacobian point doubling and field doubling on NIST curves: double(Q,q) 113
5.11 ECC Jacobian-affine point addition/subtraction and field addition/subtraction: addsub(Q,P,q,u,m,b,r) 114

List of Acronyms

ADPA	Address bit Differential Power Analysis.
AES	Advanced Encryption Standard.
AIS	Anwendungshinweise und Interpretationen.
CCD	Charge-Coupled Device.
CFI	Control Flow Integrity.
CIA	Central Intelligence Agency.
CMOS	Complementary-Metal-Oxide-Semiconductor.
CPA	Correlation Power Analysis.
CPU	Central processing Unit.
CRT	Chinese Reminder Theorem.
CVP	Closest Vector Problem.
DES	Data Encryption Standard.
DFA	Differential Fault Analysis.
DLP	Discrete Logarithm Problem.
DPA	Differential Power Analysis.
DRAM	Dynamic Random-Access Memory.
DSA	Digital Signature Algorithm.
DSS	Digital Signature Standard.
DUT	Device Under Test.
DVFS	Dynamic Voltage and Frequency Scaling.
EC	Elliptic Curve.
ECC	Elliptic Curve Cryptography.
ECDH	Elliptic Curve Diffie-Hellman.
ECDLP	Elliptic Curve Discrete Logarithm Problem.

$ECDSA$	Elliptic Curve Digital Signature Algorithm.
$ECIES$	Elliptic Curve Integrated Encryption Scheme.
$EEPROM$	Electrically Erasable Programmable Read-Only Memory.
EM	Electromagnetic.
$FBBI$	Forward Body Biased Injection.
FET	Field-Effect-Transistor.
$I2C$	Inter-Integrated Circuit.
IC	Integrated Circuit.
LED	Light-Emitting Diode.
LLL	Lenstra-Lenstra-Lovász.
LSB	Less Significant Bit.
$MOSFET$	Metal-Oxide-Semiconductor-Field-Effect-Transistor.
MOV	Menezes Okamoto and Vanstone.
MSB	Most Significant Bit.
NAF	Non-Adjacent Form.
$NIST$	National Institute of Standards and Technology.
$NMOS$	Negative-channel MOSFET.
NSA	National Security Agency.
NVM	Non-Volatile Memory.
$Nd:YAG$	Neodymium-Doped Yttrium Aluminium Garnet.
PCB	Printed Circuit Board.
$PMOS$	Positive-channel MOSFET.
RAM	Random-Access Memory.
RNG	Random Number Generator.
RPA	Refined Power Analysis.
RSA	Rivest-Shamir-Adleman.
SHA	Secure Hash Algorithm.
$SHA–1$	Secure Hash Algorithm 1.
$SHA–2$	Secure Hash Algorithm 2.
$SHA–256$	Secure Hash Algorithm 2 with 256 bits.
SOC	System On Chip.
SPA	Simple Power Analysis.
$SRAM$	Static Random-Access Memory.
ZPA	Zero-value Point Attacks.
pmf	Probability mass function.

CHAPTER 1

Introduction

This first chapter introduces the importance of information security and its relation to Integrated Circuits (IC). After a brief description of IC, non-invasive physical threats are described. These threats, side channel and fault attacks, represent a real security risk as they can allow extracting secrets from the IC. Ensure secrets protection of ICs is thus a challenge and this work aims at improving the security of Elliptic Curve Digital Signature Algorithm implementation through various contributions described at the end of this chapter.

1.1 Information Security, a necessity

Nowadays, information is everywhere and used for everything. The amount of information generated and exchanged by human is tremendous and continue to rise exponentially as new technologies are created. Every field of human activity relies somehow on information and many decisions are based on them. Thus, information trust is essential especially for critical systems and infrastructures. Military, government, financial, health and safety related information are usually considered with a special care as obviously critical to the good of our society. However, information in every field mays be considered as critical depending on the point of view. Many questions related to information may be asked. As example, how does a decision-maker is supposed to choose a good decision based on erroneous information? Where the decision-maker may be a human, a complex computer program or a simple algorithm in a basic piece of electronic. What happen and which advantages are preserved if competitors have access to the same information? Thus, how to ensure a privileged access to the information? How ensuring that the information is correct? Does the source is trustworthy? Does the communi-

cation channel preserves the integrity without eavesdropping? Does the information reaches the good recipient?
Security aims at answering these questions. More generally, information security aims at ensuring the preservation of confidentiality, integrity and availability of the information. Depending on the context, authenticity, accountability, non-repudiation and reliability can also be involved [2]. In [3] authors summarized the goals of information security according to different references.
Considering the previous questions, it becomes obvious that information security is a necessity. Cryptography is important to information security as it can provide confidentiality, integrity, authenticity and non-repudiation. As these terms are essential for cryptography, we give their definitions as follows, [4]:

Confidentiality: Ensures that data are accessible only to those that are authorized and unintelligible to others.

Integrity: Ensures that the data are not modified without a proper authorization.

Authenticity: Ensures that an entity is what it claims to be.

Non-repudiation: Ability to prove the occurrence of a claimed event or action and its originating entities

Modern cryptography considers three different kinds of algorithm, symmetric, asymmetric and hash functions. Symmetric algorithms allow encryption and decryption with the same key. The Advanced Encryption Standard (AES) is one of them and in its basic use case allows confidentiality. From a readable information called plaintext and the secret key, it generates an unreadable output called ciphertext. Only those that know the secret key can transform the ciphertext back to the plaintext. A deterministic symmetric cipher algorithm operating on fixed-length groups of bits is called block cipher. How the block cipher algorithm is used is called the mode of operation. Depending on the selected operation mode, authenticity and integrity can also be provided. In the field, the use of symmetric ciphers can be inconvenient due to the required key distribution. As opposed to symmetric algorithm, asymmetric cryptography, also named public key cryptography, uses pairs of keys. Usually referred as the private key and the public key. The private key is secretly known by one actor while the public key can be known from everyone. The public key can then be used to verify that someone knows the private key, providing authenticity. The public key can also be used to encrypt a message that only the holder of the private key can decrypt, providing confidentiality.

Hash functions aim at mapping data of arbitrary size to data of fixed size. Cryptographic hash function requires different properties such as pre-image resistance, second pre-image resistance and collision resistance. The pre-image resistance prevents to inverse the function, from a given hash value h, it should be difficult to find any message m such that $h = hash(m)$. The second pre-image resistance prevents to find another message with the same hash result. Given an input m_1 it should be difficult to find a different input m_2 such that $hash(m_1) = hash(m_2)$. Then the collision resistance, prevent to find two different messages with the same hash result. Hash function allows providing integrity and depending how used may also provide authenticity. Secure protocols use these different kinds of algorithms inside communications in order to ensure the various security requirements such as the confidentiality, integrity, authenticity and non-repudiation of the data.

Modern cryptography is complex and thus is computed by machines. Nowadays, cryptographic algorithms are usually implemented into integrated circuits (IC). Either the implementations is directly defined by a specific hardware architecture or it mays be a software running on a Central Processing Unit (CPU). The security level provided by the ICs, meaning, the security risks that ICs should be able to withstand, is thus important for the overall security. In the next section, an overview on IC is provided.

1.2 Integrated Circuit

Integrated Circuit (IC) is a small electronic circuit on a semiconductor material, such as silicon, embedded in a package. The circuit, called die, is inserted in a package and wires allow connecting it to lead frames. The lead frames allow soldering the IC to a printed circuit board (PCB) and provide connections with other components of the PCB. Figure 1.1 shows an IC with an opened package, allowing to see the internal die and bondings.

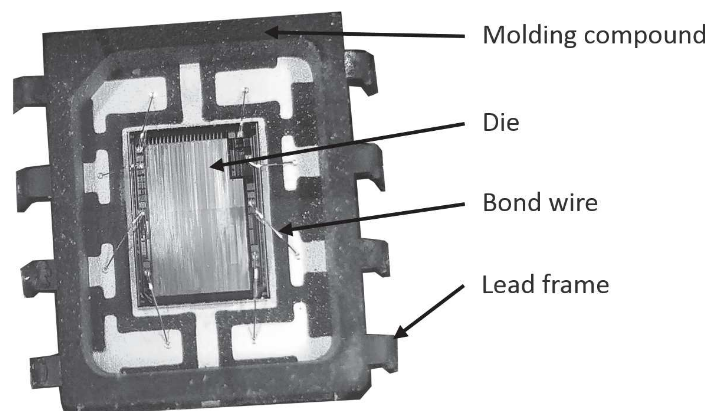

Figure 1.1: Integrated Circuit with and opened package

A semiconductor material has a conductivity between conductor and insulator and, as opposed to metals, increase alongside their temperature. This property allows to easily control the conductivity depending on the amount of energy provided to the material either with thermal or electric excitation. Moreover, their conducting properties can be locally altered by deliberately introducing impurities. This process is called doping. By inserting e.g. phosphorus impurities into silicon, it generates an excess of electron free to move and then improving the conductivity. This results to what is called n-type doped semiconductors. An opposed effect can be obtained by inserting e.g. boron impurities to silicon. By doing so, electron void in the silicon lattice structure called hole are created. These holes readily accepts electrons, decreasing the conductivity. This is called p-type doped semiconductors.

By using these properties, is it possible to build a Field-Effect-Transistor (FET) directly on a semiconductor. This is called Metal-Oxide-Semiconductor Field-Effect Transistor ($MOSFET$). A p-type doped semiconductor is used as substrate and two n-type regions are added, forming the transistor source and drain. Then a gate is added between the two regions and isolated with an oxide. This construction is depicted on the left side of figure 1.2.

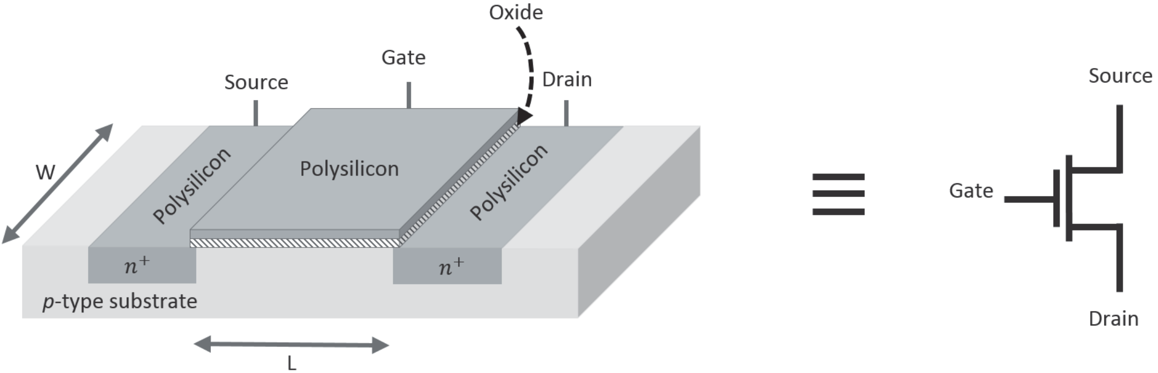

Figure 1.2: $NMOS$ transistor, left represents the physical implementation, right is the equivalent schematic representation.

The two n-type regions have a high conductivity and are separated by the substrate which has a low conductivity thus isolating the source and the drain. When a positive voltage is applied between the gate and the source, some electrons move away from the gate, creating positively charged holes in the gate. These positive charges create a field which attracts substrate electrons close to the oxide generating an inversion region which is conductive. Electrons can then freely move between the source and the drain. Such transistors are called negative-channel $MOSFET$ or $NMOS$. They can be used as a controlled switch, which is either open if the voltage between the

gate and the source is small, or closed if the voltage is higher than the threshold. By inverting the doping, i.e. using an n-type substrate and p-type doping for the source and drain, another $MOSFET$ is obtained, called positive-channel $MOSFET$ or $PMOS$. $PMOS$ works similarly to $NMOS$, however allows current to pass between source and drain when the gate voltage is low and prevents it to pass when the voltage is high.

As these transistors may act as controlled switches, it is possible to build-upon them logic cells. As example, in figure 1.3, a basic inverter is built from a $PMOS$ plus a $NMOS$ transistor.

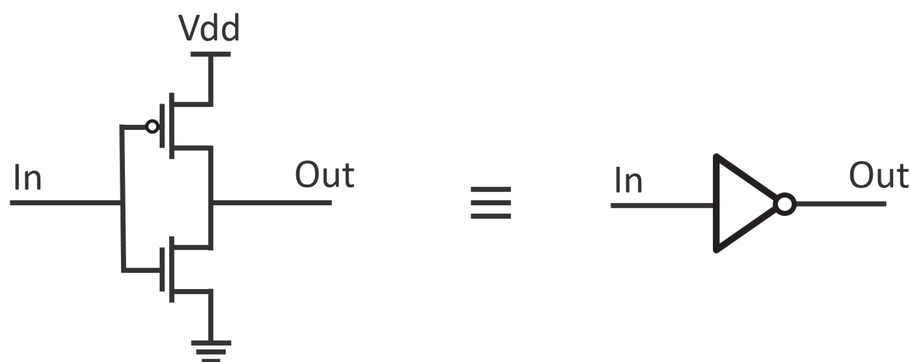

Figure 1.3: $CMOS$ inverter, left represents the transistor level representation, right is the gate level representation.

Other basic logic cells such as AND, OR, XOR, NAND can be built from transistors. This kind of logic that uses both $NMOS$ and $PMOS$ transistors is called Complementary-Metal-Oxide-Semiconductor ($CMOS$) logic. It is also possible to build any logic blocks using only $NMOS$ transistors, which is called $NMOS$ logic or using only $PMOS$ which is called $PMOS$ logic.

The use of different logic gates, allows the implementation of other elements such as flip flops and by using flip flop alongside logic elements, complex functions such as finite state machines, memories or CPUs can be built. In synchronous designs, clocked flip flops are used. The clock which is a periodic signal is propagated to all the flip flop over the design allowing them to evaluate their outputs simultaneously. The clock frequency is adjusted in such a way that signals have enough time to propagate from a flip flop to another through the logic. For a given IC, propagation times vary depending on the temperature and power supply. The slowest path between two flip flops is called the critical path and determines the maximum clock frequency that can be used in the worse temperature and power supply conditions according to the specification.

The number of a transistors in ICs double approximately every two years since the 1970s. This observation is called Moore's law and it allowed to move from thousands

of transistor per IC in the 1970s to a couple of billions in modern ICs. This rise is due to node technology improvement that moved from 10μm in 1970s, to 7nm in 2017. Originally, the technology node was defined by the length of transistors gates, i.e. the L dimension in figure 1.2. Later, the International Technology Roadmap for Semiconductors defined the technology node as the smallest half-pitch of contacted metal 1 lines allowed in the fabrication process. Nowadays, the node name is more a label than a physical meaning.

ICs are thus the physical support of various algorithm implementations including cryptography. Unfortunately, they can be observed or perturbed when operating. In the next sections, explanation about these non-invasive physical threats are provided.

1.3 Side channel attacks

While modern cryptography is considered mathematically secure, meaning that it is neither possible to recover the plaintext without the key nor to recover the key from the plaintext and the ciphertext in a reasonable amount of time, it needs a physical support to be implemented. Nowadays, ICs are used to compute cryptographic algorithms. The execution of these algorithms are not instantaneous, meaning that they require time and intermediate results are processed. During the computation, the IC can then be observed from different perspectives. Information or partial information of the running algorithm can be extracted from the observation and can be enough to reveal secrets; this is called side channel attacks.

The common observations used to recover information are:

Timing [5]: Some parts of algorithms may imply time variations depending on the processed data. By observing the time, some data can eventually be recovered.

Power consumption [6]: ICs are composed of transistors and they require energy to commute from a state to another thus generating a dynamic leakage. Once in a given state, a leakage current may also exist providing a static leakage. The overall power consumption of an IC depends on the internal activity. Thus by observing the power consumption, the algorithmic sequence can be observed providing accurate timing information. Signals amplitude may also depend on the processed data and thus may provide information on intermediate results.

Electromagnetic emanations [7]: Current flowing through transistors means moving charges and thus magnetic field. Electromagnetic (EM) emanations thus

roughly contains the same kind of information than the power consumption. However, while power consumption of an IC is observed for the whole chip, EM probes can be placed at different positions over the chip. Thus EM add spacial freedom for the observation over power consumption. This freedom may allow improving the signal to noise ratio and thus enhance the analysis.

Photonic emission [8]: Light emissions in silicon devices had been observed for the first time in 1955 [9] and is used for failure analysis [10]. $CMOS$ logic emits photon depending on the switching activity. Thus photon emission are data-dependent and by observing them, internal information of the IC can be obtained.

Acoustic [11]: Capacitor squeal and coil whine are often observed in computer systems. The noise, typically caused by voltage regulation circuits, depends on the system load and thus on the activity. By analyzing the acoustic emanation generated during a cryptography operation, information can be obtained. The main difference with power analysis is due to the low signal bandwidth that is lower than the system speed. However, [11] demonstrated the feasibility of using acoustic to recover cryptographic secrets.

It is to be noted that leakages do not necessarily provide direct information about secrets. Signal processing and statistical methods are used to leverage the leakage and obtain the information. The Differential Power Attack (DPA) [6] presented by Paul Kocher in 1999 is a good example of such practice. Current research about side channel tends to be focused on the signal processing, the distinguisher and on statistical leakage assessment tools. Side channel observation through software is also a hot topic as it allows targeting remote devices. Cache-timing attack [12], or the more recent spectre [13] and meltdown [14] attacks are interesting examples. While remote analysis does not allow accurate timing analysis, malicious software running on the target device may provide the information and unfortunately, side channel countermeasures are often omitted for non-physically accessible devices. Side channel source of information can also be the observation of the software execution behavior, the Differential Computation Analysis presented in [15] demonstrates that monitoring stack access of a withe-box cryptographic algorithm can allow recovering enough information to get the private key.

1.4 Fault injection attacks

Cryptographic algorithms are implemented in a physical support. And unfortunately this support can be manipulated or perturbed during the computation resulting in various errors. As example, the errors or faults can be a modification of the operation flow, data or memory corruption. The most common fault models are the bit-flip which inverts the value of a bit, the bit-set or bit-reset which either set a bit to 1 or to 0 and the stuck-at which locks the bit value. Faults may occur on a single bit or on a set of bits. Depending on how the fault is injected, the result can be different. Fault injection attacks aim at injecting a fault during the computation on an algorithm to obtain secret information or trigger a specific event. Its effect on the system can be various such as altering processed data, modifying addresses or changing the operation flow.

Numerous ways of inserting faults to an IC had been considered such as:

Overclocking [16]: IC clock signal sometime come from the outside and thus can be controlled by attackers. By inserting a clock with a frequency out of the specification, timing violations appear in the design resulting to faults. By syncing the clock frequency overclocking around the timing where sensitive information are processed, it is possible to insert faults during specific parts of the running algorithm.

Under-powering [17]: Under-powering the IC allows modifying the propagation delays and thus, similarly to overclocking, can generate timing violations which result into faults.

Clock or power glitching [18], [19]: By inserting a short pulse either on the power supply or on the clock, propagation delays can be violated resulting to faults. The glitch can be inserted with a high timing accuracy helping attackers to control the fault injection more precisely than overclocking or under-powering.

Temperature [20]: Propagation delays vary depending on the temperature, thus temperature is also a common way to fault an IC.

Magnetic pulses [21]: By using a coil to generate a magnetic pulse close to the IC, transient faults can be injected. The coil can be placed at different positions over the chip. Thus providing attackers fault control for both timing and position.

Laser [22]: Light contains photons which can interact with $CMOS$ transistors. By using a laser with a small beam, faults can be precisely injected in the IC with a

high time accuracy. Moreover, depending on the laser wavelength, both frontside and backside fault injection are possible.

Substrate biasing [23], [24]: The idea is to directly inject current into the IC which will generate faults. This can be done by applying a high magnitude transient voltage pulse to a needle in contact to the substrate. The attack requires a partial backside access for the needle and allows a high control over the fault injection. This kind of attack is called Forward Body Biasing Injection attack ($FBBI$).

Devices are designed according to a specification that contains working conditions. The basic question when faults are considered is what happen outside the specification? Or at the edge of the worse working condition? Attack methods can be combined (e.g. under-powering at the specification limit plus increasing the temperature and working at the maximum frequency) to stay as close as possible of the specification to generate fault and avoid detections. Fault injection techniques can also be combined with side channel analysis in order to synchronize the fault injection according to the operation flow of the running algorithm. Fault injection methods and control over the fault are in constant evolution and while remote devices were longtime considered as safe against them, the rowhammer [25] attack contradicts this feeling. Faults can also be instrumented to recover key in software implementation as demonstrated in [26] where a withebox AES implementation was instrumented to inject faults and recover the key.

1.5 Ph.D. interests

ICs are widely used for all kinds of purposes from basic usages such as simply handling LED to the more complex such as computer CPU. Computation, communication, power, all are managed by ICs. Thus Information security heavily relies on them. Unfortunately, ICs face a lot of threats such as the previously described side channel or fault attacks that are only a couple of examples. Indeed, ICs also face up reverse engineering which consists in recovering how the IC works including its secrets and various invasive attacks.

The following work was motivated by Maxim Integrated, a world leader in the semiconductor industry, as a will of constantly strengthen its products. Maxim provides a wide range of ICs for the automotive, industrial, communications, consumer and computing markets. Maxim secure microcontroller, secure manager and secure authenticator products are directly concerned about security. This work focuses on side channel and

fault countermeasures for the Elliptic Curve Digital Signature Algorithm ($ECDSA$). It focuses on hardware vulnerabilities of such algorithm implementation no matter if the implementation is fully hardware or is a software running on microcontrollers as in both case the properties of the hardware can be leveraged to attack the system. Similar researches exist and usually focus only on the elliptic curve scalar algorithm attacks and associated countermeasures. In this work, we focus mainly on leakage sources and how to prevent them or make them unusable during the $ECDSA$ signature generation. The motivation is that a leakage source may be used in different attack schemes and thus provides different attack scenarios. By fixing the leakage, we prevent existing attacks but also undiscovered or non-disclosed attacks based on the leakage. Moreover, while the elliptic curve scalar algorithm is at the heart of the security of all elliptic curve related schemes, all the system needs security. The $ECDSA$ signature is also a very leakage sensitive scheme based on elliptic curve. A small leakage on few bits may conduct to fully disclose the private key.

The $ECDSA$ can be implemented in different flavors such as in a software that runs on a microcontroller or as a hardware self-contained block or also as a mix between software and hardware accelerator. Thus, a wide range of architectures is possible to implement the $ECDSA$. For this reason, in this work, we mainly focus on algorithm countermeasures as it allows being compliant with different kinds of implementations.

In order to develop a strong $ECDSA$ implementation resistant to side channel and fault attacks, the following contributions are presented in this work:

-Experimental lattice attack results with a widely used elliptic curve is provided in order to understand the threat of even the smallest leakage inside the $ECDSA$.

-Different side channel leakage sources that allow recovering information about the scalar are discussed and experimentally demonstrated. It demonstrates leakages that can be unintentionally inserted due to the choice of coordinates representation. The leakage due to the use of the infinity point is also presented and illustrated in different cases to recover information. Finally, we demonstrate that side channel collision between signatures may allow recovering the $ECDSA$ private key without leaking any bits value.

-We show that some elliptic curve scalar algorithms are wrongly supposed to be safe-error resistant due to either the algorithm or to underlying computations. The Montgomery ladder and the coherency checking countermeasure are two examples. We also introduce the concept of dummy operand due, for example, to the infinity point that can be used when this specific point is considered and manipulated as a normal

point (e.g. with Edward curves). We demonstrate that even if in the $ECDSA$ signature, the scalar represents a nonce that is refreshed for each signature, this kind of fault is enough to recover the private key.

-Faults on both r and s parts of the $ECDSA$ signature are also discussed. It shows that $ECDSA$ signatures generated from faulted private key can be used to recursively recover the key bits. Then a similar method is applied on nonces allowing to provide enough bits of information to be used within lattice attacks. It also demonstrate that error distribution allow attackers to understand the behavior of the injected fault.

-Opportunities of injecting the considered faults in a basic architecture is discussed. This demonstrates that threats exist at the architecture level as interfaces between functional blocks can be at risk. This also demonstrates that countermeasures in the low level functional blocks are not enough.

-New elliptic curve algorithms are described that allow protecting the computation against discussed threats. Both side channel and safe-error countermeasures are provided while the security concern of eventual partial leakage is reduced.

-Approaches to protect both the $ECDSA$ private key and the nonce while computing signatures are described. The nonce countermeasure allows strengthening the elliptic curve scalar algorithm against fault attack by providing a Control Flow Integrity (CFI).

Chapter 2 provides the basic mathematical background regarding ECC and the $ECDSA$. This chapter presents the minimum mathematical knowledge to understand the following chapters from both, a computation point of view and also to understand the various presented threats. In chapter 3, side channel regarding the EC scalar operation are discussed. It presents leakage sources of some common algorithm that may allow attacker to partially recover information. It also provides an example of lattice attack that aims at showing how these small leakages can be used to fully recover an $ECDSA$ private key. Next, chapter 4 considers fault attacks. The chapter starts by exhibiting the power of safe-error that we thought is underestimated. Then faults on both r and s parts of the $ECDSA$ signature are discussed. Finally chapter 5 provides countermeasures against all presented threat prior chapter 6 which concludes about the work.

CHAPTER 2

Elliptic Curve Cryptography

In 1985, Elliptic Curves (EC) were proposed independently by Neal Koblitz [27] and Victor Miller [28] to be used for public key cryptography. Nowadays, Elliptic Curve Cryptography (ECC) are standardized and widely used in various systems ranging from the most simple ones to the most complex. ECC provides confidentiality and authenticity through various schemes. In the following, we are interested by the Elliptic Curve Digital Signature Algorithm ($ECDSA$) which aims at signing messages in order to ensure there authenticity. The $ECDSA$ is based on EC which is based on finite field arithmetic as depicted in figure 2.1.

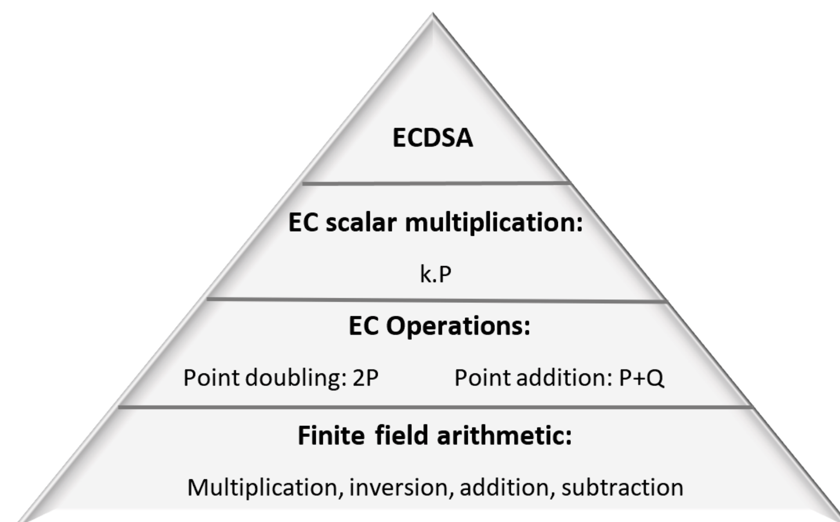

Figure 2.1: Hierarchy of $ECDSA$ operations.

This figure provides an overview of the $ECDSA$ construction and operation requirements. The following chapter is essential to the understanding of others as it explains the basic mathematical requirements behind ECC and more specifically the $ECDSA$. After explaining the advantages of using such scheme over alternatives it also explains and provides background on lattice attacks that is a real mathematical threat to the $ECDSA$.

First, group theory basics are recalled in section 2.1. Then description and generalities about ECC are provided in section 2.2. Section 2.2.1 explains how the EC scalar can easily be computed over large fields. The section 2.3 discusses the advantage of ECC over alternatives. Section 2.4 explains the coordinates choices available in ECC and how it provides design flexibility. Section 2.5 details the $ECDSA$, it shows how key pairs generation, signature and verification work. Finally, section 2.6 explains the problem of lattice attack against the $ECDSA$ prior concluding the chapter in section 2.7 .

2.1 Group, ring and field

Elliptic curve cryptography is based on group theory, we thus recall some basics about group, ring and field. A group G is a mathematical set of elements equipped with an operation $+$ that satisfies four axioms namely closure, associativity, identity and invertibility.

Closure: $(G,+) : G \times G \to G$

Associativity: $\forall a, b, c \in G : a + (b + c) = (a + b) + c$

Additive identity: $\exists 0 \in G$ such that $\forall a \in G : a + 0 = 0 + a = a$

Invertible: $\forall a \in G, \exists b \in G$ such that $a + b = b + a = 0$

A commutative group is called abelian group.

Commutativity: $\forall a, b \in G : a + b = b + a$

A ring R is an abelian group with a second operation $\times$ that is associative and distributive over the $+$ operation and has an identity element 1.

Distributivity (left): $\forall a, b, c \in R : a \times (b + c) = a \times b + a \times c$

Distributivity (right): $\forall a, b, c \in R : (a + b) \times c = a \times c + b \times c$

Multiplicative identity: $\exists 1 \in R$ such that $\forall a \in R : a \times 1 = 1 \times a = a$

A ring F is called field if all nonzero elements are invertible regarding the multiplicative operation.

Invertible: $\forall a \neq 0 \in F$, $\exists b \in F$ such that $a \times b = b \times a = 1$

The group order is the total number of elements of the group and is denoted by $ord(G)$.

The order of an element $a \in G$ denoted by $ord_G(a)$ is the smallest integer $c \mid ord(G)$ such that $c \times a = a + a + ... + a = 0$. If it exists an element g of the group such that $ord_G(g) = ord(G)$ then the group is called cyclic and g is a generator.

Example of group: $G_1 = (Z_7, +)$, is the set of integers from 0 to 6 with the addition modulo 7 as group operation and 0 as the identity. The order of the group is 7. This is an abelian group as for any $a, b \in G_1$, $a + b \mod 7 \equiv b + a \mod 7$. We can add a second operation to this group such as the multiplication modulo 7 that is associative and distributive over the modular addition. The identity of the modular multiplication is 1. As any elements of G is invertible regarding the modular multiplication, the new construction (group G_1 + the modular multiplication) is a field and is denoted by F_7. A group $G_2 = (Z_6, +)$ plus a multiplication modulo 6 as a second operation does not form a field as not all elements are invertible regarding the modular multiplication. For example, we cannot find a such that $a \times 3 \mod 6 \equiv 1$ meaning that 3 is not invertible.

A mapping between a group to another is called a morphism. An isomorphism is a morphism that admit an inverse.

Example of isomorphism: We consider the group $G_1 = (Z_{14}, +)$. We also consider the group $G_2 = (Z_2 \times Z_7, +)$, the pairs (x, y) where the x-coordinates is an integer from 0 to 1 and the y-coordinates is an integer from 0 to 6 and where addition in the x-coordinate is modulo 2 and addition in the y-coordinate is modulo 7. These two groups have the same order however contain different elements (integers from 0 to 13 or (x, y) coordinates). We can easily build a mapping: $(x, y) \rightarrow 7x + 8y \mod 14$.

$(0,0) \rightarrow 0$	$(0,4) \rightarrow 4$	$(0,1) \rightarrow 8$	$(0,5) \rightarrow 12$
$(1,1) \rightarrow 1$	$(1,5) \rightarrow 5$	$(1,2) \rightarrow 9$	$(1,6) \rightarrow 13$
$(0,2) \rightarrow 2$	$(0,6) \rightarrow 6$	$(0,3) \rightarrow 10$	
$(1,3) \rightarrow 3$	$(1,0) \rightarrow 7$	$(1,4) \rightarrow 11$	

In order to invert this mapping, we can take any element $A \in G_1$ and then recover $x \equiv A \mod 2$ and $y \equiv A \mod 7$. It is to be noted that the operation + performed in G_1 or G_2 provide equivalent results:
$(0,6) + (1,4) \equiv (1,3) \rightarrow 3$
$6 + 11 \mod 14 \equiv 3 \rightarrow (1,3)$

2.2 Generalities on ECC

An elliptic curve E over a field K is the set of points (x, y) which are solutions of a Weierstrass equation [29]:

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \tag{2.1}$$

Where $a_i \in K$ and the discriminant $\triangle \neq 0$.

Depending on the characteristic of K, (2.1) can be simplified by applying an admissible change of variables. If the characteristic of K is different from 2 or 3, the admissible change:

$$(x; y) \rightarrow (\frac{x - 3a_1^2 - 12a_2}{36}; \frac{y - 3a_1x}{216} - \frac{a_1^3 + 4a_1a_2 - 12a_3}{24}) \tag{2.2}$$

Transforms (2.1) into the short Weierstrass equation:

$$E : y^2 = x^3 + ax + b \tag{2.3}$$

In this manuscript we consider only prime fields with a characteristic > 3, other transformations exist if the characteristic is 2 or 3. The reader can refer to [29] for more mathematical details if they are interested in these other cases. For cryptographic use, elliptic curves defined over prime field F_p, binary field F_{2^m} and extension field F_{p^m} are considered. These fields come along equipped with two operations: a modular addition and a modular multiplication. In order to use elliptic curve points as an abelian group and to perform calculus on them, an additive elliptic curve group law is built and an identity $\mathcal{O}$ is added. As depicted in Figure 2.2, the operation "+" between two points P and Q of the elliptic curve consists in finding the third point that lies on the segment (PQ) and belongs to the elliptic curve and taking the opposite by inverting the y-coordinate. It is to be noted that the elliptic curve point doubling operation is similar, however it uses the point's tangent instead of a segment. This law is known as the chord-and-tangent group law.

These operations can be written as follows in the affine coordinate system:
Point addition:

$$\lambda = \frac{(y_2 - y_1)}{(x_2 - x_1)} \; ; \quad x_3 = \lambda^2 - x_2 - x_1 \quad and \quad y_3 = \lambda \cdot (x_1 - x_3) - y_1 \tag{2.4}$$

Point doubling:

$$\lambda = \frac{(3x_1^2 + a)^2}{(2y_1)} \; ; \quad x_3 = \lambda^2 - 2x_1 \quad and \quad y_3 = \lambda \cdot (x_1 - x_3) - y_1 \tag{2.5}$$

Where λ represents the segment slope or the tangent slope, $(x_1; y_1)$ is the P coordinates, $(x_2; y_2)$ the Q coordinates and $(x_3; y_3)$ the resulting $P + Q$ or $2P$ coordinates.

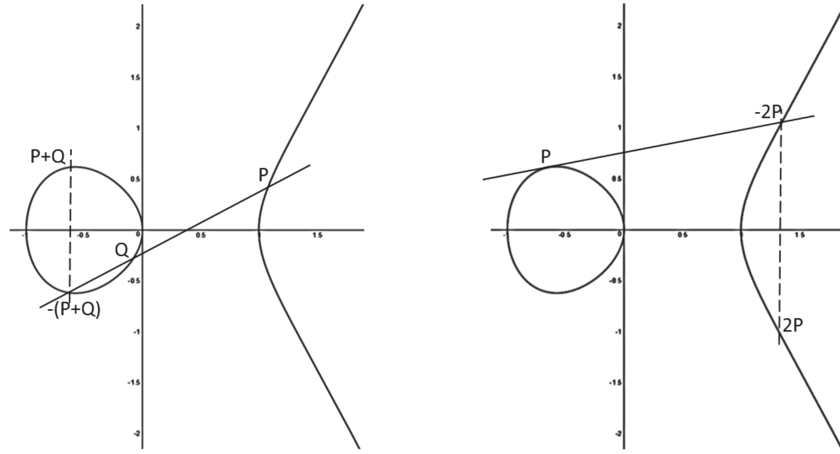

Figure 2.2: Left: Elliptic curve point addition operation over the field of real numbers. Right: Elliptic curve point doubling operation over the field of real numbers.

One should notice that Figure 2.2 is a simplified illustration which represents the elliptic curve and the point operations with real numbers. In cryptography, Galois fields are used. Thus, all calculations are modulus the field order.

The group formed by the elliptic curve points plus the neutral element $\mathcal{O}$ and the chord-and-tangent group law has an order $\#E(F_q)$ in the Hasse interval $[q+1-2\sqrt{q}, q+1+2\sqrt{q}]$. This can be expressed as $\#E(F_q) = q+1-t$ with $|t| \leq 2\sqrt{q}$ the trace of the curve. Elliptic curves such that $char(F_q) \mid t$ are called supersingular and others non-supersingular or also ordinary curve. Elliptic curves such that $\#E(F_q) = q$ are called anomalous curve. Curves candidate for secure implementation of ECC have $\#E(F_q) = h \cdot m$ with m a prime number and h a small number called cofactor.

The elliptic curve point multiplication or scalar multiplication kP with $k \in F_q$ and $P \in E(F_q)$ is build upon the additive group law and defined as:

$$kP = \underbrace{P + P + ... + P}_{\text{k times}}$$

This scalar operation is the security root of any system based on elliptic curves as it involves a complex mathematical problem, the so-called Elliptic Curve Discrete Logarithm Problem ($ECDLP$). It can be enunciated as follows: Given two points

$P \in E(F_q)$ and $G \in E(F_q)$, find $k \in [0, \#E(F_q) - 1]$ such that $G = kP$. The integer k is called the discrete logarithm of G to the base P. For a k large enough, retrieving it is computationally infeasible although computing kP is relatively easy and fast if k and P are known. Thus, for ECC, the scalar k usually represents a secret such as a private key whereas G is a public key.

Example of elliptic curve: We consider the elliptic curve $E_1 : y^2 = x^3 + 2x + 7$ defined over F_{11}. The order of this curve is 7 and the set of points is $E_1(F_{11}) = \{\mathcal{O}, (6;2), (6;9), (7;1), (7;10), (10;2), (10;9)\}$. As $\#E_1$ is prime, the group is cyclic and any element is a generator. If we consider the point (6; 2), it is possible to compute $2(6;2) = (6;2) + (6;2) = (10;9)$ from (4). Then $3(6;2) = (10;9) + (6;2) = (7;10)$ from (5). All the multiples can be recursively computed from (4) and (5) and are the following:

$$
\begin{array}{llll}
1(6;2) \rightarrow (6;2) & 3(6;2) \rightarrow (7;10) & 5(6;2) \rightarrow (10;2) & 7(6;2) \rightarrow \mathcal{O} \\
2(6;2) \rightarrow (10;9) & 4(6;2) \rightarrow (7;1) & 6(6;2) \rightarrow (6;9) &
\end{array}
$$

2.2.1 Computation of the ECC scalar operation

Cryptosystems based on elliptic curve, use an elliptic curve over a bigger field of more than 192 bits. Thus, a naive implementation of the elliptic curve scalar operation can leads to an inefficient computation time. However, simple algorithms such as algorithm 2.1 provide acceptable performance by iteratively considering each bit of the binary representation of the scalar k.

This algorithm known as the "double-and-add" algorithm first initializes a working register Q to the infinity point $\mathcal{O}$. Then a for loop iteratively double the point Q and evaluate the scalar bits, from the MSB to the LSB, and either add P to Q if the evaluated bit is 1 or do nothing. Then the final value of the working register Q is returned and contains the kP result. Algorithm 2.1 is one of the most widely known algorithm as it is one of the most simple and it is similar to the square-and-multiply algorithm used for exponentiation in other cryptosystems. A right-to-left variant of the double-and-add also exist. The difference comes from the way to parse the scalar k. The right-to-left evaluates the scalar from the LSB to the MSB whereas the left-to-right evaluates from the MSB to the LSB.

Algorithm 2.1 Left-to-right Double-and-Add

Input: $k = (k_{t-1}, k_{t-2}, ..., k_1, k_0)_2, P \in E(F_q)$

Output: kP

```
1: Q ← O
2: for i = t − 1 to 0 do
3:     Q ← 2Q
4:     if k_i then
5:         Q ← Q + P
6:     end if
7: end for
8: return (Q)
```

2.3 ECDLP and Security level of ECC

As stated previously, the ECC security rely on the $ECDLP$. An inefficient approach to solve the $ECDLP$ is to iteratively try all possible k from 0 to $\#E(F_q) - 1$ until $kP = Q$ is found. However, this is totally unpractical as soon as k is big enough ($> 80bits$) as the computation time would be in average $\frac{1}{2} \cdot \#E(F_q) \approx 2^{79}$ ECC point additions and thus too slow to be considered as a real threat. Other approaches far more efficient exist. The Pollard Rho algorithm [30] is the fastest known general method. It allows the scalar k to be recovered with a running time of $O(\sqrt{n})$ with n the order of the base point. The Pohlig-Hellman attack [31] make use of the prime decomposition of the base point P order to recover the scalar k with a complexity of $O(\sqrt{p})$ with p the largest prime divisor of $\#E(F_q)$. As a result of these attacks, the security level (i.e. the number of required step to recover the key) of ECC can be compared to the security level of a symmetric block cipher or to the RSA. Table 2.1 summarizes the key size versus the security level of ECC and RSA.

Table 2.1: Security Level of ECC vs RSA, [1].

Security level (bits)	ECC (bits)	RSA (bits)
80	160-223	1024
112	224-255	2048
128	256-283	3072
192	384-511	7680
256	512-571	15360

Figure 2.3 represents the same data as a chart. It points out the linear evolution

of the security level of ECC whereas RSA one is exponential. The expected security level of a cryptosystem aims to improve over the time in order to stay safe from new cryptanalysis techniques and computation power improvement. This difference of behavior between ECC and RSA makes ECC implementation more attractive in embedded systems than RSA. Indeed, for an RSA system, the size of manipulated data, memories, computation time and the system cost drastically increase when the security level is improved. ECC is thus more and more competitive when the required security level evolves.

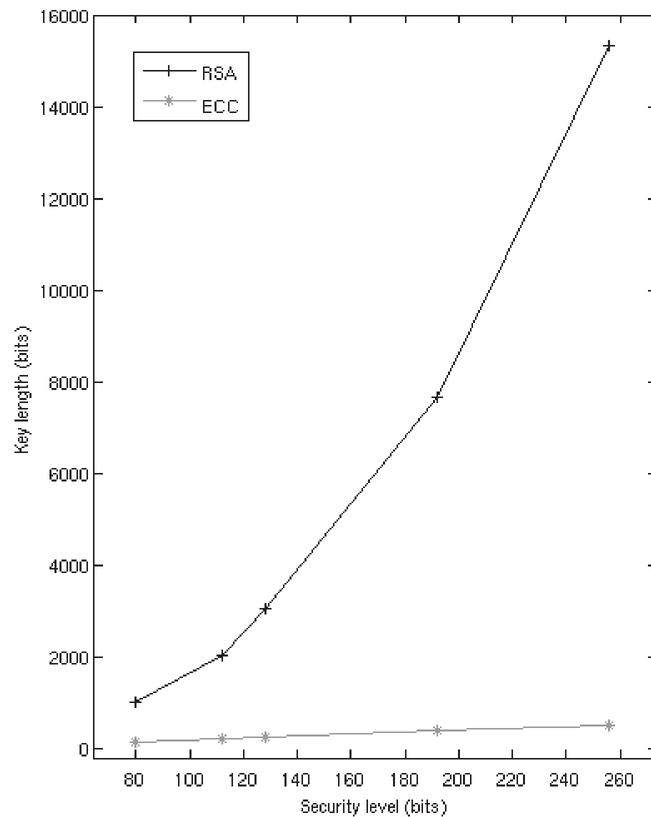

Figure 2.3: Security level of ECC vs RSA, [1].

Depending on the curve properties, other mathematical attacks can be used in order to solve the $ECDLP$. For this reason, elliptic curves are carefully selected prior to be used in a cryptosystem. Both supersingular and anomalous curves are usually avoided in ECC as they provide special mathematical properties that reduce the security level of the system. Indeed, an anomalous curve over F_p generates a cyclic group $E(F_p)$ that is isomorphic to the additive group F_p allowing thus to transform the elliptic curve scalar operation into a basic modular multiplication that is totally unsafe. Satoh, Araki [32],

Semaev [33] and Smart [34] demonstrated that such isomorphism can easily be found and used ("Smart-ASS attack"). Supersingular curves are subject to the MOV [35] and the Weil/Tate [36] pairing attacks where the main idea is to find an isomorphism between $E(F_q)$ and G a subgroup of the extension field F_{q^k} in order to end-up with an easy to solve DLP. More generally, in order to avoid this kind of transfer attack, it is required that the embedding degree is higher than a given value. The value depends on the standard (usually at least 20, e.g. [37]). The embedding degree of the cyclic group generated by $P \in E(F_p)$ with $ord_{E(F_p)}(P) = l$ is defined as the smallest k such that $p^k \equiv 1 \mod l$.

2.4 Point representation

In the previous chord-and-tangent group law formulæ, affine coordinates are used. As other coordinate systems exist and can be defined, other point addition and point doubling formulæ have been expressed [29]. The affine, standard projective and Jacobian coordinate systems are the more commonly used for ECC. They allow different kinds of performance optimizations and security features.

Projective points are represented as $(X;Y;Z) = \{(\lambda^c \cdot X, \lambda^d \cdot Y, \lambda \cdot Z); c, d \in N^*, \lambda \in F_q^*\}$. The standard projective coordinates are defined with $c = d = 1$ and Jacobian coordinates with $c = 2$, $d = 3$. Points represented with affine coordinates as $(x;y)$ can be represented in projective coordinates by calculating $(X;Y;Z) = (\lambda^c \cdot x, \lambda^d \cdot y, \lambda)$ for any $\lambda \in F_q^*$. The conversion from projective form to the affine one can be achieved by calculating $(x;y) = (X/Z^c, Y/Z^d)$. Below, as an illustration, we give elliptic curve point doubling and point addition formulæ when Jacobian coordinates are used for a characteristic of F_q different than 2 or 3.
Elliptic curve point doubling in Jacobian:

$$\begin{cases} X_3 = (3 \cdot X_1^2 + a \cdot Z_1^4)^2 - 8 \cdot X_1 Y_1^2 \\ Y_3 = (3 \cdot X_1^2 + a \cdot Z_1^4)(4 \cdot x_1 Y_1^2 - X_3) - 8 \cdot Y_1^4 \\ Z_3 = 2 \cdot Y_1 Z_1 \end{cases} \tag{2.6}$$

Elliptic curve point addition in Jacobian:

$$\begin{cases} X_3 = (Y_2 Z_1^3 - Y_1 Z_2^3)^2 - (X_2 Z_1^2 - X_1 Z_2^2)^2 \cdot (X_1 Z_2^2 + X_2 Z_1^2) \\ Y_3 = (Y_2 Z_1^3 - Y_1 Z_2^3)(X_1 Z_1^2 \cdot (X_2 Z_1^2 - X_1 Z_2^2)^2 - X_3) - Y_1 Z_2^3 \cdot (X_2 Z_1^2 - X_1 Z_2^2)^3 \\ Z_3 = Z_1 Z_2 \cdot (X_2 Z_1^2 - X_1 Z_2^2) \end{cases} \tag{2.7}$$

An affine point can be seen as a Jacobian point with a Z coordinate equal to 1. Thus, the general point addition in Jacobian can be used to add an affine point with a Jacobian point and gives a Jacobian result. This is called mixed-coordinates and allows improving the computation speed as multiplications by 1 are removed.

Table. 2.2 below summarizes the number of operation counts for different coordinates when a curve with a characteristic different than 2 or 3 is considered.

Note: $NIST$ curves (in [37]) projective point doubling formula over prime field can be simplified as $a = -3$ thus $(3 \cdot X_1^2 - 3 \cdot Z_1^4) = 3 \cdot (X_1 + Z_1^2)(X_1 - Z_1^2)$ allowing to save two field squaring.

Table 2.2: Operation counts for EC double and EC addition.

Doubling		Addition		Mixed coordinates	
Coordinates	Operations	Coordinates	Operations	Coordinates	Operations
2A	1I,2M,2S	A+A	1I,2M,1S	P+A	9M,2S
2P	7M,5S	P+P	12M,2S	J+A	8M,3S
2J	4M,6S	J+J	12M,4S		
2P $NIST$	7M,3S				
2J $NIST$	4M,4S				

Note: A = affine, P = standard projective, J = Jacobian, I = field inversion, M = field multiplication, S = field squaring.

The different coordinates systems involve different number, kind and proportion of operation. Depending on the implementation platform, changing the coordinate system to another one may improve the global performance or reduce it. Indeed, as example, usually field inversions are costly compared to field multiplications especially without a dedicated implementation. Thus, even if the total number of operation required if projective coordinates are used is higher than affine coordinates, the computation time can be reduced as no field inversions are involved. The most time efficient coordinates system depend on the implementation architecture. It is also to be noted that the memory required vary withing the different coordinates. Projective coordinates require saving three coordinates while affine ones require saving only two.

2.5 Elliptic Curve Digital Signature Algorithm

The Elliptic Curve Digital Signature Algorithm ($ECDSA$, [37]) is an alternative to the Digital Signature Algorithm (DSA) which uses elliptic curves scalar instead of modular

exponentiation [37]. Such signature schemes are used for authentication purpose. There are three distinctive operations in $ECDSA$: The key generation, the signature and the verification.

A shared curve $E(F_q)$ and a base point P of order n are used between users for these different operations. These operations are defined as follows:

2.5.1 Key generation

It aims at generating a key pair composed of a private key and a public key. The private key d is usually randomly chosen such that $0 \leq d \leq n-1$ with n the order of the base point. The associated public point Q is determined by using the elliptic curve scalar operation as defined above: $Q = d \cdot P$

2.5.2 Signature

In order to sign a message msg, first a random nonce k such that $0 \leq k \leq n-1$ is generated. Then the scalar $k \cdot P$ is computed producing a resulting point (x, y). Afterwards, $r \equiv x \mod n$ composes the first part of the signature. The second part s of the signature is calculated as: $s \equiv k^{-1}(H(msg) + d \cdot r) \mod n$ where H is a hash function (such as SHA–1, SHA–2...) and d is the private key. If the computations of r or s return "0" then the process restarts at the beginning. The signature result is the couple (r, s).

2.5.3 Verification

First, verification checks that Q (the public key) is not the point at infinity and is part of the given curve E. One verifies than that $n \cdot Q$ equals the infinity point. The signature element r and s are asserted to be between 1 and $n-1$. Afterwards, $(x, y) = (H(msg) \cdot s^{-1} \mod n) \cdot P + (r \cdot s^{-1} \mod n) \cdot Q$ is computed. The signature is correct if $r \equiv x \mod n$.

2.6 Lattice attacks on ECDSA

The lattice attack on $ECDSA$ is a powerful mathematical approach that can be used to bypass the $ECDLP$ and recover the $ECDSA$ private key and nonces used to sign messages [38]. The mathematical demonstrations behind such attack can be complex to understand, however using the concept and implementing these kinds of attacks is easy.

Thus next lines aim at describing lattice basics and provide a simple numeric example to demonstrate how easy it is to break $ECDSA$ using the lattices. Lattice attacks are composed roughly of three phases. The first one consists in recording signatures and gathering a maximum of information about the secret and nonces that were used. The second phase consists in writing equations of the $ECDSA$ scheme, inserting known information in them and transforming the system. The last phase consists in resolving the equation system and concluding if recovered secret and nonces are correct or not. Since lattice attacks only need partial known information on the secret or nonces to be useful, combining them with side channel and faults attacks furnishes a powerful mean to recover the private key. Indeed, small piece of information gathered by side channel attacks can be enough to successfully feed a lattice attack as detailed further in this part.

2.6.1 Gathering signatures and information

The attack first starts by recording $ECDSA$ signatures and for each one of them, tries to gather information. All information on the system (private key or nonce bits values, bias...) from all sources (power analysis, timing, algorithm...) can be useful and can increase the chance of success. Basic lattice attacks on $ECDSA$ require around $2.\sqrt{log(\#E)}$ signatures and $\sqrt{log(\#E) + log(log(\#E))}$ known consecutive bits of each nonce [38]. These numbers are approximation, if more bits are known, the number of signatures required decreases and vice versa. Thus, any attacks (e.g. timing, side channel, faults) can be used to gather information and increase the lattice attack efficiency. Information such as the nonce length can be exploited directly as it indicates the number of MSBs set to 0 [39]. [38] presents experimental results when nonce length can be discovered through a remote timing analysis over a computer network.

2.6.2 Computing a lattices attack

Each signature (r_j, s_j) can be written as: $s_j = k_j^{-1}(H_j(m) + d \cdot r_j)$ with $r_j = (k_j P)_x$ mod n. Thus it is an equation with two unknown values, namely the secret d and the nonce k_j. Gathering h signatures leads to h modular equations with $h+1$ unknown values (h nonces + the secret d). Therefore, the equation system cannot be normally solved. However, by inserting some knowledge of each unknown values, we can eventually solve it.

$$
\begin{aligned}
& s_j \equiv k_j^{-1}(H_j(m) + d \cdot r_j) \mod n \\
\Leftrightarrow\ & s_j k_j - d \cdot r_j - H_j \equiv 0 \mod n \\
\Leftrightarrow\ & k_j - d \cdot r_j / s_j - H_j / s_j \equiv 0 \mod n \\
\Leftrightarrow\ & k_j - d \cdot r_j / s_j - H_j / s_j - a_i \cdot n = 0
\end{aligned}
$$

From h equations as above, the lattice generated by the rows of A can be built:

$$A = \begin{pmatrix} -1 & \frac{-r_1}{s_1} \mod n & \frac{-r_2}{s_2} \mod n & \cdots & \frac{-r_h}{s_h} \mod n \\ 0 & n & 0 & \cdots & 0 \\ : & 0 & n & \ddots & 0 \\ : & : & \cdots & \ddots & 0 \\ 0 & 0 & 0 & \cdots & n \end{pmatrix}$$

By construction, it exists X such that $XA - t = (d, k_1, k_2, \ldots, k_h)$ with $t = (0, -H_1/s_1 \mod n, -H_2/s_2 \mod n, \ldots - H_h/s_h \mod n)$ If d and k_j are small enough, it is possible to solve the problem by reducing A to B with the Lenstra-Lenstra-Lovász (LLL, [40]) algorithm, rounding t to the nearest Lattice point with Babai [41] to find XB and compute $XB - t$ that gives d and k_j.

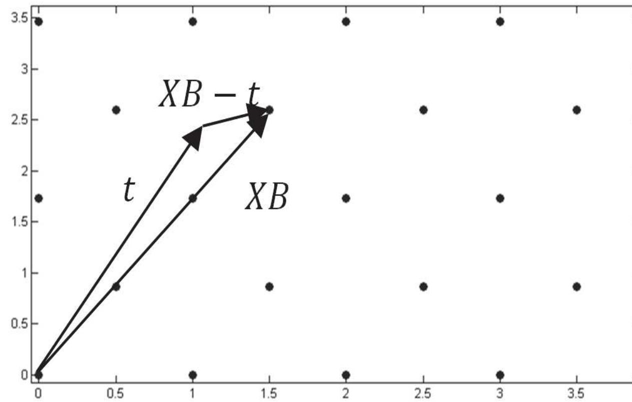

Figure 2.4: Lattice illustration in 2 dimensions

Figure 2.4 illustrates a lattice. When $(d, k_1, k_2, \ldots, k_h)$ are small, then $XB - t$ is small due to the equality $XB - t = (d, k_1, k_2, \ldots, k_h)$, thus t is close to the "good" lattice point XB and then by rounding t to the nearest lattice point, XB is found and then the correct result. If $(d, k_1, k_2, \ldots, k_h)$ are not small enough, the equation $XB - t = (d, k_1, k_2, \ldots, k_h)$ is still valid, however the wrong lattice point will be found when rounding t and thus will lead to an incorrect result. In this case, surrounding points can be used and potentially conduct to a correct result. The knowledge of information about d and k_j aims to write similar equations with smaller d' and k'_j. For example if k_1 is even, it is possible to write $k_1 = 2 \cdot k'_j$. Thus, by replacing k_1 with $2 \cdot k'_j$ and dividing by 2 the equation, leads to the same kind of equation however with a k'_1 twice smaller than k_1. If no information is known about the secret d, d can be expressed and replaced by a k_j value with known information then reducing the system to $h - 1$

equations. [38] discuss about consecutive bits knowledge in the middle of the nonce.

2.6.3 Basic numerical example of a lattice attack

The curve $E : y^2 = x^3 + 1001x + 75$ over F_{7919} and the base point $G = (4023; 6036)$ of order $\#G = 7889$ are considered.
The given public point of the system to attack is $Q = (7359; 3262)$ and three signature results are obtained.

Thanks to side channel leakages, we consider that we discovered that an important number of nonce MSBs are equal to 0 (e.g. from global timing analysis). We do not know exactly how many, due, for example, to the time measurement precision. Information gathered on the system are summed-up in the following table:

Table 2.3: Lattice example, known information summary

$(r; s)$	$H(msg)$	Nonce:	Private key used:
$(3012; 7290)$	5172	$k_1 = (0...0?...?)_2$	$d = ?$
$(4596; 2372)$	3095	$k_1 = (0...0?...?)_2$	$d = ?$
$(4808; 941)$	1350	$k_1 = (0...0?...?)_2$	$d = ?$

Note: Where $(0...0?...?)_2$ is the binary representation,
? represents an unknown bit value.

From Table 3, these three equations can be written:

$$5172 - 7290 \cdot k_1 + 3012 \cdot d \equiv 0 \mod 7889$$
$$3095 - 2372 \cdot k_2 + 4596 \cdot d \equiv 0 \mod 7889$$
$$1350 - 941 \cdot k_3 + 4808 \cdot d \equiv 0 \mod 7889$$

As no information is known about the private key d used, d is expressed in function of k_1 and replaced in all equations.

$$5172 - 7290 \cdot k_1 + 3012 \cdot d \equiv 0 \mod 7889$$
$$\Leftrightarrow d \equiv 2784 \cdot k_1 + 5310 \mod 7889$$
$$k_2 + 6978 \cdot k_1 + 283 \equiv 0 \mod 7889$$
$$k_3 + 564 \cdot k_1 + 2755 \equiv 0 \mod 7889$$

Known information about nonces are inserted in these two last equations. Nonces can be expressed as $k_j = k_{j_MSB} \cdot 2^{l_j} + k_{j_lsb}$ with l_j depending on the number of MSBs known and k_{j_MSB} know (the bits gathered).

$$k_{2_lsb} + 6978 \cdot k_{1_lsb} + (283 + (1 + 6978) \cdot k_{2_MSB} \cdot 2^{l_2}) \equiv 0 \mod 7889$$

$$k_{3_lsb} + 564 \cdot k_{1_lsb} + (2755 + (1 + 564) \cdot k_{3_MSB} \cdot 2^{l_3}) \equiv 0 \mod 7889$$

All we know is that an unknown number of nonces MSBs are equal to 0. Thereby, nonces can be expressed as $k_j = 0 \cdot 2^{l_i} + k_{j_lsb} = k_{j_lsb}$. Thus, equations are almost not modified:

$$k_{2_lsb} + 6978 \cdot k_{1_lsb} + 283 \equiv 0 \mod 7889$$
$$k_{3_lsb} + 564 \cdot k_{1_lsb} + 2755 \equiv 0 \mod 7889$$

From these two equations, we can build a lattice generated by the rows of:

$$A = \begin{pmatrix} -1 & 6978 & 564 \\ 0 & 7889 & 0 \\ 0 & 0 & 7889 \end{pmatrix}$$

We then use the LLL algorithm to reduce the lattice into B:

$$B = LLL(A) = \begin{pmatrix} 182 & 133 & -91 \\ -294 & 392 & 147 \\ -251 & 120 & -438 \end{pmatrix}$$

By construction, we know that it exists X such that:

$$XB - t = (k_{1_lsb}, k_{2_lsb}, k_{3_lsb}) \text{ with } t = (0, 283, 2755)$$

To find X, we first express t in the lattice basis. To do that, these following equations are solved:

$$\begin{cases} 182 \cdot \lambda_1 - 294 \cdot \lambda_2 - 251 \cdot \lambda_3 = 0 \\ 133 \cdot \lambda_1 + 392 \cdot \lambda_2 + 120 \cdot \lambda_3 = 283 \\ -91 \cdot \lambda_1 + 147 \cdot \lambda_2 - 438 \cdot \lambda_3 = 2755 \end{cases} \Leftrightarrow \begin{cases} \lambda_1 = -16097/7889 \\ \lambda_2 = -22964/7889 \\ \lambda_3 = -5510/1127 \end{cases}$$

We can solve the Closest Vector Problem (CVP) and find X by using Babai rounding off method. It consists in simply rounding previous result to the nearest integer. $X = (-2, 3, -5)$ is found.

Then: $XB - t = (9, 310, 2813) - (0, 283, 2755) = (9, 27, 58) = (k_{1_lsb}, k_{2_lsb}, k_{3_lsb}) = (k_1, k_2, k_3)$

Thus: $d \equiv 2784 \cdot 9 + 5310 \mod 7889 \equiv 6699 \mod 7889$

As $6699 \cdot G = (7359; 3262) = Q$, we can conclude that the recovered private key is correct.

2.6.4 Lattice attack results on NIST P256

In order to have a better understanding of the lattice attack effectiveness against the $ECDSA$, an experiment had been conducted. It simply consists in generating $ECDSA$ signatures based on the standardized $NIST$ P256 curve with some consecutive known MSBs of each nonces and try to solve the CVP with the minimum of them. Figure 2.5 shows the experimental results.

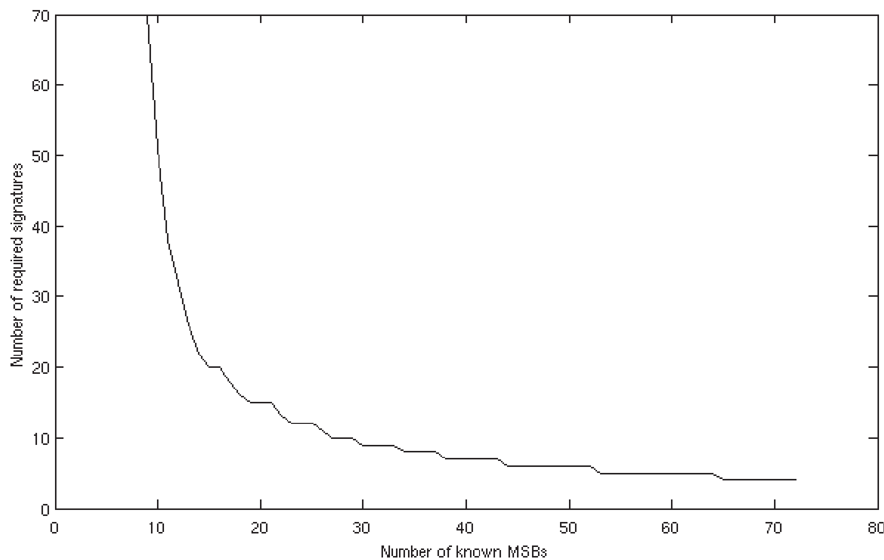

Figure 2.5: Lattice attack result against $NIST$ P256

The x-axis represents to number of consecutive known MSBs and the y-axis the number of required signatures to successfully recover the private key. With the basic implementation used, at least 9 consecutive nonce MSBs should be known in order to recover the private key with 70 signatures. With 65 leaked bits, only 4 signatures are required. Even if the implementation used does not represents the state of the art, these results are enough to underline the consequence of small leakages and thus the importance to protect all bit of the nonce when performing an $ECDSA$. This implementation also underline some limitation, indeed lattice attacks require a couple of bits for each nonce in order to succeed. In the experiment, at least 9 consecutive bits where required for $NIST$ P256. While more recent implementation of lattice attack can slightly reduce this number [42], other mathematical approaches such as the Bleinchenbacher attack [43] can overcome this limitation.

2.7 Summary

Elliptic Curve Cryptography is based on elliptic curves over finite fields. By using the algebraic structure, it allows to define a group composed of elliptic curve points. As this group is only equipped with an additive law, the so called chord-and-tangent group law allowing to addition any two points of the elliptic curve, then the EC scalar operation which is a successive addition of the same point P can be defined as $Q = k \cdot P$. The EC group is not equipped with any multiplicative law nor multiplicative inverse, thus recovering the scalar k from the result Q is difficult. The EC parameters used to implement cryptography are selected to ensuring the scalar cannot be recovered easily. The EC scalar operation is thus a one way function that can be used to implement public key cryptography. By doing so, it provides an interesting security level/key size ratio allowing to obtain public key cryptography systems with better performance, more power efficient and at lower cost compared to alternatives. Moreover, various possibilities exist to implement EC, providing flexibility.

By using the EC scalar operation, different classic public key cryptography schemes can be implemented, such as a Diffie-Hellman key exchange or the Digital Signature Algorithm denominated as $ECDH$ or $ECDSA$ when used with EC. While other schemes exist, this work focuses mainly on the $ECDSA$. Unfortunately, this scheme is extremely sensitive to even the smallest leakage of nonces and private key bits. Indeed, the lattice [38] or the Bleichenbacher [43] attacks allow recovering the private key from many signatures when few bits or information about the secret are known. The implementation of such scheme in a device that can be physically accessed is thus challenging. The next chapters aim to overview various non-invasive attacks that allow to partially recover secret information and then countermeasures against these attacks.

CHAPTER 3

Side Channel Against ECC and ECDSA

It is known since the end of world war II that electromagnetic emanations (EM) can be used to eavesdrop communication or devices. In 1943 a Bell Laboratory researcher accidentally discovered that a spike appeared on a nearby oscilloscope each time the Bell-telephone 131-B2 stepped [44]. At this time, this equipment was used as a secure teletypewriter to encrypt the U.S. Army and Navy communications. After examining these spikes, the researcher found that he could recover the plaintext. To demonstrate its finding to the Signal Corps he recorded signals during one hour from a building 80 feets aways from a Signal Corps cryptocenter and recovered 75% of the plaintext. Bell Labs improved the teletypewriter into the 131-A-1 which contains shielding and filtering techniques. However, due to constraints, the 131-B2 was still used and on the field the countermeasure was simply to control a zone around the teletypewriter. In 1951, the CIA rediscovered the same phenomena on the 131-B2 from a quarter mile. The NSA then started to examine every cipher machine and discovered that they all radiated in some way. The voltage of the power lines of rotor machines fluctuates as a function of the number of rotors moving, this was called power line modulation. Acoustic leakage was also investigated. In fact, it was found that any information-processing equipment such teletypewriters, duplicating equipment, intercoms, facsimile, computers and so on radiated information. This problem of compromising radiation and associated countermeasures has been given the cover name TEMPEST. The NSA declassified in 1972 a paper about these discoveries [44]. Since, government programs and guidances exist in order to avoid leakage in newer equipments [45], [46].
Wim Van Eck is the first in 1985 to publicly demonstrate that it is possible to eavesdrop

on video display units from a remote location thanks to EM [47]. Latter, in 1996 Paul Kocher demonstrated that it is possible to recover secret key from timing characteristic when asymmetric cryptography such as RSA, Diffie-Hellman or DSS are used [5]. Then in 1999, when smart-cards were widely used for strong and cheap authentication, Paul Kocher et al. showed that power analysis can be used to recover secret keys from smart-cards [6]. In the paper, both simple power analysis and differential power analysis were described for the first time and used the DES algorithm as example. Since, side channel attracts a wide attention from academic researchers that resulted at improving data acquisition techniques, pre-processing, power consumption models and distinguisher. The most considered side channel attacks are timing [5], power analysis [6] and EM [7], [48]. Depending on the targeted cryptographic algorithm, side channel leakage sources and the effect on security can vary.

In the following chapter, the $ECDSA$ signature side channel leakage sources that allow recovering information about the EC scalar are considered and experimentally demonstrated in section 3.1. In this section, we demonstrates leakages that can be unintentionally inserted due to the choice of coordinates representation. Leakages due to the use of the infinity point is also presented and illustrated in different cases to recover information. In section 3.2, leakages due to underlying algorithm are discussed, and the effect of data-dependent leakages on the EC scalar operation are demonstrated. Finally, in section 3.3, we demonstrate that side channel collision between signatures may allows recovering the $ECDSA$ private key without leaking any bits value through a lattice attack example. This chapter that aims at demonstrating that current EC scalar algorithms are not perfect against small side channel leakage is concluded in section 3.4.

3.1 Scalar algorithms and leakages sources

SPA stands for Simple Power Analysis [49]. The principle is to run the target once or a few times, record the power consumption and analyze it to recover the secret. This kind of attack can be, in some case, difficult to perform due to the noise and signal quality. Thus multi-shot SPA are used to average the signal and reduce the noise. However, in $ECDSA$ secrets are used only once. As a consequence, only single-shot SPA can be performed. Single-shot aims to recover the secret from only one trace. This kind of attack can be dangerous since randomness due to nonce generation over different executions of the system does not interfere with the analysis. In the following, we experimentally evaluate SPA attacks against various algorithm designed for elliptic curve scalar operation.

3.1.1 Double-and-Add

The double-and-add algorithm is a well-known and widely used algorithm to perform the Elliptic curve scalar operation. Two versions of this algorithm exist, the right-to-left and the left-to-right. The difference comes from the way to parse the scalar k. The right-to-left parses from the LSB to the MSB whereas the left to right parses from the MSB to the LSB.

Algorithm 3.1 Left-to-right Double-and-Add

Input: $k = (k_{t-1}, k_{t-2}, ..., k_1, k_0)_2, P \in E(F_q)$

Output: kP

1: $Q \leftarrow \mathcal{O}$
2: **for** $i = t - 1$ to 0 **do**
3: $\quad Q \leftarrow 2Q$
4: $\quad$ **if** k_i **then**
5: $\quad\quad Q \leftarrow Q + P$
6: $\quad$ **end if**
7: **end for**
8: **return** (Q)

This algorithm is also known to be SPA sensitive [50]. Depending on the parsed secret bit value, either an elliptic curve double plus an elliptic curve addition are done or only a single elliptic curve double operation is performed. Thus, as point doubling and point addition have different durations, by performing an SPA, attackers can totally extract the secret bit per bit. Figure 3.1 shows a partial power trace of a left-to-right software implementation on ARM926 of P256 obtained by simply using a differential probe on a 1Ω resistor placed in series with the power supply.

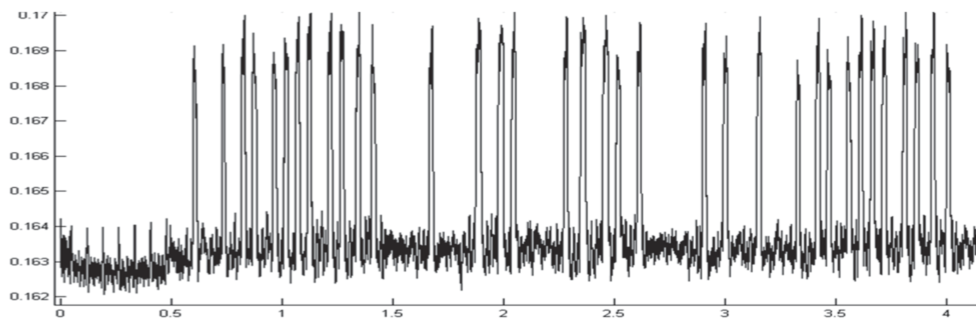

Figure 3.1: Double-and-Add left to right power trace
x-axe: Time y-axe: Power consumption.

In this trace we can clearly see that it contains information and we can distinguish different patterns. Indeed, a pattern composed of 1 peak followed by 11 peaks then 1 peak (denoted as 1–11–1 peaks) appears at the beginning of the Figure 3.1 and partially appears at the end. Between these two apparitions, another pattern of 3–5–2 peaks appears. A visual analysis of the full trace shows that these two patterns are recurrent over the encryption and no other patterns are observed. From Algorithm 3.1, we can suppose that the first pattern (1–11–1 peaks) corresponds to the elliptic curve point doubling operation. By performing a correlation between this pattern and the power trace, we obtain Figure 3.2.

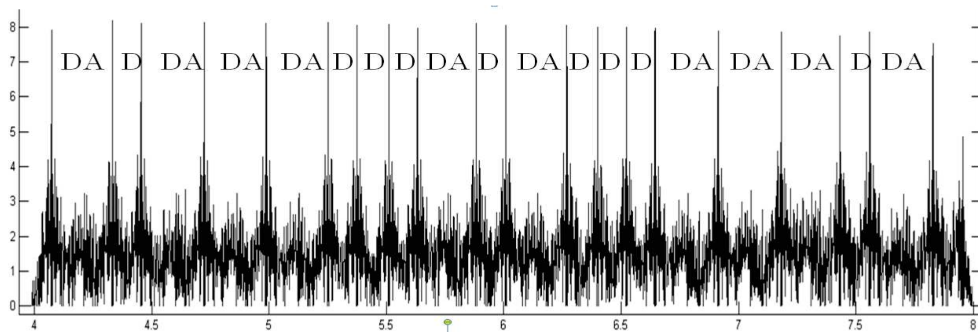

Figure 3.2: SPA result of left to right
x-axe: Time y-axe: Power consumption.

In Figure 3.2, big peaks thus represent the supposed elliptic curve double operation. We can clearly distinguish large gaps and small gaps between doubling operations. From Algorithm 3.1, we can easily conclude that large gaps are observed when an elliptic curve double and addition operations are done while small gap happens when just a point doubling is performed in the loop iteration. Large gaps are thus labeled "DA" for Double and Add and small gap "D" for Double. As a point addition only occurs if the parsed secret bit is equal to 1, we can directly replace DA by '1' and D by '0'. It results into: $101110001010001\ldots$ The key used was therefore $0xDC51\ldots = (1101110001010001\ldots)_2$. As we can see, it matches except that the first '1' is missed. This happens since no computation is done the first time, as Q is equal to the infinity point. Thus, no calculation is needed to double it and only a data transfer occurs ($Q = P$) instead of a point addition.

In the double-and-add algorithm, the security breach comes from two facts. First, the elliptic curve double and the elliptic curve add operations have different durations. Secondly, a condition based on the secret bits values directly affects the operation flow.

3.1.2 Fixed-base windowing

Another conventional algorithm to perform the elliptic curve scalar operation is the ”fixed-base windowing” algorithm [29]. This algorithm is useful to accelerate the scalar operation in systems which allow pre-computing and recording elliptic curve points.

Algorithm 3.2 Fixed-base windowing EC scalar multiplication algorithm

Input: window width w, $d = \lceil t/w \rceil$, $k = (K_{d-1}, ..., K_1, K_0)_{2^w}$, $P \in E(F_q)$
Output: kP
Pre-computation: compute $P_i = 2^{wi}P$, $0 < i < d-1$

```
1:  A ← O, B ← O
2:  for j = 2^w − 1 downto 1 do
3:      for j = 2^w − 1 downto 0 do
4:          if K_i = j then
5:              B ← B + P_i
6:          end if
7:      end for
8:      A ← A + B
9:  end for
10: return (Q)
```

As this algorithm uses only elliptic curve point additions, at a first glance it may seem that the operation flow is independent of the key. This is actually not the case as the global time depends on the number of $K_i = 0$. Indeed, the condition $K_i = 0$ is never evaluated and thus no point addition is performed in this case. In addition the use of mixed coordinates can drastically modify the operation flow. In case of mixed Affine-Jacobian coordinates for example, $B = B + P_i$ will usually use B in Jacobian and P_i in affine in order to reduce the memory footprint. Thus the + operation consists in this case in the addition operation between an elliptic curve point represented in Jacobian coordinates and another in affine ones. The returned result is represented in Jacobian. The operation $A = A + B$ consists in an addition between two points represented in Jacobian. Thus the two ”+” occurrences in Algorithm 3.2 can hide two different operations and then the resulting operation flow significantly differs depending on the key value. Figure 3.3 is a power trace of such an implementation with the same measurement setup used to obtain Figure 3.1.

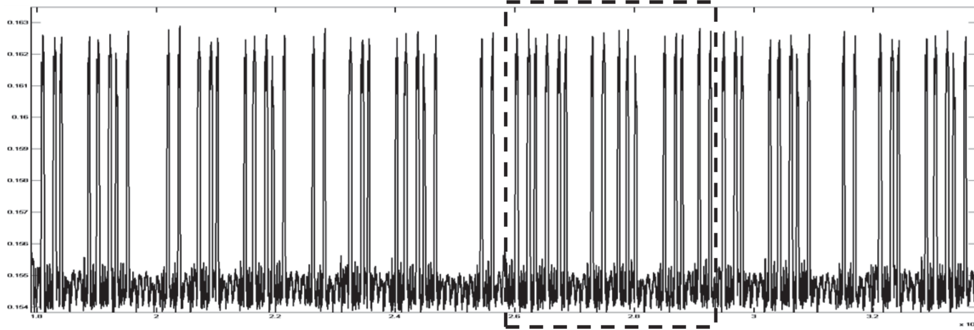

Figure 3.3: Fixed-base windowing power trace
x-axe: Time y-axe: Power consumption.

We can see a pattern of 3–5–2 peaks that appears several times in the power trace. This pattern, as in the previous analysis, represents the mixed coordinates Affine-Jacobian addition. Another pattern of 6–5–5 peaks appears in the middle of the Figure 3.3 (in the dashed frame) and thirteen other times in the full trace. This pattern represents the Jacobian plus Jacobian operation. In order to automate the pattern extraction, we can set time and amplitude thresholds. The result is given Figure 3.4.

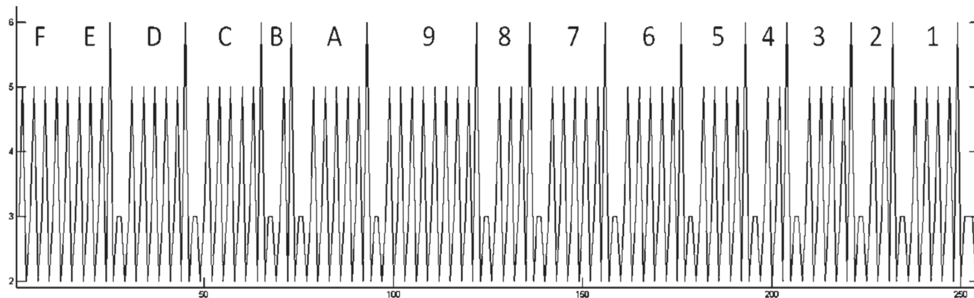

Figure 3.4: SPA result of the fixed-base windowing implemented with Jacobian representation.
x-axe: Time y-axe: Number of peaks

In this trace, due to the time threshold value, the Jacobian plus Jacobian operation is represented as a succession of 6–2–3–3–2 peaks. This operation thus clearly appears in the trace as it has the most number of consecutive peaks. The 14 big peaks appearing in Figure 3.4 thus represent 14 Jacobian plus Jacobian operations. From this point, we can deduce the value of the "w" parameter (4 as $2^4 - 1 = 15$). As the A register is initialized at infinity, the first "$A = A + B$" is just a transfer of the B value to A and thus does not wield any elliptic curve operation. This explains why 14 Jacobian plus Jacobian operations are counted instead of 15. By counting the number of peaks

occurrences with an amplitude of 5 between big peaks, it allows retrieving the number of $K_i = j$. The number of $K_i = 0xF$ and $0xE$ can be retrieved by looking time between operations. A time glitch appears (not represented in Figure 3.4). By measuring the time between Jacobian plus Jacobian operations and Jacobian + Affine operations, the approximate order of the first occurrence of each values and their position in the key can be recovered. By measuring the time between Jacobian plus Affine operations, the offsets between occurrences of a specified value can also be approximated and thus the (secret) scalar value can be compromised. These time variations are due to the "for" loop used to search $K_i = j$. The smaller is the first i as $K_i = j$, the more loop iterations will occur before an elliptic curve operation. Therefore, the duration will appear longer. In the case where both "+" operations are the same (e.g. affine coordinates case), the power trace would look more homogeneous as only elliptic curve point addition would occur, always with the same formulæ. However, information could be recovered from this "for" loop timing leakage as it is still presents.

This algorithm demonstrates two possible problems. The first one is that the operations flow at the algorithm level can seem constant in time while this it is not the case when mixed coordinates systems are used. The second one is caused by searching for a particular value in the key and doing an operation as soon as it is found. These two security flaws are independent thus in case of the use of one coordinate system, the operation flow will be constant. Nevertheless, timing leakage could help to find out information about the scalar value.

3.1.3 Fixed-base comb method

The fixed-base comb method [29] is another algorithm for elliptic curve scalar operations which aims at improving performances when point pre-calculation is allowed. Different configurations exist, depending on the width value and the number of pre-calculated table stored in the system. A two table configuration is presented below as Algorithm 3.3.

In order to speed up the calculation time, the scalar k is represented as a matrix of w lines of consecutive bits. During the "for" loop, two columns of the matrix are parsed thus reducing the total number of iterations. All possible base point P multiples associated to column values are pre-calculated. This algorithm has some similarities with the "double-and-add" algorithm presented previously. First a variable is set to the infinity point. Then a "for" loop parses the secret. In algorithm 3.3, this loop contains a point doubling of the initialized variable and two point additions that depend on different parts of the secret (two different columns). This algorithm also presents the same kind of leakage than algorithm 3.1. If the secret bits parsed are 0s then the addition is skipped. Thus during a loop iteration, either a single point doubling occurs

Algorithm 3.3 Fixed-base comb method with two tables

Input: window width w, $d = \lceil t/w \rceil$,$e = \lceil d/2 \rceil$ $k = (k_{t-1}, ..., k_1, k_0)_2$, $P \in E(F_q)$

Output: kP

Pre-computation: compute all possible $[a_{w-1}, ..., a_1, a_0] \cdot P$ and $2^e[a_{w-1}, ..., a_1, a_0] \cdot P$, where $[a_{w-1}, ..., a_1, a_0] = a_{w-1} \cdot 2^{w-1}P + ... + a_1 \cdot 2^1 P + a_0 \cdot P$ and $a_i \in \{0, 1\}$

Represent k as: $\begin{pmatrix} k^0_{d-1} & \cdots & k^0_1 & k^0_0 \\ k^1_{d-1} & \cdots & k^1_1 & k^1_0 \\ \vdots & \ddots & \ddots & \cdots \\ k^{w-1}_{d-1} & \cdots & k^{w-1}_1 & k^{w-1}_0 \end{pmatrix}$ //if necessary, pad 0s as k MSBs.

1: $Q \leftarrow \mathcal{O}$
2: **for** $i = e - 1$ to 0 **do**
3: $\quad Q \leftarrow 2.Q$
4: $\quad Q \leftarrow Q + [k^{w-1}_i, ..., k^1_i, k^0_i]P + 2^e[k^{w-1}_{i+e}, ..., k^1_{i+e}, k^0_{i+e}]P$
5: **end for**
6: **return** (Q)

or a point doubling plus one point addition or a point doubling plus two point additions. The Figure 3.5 below represents a pattern extraction performed as in previous section of such an algorithm implementation.

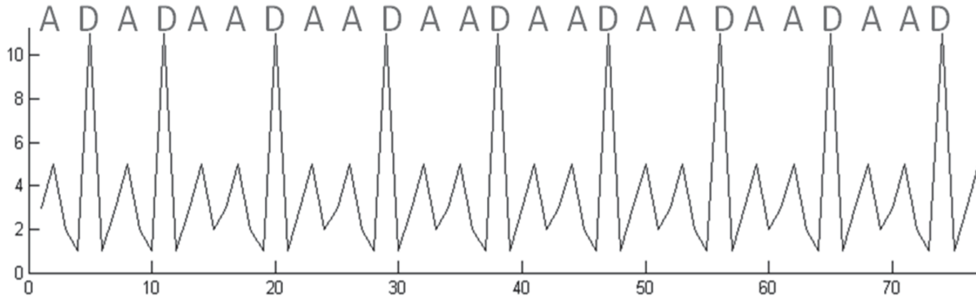

Figure 3.5: Patterns extracted from a power trace of Algorithm 3.3 execution. x-axe: Time y-axe: Number of peaks.

Point additions are again represented by a pattern of 3–5–2 peaks and point double by a pattern of 1–11–1 peaks. As it can be seen from Figure 3.5, usually a point doubling "D" and two points additions "A" occur for each iteration. In Figure 3.5, this did not happen only two times at the beginning. According to Algorithm 3.3 we know that the first elliptic curve operation that occurs is the point doubling. However, in the trace the first thing that appears is a point addition. This is due to the initialization of Q to the infinity point. As the infinity point is a neutral element, nothing happens. The first point addition also does not appear for the same reason, instead only a value transfer

occurs. However, the only reason that explains the second orphan point addition is that one of the two parsed columns of the secret key is set to zero. The secret key used is given Figure 3.6.

```
↓                                   ↓
0010 0111 1111 0001 0110 0001 0100 1100 1000 1000 1010 0111 1111
0000 1001 0011 0100 1110 0111 0000 1001 0111 0101 0101 1001 1100
0010 1111 1100 1010 1001 1001 1101 1010 0111 1110 0110 1110 1111
1011 1010 1100 1101 1110 0011 0011 1101 1001 0110 1111 1001 1001
0000 1101 1100 0101 0001 1101 0011 1000 0110 0110 1010 0001 0101
```

Figure 3.6: Key used with Algorithm 3.3 and Figure 3.5. Arrows represent the parsed secret columns when the 2nd orphan point addition appears

Figure 3.6 confirms that one of the two parsed columns is 0. Each column is a set of five bits, however we discovered only four bits of the secret as the fifth bit of the column is a part of the padding. This algorithm thus provides the exact same source of leakage than Algorithm 3.1. Nevertheless, the leakage is not systematic as a column has a probability of $1/2^w$ to be set to 0 ($1/2^{w-1}$ for MSBs columns if the secret is parsed with 0s). Also when one of the two parsed column is set to 0, two possibilities exist, then these bits does not instantly leak. Another leakage that could happen using such an algorithm is relative to the pre-computed values management. The time to fetch the required pre-computed value for an iteration can vary. For instance, when a cache is used, a cache hit/miss can eventually leak information [51].

3.1.4 Double-and-Add countermeasures

Section 3.1 "Double-and-add" pointed out that the leakage of the double-and-add algorithm comes from the association of two things: The difference between elliptic curve add/double operations and the operation flow that depends on the secret. Thus two paths of countermeasures have been proposed. In the following, we evaluate them. First, the most obvious countermeasure is a double and always add algorithm. Algorithm 3.4 presents such countermeasure. The double-and-always-add algorithm consists in doing a dummy operation (i.e. $G = Q + P$ in algorithm 3.4) when the parsed bit of the secret scalar is 0. Doing so, the operation flow become regular in time and consists in a succession of point doubling and point addition. Thus attackers should not be able to recover the secret from the operation flow by distinguishing double operation with a double and add one, since a double and add is always performed whatever is the value of the bit of k being processed.

Algorithm 3.4 Left to right Double-and-Always-Add

Input: $k = (k_{t-1}, ..., k_1, k_0)_2, P \in E(F_q)$

Output: $k.P$

1: $Q \leftarrow \mathcal{O}$
2: **for** $i = t-1$ to 0 **do**
3: $\quad Q \leftarrow 2Q$
4: $\quad$ **if** k_i **then**
5: $\quad\quad Q \leftarrow Q + P$
6: $\quad$ **else**
7: $\quad\quad D \leftarrow Q + P$ // D is a dummy register
8: $\quad$ **end if**
9: **end for**
10: **return** (Q)

Figure 3.7 below presents SPA results over an implementation of Algorithm 3.4. These results were obtained by the same process used for Figure 3.2. A correlation between the point doubling and the power trace of the execution of Algorithm 3.4 was performed.

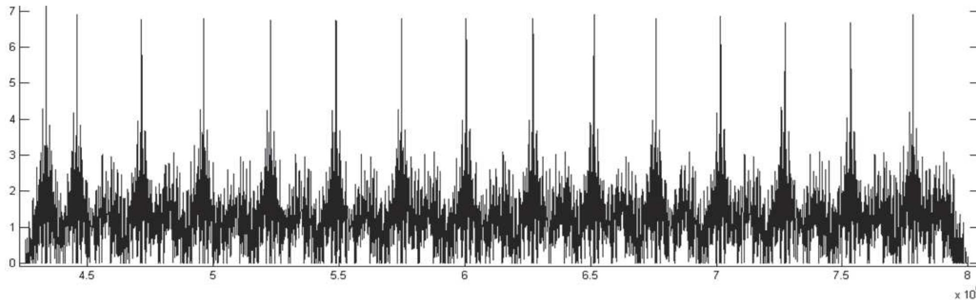

Figure 3.7: SPA results of Algorithm 3.4.
x-axe: Time y-axe: Correlation value.

Big peaks in Figure 3.7 represent point doubling operations. This figure is far more regular compared to Figure 3.2. The gaps between peaks seem equal, except the first one. This leakage is generated by the initialization of Q to the point at infinity. Operations with the infinity point are not real operation as the point at infinity is a neutral element of the elliptic curve group. Thus this leakage allows recovering the secret length as until a 1 is met in the parsed scalar, the point at infinity will be used and thus small gaps appear. In the illustration of Figure 3.7, where only one small gap appears, we can deduce that the MSB of the key is equal to 1. The Table 2 below details the elliptic curve operations performed with Algorithm 3.4 depending on the scalar MSB value.

Table 3.1: Detailed operation flow of Algorithm 3.4 depending on the MSB

$k = 11...$		$k = 01...$	
$Q = \mathcal{O}$	//initialization	$Q = \mathcal{O}$	//initialization
$Q = 2 \cdot \mathcal{O}$	//nothing to do	$Q = 2 \cdot \mathcal{O}$	//nothing to do
$Q = \mathcal{O} + P$	//data transfer, P to Q	$Q = \mathcal{O} + P$	//data transfer, P to Q
$Q = 2P$	//Point double	$Q = 2 \cdot \mathcal{O}$	//nothing to do
$Q = 2P + P$	//Point addition	$Q = \mathcal{O} + P$	//data transfer, P to Q
$\vdots$		$\vdots$	

The infinity point is generally a special case for elliptic curve operations as it is a neutral element of the elliptic group and in this case formulæ either does not apply or are simplified. For instance, the Jacobian point addition formulæ given in the background section does not work to addition a point with the point at infinity. The point at infinity is represented in Jacobian as $(1, 1, 0)$. Thus $P + \mathcal{O}$ using the given formulæ result in $(0, 0, 0)$. It is obvious that this result is wrong as, except for the infinity point, the Z coordinate cannot be equal to 0. Due to this leakage, attackers can deduce the secret length and as demonstrated in [39] and explained in the lattice section of this paper, such knowledge is enough to break the $ECDSA$. The second approach to counter operation flow leakage described section 3.1.1 consists in keeping Algorithm 3.1 and having a unified formula [52] for both point double and point addition. Thanks to unified formula, one cannot distinguish a double operation from an add operation. Thus, the power trace does not leak any more information on the type of operation (i.e. elliptic curve double or add) which is performed. It is to be noted that this approach leaks the Hamming weight of the scalar as the global execution time depends on it. Indeed, the number of peaks gives the total number of elliptic curve double and addition operations which have been processed and just the number of elliptic curve point addition varies. As demonstrated above, another leakage also exists due to the initialization. Both countermeasures (i.e. Algorithm 3.4 and Algorithm 3.1 with unified formulæ) then work against SPA in the traditional case where attackers try to fully recover the secret from one power trace by identifying elliptic curve operations (i.e. point doubling and point add). However, they are not enough in the case of $ECDSA$ as partial leakages drastically reduce the security (this point is developed further in section 2.6).

Another well-known leakage in the description of Algorithm 3.4 and Algorithm 3.1 is the use of the if-else condition. Indeed, in a software implementation and depending on the architecture of the core, a conditional branch may induces cache hit/miss, pipeline flushing, wrong branch prediction and so on. This can result in a different power consumption either in level or timing depending on the branch that is used. Figure 3.8

below is a good illustration of the leakage.

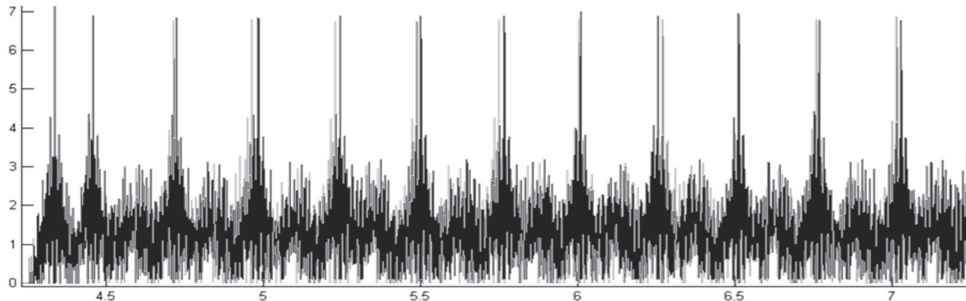

Figure 3.8: Comparison of two different scalar operations based on Algorithm 3.4. x-axe: Time y-axe: Correlation value.

Due to this leakage, attacker can directly know which branch is used and thus the value of the secret bit that is considered for each loop iteration.

3.1.5 Global timing

A global timing analysis can leak some useful information. Depending on the algorithm, either MSBs, LSBs or Hamming weight of the secret can leak via timing information. In the case of Algorithm 3.4 for example, a global timing analysis can lead to recover the secret length and thus the number of MSBs set to 0. Indeed, until a 1 is met in the parsed scalar, the point at infinity will be used and thus point doubling and point addition performed are not real operations and can be shorter in time as demonstrated Figure 3.7. In some cases (e.g. OpenSSL v0.9.8o [39]), this leakage is due to the scalar multiplication algorithm implementations that start the loop index at the first non-zero MSB of the secret.

3.2 Leakages on underlying algorithms

As SPA is very simple to implement and efficient to recover ECC secrets, various countermeasures against SPA are widely used. Unfortunately, some of them are not perfect and due to the underlying algorithms used for the specific ECC arithmetic, leakages, exploitable with SPA, can remain. In the following, some of them are detailed. Coron initially suggested the double-and-always-add in [50]. His implementation is different from Algorithm 3.4. Algorithm 3.4 was discussed in this paper to illustrate a common mistake. The always double-and-add SPA resistant algorithm given by Coron is detailed in Algorithm 3.5.

Algorithm 3.5 Coron always Double-and-add

Input: $k = (k_{t-2}, ..., k_1, k_0)_2, P \in E(F_q)$

Output: $k.P$

1: $Q[0] \leftarrow P$
2: **for** $i = t-2$ to 0 **do**
3: $\quad Q[0] \leftarrow 2Q[0]$
4: $\quad Q[1] \leftarrow Q[0] + P$
5: $\quad Q[0] \leftarrow Q[k_i]$
6: **end for**
7: **return** $(Q[0])$

In this algorithm, two differences appear compared to Algorithm 3.4. First $Q[0]$ is initialized to P, this initialization prevents the use of the point at infinity in the loop, however it forces the scalar MSB to 1. Secondly, no "if" condition is used, instead a two index table is used for Q and the correct index is selected depending on the parsed bit. This prevents branching in software implementation. Branching can in some cases be exploited as it may results in different timing executions or power consumption depending on the no code jumping case and code jumping case [51]. Algorithm 3.5 is resistant against previously described attacks. Nevertheless, it is not enough if we consider underlying algorithms (i.e. the ones used to perform low level operations such as modular addition, modular multiplication...). Point addition and point double are based on modular operations such as modular multiplication and modular division. Depending on how modular operations are managed, information can leak. It is common to see modular operations with an execution time which depends on the operands. Indeed in a standard multiplication or division, execution time can vary depending on operands. The number of modular reduction operations used can also vary depending on operands. Thus combining both to get a modular multiplication/division with a constant execution time is not obvious, especially when high performances are needed. Moreover, when constant time modular operations are reached, they can still leak information due to internal conditions and registers values which influences the power consumption. It can be remarked from Algorithm 3.5 that at the second iteration of the loop, the two first operations are:

- Either $Q[0] = 2 \cdot (2P)$ and $Q[1] = 4P + P$

- Or $Q[0] = 2 \cdot (3P)$ and $Q[1] = 6P + P$

This depends on the previously parsed bit. Thus, only two possibilities exist and discriminating them can be easy. Most designs can allow attackers to capture a power

trace of an elliptic curve scalar operation when using sensitive information such as a private key or a nonce. Some of them can allow attacker to use the implemented scalar operation with their own value. These kinds of designs can be subjected to a simple attack consisting in recording a power trace when using the sensitive information, recording power trace with the two possibilities, and comparing them with the original trace and concluding. By iteratively doing that, attackers can expect to break a 256-bit secret in 512 steps. The next figures illustrate such attack. The initial scalar used was random and equal to $0x8AC \cdots 325C0$.

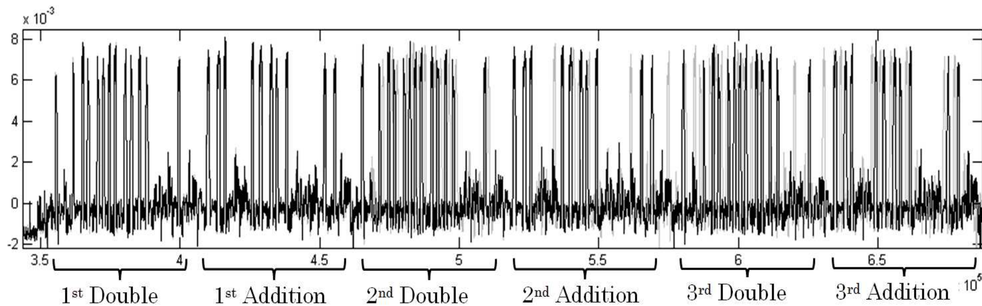

Figure 3.9: Comparison between genuine scalar and attacker scalar.
Black: genuine, grey: attacker, $0xE00 \cdots 0000$
x-axe: Time y-axe: Power consumption.

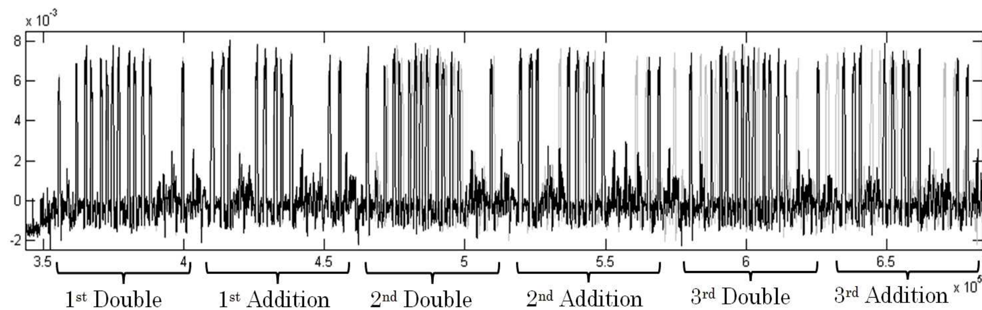

Figure 3.10: 2nd comparison between genuine scalar and attacker scalar.
Black: genuine, grey: attacker, $0xC00 \cdots 0000$
x-axe: Time y-axe: Power consumption.

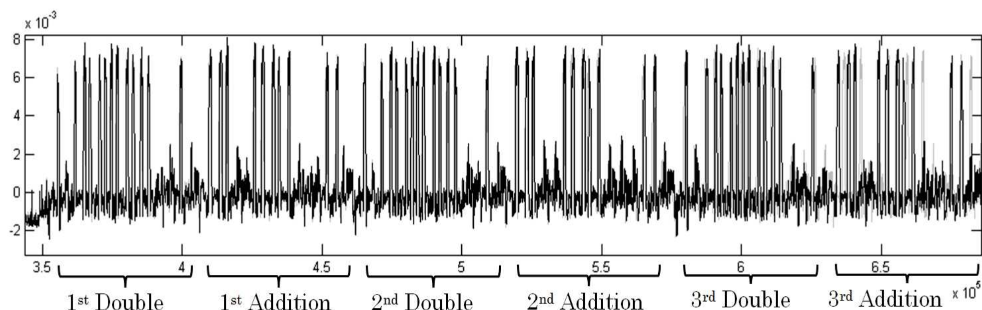

Figure 3.11: 3rd comparison between genuine scalar and attacker scalar.
Black: genuine, grey: attacker, $0xA00 \cdots 0000$
x-axe: Time y-axe: Power consumption.

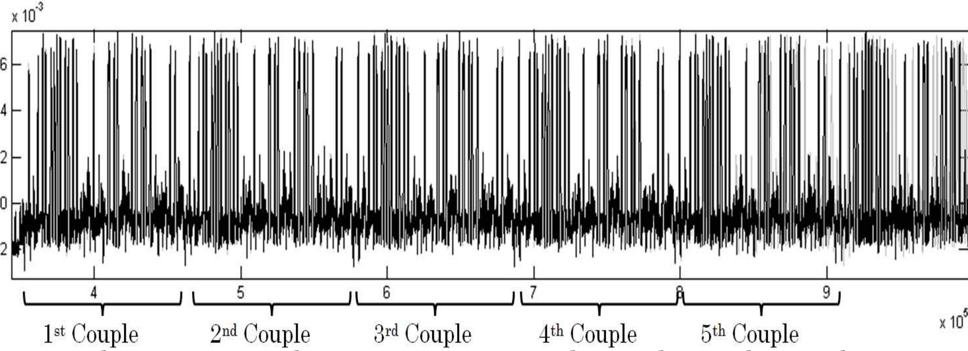

Figure 3.12: 4th comparison between genuine scalar and attacker scalar.
Black: genuine, grey: attacker, $0x800\cdots0000$
x-axe: Time y-axe: Power consumption.

From Figure 9 to Figure 12, we can clearly distinguish parts of the trace that match from part that does not without any post-process. Table 3.2 summarizes the results.

Table 3.2: Figure 3.9 to 3.12 results summary

	Binary	Nb. of MSB match	Nb. of D/A that match
Key: $0x8A$...	10001010	-	-
Hyp1: $0xE0$...	**1**1100000	1	1
Hyp2: $0xC0$...	**1**1000000	1	1
Hyp3: $0xA0$...	**10**100000	2	2
Hyp4: $0x80$...	**1000**0000	4	4

Note: Compares the number of matched operations to the number of bits match between key and hypothesis.

Table 3.2 shows that the number of correctly guessed bits directly depends on the number of couple double/add that matches the genuine trace. Thus breaking the system in this case is easily performed since attackers only have to recursively try two hypotheses until the whole secret is recovered. This is a data dependent leakage, which can be recovered from timing information. It is also possible that the leakage is not visible from the timing but from the signal amplitude. When not clearly visible, correlation or template attacks [53] and [54] can be done to enhance the probability of attack success. This approach is easy to perform in our case as attackers can control the scalar input and thus directly reuse the design as a model that perfectly fits the reality. This can happen in real life systems when for example the design allows users to set a private key and calculate the public one. In the cases where the design reuse

is not allowed to attackers, the knowledge of the design and/or reverse engineering can allow building a model to perform the same attack. Sometimes a code dumping when unprotected plus an execution of the interesting function on another CPU with a same architecture is enough to build a model. The underlying algorithms leakages can also be used without having access to any model in the case of $ECDSA$. By power traces comparison, enough information can be gathered; this is explained in the next section. It has to be noted that the Montgomery ladder algorithm [29] to perform the Elliptic scalar operation has the same source of leakage and thus is also vulnerable to this attack.

3.3 Leakages usage against the ECDSA

Secret leakages may be only partial. Indeed, instead of leaking all the secret, an algorithm may leak only partial information such as the value of a couple of bits or that the same couple of bits are used in different nonces and so on. These informations alone, may be not enough to directly fully recover the secret. However, as explained in section 2.6 these information can be used inside a lattice attack to recover the secret. In the following, a concrete example of such an attack is used against a device that implement an $ECDSA$ based on $NIST$ P-256 curve with an EC scalar algorithm that provides a data dependent leakage.

Example on NIST P-256

In this example, we consider a system based on $NIST$ P-256 implemented as previously on an ARM926 using as hash function SHA–256. The public point is:

$$Q : \begin{cases} x_Q = 0xf703e67604e1187cbe40f2176dc86d7e6b168f8a160c6e8f106bf90d184c5ffc \\ y_Q = 0x4ca233f8ae175e7c21eac1c1a705cdf50d6ef9f9bb65a8dd44aa5109ed02c567 \end{cases}$$

We do not know the implementation and we do not have access to any open system or model that would allow us to build templates or perform miscellaneous experiments. We consider that we can only know or input the message, get signatures and record power traces of the DUT during signature computation.

The EC scalar operation is easily located in the trace as it is the most time consuming operation and its shape is homogeneous over the time. A couple of power trace of the scalar are thus captured as the one illustrated Figure 3.9. From this, we can first deduce that the EC scalar operation is implemented with a constant time algorithm. The

number of pattern repetition (255) allows concluding that the implemented algorithm does not use pre-calculation as this would reduce the number of EC operation.

This first analysis allows reducing the assumptions about the implemented EC scalar multiplication algorithm as all pre-calculated algorithms are excluded. The basic double-and-add algorithm or unified formulae does not apply as both have an execution time depending on the Hamming Weight of the scalar. From [29] and [55] lists of algorithms, only the double-and-always-add, the Montgomery ladder, the BRIP [56] and variants can be considered.

Operations always seems similar, especially at the beginning of the power traces. From section 3.4, this let us suppose that the infinity point is not used. Thus, the working register is initialized either to the base point or to a random one before a loop that parse the scalar. This observation is coherent with the 255 patterns repetition instead of 256 that could be expected for a 256-bit system.

A closer look on the captured power traces reveals that at each new parsed bit, the power traces fork in two groups except for the first EC operation. This is illustrated by Figure 3.13 where different traces have been superposed.

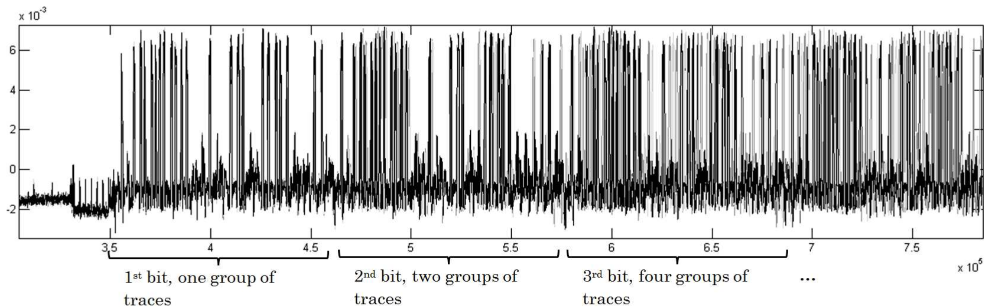

Figure 3.13: Superposition of power traces from different signatures.
x-axe: Time y-axe: Power consumption.

This gives the information that no randomness is used in the system and thus that the scalar implementation is vulnerable to DPA/CPA and template. Unfortunately, DPA/CPA cannot be used to target this scalar as a random nonce is used and template cannot easily be build due to our hypothesis. However, this leakage allows us to gather signatures with the same nonce MSB or LSB depending if the implemented EC scalar multiplication algorithm is a left-to-right or a right-to-left algorithm. In the next few lines, we make the assumption that the algorithm used is a left-to-right and thus that the first scalar bit represents the MSB.

From here, $500k$ signatures with the same messages are computed and the associated power consumption recorded. Traces are recursively categorized in two groups A and B from the first operation and for the next 14 ones thus forming a rooted tree. Each

leaf represents a 15 bits unknown value.

Figure 3.13 shows that when parsing the first scalar bit, all traces are well superposed. As 255 EC operations are performed instead of 256 (scalar length), we can conclude that the MSB is constant (and equal to 1) and 500k traces provide 2^{14} leafs with $500k/2^{14} \approx 30$ signatures. In order to remove wrongly categorized curve, each of the 30 curves are compared to the average of all power traces of the leaf and only the 25 best match are keep.

We work now only on one leaf, thus a set of 25 signatures with the same 15 MSB. The message used is "This is a test message", the SHA–256 of the message is:
$0x6f3438001129a90c5b1637928bf38bf26e39e57c6e9511005682048bedbef906 =$
50298988739713912396315566208261820918074097769561071417660208658292003502342

As an example, we give here three signatures, represented in decimal, however we work with all the set.
(r_1, s_1) :
40818870527619451644719711244939765778748181053397403427788471677447175829217
107539402501435392702812634518510467209619979448516202477004316951198642568026
(r_2, s_2) :
88437061408827753365795462915968315641110118768994223456506347312347102003787
98727873297874123848231526138109165393717483597725918105537237067528884092968
(r_3, s_3) :
45979991908241572238836614597429932250867720838767671374887652044439711494771
94060224568977366626160538534008548053491779844689742297995436580723181795508

From the set of 25 signatures, 25 equations can be written, the three following equations correspond to the three previous signatures (truncated to ($MSB...LSB$) in decimal due to writing constraints):

$$(5029...2342) - (1075...8026) \cdot k_1 + (4081...9217) \cdot d \equiv 0 \mod (1157...4369)$$
$$(5029...2342) - (9872...2968) \cdot k_2 + (8843...3787) \cdot d \equiv 0 \mod (1157...4369)$$
$$(5029...2342) - (9406...5508) \cdot k_3 + (4597...4771) \cdot d \equiv 0 \mod (1157...4369)$$

No information are known about the private key d, thus d is removed from the equations.

$$(5029...2342) - (1075...8026) \cdot k_1 + (4081...9217) \cdot d \equiv 0 \mod (1157...4369)$$
$$\Leftrightarrow d \equiv (7623...9043) \cdot k_1 + (8257...4784) \mod (1157...4369)$$
$$k_2 + (4253...8423) \cdot k_1 + (4003...1090) \equiv 0 \mod (1157...4369)$$
$$k_3 + (1461...6647) \cdot k_1 + (1135...7846) \equiv 0 \mod (1157...4369)$$

All we know about nonces is that they share the same 15 MSB bits. Thus, we can write the above equations as following with $0 \leq C < 2^{15}$:

$$k_{2_lsb} + (4253...23) \cdot k_{1_lsb} + [(4003...90) + (1 + (4253...23)) \cdot C \cdot 2^{251}] \equiv 0 \mod (1157...369)$$
$$k_{3_lsb} + (1461...47) \cdot k_{1_lsb} + [(1135...46) + (1 + (1461...47)) \cdot C \cdot 2^{251}] \equiv 0 \mod (1157...369)$$

From the 24 remaining equations, we can build a lattice of dimension 25 generated by the rows of:

$$A = \begin{pmatrix} -1 & (4253...23) & (1461...47) & \cdots \\ 0 & (1157...69) & 0 & \cdots \\ 0 & \ddots & (1157...69) & \ddots \\ 0 & \cdots & 0 & (1157...69) \end{pmatrix}$$

This lattice is reduced to get B:

$$B = LLL(A) = \begin{pmatrix} (-6856...88) & (3054...68) & \cdots & (-4495...47) \\ (-3320...20) & (1325...97) & \cdots & (-9601...69) \\ 0 & \ddots & \cdots & \ddots \\ (3707...38) & \cdots & \cdots & (1387...22) \end{pmatrix}$$

By construction, we know that it exists X such that:

$$XB - t = (k_{1_lsb}, k_{2_lsb}, k_{3_lsb}, ..., k_{25_lsb}) \text{ with}$$
$$t = (0, [(4003...90) + (1 + (4253...23))C \cdot 2^{251}], [(1135...46) + (1 + (1461...47))C \cdot 2^{251}], ...)$$

To find X, we first express t in the lattice basis and round the result.

$$\begin{cases} (-6856...88) \cdot \lambda_1 + (-3320...20) \cdot \lambda_2 + ... + (-4495...47) \cdot \lambda_{25} = 0 \\ (-3320...20) \cdot \lambda_1 + ... + (-9601...69) \cdot \lambda_{25} = [(4003...90) + (1 + (4253...23))C \cdot 2^{251}] \\ (-3396...70) \cdot \lambda_1 + ... + (-1179...340) \cdot \lambda_{25} = [(1135...46) + (1 + (1461...47))C \cdot 2^{251}] \\ ... \end{cases}$$

We solve this system 2^{14} times for all possibilities of C with the MSB set to 1. This will give us 2^{14} different X and thus 2^{14} values of $(k_{1_lsb}, k_{2_lsb}, k_{3_lsb}, ..., k_{25_lsb})$ and thus 2^{14} possibilities of d.

By comparing all $d.G$ value to the public key Q, we can determine if one of the guessed value is good or not.

In our case, we found the correct secret key that is:
$d = 0xc166ea345491b1576ff9e8166df96b5f4cd3ae47350efff2446f28f29a5883ee$
The nonces values used during the given signatures was the following:

$k_1 = 0x8a1234a8b3739b347af417cdbf8ff73649c5e62ef81767e20626adff81ade12c$
$k_2 = 0x8a125c71934ca50d396c299c01c02aba94e1f97fec13d5edb7932aab7aabc2c9$
$k_3 = 0x8a131fe2d71cd3a81aa6786a283ca05dfe4de37dddca27b6b5ad9f6847a9ba53$

We are thus able to recover the secret without knowing the value of any bit and with a limited knowledge and access to the target. The data dependent leakage discussed in section 3.2 allowed us to find signatures with the same MSBs. This knowledge is enough to break the system as only few bits have to be brutforced to fully recover the key. Our inefficient implementation (Matlab/Mupad) is able to solve all 2^{14} possibilities in around 3 hours on a basic laptop. It is also to be noted that in this example case, the LLL does not have to be recomputed for each hypothesis (A does not rely on C) and allowed to save a lot of time. Indeed, the most time consuming operation for the lattice section is the lattice reduction. It is also to be noted that fewer power acquisition is necessary to break the system. By performing the lattice attack on a group of signature with the same 14 bits (instead of 15), and by using the knowledge of which signature nonce have the 15th bit set to A or B, we do not increase the brutforce complexity and divide per two the total number of signatures power trace.

3.4 Summary

Different side channel leakage sources that allow recovering information about the EC scalar were described and demonstrated. While some leakages were already well known, such as the operation flow that is scalar dependent inside the double-and-add EC scalar algorithm, less obvious leakages were presented. Demonstration was provided that leakages can be unintentionally inserted due to the choice of coordinates representation which result in having different "+" operations that can be distinguished. Various leakages due to the use of the infinity point were also discussed and illustrated in different cases to recover information even when EC points are blinded. These leakages can be used either with power consumption or also with a basic timing attack.

While such leakages do not allow to fully recover the EC scalar, they may be enough to get some bit values. Mathematical attacks allowing to recover the $ECDSA$ private key if partial information of the nonces are available are relatively easy to implement and use without requiring any extended mathematical background. Thus these leakages even small can be enough to jeopardize the security of an $ECDSA$ implementation. Indeed, we demonstrated that side channel collisions between signature generations may allows recovering the $ECDSA$ private key thanks to lattice attacks and brutforce without even requiring to know any bits value.

From the various side channel evaluations and observations, it becomes obvious that

EC scalar algorithms should be carefully chosen and provides side channel countermeasures which allow to protect every single bits of the scalar. As side channels are not the only non-invasive threat, the next chapter aims at studying how fault injection attacks can be used to also recover some bits of the EC scalar. Studying both is important as countermeasures against side channel or fault injection attacks should not jeopardize the security of each others.

CHAPTER 4

Fault Attacks Against ECC and ECDSA

The effect of the environment on semiconductors is studied since a long time. Back in 1957, [57] studied the effect of nuclear radiation on semiconductor devices. It was observed various modifications of device characteristics due to gamma flux either transient or permanent on both germanium and silicon based devices. The transient behavior and response due to ionized radiation are then studied in [58] and [59] by using X-ray. In [60] the author determined that a pulsed-infrared laser is an inexpensive and effective way to simulate the effects caused by gamma ray sources in semiconductor devices. In 1975, after anomalies in communication satellite, [61] investigated the interaction of galactic cosmic rays with devices. In [62], single-bit soft error in dynamic RAMs and CCDs due to alpha particles are observed. Then authors in [63] provide a method for evaluating the effects of cosmic rays on computer memories. While these researches focus on harsh environment effects on semiconductor and possible countermeasures mostly for aerospace systems, more modern researches aimed at intentionally inducing errors to break implementation of cryptographic algorithm. The first academic discussion of such an attack is [64] in 1997 which presents vulnerabilities in various RSA implementations. Then in [65] the first Differential Fault Analysis (DFA) is presented against a DES cipher. In 2003, [66] details the first real-world implementation of such attack. Since then, different attacks and countermeasures have been proposed. Numerous ways of inserting faults had been considered such as using clock or power glitching [18], [19], overclocking [16], [67], [68], [69] and [70], under-powering [17], [71], [72], temperature [20] or EM [21]. Currently, the most effective fault injection method is obtained by using a laser such as in [22], [73], [74] or by using Forward Body Biasing Injection attack

($FBBI$) [23], [24] which simply consists in applying a high magnitude transient voltage pulse to a needle near the backside of the IC. As depicted in figure 4.1, the IC is decapsulated and placed on an XY positioning table allowing placing the target under the laser beam or the $FBBI$ needle.

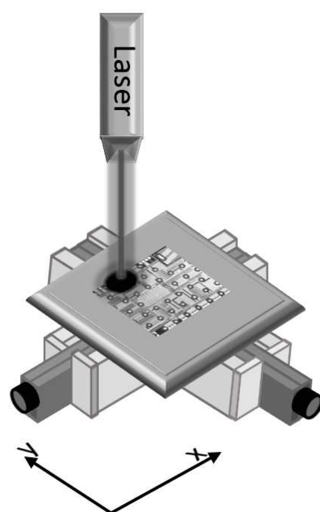

Figure 4.1: Illustration of a laser-based fault injection setup

The XY table allows targeting specific area of the IC. The attack can happen both on front-side or in backside. It is interesting to notice that laser evolution, moving from $Nd : YAG$ laser as in [75] to LED based laser, greatly improve attack possibilities. Indeed, LED based lasers allow reducing the jitter of the beam while providing a wide range of pulse durations ranging from $3ps$ to continuous wave at a decent power. This greatly help to target a specific operation of the IC at low cost. Most modern laser attack stations contain different lasers that can be used simultaneously to fault different parts of the IC [76]. More recently, software based fault injection methodologies are also researched allowing remote fault injection. The Rowhammer attack [25] is a good example, it consists in repeating the toggling of a $DRAM$ row's wordline that accelerates charge leakage from nearby rows which cannot be compensate by the memory refresh system and end up with faulted bits. Another example is the Clkscrew [77] attack which by using the SOC $DVFS$ allows forcing the chip to operate outside its operating range resulting to a fault. Combined with an embedded software for timing profiling, both attack allow to overcome ARM Trustzone isolation [78].

Different faults and countermeasures in various cryptosystems are reviewed in [79]. Unfortunately, fault injection to recover secret keys is still an active field of research and previously presented countermeasures become not enough. In the following chapter,

various attack scenarios that can be used on some modern $ECDSA$ implementations are described. As fault injection result depends on the architecture, chip technology and so on, the success rate of the fault injection is not studied. However, in the following the considered faults required for each attack do not need to be very specific. Indeed, random fault on different group of bits are considered instead of, as example, single bit flip. Moreover, for some described fault attacks, attackers may have many cycles (≈ 3000 or more due to the elliptic curve arithmetic) of opportunity allowing them to insert multi faults to ensure a success rate of fault insertion inside the computation of around 1.

The aims of this chapter is to identify security threats related to fault injection to determine later, in chapter 5, proper countermeasures independently of the success rate of the fault injection. We show that some elliptic curve scalar algorithms are wrongly supposed to be safe-error resistant due to either the algorithm or to underlying computations. The Montgomery ladder and the coherency checking countermeasure are two examples. We also introduce the concept of dummy operand. We demonstrate that even if in the $ECDSA$ signature, the scalar represents a nonce that is refreshed for each signature, safe-errors are enough to recover the private key.

Section 4.1 exhibits the power of safe-error against the EC scalar operation that we thought is underestimated. Then in section 4.2, faults on the s part of the $ECDSA$ signature are discussed. It shows that $ECDSA$ signatures generated from faulted private key can be used to recursively recover the key bits. Then a similar method is applied on nonces allowing to provide enough bits of information to be used within lattice attacks. It also demonstrates that the error distribution allow attackers to understand the behavior of the injected fault. Section 4.2 also discusses opportunities of injecting the considered faults in a basic architecture. Finally, the chapter is concluded in section 4.3.

4.1 Safe-error attack against the scalar algorithm

Some elliptic scalar point multiplication algorithms are designed to resist against SPA by adding dummies operations. These operations aim at providing a constant operation flow without being useful from an arithmetic point of view. Algorithm 4.1 is a good example of such practice.

Algorithm 4.1 Coron always Double-and-add

Input: $k = (k_{t-2}, ..., k_1, k_0)_2, P \in E(F_q)$
Output: $k.P$

1: $Q[0] \leftarrow P$
2: **for** $i = t - 2$ to 0 **do**
3: $\quad Q[0] \leftarrow 2Q[0]$
4: $\quad Q[1] \leftarrow Q[0] + P$
5: $\quad Q[0] \leftarrow Q[k_i]$
6: **end for**
7: **return** $(Q[0])$

C safe-error attacks aim at injecting a fault inside a computation and, thanks to the error propagation or non-propagation, to conclude about a secret value from the correct or incorrect result. In [80], authors have presented the C safe-error attack against the dummy operation inside an RSA exponentiation. The basic idea behind this attack is to inject a fault during a dummy operation and to see if it is propagated or not. From algorithm 4.1 in the case $k_i = 0$ then the result of $Q[1] = Q[0] + P$ is never used. Then, if one inject an error during this computation, the computation result will not be altered and then the attacker can deduce that the bit k_i was 0 at this time. Thus, an attacker can easily conclude about the k_i value if he is able to inject a fault inside the $Q[0] + P$ computation for the i index and observe if the result is correct or incorrect. This kind of fault injection is really powerful as attackers does not have to know which kind of fault is injected nor where exactly. Synchronization between the algorithm execution and the fault injection is simple in this case as algorithm 4.1 always execute the same operation flow. Moreover, the operation $Q[1] = Q[0] + P$ require many clock cycles (E.g. > 2816 cycles in the case of mixed jacobian-affine coordinates and a modular multiplication performed in 256 cycles) providing time and opportunity to the fault attack.
While safe-error are known since at least 15 years, we think that the effect on security is highly underestimated. Indeed, many EC algorithms are vulnerable to safe-error and even some algorithms presented as immune to this kind of attack. M safe-error attacks are similar to C safe-error attack and aim at targeting a memory value instead of a computation. The attacker concludes also depending on the propagation or not of the fault to the result.
In the following, we present different situations where safe-error can be used to recover enough information on the $ECDSA$ nonce during the EC scalar algorithm to mount a lattice attack.

4.1.1 Local dummy operations, C safe-error against the Montgomery ladder

Most common ECC scalar algorithms are known to be sensitive against the C safe-error attack and require specific countermeasures. In the literature only the Montgomery ladder presented in algorithm 4.2 is referenced as resistant against this attack by construction. In [81], the authors claim this resistance as there are no dummy operations in the Montgomery ladder. Also, they described the M safe-error attack against the Montgomery ladder that targets a memory value instead of a computation in order to see if it is used or not and then they provide a modified Montgomery ladder 'protected' against side-channel, M safe-error and also C safe-error per the previous claim.

Algorithm 4.2 Montgomery scalar operation

Input: $k = (1, k_{t-2}, ..., k_1, k_0)_2, P \in E(F_q)$

Output: $k.P$

1: $R_0 \leftarrow P$
2: $R_1 \leftarrow 2P$
3: **for** $i = t - 2$ to 0 **do**
4: $\quad R_{1-k_i} \leftarrow R_0 + R_1$
5: $\quad R_{k_i} \leftarrow 2R_{k_i}$
6: **end for**
7: **return** (R_0)

One missing information is that, from algorithm 4.2, if $k_i = 0$ the operation $R_{1-k_i} \leftarrow R_0 + R_1$ may become a dummy operation. It is clearly visible that for $k = (1, 0, ..., 0, 0)_2$ the register R_1 is never involved in the result. Algorithm 4.3 is an illustration of the Montgomery scalar operation with this specific scalar value.

This observation can be used in order to break an $ECDSA$. Indeed, if a nonce is randomly selected and then used as the scalar k in algorithm 4.2, an attacker may inject a fault into $R_{1-k_i} \leftarrow R_0 + R_1$ during k LSBs, and then deduce if k_{LSB} are equal to 0. After gathering a couple of signature with nonces LSBs set to 0, it is possible to perform a lattice attack [38] to recover the private key. As we did not find any public record of such an attack, here is a short example of the principle. An attacker looking for 70 signatures with 9 known bits of nonces LSBs (see section 2.6.4) in order to perform a lattice attack against an $ECDSA$ based on $NIST$ P-256 will need to fault $R_0 + R_1$ during evaluation of the nine LSBs for $2^9 \cdot 70 = 35840$ signatures generation.

Algorithm 4.3 Montgomery scalar operation with a specific scalar value

Input: $k = (1, 0, ..., 0, 0)_2, P \in E(F_q)$

Output: $k.P$

1: $R_0 \leftarrow P$
2: $R_1 \leftarrow 2P$
3: **for** $i = t - 2$ to 0 **do**
4: $\quad R_1 \leftarrow R_0 + R_1$
5: $\quad R_0 \leftarrow 2R_0$
6: **end for**
7: **return** (R_0)

This is due to the uniform distribution of the nonces, the probability of having nine LSBs set to 0 is $1/2^9$, thus 2^9 signatures are needed to find one signature with the nine nonce LSB equal to zero. A valid signature means that the faults did not propagate to the result and thus that the nine LSB were set to 0. This attack scheme is realistic as each elliptic curve operation requires time, any fault on the targeted operation can be used and the algorithm is highly homogeneous facilitating the fault injection. As all faults can be injected in the same physical location of the IC, a single LED based laser is enough to perform the attack. Moreover, the targeted operation $R_{1-k_i} \leftarrow R_0 + R_1$ requires time and multiple clock cycles allowing attacker to inject multiple faults in the computation to ensure the fault injection. Both data and control flow of low level operations (e.g. modular operations) can be targeted as long as the computation is corrupted without altering the signature process. Once R_1 contains a faulted value, it will propagate to the end releasing the need to fault followings $R_{1-k_i} \leftarrow R_0 + R_1$ operations. Regarding section 2.6.4, our number in term of required signatures (35840) is realistic in order to break an $ECDSA$ based on $NIST$ P-256 with a basic, non optimized lattice attack and does not take into account the information of the scalar MSB bit directly given by the algorithm. The MSBs leakage, in this case, can easily be taken in account inside the lattice attack. Indeed, from [37], the s part of the $ECDSA$ signature is equal to:

$$s_j \equiv k_j^{-1}(H(m_j) + d \cdot r_j) \mod n$$
$$\Leftrightarrow k_j - d \cdot r_j/s_j - H(m_j)/s_j \equiv 0 \mod n$$
$$\Leftrightarrow k_{j_{MSB}} + k_{j_{unknow}} \cdot 2^x + k_{j_{LSBs}} - d \cdot r_j/s_j - H(m_j)/s_j \equiv 0 \mod n$$

Where $H(m_j)$ is the hash result of the message m_j, d is the private key, k_j the random nonce, r_j the r part of the signature, j represents the jth signature and $k_{j_{unknow}}$ represents all the unknown bits of k . x depends on the number of LSB set to 0.

By considering a 256 bit curve, $k_{j_{MSB}} = 1$ and $k_{j_{LSBs}} = 0$:
$\Leftrightarrow k_{j_{unknow}} \cdot 2^x - d \cdot r_j/s_j - H_j/s_j + 2^{255} \equiv 0 \mod n$
$\Leftrightarrow k_{j_{unknow}} - d \cdot r_j/(s_j \cdot 2^x) - H_j/(s_j \cdot 2^x) + 2^{255-x} \equiv 0 \mod n$

Thus, if the MSB bit is given away, attackers can easily insert this knowledge inside the equations in order to reduce the researched values to the minimum improving lattice attack success rate (please refer to section 2.6 for details on the lattice attack). As the MSB of k is set to 1 and then provided, attackers will seek one bit less and then the expected consequence is to divide by almost two the total number of required signatures to ask to the system and thus the attack time. Indeed, in our example, with $2^8 \cdot 70 = 17920$ signatures, attackers will obtain 70 of them with 9 known bits (8 LSBs + 1 MSB). The M safe-error countermeasure presented in [81] does not affect this attack as it simply changes the operand order depending on the secret bit. We can then conclude that the Montgomery ladder, as opposed to a common thought ([81], [82], [55], [83]...), is in some case sensitive to C safe-error attacks. It also demonstrates that safe-error attacks can be used even with an ephemeral scalar value in the $ECDSA$ case. A good practice could be to systematically verify that $R_0 - R_1 = P$ prior any result exposure. However, this is not enough. Indeed, during the operation $R_{1-k_i} \leftarrow R_0 + R_1$, a fault may occur on R_1 once R_1 value is used in the computation and prior the update of R_{1-k_i}. This is called M safe-error attack and similarly to C safe-error attack, it lead to know the value k_i through correct or incorrect result. The countermeasure in the case of elliptic curve is not obvious, especially considering the complexity of the elliptic curve point addition as detailed later in section 4.1.4.

4.1.2 Unused memory values

Similarly, errors can be inserted on a memory saved value which is no longer used. To illustrate the problem, the algorithm described in [84] based on the binary expansion and the randomized initial point (RIP, initially presented in [85]) is considered. The presented algorithm aims to be resistant against Simple Power Analysis (SPA), Differential Power analysis (DPA) [50], Refined Power Analysis (RPA) [86] and Zero-value Point Attacks (ZPA) [87]. It is described as algorithm 4.4.

For $k = (0, ..., 0, 0)_2$, T is never used, therefore, an attacker can fault T inside the memory prior evaluation of the scalar LSBs and thus detect if they are equal to zero similarly to Section 4.1.1. If an attacker is able to induce a temporary fault inside T during a loop iteration, the result is even worse. Indeed, in this case attackers could target any part of the scalar as after attacking during k_i bit, the fault no longer exist allowing the system to work properly and pursuing operation. It is worth to notice that this unused value problem may arises with different algorithm and most algorithms that

Algorithm 4.4 Binary Expansion with RIP (BRIP)

Input: $k = (k_{t-1}, ..., k_1, k_0)_2, P \in E(F_q)$

Output: $k.P$

1: $R \leftarrow randompoint()$
2: $T \leftarrow P - R$
3: $Q \leftarrow R$
4: **for** $i = t - 1$ to 0 **do**
5: $\quad Q \leftarrow 2Q$
6: $\quad$ **if** k_i **then**
7: $\quad\quad Q \leftarrow Q + T$
8: $\quad$ **else**
9: $\quad\quad Q \leftarrow Q - R$
10: $\quad$ **end if**
11: **end for**
12: **return** $(Q - R)$

use pre-calculated to accelerate the computation may be subject to the same kind of attack as the one previously described. Indeed, attackers may fault some pre-computed values at some instant of the algorithm execution and deduce if they are used or not from a correct or incorrect result leading to provide information about the secret scalar value.

4.1.3 The infinity point and dummy operands

Despite the fact that the infinity point (i.e. the neutral element) usually generates particular cases in the computation as usual elliptic curve point doubling and point addition formulæ does not work with it, some scalar point multiplication algorithms even in the most modern implementations still use it. Algorithm 4.5 is an example of such algorithms. It uses pre-calculated points in order to speed up the computation and a simple loop to sequentially add the correct points. The scalar k is represented with both positive and negative coefficients in order to reduce the total number of pre-calculated points and thus the memory and implementation cost.

As defined above, this algorithm uses the infinity point for the initialization and also as a pre-calculated point. It can then be expected that, in most cases, attackers may recover all $r_j = 0$ through side channel analysis due to the specific arithmetic involved as explained in [88]. In [89] the authors describe the same algorithm used with an Edwards curve. An interesting property of Edwards curves is the fact that the elliptic curve point addition formula is a unified and complete addition law that works for both

Algorithm 4.5 Scalar operation with pre-computed points

Input: $k = (r_{\lceil t/4 \rceil}, ..., r_1, r_0)_{2^4}$, $|r_j|.16^a.P \in E(F_q)$, with $0 \leq a \leq \lceil t/4 \rceil$ for any $r_j \in \{-8, -7, -6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7\}$

Output: $k.P$

1: $Q \leftarrow \mathcal{O}$
2: **for** $i = \lceil t/4 \rceil$ to 0 **do**
3: $\quad Q \leftarrow Q + r_j.16^i.P$
4: **end for**
5: **return** (Q)

point addition, point doubling and also with the neutral element without any special condition. However, the neutral element is still special. Indeed, the neutral element $(0, 1)$ under the twisted Edwards addition law presented in [90] generates special arithmetic cases. The given law is the following:

$$(x_1, y_1) + (x_2, y_2) = (\frac{x_1 \cdot y_2 + x_2 \cdot y_1}{1 + d \cdot x_1 \cdot x_2 \cdot y_1 \cdot y_2}, \frac{y_1 \cdot y_2 + x_1 \cdot x_2}{1 - d \cdot x_1 \cdot x_2 \cdot y_1 \cdot y_2}) \quad (4.1)$$

With the neutral element, we obtain:

$$(x_1, y_1) + (0, 1) = (\frac{x_1 \cdot 1 + 0 \cdot y_1}{1 + d \cdot x_1 \cdot 0 \cdot y_1 \cdot 1}, \frac{y_1 \cdot 1 + x_1 \cdot 0}{1 - d \cdot x_1 \cdot 0 \cdot y_1 \cdot 1}) = (x_1, y_1) \quad (4.2)$$

In order to avoid side channel attacks, the system should perform multiplication operations similarly with random operands than with specific values such as 0 or 1. This mean $x_1 \cdot 1 \mod p$ similarly to $x_1 \cdot y_2 \mod p$, $0 \cdot y_1 \mod p$ similarly to $x_2 \cdot y_1$ and so on. And also perform inversions without leaking information, i.e. $A/1 \mod p$ similarly to $A/B \mod p$ for any A and B. With "similarly" meaning, without any noticeable timing difference due to a simplification nor without a different power consumption that could be due to carry propagation, modular reduction or other. Such system is not obvious and this prerequisite limit the Even considering the design is safe against side channel analysis, other problems appear if fault injection is considered. Indeed, in equation (4.2), a fault can be injected in y_1 operand when $0 \cdot y_1$ is computed. Faults can also be injected in dx_1 operands or computation without affecting the result. Such faults will not be propagated to the result in the case where the neutral element is used. Thus, due to this special point the design may face up safe-error attacks. Table 4.1 provides the identity element representation in different systems. From this table, it can be expected that the neutral element, due to the zero and one values, will generate special computation cases in most systems that can be detected thanks to side channel or fault injection and thus should be avoided.

Table 4.1: Representation of the neutral element in different systems

System	$\mathcal{O}$
Weierstrass, affine coordinates	none
Weierstrass, projective coordinates	(0:1:0)
Weierstrass, Chudnovsky coordinates	(1:1:0:0:0)
Edwards, affine coordinates	(0,1)
Edwards, homogeneous projective coordinates	(0:1:1)

4.1.4 Safe-error on underlying algorithms

As a safe-error resistant algorithm, [91] suggests to use a countermeasure based on coherency checking. Algorithm 4.6 presents this countermeasure.

Algorithm 4.6 Right-to-left double-and-add always with coherency checking

Input: $k = (k_{t-1}, ..., k_1, k_0)_2, P \in E(F_q)$

Output: $k.P$

1: $Q_0 \leftarrow \mathcal{O}$
2: $Q_1 \leftarrow \mathcal{O}$
3: $Q_2 \leftarrow P$
4: **for** $i = 0$ to $t - 1$ **do**
5: $\quad Q_{k_i} \leftarrow Q_{k_i} + Q_2$
6: $\quad Q_2 \leftarrow 2Q_2$
7: **end for**
8: **if** $Q_0 \in E(F_q)$ and $Q_1 \in E(F_q)$ and $Q_2 = Q_0 + Q_1 + P$ **then**
9: $\quad$ **return** (Q_1)
10: **else**
11: $\quad$ **return** $(\mathcal{O})$
12: **end if**

Despite the fact this algorithm may leak through side channel the number of times the MSB value repeats itself due to the use of the infinity point (Q_0 and Q_1 are both initialized to $\mathcal{O}$ and will be updated at the first $k_i = 0$ and $k_i = 1$ which can be detected), the coherency checking provides an interesting approach to validate that all computations are done correctly. Indeed, all computations are not necessary to end up with the good result in Q_1 as some of them are dummy operations (i.e. operations involving Q_0). However, the checking $Q_2 = Q_0 + Q_1 + P$ line 8 involves both Q_0 and

Q_1. Thus, it is expected that faults on any $Q_{k_i} \leftarrow Q_{k_i} + Q_2$ computations, line 5, or memory addresses will be detected independently of the scalar value k.

Nevertheless, the "+" operation is an elliptic curve point addition which is a complex operation. Algorithm 4.7 is an example of ECC point addition operation performed in Jacobian coordinates. If we consider the line 8 of algorithm 4.6, the operation $Q_{k_i} \leftarrow Q_{k_i} + Q_2$ is performed, meaning that either Q_0 or Q_1 will be updated with the result. If algorithm 4.7 is used, Q_0 or Q_1 will be updated with the $(X_3 : Y_3 : Z_3)$ result of algorithm 4.7. A fault on Q_0 x-coordinate between line 7 and line 19 of algorithm 4.7 will either generate an error or not during the coherency checking. Indeed, if $k_i = 0$, then Q_0 will be updated with the result $(X_3 : Y_3 : Z_3)$ of algorithm 4.7. Thus, the faulted x-coordinate will be updated with the correct result X_3 and then no error will be detected. As opposed, if $k_i = 1$, Q_1 will be updated with the result $(X_3 : Y_3 : Z_3)$ of algorithm 4.7 and the error on Q_0 x-coordinated will remain and be detected during the coherency checking. An attacker can thus guess the value of k_i by using an M safe-error attack despite the countermeasure. The same attack can be applied also on Q_1. And other lines of algorithm 4.7 can be targeted (e.g. X_3, Y_3, Z_3 can be used).

Algorithm 4.7 ECC Jacobian point addition

Input: $P = (X_1 : Y_1 : Z_1)$ and $Q = (X_2, Y_2, Z_2)$ on $E(F_q) : y^2 = x^3 - 3x + b$

Output: $P + Q = (X_3 : Y_3 : Z_3)$

1: $T_1 \leftarrow Z_1^2$
2: $T_3 \leftarrow Z_2^2$
3: $T_2 \leftarrow T_1 \cdot Z_1$
4: $T_1 \leftarrow T_1 \cdot X_2$
5: $T_4 \leftarrow T_3 \cdot Z_2$
6: $T_3 \leftarrow T_3 \cdot X_1$
7: $T_2 \leftarrow T_2 \cdot Y_2$
8: $Y_3 \leftarrow T_4 \cdot Y_1$
9: $T_1 \leftarrow T_1 - T_3$
10: $T_2 \leftarrow T_2 - Y_3$
11: **if** $T_1 == 0$ **then**
12:     **if** $T_2 == 0$ **then**
13:         $(X_3 : Y_3 : Z_3) = Compute(2Q)$
14:         **return** $(X_3 : Y_3 : Z_3)$
15:     **else**
16:         **return** $(\mathcal{O})$
17:     **end if**
18: **end if**
19: $Z_3 \leftarrow Z_1 \cdot T_1$
20: $Z_3 \leftarrow Z_2 \cdot Z_3$
21: $X_3 \leftarrow T_1^2$
22: $T_4 \leftarrow X_3 \cdot T_1$
23: $T_3 \leftarrow X_3 \cdot T_3$
24: $T_1 \leftarrow 2T_3$
25: $X_3 \leftarrow T_2^2$
26: $X_3 \leftarrow X_3 - T_1$
27: $X_3 \leftarrow X_3 - T_4$
28: $T_3 \leftarrow T_3 - X_3$
29: $T_3 \leftarrow T_3 \cdot T_2$
30: $T_4 \leftarrow T_4 \cdot Y_3$
31: $Y_3 \leftarrow T_3 - T_4$

4.2 Fault attacks against the ECDSA signature

The $ECDSA$ signature is a complex operation as it involves different computations with different secrets such as the private key or the nonce. Thus, the number of attack possibilities is large. Indeed, the elliptic curve scalar operation can be targeted as previously detailed, however, the random number generator, modular operations or memory access can also be faulted. Previously, we focused on the elliptic curve scalar operation, in this section, we overview the consequences of faults on the private key, the nonce or on intermediate values outside the scalar operation.

4.2.1 Faulted secret key

Each $ECDSA$ signature is generated from a nonce k and the private key d. We recall here the $ECDSA$ key generation and signature.
Key generation:
- Select a random d such that $0 \leq d \leq n-1$
- Compute $Q = d.P$ with $P \in E(F_q)$
- d is the private key, Q is the public one.

Signature:
- Generate a nonce k such that $0 \leq k \leq n-1$
- Compute $G = (x, y) = k.P$ with $P \in E(F_q)$
- $r \equiv x \mod n$
- $s \equiv k^{-1}(H(msg) + d \cdot r) \mod n$
- (r, s) is the signature.

The signature (r, s) can be verified by using the public key Q and $E(F_q)$ as explained in section 2.5. If an attacker is able, somehow, to fault d into $d' = d \pm e \cdot 2^l$ during signature, the following will happen:
- Generate a nonce k such that $0 \leq k \leq n-1$
- Compute $G = (x, y) = k.P$ with $P \in E(F_q)$
- $r \equiv x \mod n$
- $s' \equiv k^{-1}(H_j(m) + (d \pm e \cdot 2^l) \cdot r) \mod n$
- (r, s') is the faulted signature.

In this case, the signature (r, s') cannot be verified from $Q = d \cdot P$ and $E(F_q)$. Indeed, from the verification formula:
$(x, y) = (H(msg) \cdot s'^{-1} \mod n) \cdot P + (r \cdot s'^{-1} \mod n) \cdot Q$
$\Leftrightarrow (x, y) = (H(msg) \cdot k \cdot (H_j(m) + (d \pm e \cdot 2^l) \cdot r)^{-1} \mod n) \cdot P + (r \cdot k \cdot (H_j(m) +$

$(d \pm e \cdot 2^l) \cdot r)^{-1} \mod n) \cdot Q$
$\Leftrightarrow (x,y) = [k \cdot (H(msg) + d \cdot r) \cdot (H(msg) + (d \pm e \cdot 2^l) \cdot r)^{-1} \mod n] \cdot P$
$\Leftrightarrow (x,y) \neq k \cdot P$ and $(x,y) \neq -k \cdot P$
$\Leftrightarrow x \neq r \mod n$

If $\pm e$ is small enough, i.e. can be exhaustively tested, and the position l known, an attacker can try to compute all possible $Q' = Q \pm e \cdot 2^l \cdot P$ until the signature (r, s') can be verified from Q' and $E(F_q)$.
Once e is found, the verification will pass:
$(x,y) = (H(msg) \cdot s'^{-1} \mod n) \cdot P + (r \cdot s'^{-1} \mod n) \cdot Q'$
$\Leftrightarrow (x,y) = [k \cdot (H(msg) + (d \pm e \cdot 2^l) \cdot r) \cdot (H(msg) + (d \pm e \cdot 2^l) \cdot r)^{-1}] \cdot P$
$\Leftrightarrow x \equiv r \mod n$

From the knowledge of $\pm e$, attackers obtain information about d. Indeed, as example, lets consider that the fault is simply a one bit flip on the LSB. Thus $e = \pm 1$. If $e = 1$, this means that the LSB flip from 0 to 1. If $e = -1$, this means that the LSB flip from 1 to 0. Thus the LSB is recovered.
Multi-bits attack is also possible however it is not as simple. Indeed, some e values can be injected in different ways. As example, if we consider an 8 bits register that is faulted and e the error injected. Then $e = +1$ means that either the LSB flips from 0 to 1 or that it flips from 1 to 0 and the fault also flips bits according to a carry propagation (e.g. if $0x01$ is faulted into $0x02$, the error e is 1 while the LSB flips from 1 to 0). Both solutions are possible and depends on the fault model (i.e. one bit flip, multi-bits flip, flip only 0 to 1...). A +1 error may happen on any value, except on 255 as 256 cannot be encoded into the 8 bits register. The same applies for every possible e. It is interesting to notice that an 8 bits register allows encoding values from 0 to 255 and thus for each value a unique range of error is possible. E.g. if the register encodes the value 6 then the error range is from -6 to $+249$ as $6 - 6 = 0$ and $6 + 249 = 255$. For the value 7, the range is $-7/+248$. This means that, once the error range is determined, the value is obtained. Figure 4.2 is a simulated example of the number of faults required to recover an arbitrary 8 bits value (0x84) thanks to the error range. It considers that a fault on the 8 bits register generates a uniformly random 8 bits value. The x-axis represents the number of faults. Y-axis is the probability of having the good result. E.g. 0.5 means that two possible values remain. In Figure 4.2 example, around 1500 faults were required to reduce the number of possibilities to a single one. This means the highest error minus the lowest equal 255. As we can see, couples of faults are enough to start to reduce the possibilities while a lot of them are required to end-up with the correct value.

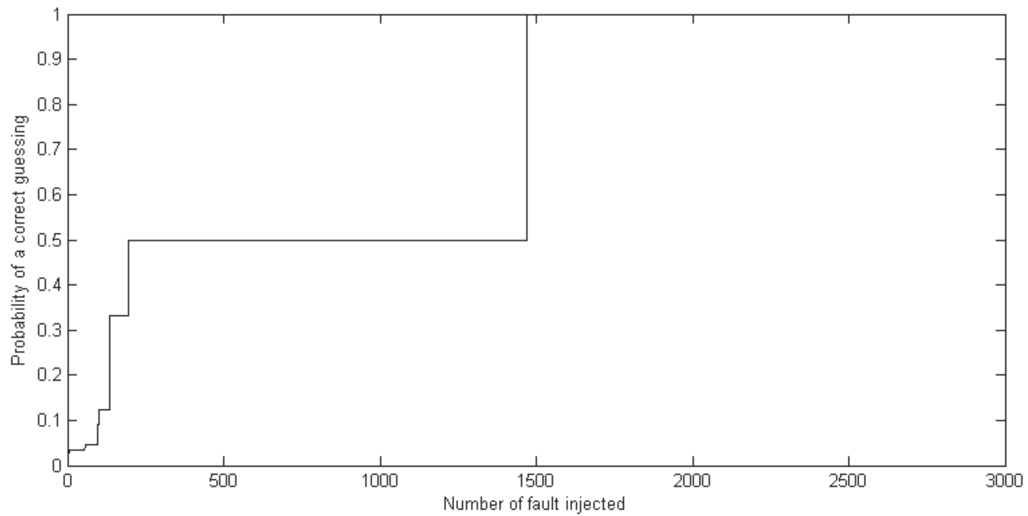

Figure 4.2: Illustration of the number of faults required to recover an 8 bits value arbitrary set to 0x84

Figure 4.3 represents the average of the results of different simulations with random initial values. From this figure, around 1300 faults are required to end-up with the good result with 90% of correct guessing. 290 are required to obtain two possibilities.

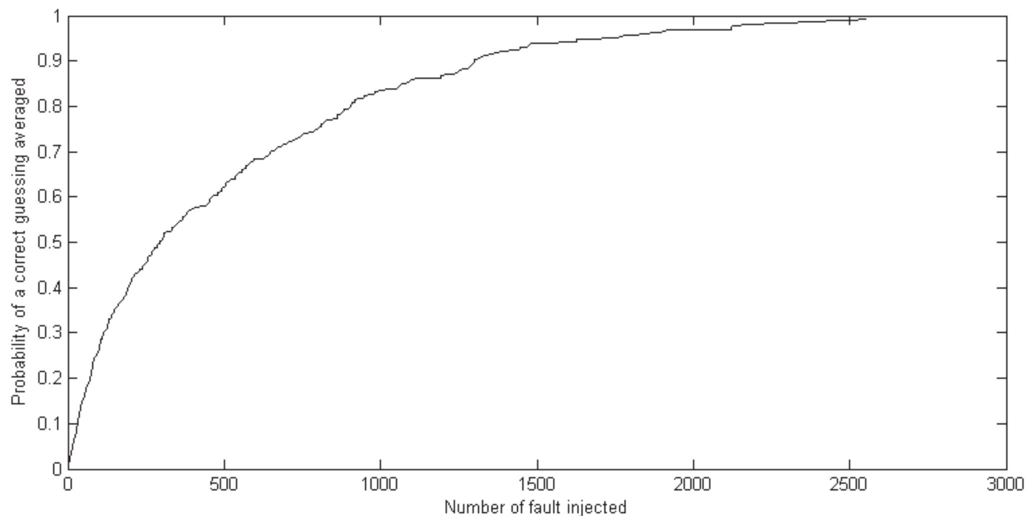

Figure 4.3: Average of the number of fault required to recover an 8 bits value

If someone is able to insert such faults on a 256 bits key length system, 1300 faults on 8 bits of the secret and a maximum of 2^9 computations after each fault to recover $\pm e$

allows recovering these 8 bits with 90% of chance. Doing so 32 times ($41k$ faults) allows recovering the full key with in average $log(1/0.9)/log(2) \cdot 32 = 4.86$ missing bits that can easily be brutforced. The number of necessary faults can be reduced depending on the computation power available, e.g. $\sim 20k$ faults and 2^{40} computations to recover $\pm e$ can also recover the full key. It is obvious that, if for some reasons attackers know the result value of the faulted register, by recovering $\pm e$, the genuine value is recovered. It is also to be noted that in a non-contiguous faulted bits case, the carry propagation cannot be generated by the fault, and thus, similarly to the previous LSB bit flip example, attackers directly recover the flipped bits. E.g. if the fault model considers fault only on MSB and LSB, a $+1$ error necessary reveals that the genuine LSB is 0. Indeed, the second bit cannot be faulted due to the model preventing the fault to flip bit according to the carry propagation required if the LSB is 1. This attack demonstrates that an error inserted on the $ECDSA$ private key during signature generation can be recovered from the faulted signature. Due to the binary representation of the private key, it is then possible to recover information about the private key.

4.2.2 Faulted nonce

For each new signature, a nonce k is selected and used for both the r and s parts of the signature. In the following, it is considered that an attacker is able to fault the nonce in such a way that r and s are computed with a slightly different k (with a couple of flipped bits).

In this case, the $ECDSA$ signature generation can be summarized as following:

- Generate a nonce k such that $0 \leq k \leq n-1$
- Compute $G = (x, y) = k.P$ with $P \in E(F_q)$
- $r \equiv x \mod n$
- $s' \equiv (k \pm e \cdot 2^l)^{-1}(H(msg) + d \cdot r) \mod n$
- (r, s') is the faulted signature.

As in the previous section, the signature will not pass the verification as r and s are not computed with the same nonce. And as previously presented, if e is small enough, i.e. can be exhaustively tested, and the position l known, an attacker can try to compute all possible $G' = (x', y') = kP \pm (e \cdot 2^l) \cdot P$ and $r' \equiv x' \mod n$ until the signature (r', s') can be verified from G' and $E(F_q)$. In $E(F_p)$, due to the Hasse interval ($n \in [p + 1 - 2\sqrt{p}, p + 1 + 2\sqrt{p}]$), the probability to get $r > n$ is less than $1/2^{127}$ for a 256 bits curve. Thus, most of the time $r = x$, and $kP = (x, y)$ can be recovered.

Once e and r' are found, the verification will pass:

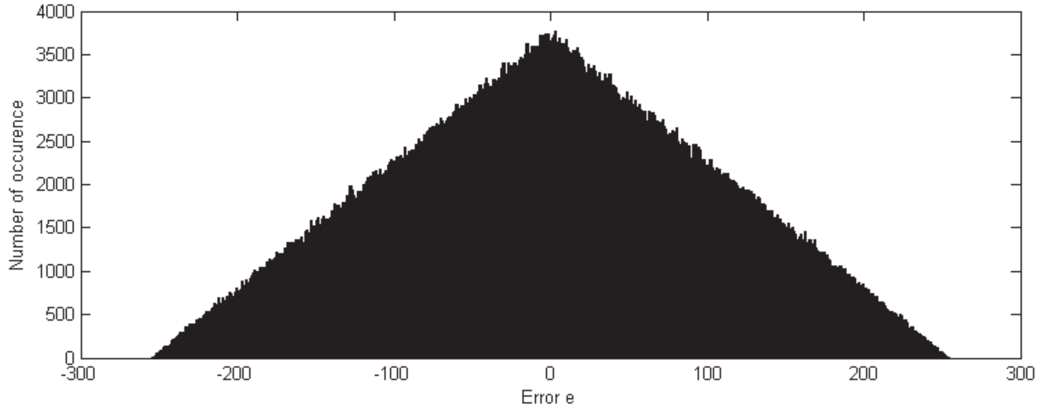

Figure 4.4: Error e distribution on a random 8 bits register

$$
\begin{aligned}
&(x, y) = (H(msg) \cdot s'^{-1} \mod n) \cdot P + (r' \cdot s'^{-1} \mod n) \cdot Q \\
&\Leftrightarrow (x, y) = [(k \pm e \cdot 2^l) \cdot (H(msg) + d \cdot r) \cdot (H(msg) + d \cdot r)^{-1}] \cdot P \\
&\Leftrightarrow (x, y) = kP \pm (e \cdot 2^l) \cdot P \\
&\Leftrightarrow r' \equiv x \mod n
\end{aligned}
$$

It is to be noted that two possible values of G' allow the verification. One is $G' = kP \pm (e \cdot 2^l) \cdot P$ and the second one is $G' = (n-k)P \pm (e \cdot 2^l) \cdot P$ with the sign of e inverted. This is due to the fact that $kP + (n-k)P = \mathcal{O}$, thus if $kP = (x, y)$ then $(n-k)P = (x, -y)$. As both x are the same, it results to the same r value.

Similarly to the previous section, from the knowledge of $\pm e$, attackers obtain information about k due to its representation. The difference is that k is a nonce that changes for every signature thus it is not possible to perform multiple fault attacks and learn k from all the answers. Nevertheless, as explained in the previous section, single bit fault and non-contiguous multi-bits faults (i.e. faults on non-adjacent bits) allow recovering all flipped bits in one attack.

In the case of contiguous multi-bits faults, it is interesting to notice that the distribution of the error e is not uniform. Indeed, per example, if we consider an 8 bits register that is faulted and e the error injected. And if we consider that a fault on the 8 bits register generates a uniformly random 8 bits value. Then the error distribution is as depicted in figure 4.4.

This is due to the fact that the error $e = +255$ may happen only if the register value is 0. The error $e = +254$ may happen only if the register value is 0 or 1. The value $e = 0$ (no error) may happen on any value of the register. And, as opposed, $e = -255$ may happen only on a +255 registers and so on. Thus the probability mass function (pmf) is $P(e) = (2^8 - |e|)/(2^{2 \cdot 8}), e = [-255; 255]$. This can be generalized to any register of size m with the function $P(e) = (2^m - |e|)/(2^{2 \cdot m}), e = [-2^m + 1; 2^m - 1]$.

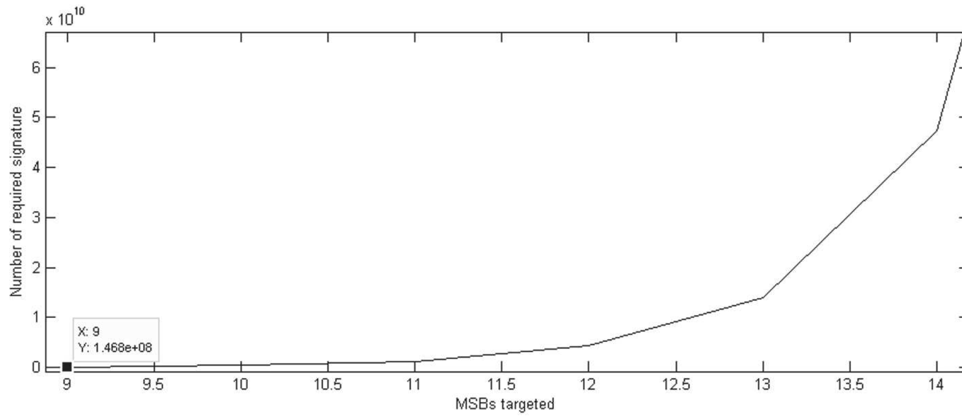

Figure 4.5: Total number of required signatures to recover the private key thanks to a lattice attack and random fault on MSB bits on NIST-P256

From this observation, we know that large $|e|$ values allow attackers to recover the MSBs of the faulted register while small $|e|$ value such as 0 provide less information. E.g. if an attacker obtains $e \geq 240$, then he directly knows that the genuine 8 bits register value is smaller than 15, thus the four MSBs are set to 0. Someone looking for x MSBs need to find $|e| \geq 2^m - 2^{m-x}$. The probability to find such an error value is $p_x \approx 1/2^{2x}$. Thus, $\approx 2^{2x+3}$ faults can be injected in order to obtain at least one which will provide x bits of information with a high probability. Doing so by targeting the $ECDSA$ signature nonces would enable a lattice attack as described previously in section 2.6. If the result on $NIST$ P256 provided section 2.6.4 are considered, then the total number of signatures required to recover the secret can be determined. Figure 4.5 provides these numbers for different number of targeted MSB.

As two possible values of G' exist that are kP and $(n-k)P$, then the guessed MSBs are either for k or $n-k$. These values are related. On $NIST$ P256, MSBs of the modulo n are set to 1s [37]. Thus, if the MSBs of k are set to 0s then the MSBs of $(n-k)$ are set to 1s and vis versa. For each gathered signature such as $|e| \geq 2^m - 2^{m-x}$, two solutions exist, either x MSBs of k are set to 0s or x MSBs of k are set to 1s. In the case where w signatures are required, then attacker should perform at most 2^w times the lattice attack with the different possibilities to end-up with the correct result. Nevertheless, the most time consuming operation of the lattice attack, namely the lattice reduction, is to be performed only one time if nonces MSBs are targeted as explained in section 3.3. From our result figure 2.5 and figure 4.5, 1.27 billions of signatures and 2^{32} CVP solving are required to obtain the private key. While this number seems huge, as explained section 2.6.4, our lattice implementation is far from the state of the art and less signature should be enough. In fact, this number represents the worst case. Indeed, it was previously considered that a uniformly distributed random

value is obtained in the faulted register after the fault insertion forcing inserting a lot of faults on different signatures to find particular ones. If, for some reasons, attackers know the result of the faulted register, by recovering e, the genuine value is recovered. Also, if the faulted register always end-up with a particular value or inside a set of value, then this can be detected and used (i.e. see section 4.2.4).

Figure 4.6 represents the error distribution when 8 bits of $ECDSA$ signatures nonces are faulted and forced to a value v.

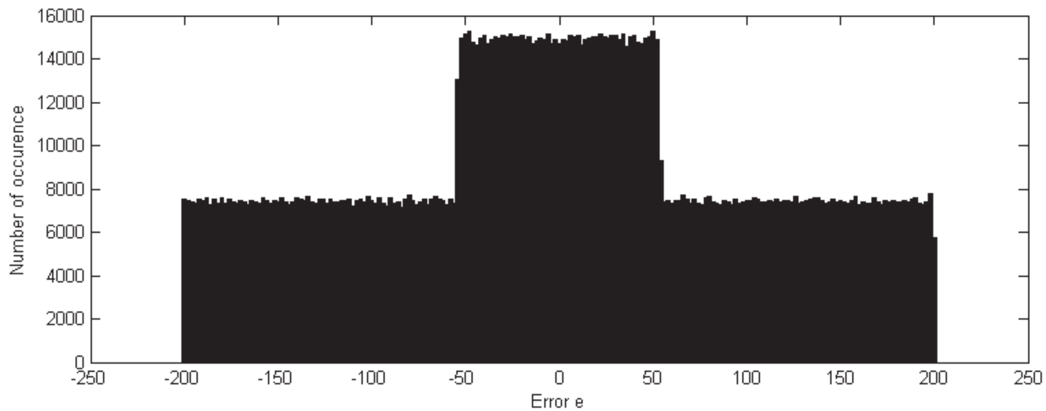

Figure 4.6: Number of occurrences of error e on a random $m = 8$ bits register that is forced to a constant value $v = 54$

The distribution should be uniform, however, due to the fact that two values of G' exist and conduct to recover either e or $-e$, the distribution is as depicted. It is the sum of two uniform distributions symmetrical to 0 centered around $v - (2^m - 1)/2$ and $-(v - (2^m - 1)/2)$ and with a width of $2^m - 1$. From the observed distribution, attackers have two candidates for the value v that are one of the falling edge (i.e. 54 or 201 in figure 4.6). Distributions on figure 4.4 and figure 4.6 can easily be distinguished, meaning that an attacker can know if the faulted register end-up with a random value or a constant one. In the last case, the number of required faulted signatures to recover the secret key is the number required to identify the error distribution ($\approx 10 \cdot 2^m$ can be enough, working with the absolute value allow reducing this number). Indeed, once attackers obtain it, a basic lattice attack can be conducted as the faulted nonces share the same m bits and are equal to one of the falling edge of the distribution.

If the faulted register results end-up in a small set of values, then the error distribution is the sum of the different distributions as depicted figure 4.7 where the faulted register either end-up to the value v_1 or v_2. In this case, more faults are required to identify

the distribution.

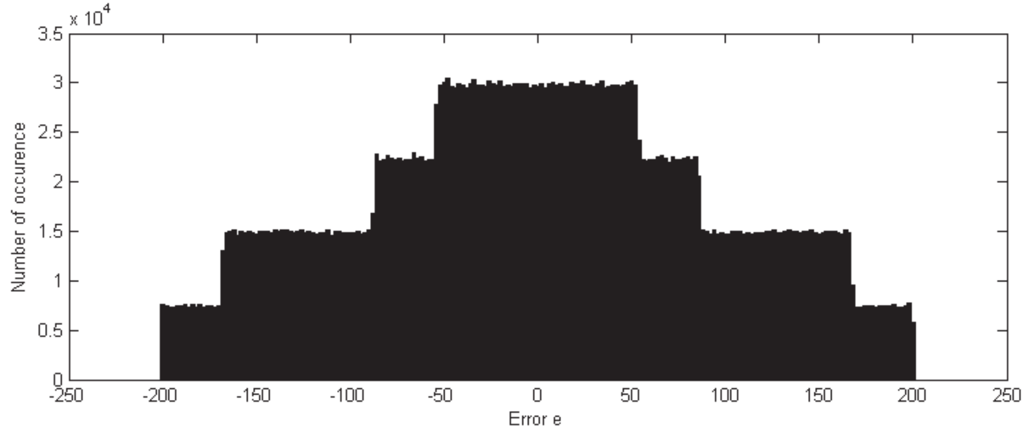

Figure 4.7: Error e distribution on a random $m = 8$ bits register that lead to two different values $v_1 = 54$, $v_2 = 168$

Depending on the error result, the register value can be categorized into v_1. Indeed, from figure 4.7 it can be determined that $v_1 = 54$ or 201 and $v_2 = 87$ or 168. If an attacker observes $|e| > 168$ then he knows that v_1 is the correct value. In the figure 4.6 example, around 16% of the faulted signatures end-up with the v_1 value. Around $10k$ faulted signatures are sufficient to identify the distribution edge +168, resulting in ≈ 1600 signatures with the v_1 value. 1600 signature with 8 known bits are enough in order to recover the private key thanks to a lattice attack. This represents the ideal case and the success rate of the fault injection may vary depending on the system and architecture choices.

4.2.3 Faulted intermediate values

The partial knowledge of intermediate values during an $ECDSA$ signature can also compromise the security of the secret key. Indeed, as demonstrated in [92], a partial knowledge of $(H(msg) + d \cdot r)$ allows a lattice attack to recover the secret d. Same apply on $d \cdot r$. It is possible to write:

$$(H(msg) + d \cdot r) \equiv A \mod n$$
$$\Leftrightarrow A - d \cdot r - H(msg) \equiv 0 \mod n$$

A represents an unknown value which is an intermediate result of the signature generation. This equation is similar to the one targeted by lattice attack in section 2.6.

Thus if partial information about A are leaked on enough signatures, then attackers can use a lattice attack to fully recover the private key d. If information about $d \cdot r$ leak, then $H(msg)$ is removed from the equation and the attack also work.

While [92] demonstrates an approach to reveal partial information in the specific case where operand scanning multiplier is used, other method can be used to recover the information. Indeed, a similar approach as in section 4.2.2 can be used. Indeed, assuming $d \cdot r$ is faulted with a fault e then the $ECDSA$ signature generation can be summarized as following:
- Generate a nonce k such that $0 \leq k \leq n-1$
- Compute $G = (x, y) = k.P$ with $P \in E(F_q)$
- $r \equiv x \mod n$
- $s' \equiv k^{-1}(H(msg) + d \cdot r \pm e \cdot 2^l) \mod n$
- (r, s') is the faulted signature.

The signature will not pass the verification. However, if e can be exhaustively tested, and the position l known, attacker can try to verify (r, s') with every possible $(H(msg) \pm e \cdot 2^l)$ combination. Once the signature pass the verification, e is discovered. The interpretation of the error e is similar to the one in section 4.2.2. Indeed, as r depends on the nonce and change for each signature then $d \cdot r$ will be different for each signature. The distribution of e over different signature will provide information about the kind of injected fault (random or constant value). Then signature with specific e value will leak information about the intermediate result A. The exact same happen if $d \cdot r$ is targeted instead of $(H(msg) + d \cdot r)$.

4.2.4 Example of architecture and fault opportunities

Architecture description

In the following, we consider a basic architecture that allows the generation of $ECDSA$ signatures. The architecture is described and then different situations allowing to recover the private key are discussed alongside how to fault/alter the system to reach these situations. Figure 4.8 is an overview of this architecture. It is composed of an $I2C$ communication block in order to enable interaction from the outside, a cryptographic core allowing $ECDSA$ signatures computation and two memories (a NVM plus an $SRAM$) to save information and as working space. The cryptographic core is composed of a RNG allowing nonces generation, an ECC accelerator that allows computing elliptic curve operations, a SHA block for messages hashing and an AES to internally encrypt/decrypt secrets.

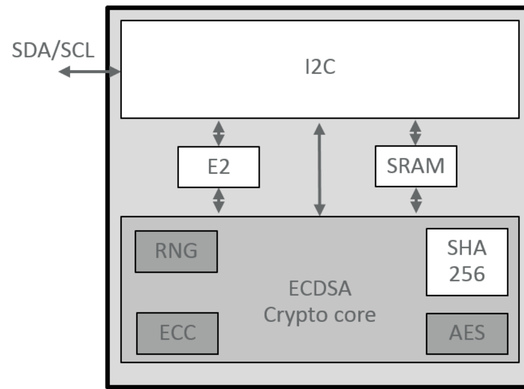

Figure 4.8: Overview of the considered architecture. Grey blocks containing some side channel and fault countermeasures

In this architecture, memories are shared and partitioned between the communication block and the cryptographic core. It is considered that $EEPROM$ and $SRAM$ are not accessible to the $I2C$ block while containing sensitive or partial information. We also consider that the RNG is perfect (i.e. uniformly random and not controllable from outside) and the ECC block contains strong side channel and fault countermeasures (i.e. resistant to any fault and side channel attack). The private key d is securely stored in the $EEPROM$ protected thanks to an AES cipher with also both side channel and fault countermeasures. The elliptic curve considered is $NIST$ P256. Memories are 32 bits width and 8 memory accesses are performed each time to read or write 256 bits ($8 \cdot 32 = 256$), the SHA and AES blocks are built with a 8 bits datapath.

The $ECDSA$ signature generation is as depicted in the chronograph presented in figure 4.9.

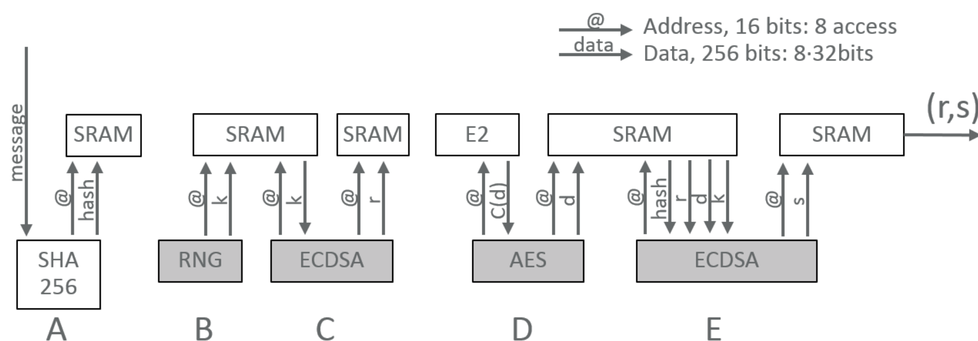

Figure 4.9: Chronograph of a $ECDSA$ signature generation

First, the message to sign is hashed thanks to the SHA 256 block. The result is then saved in the $SRAM$ to a specific address. A 256 bits nonce k is generated from the RNG and saved into the $SRAM$. The $ECDSA$ block then reads the nonce from the $SRAM$ when needed (for each bit evaluation, as shown in previously described elliptic curve scalar algorithm), performs an elliptic curve scalar operation and compute the r part of the signature that is saved into the $SRAM$ at another address. The ciphertext $C(d)$ of the private key d is read from the $EEPROM$ and then decrypted by the AES block. The result d is then saved into the $SRAM$. The $ECDSA$ block then reads the hash result, the r part of the signature, the private key d and the nonce k from the $SRAM$ and generates the s part of the signature. Once all sensitive information are cleared, the $ECDSA$ signature (r, s) of the message is then exposed to the outside. This architecture provides countermeasures against side channel and fault attacks at the cryptographic block level and not overall in the system. Errors can thus happen in blocks interfaces, in memories, or on memory controllers that may affect data during transfer or address used. In the following, based on this observation, different situations allowing security breach are studied.

Signatures with shared nonces

Two signatures computed with a different message and the same nonce are enough to recover the private key. Indeed, each signature j, provides an equation with two unknown values, the nonce and the private key.

$s_j = k_j^{-1}(H(m_j) + d \cdot r_j)$

Thus two signatures provide two different equations with three unknown values, two nonces and the private key. If the nonces are equals, then there are two different equations with two unknown values, the system is thus solvable as detailed in equation (4.3).

$$\begin{cases} s_1 \equiv k^{-1}(H(m_1) + dr) \mod n \\ s_2 \equiv k^{-1}(H(m_2) + dr) \mod n \end{cases}$$

$$\Leftrightarrow \begin{cases} k \equiv s_1^{-1}(H(m_1) + dr) \mod n \\ d \equiv (s_2 H(m_1) - s_1 H(m_2)) \cdot (r(s_1 - s_2))^{-1} \mod n \end{cases} \tag{4.3}$$

In the considered architecture, nonces are refreshed from the RNG for each signature. However, one can try to interfere into this process in order to end-up with two signatures with the same nonces. In order to end-up with this result within the provided architecture, first, a signature can be asked. Then a second signature is requested and during operation B in figure 4.9 where the 256 bits nonce k is generated from the RNG and saved into the $SRAM$, one can try to fault the writing addresses or the read/write signal. If the 8 faults injection are successful (recall, 8 writing are performed

for updating the 256 bits nonce), then the new nonce k will be written in another location, unrelated to the nonce or not written at all. As the previous nonce is in the memory, then the new signature will be computed with the same nonce that the previous one. It is to be noted that during operation B, the $SRAM$ contains only the hash result. Meaning that the nonce addresses can be faulted and transformed into almost any other addresses without corrupting important data for the process. If the hash is corrupted, the signature will be invalid and thus can be rejected by the attackers. It is also to be noted that, if this kind of faults are injected at the first signature then the nonce used will be the initialization value of the $SRAM$. The success of this attack can be detected by having two signature with the same r value.

Signature nonces with mostly shared bits

Signatures nonces with mostly shared bits ($k_1 = k_2 + X \cdot 2^l$ with X small and l known) also allow recovering the private key. Two signature equations allow to write the following:

$$\begin{cases} s_1 \equiv k_1^{-1}(H(m_1) + dr_1) \mod n \\ s_2 \equiv k_2^{-1}(H(m_2) + dr_2) \mod n \end{cases}$$

If $k_2 = k_1 \pm X \cdot 2^l$:

$$\Leftrightarrow \begin{cases} k_2 \equiv s_1^{-1}(H(m_1) + dr_1) \pm X \cdot 2^l \mod n \\ d \equiv (s_2 H(m_1) - s_1 H(m_2) \pm s_1 s_2 X \cdot 2^l) \cdot ((r_2 s_1 - r_1 s_2))^{-1} \mod n \end{cases} \tag{4.4}$$

Thus, if X is small enough, attackers can try all possible values and then guess the private key d. As in the previous case, attackers may try to interfere with the nonce update. The difference is that less successful fault injections are needed. Indeed, in the given architecture, each faulted addresses or read/write alterations during operation B allows avoiding the update of 32 consecutive bits. In the previous case, 8 successful faults were needed. In this new case, 6 or 7 successful faults injection allow obtaining two signatures with either 192 or 224 shared bits. Thus, either 64 or 32 bits should be brutforced. It is interesting to notice that the fault injection that should be performed is exactly as in the previous case. The difference is only the number of successful fault and the computation power required to recover the private key. The previous case is in fact just a special case where X equals 0.

Nonces with some known or shared bits

This case follow up the two previous one. In this case, only a few bits of the random nonce are equal.

$$\begin{cases} k_1 = X \cdot 2^l + Rdm_1 \\ k_2 = X \cdot 2^l + Rdm_2 \\ ...\ k_j = X \cdot 2^l + Rdm_j \end{cases}$$

As previously, this can happen in the studied architecture if the nonce update is compromised. A single fault during operation B on the address used or the read/write signal may lead to avoid a word update. As explained in section 3.3, if enough such signature are gathered, a lattice attack can be used with a brutforce on the shared bits. It is to be noted that, even if 32 bits or more are shared, the brutforce can be applied on fewer bits and attackers will need more signatures to succeed. Thus, a trade off between number of signature and computation power is possible.

Signature generations with faulted secret key

Section 4.2.1 details how a faulted private key used inside an $ECDSA$ signature can be used to recover information when it is slightly different from the genuine private key used during the key generation. In the described architecture, different possibilities may exist to inject such fault. The private key is stored encrypted in an NVM. In order to use it, first it is decrypted thanks to an AES block and then stored in the $SRAM$ during operation D. Faults injections and countermeasures in AES as discussed in [93] consider faults inside the block cipher. Hence, despite the fact the AES contains fault countermeasures at the block level, the interface between it and the $SRAM$ can eventually be faulted. In our case, considering section 4.2.1, attackers are interested to fault the output result of the AES prior to be used to generate the s part of the $ECDSA$ signature. The AES block internally uses a 8 bits datapath and the $SRAM$ interface uses 32 bits of data. Thus, in this case either the AES result is accumulated in hardware until 32 bits are ready to be written in the $SRAM$ or multiple read and write are used to end up with the full word written correctly in the memory. The fact that the AES uses an 8 bits datapath is interesting for attackers. Indeed, this may allow them to target 8 bits instead of 32. Figure 4.10 illustrate how bits can be accumulated to 32 bits prior an $SRAM$ writting.

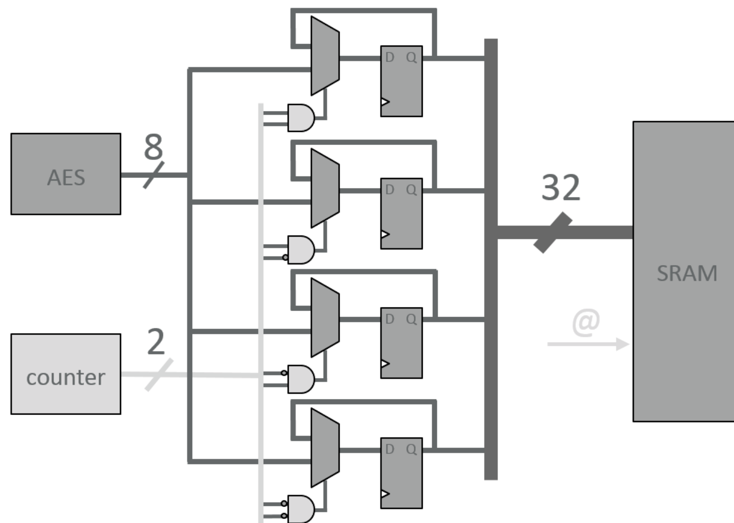

Figure 4.10: Illustration of an 8 bits to 32 bits interface

A counter starts from 0 and count until 3. For each new value of the counter, the AES provides 8 bits that are stored in an 8 bits register according to the counter. At the end, the 32 bits accumulated in the different registers are then written into the $SRAM$ at a specific address. Attackers can try to directly target the AES output, or the registers used for accumulation. It is also possible to fault the counter register, the address or the read/write signal. Due to this structure, various effect can be obtained such that faulting 8, 16, 24 or 32 bits of data, prevent word writing or shift the private key. In the case where multiple read and write are used in order to write the private key into the $SRAM$, the system is still vulnerable to faults. Table 4.2 bellow is an illustration of the normal behavior and what can be expected from attackers.

Faults occurring either on the data or on the read/write address can lead to fault $1-4$ bytes of data. It is interesting to notice that faults on the writing address allow attacker to target any specific byte of the word. Faults on the private key used during the generation of the s part of the signature can also happen in the $SRAM$ prior operation E or when reading the value during operation E. Indeed, as previously either the data read or the address used can be faulted, resulting in 32 faulted bits.

Signature generations with faulted nonces

As explained in section 4.2.2, if the r and s parts of the $ECDSA$ signature are generated with a slightly different nonce, then the difference can be recovered providing information on the nonce that can eventually be used inside a lattice attack (see section 2.6). From the architecture, provided in figure 4.9, this can happen almost

Table 4.2: AES bits accumulation through $SRAM$ read/write iteration

	Expected		Faulted data or read @		Faulted write @	
$Read1$	@	$0x0000$	@	$0x0000$	@	$0x0000$
$Write1$	@	$0xA_1000$	@	$0xA_1000$	@	$0xA_1000$
$Read2$	@	$0xA_1000$	@′	$0xF_1F_2F_3F_4$	@	$0xA_1000$
$Write2$	@	$0xA_1A_200$	@	$0xF_1A_200$	@′	$0xA_1A_200$
$Read3$	@	$0xA_1A_200$	@	$0xF_1A_200$	@	$0xA_1000$
$Write3$	@	$0xA_1A_2A_30$	@	$0xF_1A_2A_30$	@	$0xA_10A_30$
$Read4$	@	$0xA_1A_2A_30$	@	$0xF_1A_2A_30$	@	$0xA_10A_30$
$Write4$	@	$0xA_1A_2A_3A_4$	@	$0xF_1A_2A_3A_4$	@	$0xA_10A_3A_4$

@: address, A_i: i^{th} byte of AES result, F_i: Faulted byte

anytime between operations C and E. Indeed, the nonce read during operation C can be faulted either directly on the data bus or by faulting the read address. Similarly, the same can happen during operation E. A fault can also occur in the $SRAM$ between operations C and E. As during operation C the $ECDSA$ block read the nonce from the $SRAM$ for each bit evaluation (i.e. for each loop iteration of the previously described elliptic curve scalar algorithm), attackers can target a few bits of the nonce with faults. Indeed, even if 32 bits of the nonce are faulted during the nonce reading of operation C, only the bits required by the elliptic curve scalar algorithm are used. Thus, the fault propagate to the result only on those bits.

4.3 Summary

We demonstrated that some elliptic curve scalar algorithms are wrongly supposed to be safe-error resistant due to either the algorithm or to underlying computations. The Montgomery ladder and the coherency checking countermeasures are two examples. The problem of the Montgomery ladder is that, depending on the scalar value, operations may become useless to the computation of the correct result. Thus by injecting a fault, attackers can detect the particular value of the scalar. While it cannot be expected to fully recover the EC scalar value with this method, it allows to get couple of bits. The coherency checking algorithm faces another problem that is due to the scalar dependent update of a working register alongside the complexity of EC operation that allows fault injection opportunities. By using a memory safe-error attack on an EC input point of the point addition algorithm after it is used inside the computation, attackers know if the register update corrects the fault or not and thus can obtain the scalar bit value. The concept of dummy operand was introduced due, for example, to the infinity

point that can be used when this specific point is considered and manipulated as a normal point (E.g. with Edward curves). We demonstrated that even if in the $ECDSA$ signature, the scalar represents a nonce that is refreshed for each signature, safe-error attacks are enough to recover the private key due to the fact that obtaining only a couple of bits per signature allows mathematical attacks.

The EC scalar operation represents only the r part of $ECDSA$ signature. The s part of the signature can also be targeted. We thus showed that $ECDSA$ signatures generated from faulted private key can be used to recursively recover the key bits. We demonstrated that a slightly faulted private key used during the computation of the s part of the signature may generate a small error that can be recovered from the faulted signature. Thanks to the knowledge of how the private key is represented in registers and the error, information about the key can be obtained. Indeed, the possible error range generated by a register of a given size depends on the size and the genuine value. By recursively generating signature with faulted private key, it is possible to recover the error range and with the knowledge of the register size and how the secret is represented, the private key can be fully recovered. A similar method was also applied on nonces. While it is not possible to recursively attack a given nonce, signature can be selected according to the obtained error due to the fault. This allows to provide enough bits of information to be used within lattice attacks and then to recover the private key. We also demonstrated that the error distribution allows attackers to understand the behavior of the injected fault and then to greatly improve the attack speed.

A basic architecture of an $ECDSA$ system was described allowing to compute signatures. This allowed to understand the wide range of fault injection possibilities and opportunities for attackers and to understand how realistic the described fault injection attacks can be. This also demonstrated that countermeasures in the low level functional blocks are mandatory but unfortunately are not enough to ensure the security regarding fault injection.

Safe-error attacks against the EC scalar algorithm and consequences are underestimated in the case of $ECDSA$ computations and other fault injections may allow recovering information in other parts of the $ECDSA$. Similarly to side channel, while fully recover secret information with fault injection is not obvious, it seems easy to recover at least partial information enabling mathematical attacks and then to recover the private key of $ECDSA$ systems. Countermeasures against both side channel and fault attacks are thus mandatory and should protect all the secret bits. In the next chapter, countermeasures attempting to cover all these threats are presented and discussed.

CHAPTER 5

Side Channel and Fault Countermeasures

In order to speedup or ease the computation, some elliptic curve scalar point multiplication algorithms require the scalar in a non-conventional representation. In the previous chapter, algorithm 4.5 is one example of such practice. The Non-Adjacent Form (NAF) and the Join Sparse Form [29] are also often used. As the secret scalar may be used elsewhere in the system, this ends up with different representation, manipulation and use case of a same secret. This may extend the attack surface and introduce new security risks. For example, from [37], the implementation of $ECDSA$ signature requires the elliptic curve scalar operation $k \cdot P$ with a random k for the r part of the signature. It also requires the computation of $k^{-1}(H(msg) + d \cdot r) \mod n$ with the same k for the s part. If the scalar $k \cdot P$ is computed with algorithm 4.5, it is highly possible that the system uses a conversion function $\phi(k) = k'$ to convert the random nonce k from a binary representation to k', the signed representation required in the algorithm. In this case, it is necessary to design such a function with inherent side channel and fault countermeasures. Indeed, the carry attack against the scalar blinding presented in [94] is a good demonstration that any scalar manipulation can leak information through side channel. While the scalar blinding aims at avoiding leakage during the EC scalar computation, Fouque et al. demonstrated that a simple scalar addition with a random value can put the scalar at risk due to the carry propagation between words. Indeed, they show that the carry can be detected thanks to side channel and then after many experiment, the number of time the carry is propagated depends on the secret value and thus can be recovered. The fault countermeasure is also important in order to guarantee that k and k' represent the same value independently of the representation

otherwise the system may be vulnerable to a lattice-based fault attack as explained in chapter 4. As example, if k' is used to compute r, and k to compute s, and if, due to a fault, a bit flip such that k' represents the value $k \oplus 2^i$ then the signature (r, s) generated by the system will be invalid. As demonstrated in chapter 4, in this case an attacker can easily recover the injected fault and then deduce the value of the flipped bit of k. If a larger fault (i.e on 8 bits) is injected, more bits can be recovered and then used inside a lattice attack to recover the private key. Section 4.2.2 demonstrates some attack possibilities when attackers have the opportunity to fault nonces.

In this following chapter, we detail algorithms, improvement and implementation strategies that aim to obtain an $ECDSA$ signature implementation secure against the previously described attacks. First in section 5.1, new elliptic curve algorithms are described that allow protecting the computation against both side channel and safe-error while reducing the security concern in case of partial leakages while computing the r part of the signature. Section 5.2 discusses about the implementation of such algorithms. It presents a selection of parameter allowing merging most parts of algorithms and also presents improved way of using such algorithms to, as example, efficiently compute the EC scalar operation with a blinded scalar. Then in section 5.3, approaches to protect both the $ECDSA$ private key and the nonce while computing the s part of signatures are described. The nonce countermeasure also allows to strengthen the elliptic curve scalar algorithm against fault attack by providing a Control Flow Integrity (CFI). This CFI aims at ensuring that the operation flow is correct and not modified by a fault and also provides some detection capabilities of data modification. Finally, the various algorithms and countermeasures performances, cost, and countermeasures are summarized in section 5.4.

5.1 Our secure scalar point multiplication

If we consider current attacks and the aforementioned problems, from a security point of view a secure scalar point multiplication algorithm should use an operation flow independent of the scalar. Indeed such a deterministic flow can easily be online checked with a control flow integrity (CFI) system. It should also avoid the use of any dummy operation or operand, even local as this can be detected. The infinity point should not be used and no scalar bit should be given away. The algorithm should also avoid using specific representation of the scalar that are different than required in other computation involving the scalar. From both a performance and design cost point of view, the algorithm should use a minimum number of working registers and allows acceleration through pre-calculation. Plenty of elliptic curve scalar multiplication algorithms already

exist, however, as far as we now, no published algorithms meet all these requirements. In this section we describe a solution that meets all these criteria.

5.1.1 Scalar operation

Our proposition presented in algorithm 5.1 is based on the classical double-and-add algorithm and aims at correcting the different security issues.

Algorithm 5.1 Scalar operation

Input: $E(F_q), k = (k_{t-1}, ..., k_1, k_0)_2, P \in E(F_q)$
Output: $2k.P$

1: $Q \leftarrow P$
2: **for** $i = t - 1$ to 0 **do**
3: $\quad Q \leftarrow 2.Q$
4: $\quad Q \leftarrow Q + (-1)^{\overline{k_i}}.P$ //add or subtract P, depending on k_i
5: **end for**
6: $Q \leftarrow Q - P$
7: **return** (Q)

The idea behind our algorithm is to first initialize Q to a point $E \in E(F_q)$ instead of the infinity point as in the standard double-and-add algorithm. This will be transformed to $2E$ during the loop due to the point doubling. In order to maintain the point E over the loop indexes and reject the doubling effect, E should be subtracted from Q. This leads to two possibilities, if $k_i = 1$, $Q \leftarrow Q + (P - E)$ otherwise, $Q \leftarrow Q - E$. At the end of the loop, E is subtracted from Q in order to remove the initialization. By carefully selecting $E = \frac{P}{2}$, the two solutions become, if $k_i = 1$, $Q \leftarrow Q + \frac{P}{2}$ otherwise, $Q \leftarrow Q - \frac{P}{2}$. By using P instead of $\frac{P}{2}$, this leads to algorithm 5.1 that compute $2k \cdot P$. This proposed algorithm removes most problems encountered in the basic double-and-add. It ensures that no infinity point is used without forcing any bit of the scalar. The operation flow is homogeneous and branches condition are removed. It is to be noted that algorithm 5.1 avoids dummy operations that can be detected through safe-error attacks, does not force any specific treatment on any bits and also minimizes the number of required registers to prevent faults on unused memory values. Thanks to the use of either $+P$ or $-P$, the algorithm is also immune to Address bit DPA ($ADPA$) [95] as the addresses flow is constant and independent of the scalar. The number of working registers is the same than a standard double-and-right, left-to-right algorithm.

The data dependent leakage is however not removed as operands values depends on the scalar. It is also to be noted that algorithm 5.1 adds a new horizontal side channel threat. Indeed, at the beginning and until a scalar bit equal to one is encountered; the computation of $Q \leftarrow Q - P$ nullifies the double operation. Thus, the following elliptic curve double operation will double the same point as the previous operation. This will helps attackers to detect the scalar length through collision correlation analysis [96] that allow detecting the number of time the loop performs the same operations. However existing solutions can be used and are discussed in Section 5.2.

If the platform does not provide indistinguishable field addition/subtraction, the elliptic curve point addition/subtraction can be computed with algorithm 5.2.

This algorithm is similar to standard ECC mixed Affine/Jacobian point addition excepted from lines 4 to 9. These lines compute the intermediate result for both ECC point addition and ECC point subtraction and save them randomly in $T_3[0]$ and $T_3[1]$ depending on r. Line 6 selects which intermediate result is to be used regarding the asked operation and the random bit r. This randomness aims at avoiding fault injection inside the memory (M safe-error) as attackers will not know which value $-T_2$ or T_2 is faulted. Lines 8 and 9 aim at preventing C safe-error attacks when $-T_2$ is computed.

Algorithm 5.1 improves the security level compared to the algorithm presented in [97] that uses the zeroless signed-digit expansion. In [97], the authors note that any group of w bits 00...01 can be replaced with a group of w signed digits $1\bar{1}\bar{1}...\bar{1}$ with $\bar{1} = -1$. This remark leads to a zeroless signed representation of the scalar k. An on the fly conversion is possible by initializing Q to the base point P and then by performing a for loop for all bits (except the LSB) that contains a point double $Q \leftarrow 2Q$ and two

Algorithm 5.2 ECC Jacobian-affine point addition/subtraction

Input: $P = (X_1 : Y_1 : Z_1)$ in Jacobian and $Q = (x_2, y_2)$ in affine $\in E(F_q)$, b operation selection, r a random bit

Output: if $b = 0$, $P - Q$ else $P + Q$

1: $T_1 \leftarrow Z_1^2$
2: $T_2 \leftarrow T_1 \cdot Z_1$
3: $T_1 \leftarrow T_1 \cdot x_2$
4: $\boldsymbol{T_3[r] \leftarrow -T_2}$
5: $\boldsymbol{T_3[\overline{r}] \leftarrow T_2}$
6: $\boldsymbol{T_2 \leftarrow T_3[b \oplus r] \cdot y_2}$
7: $\boldsymbol{T_1 \leftarrow T_1 - X_1}$
8: $\boldsymbol{T_2 \leftarrow T_2 + T_3[0]}$
9: $\boldsymbol{T_2 \leftarrow T_2 + T_3[1]}$
10: $T_2 \leftarrow T_2 - Y_1$
11: **if** $T_1 == 0$ **then**
12: **return** (error)
13: **end if**
14: $Z_3 \leftarrow Z_1 \cdot T_1$
15: $T_3[0] \leftarrow T_1^2$
16: $T_3[1] \leftarrow T_3[0] \cdot T_1$
17: $T_3[0] \leftarrow T_3[0] \cdot X_1$
18: $T_1 \leftarrow T_3[0] + T_3[0]$
19: $X_3 \leftarrow T_2^2$
20: $X_3 \leftarrow X_3 - T_1$
21: $X_3 \leftarrow X_3 - T_3[1]$
22: $T_3[0] \leftarrow T_3[0] - X_3$
23: $T_3[0] \leftarrow T_3[0] \cdot T_2$
24: $T_3[1] \leftarrow T_3[1] \cdot Y_1$
25: $Y_3 \leftarrow T_3[0] - T_3[1]$

possibilities, $Q \leftarrow Q + P$ if $k_i = 1$ or $Q \leftarrow Q - P$ if $k_i = 0$. Nevertheless, the zeroless signed representation works only for an odd scalar k thus forcing the LSB to one. Another downside of [97] is that an on the fly conversion with pre-calculated points seems harder to implement. Also, as mentioned in Chapter 5, recoding the scalar value in a signed representation prior using the elliptic curve scalar operation may extends attack possibilities. Thus, our algorithm 5.1 improves the security as no scalar extra manipulation is required and no scalar bit is constrained to any value while allowing performance/cost flexibility thanks to pre-calculation possibilities as presented in the next section. Compared to algorithm 4.4, algorithm 5.1 losts some randomness provided by the random point. However, this allows preventing M safe-errors and $ADPA$ as all registers are used during each loop independently of the secret. Randomness can be recovered as described in Section 5.2 without effecting the M-safe resistance property of the algorithm.

5.1.2 Scalar operation with pre-calculation

When performance is needed, point precalculation is usually used. The number of pre-computed points can be used to determine a trade-off between implementation cost and performance. In this section we present how our approach can be applied to the fixed-base comb method. This leads to similar performance with half of the required pre-calculated points. Compared to [98], in our case the scalar k does not need any modified representation simplifying the implementation. In order to achieve that, the same reasoning than algorithm 5.1 is applied. The working register Q is initialized to a carefully selected point instead of the neutral element. The introduced error is partially corrected over the for loop and then fully removed after the loop. By carefully selecting the initialization point, the number of pre-calculated points is divided by two and the neutral element is removed from the list. Algorithm 5.3 describes this approach. Each pre-calculated point is used either in an elliptic curve point addition or subtraction. This leads to a better performances/cost ratio if pre-computation is allowed in the system. It is to be noted that the initialization point is one of the pre-calculated ones and the neutral element is never used. As described, algorithm 5.3 needs to pre-compute 2^{w-1} points and uses $\lceil \frac{t}{w} \rceil$ elliptic curve point double operations and $\lceil \frac{t}{w} \rceil + 1$ elliptic curve point addition operations.

The 'represent k as:' does not necessarily mean that the scalar representation change in the register. Indeed, an hardware implementation can easily use the standard binary representation of k and evaluate the required bit without transforming the scalar.

Algorithm 5.3 Modified Comb method

Input: $E(F_q), k = (k_{t-1}, ..., k_1, k_0)_2, P \in E(F_q)$, window width $w, d = \lceil \frac{t}{w} \rceil$

Output: $2k.P$

Pre-computation: compute $2 \cdot [1, a_{w-2}, ...a_1, a_0].P - [1_{w-1}, ...1_1, 1_0].P$ for all possible binary values of $a_{w-2}, ...a_1, a_0$, with $[1, a_{w-2}, ...a_1, a_0].P = 2^{(w-1)d}P + ... + a_1 2^d P + a_0 P$.

Represent k as: $\begin{pmatrix} k_{d-1}^0 & \cdots & k_1^0 & k_0^0 \\ \vdots & \ddots & \ddots & \vdots \\ k_{d-1}^{w-1} & \cdots & k_1^{w-1} & k_0^{w-1} \end{pmatrix}$ //if necessary, pad 0s as k MSBs.

1: $Q \leftarrow [1_{w-1}, ...1_1, 1_0].P$ //represents the highest pre-calculated point.
2: **for** $i = d-1$ to 0 **do**
3: $\quad Q \leftarrow 2.Q$
4: $\quad Q \leftarrow Q + (-1)^{\overline{k_i^{w-1}}}.\overline{[[k_i^{w-1}, ..k_i^{w-1}] \oplus [k_i^{w-2}, ...k_i^1, k_i^0]]}.P$
5: **end for**
6: $Q \leftarrow Q - [1_{w-1}, ...1_1, 1_0].P$
7: **return** (Q)

5.1.3 The non-standard $2k \cdot P$ result and exceptional cases

Standard ECC schemes usually require the computation of $k \cdot P$, instead of $2k \cdot P$ as presented algorithms. While elliptic curve point halving is possible, it requires to implement another specific operation and thus it is not efficient in term of required memory. Different simple other solutions exist. First, the less recommended solution would be to compute $c = k \cdot 2^{-1} \mod n$ prior using our algorithms with c. This solution is not the best one as it extends the number of manipulation of the secret k thus extending the attack surface (k is a secret but also c in this case). Another solution would be to save $\frac{P}{2}$ in the system instead of the fixed base point P during the system development. However, this solution cannot be applied if algorithms are intended to be used with an elliptic curve point coming from outside the system (e.g. a Diffie-Hellman). Finally, a last solution could be to consider during the secret generation that $k \cdot 2^{-1} \mod n$ is generated and saved in memory instead of k and taken into account if needed. For example, during an $ECDSA$ signature, c can be generated using a random number generator. Then for the r part of the signature, by using c with our algorithms, $k \cdot P$ will be returned. The s part can be changed into: $s = c^{-1}(H(m) + d \cdot r) \cdot 2 \mod n$ leading to the good signature. This solution seems to be the best as it can be applied to most ECC schemes without extending the attack surface nor requiring special computation such as point halving. It is also to be noted that few exceptional cases exist within our algorithms depending on the scalar. Indeed, per example, if n represents the ECC

point order used within algorithm 5.1, then if $k = 0$, $k = n - 1$, $k = n - 2$, $k = n - 3$, $k = \frac{n-1}{2}$ and $k = \frac{n-1}{2} - 1$, special computation such as $P + P$, $P - P$, $-P + P$, $-P - P$ or $2 \cdot \infty$ will happen in the EC point addition formula leading to special power consumption or fault exposure as explained in Section 4.1.3. However, the probability to encounter these cases is really low (around $\frac{5}{2^{256}}$ for $NIST$ P256) and thus attacker cannot expect to observe such behavior.

5.2 Secure implementation strategy

Usually, systems requiring ECC require the implementation of different schemes such as $ECDSA$, $ECDH$ or $ECIES$. These schemes face up different threat models as they use the elliptic scalar operation in different scenario. In an $ECDH$, the scalar is a private key while the base point is a given public key. Thus chosen point attacks such as RPA[87]/ZPA[86] or attacks requiring multiple executions with the same scalar such as DPA can be used to attack the device and fully recover the secret. However, during an $ECDSA$ signature, the elliptic scalar operation is used with a refreshed nonce and the curve base point. Thus, these attacks cannot be used. Nevertheless, due to Lattice attack a partial leakage of the scalar is enough to recover the secret [88]. The work presented in [82] surveys different active and passive attacks, their prerequisites and countermeasures. Resource-restricted systems such as embedded system or smart-card need clever implementations in order to provide both functionalities and security with acceptable performance. In this section we present an efficient implementation strategy based on previous algorithms that can be used in different scenarios. Three use cases are considered. Firstly the elliptic curve scalar operation $k \cdot P$ with P the curve base point which can be hardcoded. Secondly the sum of two elliptic curve scalar operation $k \cdot P + v \cdot G$ with two different points P and G that can be different from the curve base point. Finally, these algorithms will lead to a security enhanced elliptic curve scalar operation $k \cdot G$ that considers any point G. It is to be noted that the different cases differ only from slight algorithmic modifications. Thus, a single implementation can be used over these use-cases in order to save memory or silicon as only the input and configuration will change.

5.2.1 Scalar with a fixed base point: $k \cdot P$

The $k \cdot P$ operation can be implemented using ECC point precomputation techniques as described in algorithm 5.3. In algorithm 5.4, we use a window width parameter of two, allowing to speed-up the computation with a factor of $\times 2$ compared to computation without the precomputed points. It requires only two ECC points on the curve: $2^{t/2}P + P$ and $2^{t/2}P - P$.

In this algorithm, $RandomAfftoJac()$ represents the common affine to Jacobian random representation conversion and provides countermeasure against data dependent leakage. $Shuffleregisters()$ is a function that randomly reassign $P[0]$ and $P[1]$ in order to avoid address dependent leakages. $Verify(Q)$ ensure that the input point is on the curve. Lines 12-14 aim to involve $P[r]$ and $P[\overline{r}]$ in the result prior verification for integrity check. Without these two operations, an attacker can fault $P[r]$ prior the last l bits of the scalar and use the consequences to detect l consecutive 1 inside the LSBs (M-safe-error attacks). This countermeasure does not consider transient faults as only the value at the end is verified.

Algorithm 5.4 $2kP$ operation optimized with two precomputed points

Input: $k = (k_{t-1}, ..., k_1, k_0)_2$, P and $2^{t/2}P \in E(F_q)$
Output: $2k.P$

1: $r \leftarrow$ *randombit()*
2: $P[r] \leftarrow 2^{t/2}P - P$ //in affine coordinates
3: $P[\overline{r}] \leftarrow 2^{t/2}P + P$ //in affine coordinates
4: $Q \leftarrow$ *RandomAfftoJac*($P[1]$) //use random affine to Jacobian conversion
5: *Verify(Q)*
6: **for** $i = t/2 - 1$ to 0 **do**
7: $Q \leftarrow 2Q$
8: $Q \leftarrow Q + (-1)^{\overline{k_{i+t/2}}} P[r \oplus \overline{(k_i \oplus k_{i+t/2})}]$
9: $r \leftarrow$ *randombit()* //refresh the random r
10: *shuffleregisters*($P[0]$,$P[1]$,r) //shuffle P[0] and P[1] according to r
11: **end for**
12: $Q \leftarrow Q + P[r]$ //add $P[r]$ for system integrity
13: $Q \leftarrow Q - P[\overline{r}]$
14: $Q \leftarrow Q - P[r]$ //remove $P[r]$
15: $Q \leftarrow$ *JactoAff(Q)* //use Jacobian to affine conversion
16: *Verify(Q)*
17: **return** Q

This algorithm requires $t/2 + 5$ ECC point additions and $t/2$ ECC point doubling operations while requiring 2 precomputed points. In a similar configuration, the classical comb method would require 3 ECC points to perform in $t/2$ ECC point doubling and ECC point additions. As the base point is fixed, precomputation cost is neglected.

5.2.2 Sum of two scalars: $k \cdot P + v \cdot G$

$ECDSA$ verifications for example require the computation of $k \cdot P + v \cdot G$ with always the same known ECC point P and G another ECC point representing a public key. Usually precomputation is not convenient to use in this case as the point G is not predictable. The best known solution to speed-up the computation is Shamir's trick [29] that aims to simultaneously compute both scalar. In our implementation, we use Shamir's trick combined with our previous presented solution. The result is described by algorithm 5.5 and allows a speed-up with a factor of $\times 2$ compared to two distinct classical ECC scalar operations. During the for loop, two bits of scalars are considered, one from k and another from v ending-up with four different cases. With the classic Shamir's trick, either $Q + \infty$, $Q + P$, $Q + G$ or $Q + (P + G)$ are considered. In our case, the four different cases become either $Q - (P + Q)$, $Q - (G - P)$, $Q + (G - P)$ or $Q + (G + P)$.

Algorithm 5.5 $2k \cdot P + 2v \cdot G$ operation

Input: $k = (k_{t-1}, ..., k_1, k_0)_2$,$v = (v_{t-1}, ..., v_1, v_0)_2$, P and $G \in E(F_q)$
Output: $2kP + 2vG$

1: $r \leftarrow$ *randombit()*
2: $P[r] \leftarrow G - P$ //in Affine coordinates
3: $P[\overline{r}] \leftarrow G + P$ //in Affine coordinates
4: $Q \leftarrow$ *RandomAfftoJac*($P[1]$) //use random affine to Jacobian conversion
5: *Verify(Q)*
6: **for** $i = t - 1$ to 0 **do**
7: $Q \leftarrow 2Q$
8: $Q \leftarrow Q + (-1)^{\overline{v_i}} P[r \oplus \overline{(k_i \oplus v_i)}]$
9: $r \leftarrow$ *randombit()* //refresh the random r
10: *shuffleregisters*($P[0]$,$P[1]$,r) //shuffle P[0] and P[1] according to r
11: **end for**
12: $Q \leftarrow Q + P[r]$ //add $P[r]$ for system integrity
13: $Q \leftarrow Q - P[\overline{r}]$
14: $Q \leftarrow Q - P[r]$ //remove $P[r]$
15: $Q \leftarrow$ *JactoAff(Q)* //use Jacobian to affine conversion
16: *Verify(Q)*
17: **return** Q

This algorithm does not aim to improve the security level of the implementation

as all manipulated values are public. Instead, it aims to end-up with a very similar implementation than the $k \cdot P$ computation. The descriptions of algorithm 5.4 and algorithm 5.5 are almost identical. The differences are the initialization of $P[r]$ and $P[\overline{r}]$, the loop length, the bits used to select the ECC point addition/subtraction and the precomputed value to use. By doing so, implementation cost can be reduced as a parameter can be used to configure the algorithm depending on the requested operation. This algorithm requires $t + 5$ ECC point additions and t ECC point doubling. The classic Shamir's trick would get slightly better performances, in average t ECC point doubling and $0.75t$ ECC point addition. While $ECDSA$ verification does not necessary requires side channel countermeasures as all parameters are public, this algorithm may be used in other situations.

5.2.3 Scalar with any base point: $k \cdot G$

The $k \cdot G$ operation is required for example in a Diffie-Hellman key agreements. It differs from the $k \cdot P$ operation as the ECC point used is not predictable meaning that usual precomputation techniques cannot be used efficiently. Another difference is from a security perspective as this operation is vulnerable to chosen point attacks. For instance, RPA[87]/ZPA[86] can defeat the random affine to Jacobian point representation conversion. These attacks rely on the fact that some special points such as $(0, Y, Z)$, $(X, 0, Z)$ remains in the same form whatever the value of Z and thus the random number used during the conversion. In this attack scenario, attackers input a chosen value Q that will be transformed to the special point at a targeted stage depending on the scalar value. E.g. in algorithm 5.4, if P is chosen such that $(2^{t/2}+1)P = (0, Y, Z)/3$ then if $k_{t-1} = k_{t/2-1} = 1$, the value $(0, Y, Z)$ will appear in the system at the end of the first loop iteration. This can be detected through a CPA by attackers allowing them to recover two bits. By iteratively performing the attack, attackers end-up with all the scalar value. To prevent this attack, we can use the $k \cdot P + v \cdot G$ algorithm as presented previously with a random ECC point G, $v = 0$ and $P = Q$.

From an arithmetic point of view, $k \cdot P + 0 \cdot G$ is equal to $k \cdot P$, however inside Algorithm 5.5, computations are based on $G - P$ and $G + P$. Thus with a random G, the algorithm computes the good result from random points. Attackers that input their own special point P will not be able to predict the values of $G - P$ and $G + P$. With this use of algorithm 5.5, the scalars represent secret values and thus countermeasures are mandatory. In the given example, the ECC point G is based on a 32bits random number that can be computed similarly than $k \cdot P$ with a reduced loop in order to reduce the effect on the overall performances. An even more interesting use of algorithm 5.5 to compute $k \cdot Q$ is with scalar splitting such that $k \cdot P = k_1 P + k_2(r \cdot P)$. This is

described in algorithm 5.6.

Algorithm 5.6 kG operation computed as $2k \cdot G = k_1G + k_2(r \cdot G)$

Input: $k = (k_{t-1}, ..., k_1, k_0)_2$, $G \in E(F_q)$

Output: $2k \cdot G$

1: $r \leftarrow random([0, 2^{32} - 1])$
2: $v \leftarrow random([0, \#E(F_q) - 1])$
3: $k \leftarrow k - v$
4: $v \leftarrow v \cdot (2r)^{-1} \mod n$
5: $Q \leftarrow AlgKP(r, G)$ //Algorithm 5.4 reduced for 32bits of scalar
6: $R \leftarrow AlgkPvG(k, G, v, Q)$ //Algorithm 5.5 kP+vQ
7: **return** R

Indeed, algorithm 5.5 security relies on indistinguishable point addition or subtraction and also indistinguishable use of $P[0]$ or $P[1]$ during each loop iteration. By using scalar splitting, attackers will be forced to look for both information at the same time. In algorithm 5.6, $2 \times t + 32$ bits of secret are used instead of t. If an attacker is able to distinguish elliptic curve point addition from point doubling, he obtains at most t bits of secret meaning that $t + 32$ bits remain. Similarly, if an attacker is able to recover which precomputed point is used, he will obtain at most $t + 32$ bits and thus t bits will remain. The use of r aims to remove the probability dependence between bits that exist and can be used in the simple additive splitting as explained in [99]. The reduction to a 32 bit loop should be carefully implemented by the designer. Indeed, if using an input parameter to configure the loop size, he has to ensure that this entry cannot be faulted in order to reduce the computation of a full size scalar to only 32 bits. Nevertheless, a CFI should easily be able to detect any iteration errors within the loop.

This algorithm requires $t + 26$ ECC point additions and $t + 16$ ECC point doubling operations. The classic always double-and-add algorithm would be faster as only t ECC point doubling and additions would be required, however, algorithm 5.5 uses a fully masked scalar. Scalar blinding technique could also be used alongside the classic always double-and-add algorithm. Nevertheless, a too small random used with the scalar blinding may conduct to a partially masked scalar [100] due to the particular modulus value or to bias the probability as described in [101] or also to face doubling attacks due to the birthday paradox [102]. From [100], a $NIST$ P256 implementation would require a random on 64 bits leading to $t + 64$ ECC point doubling and additions. From this perspective,algorithm 5.5 is around 15% faster.

5.3 Preventing attacks against ECDSA nonce and private key

The elliptic curve scalar operation is a complex operation that needs an extremely careful attention. However, this is not the only critical operation of an $ECDSA$. This scalar operation is used to generate the public key Q from the private one d and also to generate the r part of the signature by using the nonce. As previously explained in section 4.2, the s part of the $ECDSA$ signature can also be targeted as it involves both the private key d and the nonce k. In the following, we provide methods on how to protect the private key and the nonce against fault and side channel while computing the s part of the signature.

5.3.1 ECDSA signature, s part private key countermeasures

The $ECDSA$ private key d is involved in the s part of the signature with the following formula:
- $s \equiv k^{-1}(H(msg) + d \cdot r) \mod n$

Section 4.2.1 details a possible fault attack against the private key that aims to force the system to compute the s part of the $ECDSA$ signature with a slightly different key than the one used for the public key generation. The attack success depends on the attacker capability of injecting such fault and to recover the resulting error. This depends on the system architecture and implementation choices. In order to counteract this attack path, in the following suggestion, the private key d is no-longer manipulated as a single binary represented number. Instead of being generated from a single random number, the private key d is generated from two random shares (m_1 and m_2) that should be multiplied ($\mod n$) to get the good result d. By doing so, the system never manipulates d.

Suggestion of key generation to protect d:
- Select and store a random m_1 such that $0 \leq m_1 \leq n - 1$
- Select a random r_1 such that $0 \leq r_1 \leq n - 1$
- Select a random r_2 such that $0 \leq r_2 \leq n - 1$
- Compute $k = r_1 \cdot m_1 \mod n$
- Select a random r_3 such that $0 \leq r_3 \leq 2^{32}$
- Compute $G = 2r_3.P$ with $P \in E(F_q)$ //Algorithm 5.4 reduced for 32bits of scalar
- Compute $v = (r_2 \cdot (2r_3)^{-1}) \cdot m_1 \mod n$
- Compute and save $Q = 2k \cdot P + 2v \cdot G$ with P and $G \in E(F_q)$ //Algorithm 5.5
- Compute and save $m_2 = r_1 + r_2 \mod n$

- $d = 2 \cdot (m_1 \cdot m_2) \mod n$ is the private key, Q is the public one.

Suggestion of signature to protect d:
- Generate a nonce k such that $0 \leq k \leq n-1$
- Compute $G = (x, y) = k.P$ with $P \in E(F_q)$
- $r \equiv x \mod n$
- $s \equiv k^{-1}(H(msg) + 2m_1 \cdot (r \cdot m_2)) \mod n$
- (r, s) is the signature.

The modular multiplication relationship between d, m_1 and m_2 prevents the attack. Indeed, if m_1 is faulted into $m_1' = m_1 \pm e$, the s part of the signature become:
$s \equiv k^{-1}(H(msg) + 2 \cdot m_1' \cdot (r \cdot m_2)) \mod n$
$s \equiv k^{-1}(H(msg) + 2 \cdot (m_1 \pm e) \cdot (r \cdot m_2)) \mod n$
$\Leftrightarrow s \equiv k^{-1}(H(msg) + (d \pm 2 \cdot e \cdot m_2) \cdot r) \mod n$

The equivalent error injected on the private key d is $\pm e \cdot m_2 \mod n$. As m_2 is a random number such as $0 \leq m_2 \leq n-1$ then $0 \leq e \cdot m_2 \mod n \leq n-1$ thus even if $\pm e$ is small, attackers will not be able to recover the error by exhaustively trying all possibilities. The same happen if m_2 is faulted. If for some reasons, attackers know the resulting value of the faulted register, then as long as m_2 is unknown it is still not possible to recover $\pm e \cdot m_2$ and thus deduct information about d. It is to be noted that if an attacker perfectly control the fault injection (i.e. is able to force a specific bit to 0 or 1 or fostering 0 to 1 transitions over 1 to 0 [103]) then he can get information with a safe-error principle and iteratively recover information. E.g. if the LSB of m_1 is forced to 1 and the signature is valid, then the genuine m_1 LSB is 1 otherwise it is 0. It is also to be noted that m_2 can be recovered from $r \cdot m_2$ thanks to a classic DPA or CPA as r is known and then m_1 recovered from $m_1 \cdot (r \cdot m_2)$. This side channel attack is similar to the classical one against the $ECDSA$ private key [104] and [105]. Moreover, if an operand scanning integer multiplication or similar is used to compute field multiplications, then attacks described in [92] can be used. These attacks need to be recursively used as they aim at faulting a partial product which, once propagated, allow to recover a single secret word according to the fault position. In order to counter both the fault attack and the side channel, the shares that represent the private key can be updated after each signature.

Suggestion of shares update:
- Select and store a random R such that $0 \leq R \leq n-1$
- $m_1 \leftarrow m_1/R \mod n$

- $m_2 \leftarrow m_2 \cdot R \mod n$

By doing this shares update, the representation of the private key d over the elements m_1 and m_2 changes over the time. This allows preventing iterative fault attacks and vertical side channel attacks against m_1 and m_2. From a side channel perspective, m_1 and m_2 act as masks. It is however to be noted that $m_1 \cdot (r \cdot m_2) = d \cdot r$, thus the result of $m_1 \cdot (r \cdot m_2)$ is not masked and equals the private key times a known value that change over signatures. The modular multiplication relationship between the secret d and the result $d \cdot r$ does not provide relationship between hypothesis of a subset of d and the result due to the modular reduction. Thus targeting this intermediate result with a DPA or CPA like attacks is not obvious. However, partial leakage of $d.r$ can be used inside a lattice attack in order to recover d as demonstrated in section 4.2.3. Thus, a good practice should be to maintain the private key masked until the end. To this end, another random number m_3 can be used to finally obtain the following $ECDSA$ signature generation.

Suggestion of signature to protect d with side channel countermeasure:
- Generate a nonce k such that $0 \leq k \leq n-1$
- Compute $G = (x, y) = k.P$ with $P \in E(F_q)$
- $r \equiv x \mod n$
- Select a random m_3 such that $0 \leq m_3 \leq n-1$
- $s \equiv (k \cdot m_3)^{-1}(H(msg) \cdot m_3 + 2m_1 \cdot ((r \cdot m_3) \cdot m_2)) \mod n$
- Update m_1 and m_2.
- (r, s) is the signature.

If instead of the modular multiplication used between shares to generate the private key, a modular addition is used (i.e. $d = m_1 + m_2$ and $s \equiv k^{-1}(H(msg) + m_1 \cdot r + m_2 \cdot r) \mod n$) then the fault $\pm e$ can be recovered as it is not modified by the random shares allowing exhaustive search. Indeed, if $m_1' = m_1 \pm e$, then:

$s \equiv k^{-1}(H(msg) + m_1' \cdot r + m_2 \cdot r) \mod n$

$\Leftrightarrow s \equiv k^{-1}(H(msg) + m_1 \pm e \cdot r - m_2 \cdot r) \mod n$

$\Leftrightarrow s \equiv k^{-1}(H(msg) + (d \pm e) \cdot r) \mod n$

Similarly to the case without countermeasures (i.e. section 4.2.1), this would provide information on the faulted register (m_1). E.g. If an 8 bits register containing 8 bits of m_1 is targeted by a fault that generates an error $e = 1$. Then the genuine register value is < 255 as 256 cannot be encoded within the register. This error provides information to the attacker. Multiple attacks against m_1 will allow attackers to fully get the value.

As nothing prevent to do similarly with m_2, the private key d can be recovered.

The modular multiplication used between the shares used to generate the private key thus represents a better solution. The computation overhead is negligible compared to the whole signature. Only four field multiplications and two random number generations are required to protect the private key during the computation of $ECDSA$ signatures.

5.3.2 ECDSA signature, s part nonce countermeasures

Countermeasure against signatures with faulted nonce

The nonce k is involved in the s part of the $ECDSA$ signature with the following formula:

- $s \equiv k^{-1}(H(msg) + d \cdot r) \mod n$

Section 4.2.2 details a possible fault attack against the nonce that aims to force the system to compute the s part of the $ECDSA$ signature with a slightly different nonce than the one used for the r part. The success of the attack depends on the attacker capability of injecting such fault and thus depends on the system architecture and implementation choices. In order to counteract this attack path, in the following suggestion, the nonce k is manipulated only one time thus preventing the nonce alteration between r and s computation.

Suggestion of signature to protect k from faults:

- Generate a nonce k such that $0 \leq k \leq n-1$
- Select and store a random m such that $0 \leq m \leq n-1$
- Compute simultaneously $G = (x, y) = k.P$ and $k_m = k \cdot m \mod n$ with $P \in E(F_q)$
- $r \equiv x \mod n$
- $s \equiv k_m^{-1}(H(msg) \cdot m + d \cdot (r \cdot m)) \mod n$

Where "Compute simultaneously $G = (x, y) = k.P$ and $k_m = k \cdot m$" means computing $k.P$ and $k \cdot m$ by evaluating the value of the scalar k only once for both operation. The elliptic curve scalar multiplication $k.P$ computes k times the base point P. Thus, a similar algorithm can be used to compute $k \cdot m$ with m in the field. Bits evaluation of the scalar k can then be combined to compute both $k.P$ and $k_m = k \cdot m$. By doing so, r and s will be computed with the same nonce thus preventing attackers to obtain information when trying to tamper the nonce. As an example, algorithm 5.7 can be used to compute $k \cdot m$. This algorithm is similar to algorithm 5.1 that was described in section 5.1 to compute the elliptic curve scalar operation.

Algorithm 5.7 Field scalar operation

Input: $k = (k_{t-1}, ..., k_1, k_0)_2, m \in F_q$

Output: $2k.m$

1: $q \leftarrow m$
2: **for** $i = t - 1$ to 0 **do**
3: $\quad q \leftarrow 2.q$
4: $\quad q \leftarrow q + (-1)^{\overline{k_i}}.m$ //add or subtract r, depending on k_i
5: **end for**
6: $q \leftarrow q - m$
7: **return** (q)

It is important to notice that the computation is not on an elliptic curve. Thus the computation time is negligible compared to algorithm 5.1. Indeed, algorithm 5.7 requires 2 basic field operations (addition, subtraction, shift...) per loop iteration. In the case of mixed affine-jacobian coordinates with a $NIST$ curve, algorithm 5.1 requires 12 field multiplications, 7 field squaring and around 16 basic field operations per loop iteration. However, while algorithm 5.1 can provide side channel countermeasures by using a random jacobian representation of the base point P, algorithm 5.7 does not. Thus algorithm 5.7 is, as presented, subject to horizontal attacks. Indeed, as long as $k_i = 0$ then lines 3 and 4 compute $q \leftarrow 2.m$ and $q \leftarrow 2.m - m$. The redundancy of these operations can be detected leading attackers to conclude about the scalar length. Another side channel threat is how $q \leftarrow q + (-1)^{\overline{k_i}}.m$ is computed. Indeed, either the modular addition/subtraction is computed and then should not be different from a side channel point of view, or a two indexes table $v[0] = -m$, $v[1] = m$ can be used. In this last case, the addresses of $v[0]$ and $v[1]$ and the values $-m$ and m will be constant over the execution. Attackers can thus use a horizontal attack to try to differentiate whether $v[0]$ and $-m$ or $v[1]$ and m are used. Moreover, as discussed in section 4.1, unused values can be detected with safe error attack.

In order to counter these threats, algorithm 5.7 is improved into algorithm 5.8. The random value u is multipurpose. First it allows countering horizontal side channel attacks. The value u randomizes all intermediate values and the indexes of the table v is randomized by using a random bit r. By doing so, both data and addresses will vary independently of the secret bit k_i. Secondly, it allows countering safe error attacks, as both indexes of the table v are used during each loop iteration (in line 9) for updating u. If either $v[0]$ or $v[1]$ are faulted, then u will be faulted conducting the result q to be faulted and then also the signature (r, s). Finally, the random u also acts as a CFI as it allows ensuring the correct execution of the algorithm. Indeed, u is involved in

all operations, evolves over the execution and is removed by using the product with a constant value X prior returning the result.

Algorithm 5.8 Side channel and fault resistant field scalar operation

Input: $k = (k_{t-1}, ..., k_1, k_0)_2, m, n \in F_q$
Output: $2k.m \mod n$
1: $q \leftarrow random([1, 2^t - 1])$
2: $u \leftarrow q - m \mod n$
3: $u \leftarrow 2u$
4: **for** $i = t - 1$ to 0 **do**
5: $\quad r \leftarrow randombit()$
6: $\quad u \leftarrow (2^{t-1} - 1) \cdot u \mod n$
7: $\quad v[\overline{r}] \leftarrow u + m \mod n$
8: $\quad q \leftarrow 2q \mod n$
9: $\quad v[r] \leftarrow u - m \mod n$
10: $\quad q \leftarrow q + v[\overline{(k_i \oplus r)}] \mod n$
11: $\quad u \leftarrow v[\overline{r}] + v[r] \mod n$
12: **end for**
13: $q \leftarrow q - m \mod n$
14: $q \leftarrow q - X \cdot u \mod n$ //$X = 2^{t-1} \cdot (1 - (2^{t-1} - 1)^{t+1})/((1 - (2^{t-1} - 1)) \cdot (2^t - 2)^t)$
15: **return** (q)

It is interesting to notice that even if a fault on $v[0]$ or $v[1]$ generates a different error on the result q depending on the value of k_i, this difference is not usable by attacker to conclude about k_i due to the random m. As an example, if $u + m$ is faulted into $u + m + e$ during the last loop iteration then if $k_0 = 0$ the final result will be $q' = 2 \cdot k \cdot m + X \cdot e$, otherwise, if $k_0 = 1$ then $q' = 2 \cdot k \cdot m + (X + 1) \cdot e$. Attackers will observe a signature (r, s') either with $s' \equiv (2k + X/m \cdot e)^{-1} \cdot (H(msg) + d \cdot r) \mod n$ or $s' \equiv (2k + (X + 1)/m \cdot e)^{-1} \cdot (H(msg) + d \cdot r) \mod n$. The equivalent error on the nonce $2k$ is thus either $X/m \cdot e$ or $(X + 1)/m \cdot e$, both depending on m which is on t bits. The error search space is thus to big to allow recovering the error on $2k$, independently of the initial fault that generates e.

Independent execution of algorithm 5.1 and algorithm 5.8 does not provide evidence that the same scalar is used in both cases as it can be tampered between executions. In order to do so, they should be merged into a single algorithm. Algorithm 5.9 describes the convergence of algorithm 5.1 and algorithm 5.8.

Algorithm 5.9 EC and field scalar operation with a CFI

Input: $E(F_q), k = (k_{t-1}, ..., k_1, k_0)_2, P \in E(F_q)$ and $m \in F_q$

Output: $2k \cdot P$ and $2k \cdot m$

1: $q \leftarrow random([1, 2^t - 1])$
2: $u \leftarrow q - m \mod n$
3: $u \leftarrow 2u$
4: $Q \leftarrow$ *RandomAfftoJac(P)*
5: *Verify(Q)*
6: **for** $i = t - 1$ to 0 **do**
7: $r \leftarrow$ *randombit()*
8: $u \leftarrow (2^{t-1} - 1) \cdot u \mod n$
9: $(Q, q) \leftarrow double(Q, q)$
10: $(Q, q) \leftarrow addsub(Q, P, q, u, m, k_i, r)$ //if $k_i = 0$, subtract else, add
11: **end for**
12: $(Q, q) \leftarrow addsub(Q, P, q, u, m, 0, 0)$
13: $q \leftarrow q - X \cdot u \mod n$ //$X = 2^{t-1} \cdot (1 - (2^{t-1} - 1)^{t+1})/((1 - (2^{t-1} - 1)) \cdot (2^t - 2)^t)$
14: $Q \leftarrow$ *JactoAff(Q)* //use Jacobian to affine conversion
15: *Verify(Q)*
16: **return** (Q, q)

The functions $double(Q, q)$ and $addsub(Q, P, q, u, m, k_i, r)$ aim at tying up the EC scalar and the field scalar operations. The purpose of doing this with $double(Q, q)$ is to extend algorithm 5.8 CFI to cover some operations of the EC scalar operation. Alongside the final point verification, this allows to be confident about the correct execution of the EC point doubling operation. The purpose of $addsub(Q, P, q, u, m, k_i, r)$ is similar, however it also ensures that EC point addition or subtraction choice is done in accordance to the field addition of subtraction.

Algorithm 5.10 details the $double(Q, q)$ function for $NIST$ curves in Jacobian coordinates. The modification between this algorithm and a standard EC point doubling algorithm allowing to tying the EC and the field scalar operations are between lines 6 to 10. Due to the CFI, the field doubling $2q$ is ensured to be computed otherwise the signature result will be completely corrupted with a random value. As this doubling is computed inside $double(Q, q)$, this allows ensuring that this function is executed and not bypassed due to a fault or any other reason.

Instead of directly computing the doubling, a partial result composed of both q and an EC element (Y_1) is computed in line 6, then doubled and replace the EC element Y_1 in line 8. At this point the EC element Y_1 is modified with $2q$ and thus $(X_1 : Y_1 : Z_1)$

Algorithm 5.10 ECC Jacobian point doubling and field doubling on NIST curves: double(Q,q)

Input: $P = (X_1 : Y_1 : Z_1)$ in Jacobian $\in E(F_p)$, $q \in F_p$
Output: $(X_1 : Y_1 : Z_1) \leftarrow 2P$ and $q \leftarrow 2q \mod n$

1: $T_1 \leftarrow Z_1^2 \mod p$
2: $T_2 \leftarrow X_1 - T_1 \mod p$
3: $T_1 \leftarrow X_1 + T_1 \mod p$
4: $T_2 \leftarrow T_2 \cdot T_1 \mod p$
5: $T_2 \leftarrow 3T_2 \mod p$
6: $\boldsymbol{T_3 \leftarrow Y_1 + q}$
7: $\boldsymbol{q \leftarrow 2Y_1 \mod p}$ // $2Y_1$
8: $\boldsymbol{Y_1 \leftarrow 2T_3}$ //$2Y_1 + 2q$
9: $\boldsymbol{q \leftarrow Y_1 - q}$ //$2q$
10: $\boldsymbol{Y_1 \leftarrow Y_1 - q \mod p}$ //$2Y_1$
11: $\boldsymbol{q \leftarrow q \mod n}$ //$2q \mod n$
12: $Z_1 \leftarrow Y_1 \cdot Z_1 \mod p$
13: $T_3 \leftarrow Y_1^2 \mod p$
14: $Y_1 \leftarrow T_3 \cdot X_1 \mod p$
15: $T_3 \leftarrow T_3^2 \mod p$
16: $T_3 \leftarrow T_3/2 \mod p$
17: $X_1 \leftarrow T_2^2 \mod p$
18: $T_1 \leftarrow 2Y_1 \mod p$
19: $X_1 \leftarrow X_1 - T_1 \mod p$
20: $T_1 \leftarrow Y_1 - X_1 \mod p$
21: $T_1 \leftarrow T_1 \cdot T_2 \mod p$
22: $Y_1 \leftarrow T_1 - T_3 \mod p$

is no longer on the curve. The register q which is also modified is then used with Y_1 to end up with the expected computation $2q$. This result is then used with Y_1 to obtain the good EC result which is two times the original Y_1 value. It is obvious that skipping a line among lines 6 to 10 conduct to fault q which is part of the CFI. It is also obvious that random faults on data will either fault Y_1, which will move the computation outside the curve, or q.

The $addsub(Q, P, q, m)$ function uses similar tricks and is presented as algorithm 5.11. This algorithm is based on algorithm 5.2 of section 5.1. The main differences are using Y_1 as a working register instead of T_2 due to line 2, then lines 5 to 13 aims at selecting addition or subtraction for both EC and field operations. Then lines 16 to 19 aim at checking the integrity of all tables which contain addition or subtraction possibilities. Using Y_1 instead of T_2 for intermediate value aims at ensuring that Y_1 is modified by the algorithm thanks to the CFI. In order to pass the point verification at the end of the EC scalar operation, X_1 and Z_1 should be modified accordingly to the modified Y_1. This ensure that either a point addition or a point subtraction is performed. In lines 5 to 9 two two-indexes tables are fulfilled with addition or subtraction possible values, for EC elements summed with field elements in one table (T_3) and field elements alone in the second (T_4). In line 11 the correct possibility, according to the addition or subtraction bit selection b, is transfered from T_3 into Y_1. From Y_1, an intermediate q value is then computed during line 12. Line 13 aims at removing the field component

from Y_1. Then the EC element Y_1 is removed from q.

Algorithm 5.11 ECC Jacobian-affine point addition/subtraction and field addition/subtraction: addsub(Q,P,q,u,m,b,r)

Input: $P = (X_1 : Y_1 : Z_1)$ in Jacobian and $Q = (x_2, y_2)$ in affine $\in E(F_p)$, q, u and $m \in F_p$, b operation selection, r a random bit

Output: if $b = 0$, $(X_1 : Y_1 : Z_1) \leftarrow P - Q$, $q \leftarrow q + u - m \mod n$ and $u \leftarrow 2u \mod n$, else $(X_1 : Y_1 : Z_1) \leftarrow P + Q$, $q \leftarrow q + u + m \mod n$ and $u \leftarrow 2u \mod n$

1: $T_1 \leftarrow Z_1^2 \mod p$
2: $T_2 \leftarrow Y_1$
3: $Y_1 \leftarrow T_1 \cdot Z_1 \mod p$
4: $T_1 \leftarrow T_1 \cdot x_2 \mod p$
5: $\boldsymbol{T_3[\overline{r}] \leftarrow Y_1}$
6: $\boldsymbol{T_4[\overline{r}] \leftarrow u + m \mod n}$
7: $\boldsymbol{T_3[\overline{r}] \leftarrow T_3[\overline{r}] + T_4[\overline{r}]}$
8: $\boldsymbol{T_3[r] \leftarrow -Y_1}$
9: $\boldsymbol{T_4[r] \leftarrow u - m \mod n}$
10: $\boldsymbol{T_3[r] \leftarrow T_3[r] + T_4[r]}$
11: $\boldsymbol{Y_1 \leftarrow T_3[b \oplus r]}$
12: $\boldsymbol{q \leftarrow Y_1 + q}$
13: $\boldsymbol{Y_1 \leftarrow Y_1 - T_4[\overline{b} \oplus \overline{r}] \mod p}$
14: $\boldsymbol{q \leftarrow q - Y_1 \mod n}$
15: $\boldsymbol{Y_1 \leftarrow Y_1 \cdot y_2 \mod p}$
16: $\boldsymbol{Y_1 \leftarrow Y_1 + T_3[0] \mod p}$
17: $\boldsymbol{Y_1 \leftarrow Y_1 + T_3[1] \mod p}$
18: $\boldsymbol{Y_1 \leftarrow Y_1 - T_4[0] \mod p}$
19: $\boldsymbol{Y_1 \leftarrow Y_1 - T_4[1] \mod p}$
20: $\boldsymbol{u \leftarrow T_4[\overline{r}] + T_4[r] \mod n}$
21: $Y_1 \leftarrow Y_1 - T_2 \mod p$
22: $T_1 \leftarrow T_1 - X_1 \mod p$
23: **if** $T_1 == 0$ **then**
24:     **return** (error)
25: **end if**
26: $Z_1 \leftarrow Z_1 \cdot T_1 \mod p$
27: $T_3[0] \leftarrow T_1^2 \mod p$
28: $T_3[1] \leftarrow T_3[0] \cdot T_1 \mod p$
29: $T_3[0] \leftarrow T_3[0] \cdot X_1 \mod p$
30: $T_1 \leftarrow T_3[0] + T_3[0] \mod p$
31: $X_1 \leftarrow Y_1^2 \mod p$
32: $X_1 \leftarrow X_1 - T_1 \mod p$
33: $X_1 \leftarrow X_1 - T_3[1] \mod p$
34: $T_3[0] \leftarrow T_3[0] - X_1 \mod p$
35: $T_3[0] \leftarrow T_3[0] \cdot Y_1 \mod p$
36: $T_3[1] \leftarrow T_3[1] \cdot T_2 \mod p$
37: $Y_1 \leftarrow T_3[0] - T_3[1] \mod p$

The addition or subtraction bit selection b is used two times, in line 11 and in line 13. If during both these lines, b is the same value, then either an EC point subtraction alongside a field subtraction is performed ($b = 0$) or an EC point addition alongside a field addition is performed ($b = 1$). If the b value is, for some reason (e.g. a fault), different in lines 11 and 13, then line 13 is equivalent to either $Y_1 \leftarrow -Y_1 - 2m$ or $Y_1 \leftarrow Y_1 + 2m$. The Y_1 value is thus corrupted with the random m and the result $(X_1 : Y_1 : Z_1)$ will not be on the curve which will be detected.

Using the modified EC point doubling and EC point addition formulæ generate around 8% of time overhead compared to standard jacobian-affine formulæ. The complete solution which consist in algorithm 5.9, algorithm 5.10 and algorithm 5.11 generates 13% time overhead and requires 5 more working registers compared to a classical double-and-add algorithm with also mixed jacobian-affines coordinates. This overhead allows ensuring that the operation flow is not modified. It also allows ensuring that the same nonce is used in both the r and s part of the signature by evaluating it simultaneously for both. To our knowledge, the closest solution is the one presented in [106], which involves more than 30% of overhead and requires the implementation of the $CRT()$ function. While in [106], authors do not consider fault on the s part of the signature, their point addition and point doubling formulæ can be used similarly than in our solution.

Countermeasure against nonce updating tampering

While AIS 31 or $NIST$ 800–90A/B standards require embedded tests to ensure that the RNG works properly and its output contains a high entropy, nothing allows ensuring that the output is correctly inserted inside the $ECDSA$ signature. In section 4.2.2, different attack scenarios are described which take place between the nonce generation by the RNG and its use during the computation of the $ECDSA$ signature. The described attack targeted addresses used to either read or write the nonce. By faulting addresses, the nonce can be incorrectly updated or the system mays use another value stored in other memory locations with poor entropy instead of the nonce generated by a high quality RNG. An interesting feature about the countermeasure presented in section 5.3.2 is the ability to compute the signature with an on the fly nonce generation. Indeed, by computing both $k \cdot P$ and $k \cdot m$ at the same time allows evaluating the nonce bits simultaneously for both the r and s parts of the signature. Thus, the nonce no longer need to be saved and the RNG output can be used on the fly. This can be used to circumvent attacks described in section 4.2.2 by directly connecting the RNG to the $ECDSA$ block avoiding the storage of the nonce in a memory. By doing so, the nonce value does not persist in any memory and then nonce reuse over signatures is not possible. As explained in section 4.2.2, the random bit b used in both lines 11 and 13 of algorithm 5.11 required to be unchanged. Thus, this provides reading redundancy over the value.

5.4 Algorithms Summary

Table 5.1 summarizes the proposed algorithms and compares them to basic well known algorithms. It shows required memory, performances, leakage vulnerabilities and various information on how the scalar is processed for a scalar of size t. The "$'$" algorithms are simply the scalar algorithm modified to compute both in EC field and in the basic field to provide CFI and avoid some other attacks as described in section 5.3.2. Algorithm "$5.4bis$", is the same as algorithm 5.4 however with four pre-computed points and scalar blinding $k' = k + \lambda n$ with λ having a size of $t/2$ bits. This choice allows acceleration over classic algorithm while blinding the scalar. While this kind of scalar blinding does not prevent lattice attacks, as explained in [42], it forces attacker to get more bits and to work on both selection methods due to the λ parameter. Indeed, the processed scalar is used to both select EC point addition/subtraction and select which precomputed point is used. With a parameter λ having a size of $t/2$ bits, if the EC point addition/subtraction selection somehow leak, only $t/2$ bits of information leak and thus it is not enough to mount a lattice attack. As opposed, if the precomputed point selection leak then, as pre-computed points are used in two different cases either in an addition or in a subtraction, it is not enough to mount a lattice attack. The last lines of the table, aim at giving the number of bit that are processed, the number of usecases these bits are manipulated, the repartition between them and finally the number of usecase manipulation that attacker should find leakages in order to get enough information to be able to mount a lattice attack in case of $ECDSA$.

Algorithm with Jacobian	ADA	Coron ADA	Montgomery $(+y)$	5.1	5.9	5.4	5.4' $k \cdot P \& k \cdot m$	5.4bis	5.4bis' $k \cdot P \& k \cdot m$	5.6	5.6' $k \cdot P \& k \cdot m$
Total working registers	7	10	10	7	11	7	11	7	11	13	17
Inputs	3	3	3	3	3	5	6	9	12	3	3
Performances	$tD + tA$	$tD + tA$	$(t+1)D + tA$	$tD + (t+1)A$	$\times 1.13$	$t/2D + (t/2+3)A$	$\times 1.13$	$t/2D + (t/2+5)A$	$\times 1.13$	$(t+26)D + (t+16)A$	$\times 1.13$
Constant EC operation flow	✓	✓	✓	✓	✓	✓		✓		✓	
Conditional jump	✗	✓	✓	✓	✓	✓		✓		✓	
Timing	✗	✓	✓	✓	✓	✓		✓		✓	
Data dependent	✗	✗/✓	✗/✓	✗/✓	✓	✓		✓		✓	
Infinity point	✗	✓	✓	✓	✓	✓		✓		✓	
Dummy operations	✗	✗	✗	✓	✓	✓		✓		✓	
Dummy operands	✓	✓	✗	✓	✓	✓		✓		✓	
Unused memory values	✗	✗	✗	✓	✓	✓		✓		✓	
Register update scalar dependents	✓	✓	✗	✓	✓	✓		✓		✓	
Integrated control flow	✗	✗	✗	✗	✓	✗	✓	✗	✓	✗	✓
Number of secret bits	t	t	t	t	t	t		$1.5 \cdot t$ $(kP + \lambda \cdot n)$		$2 \cdot t + 32$ $(k_1 \cdot P + k_2(r \cdot P))$	
Number of bit usecase	1	1	1	1	1	2 (add/sub, point selec.)		2 (add/sub, point selec.)		2 (add/sub, point selec.)	
Usecase repartition	t	t	t	t	t	$\frac{t}{2}/\frac{t}{2}$		$t/\frac{t}{2}$		t/t	
Nb of usecase to discriminate for attack	1	1	1	1	1	1		2		2	

Table 5.1: Algorithms summary

CHAPTER 6

Conclusion

Information security is a necessity and as it finally relies on IC, securing the hardware running the system is a necessity. This work focuses on hardware vulnerabilities (specifically non invasive attacks) of integrated circuits running $ECDSA$ algorithms. It provides a study of existing ECC algorithms and schemes in order to demonstrate their vulnerabilities against such threats. Indeed, despite the fact that many algorithms dedicated to ECC include countermeasures against side channel and fault attacks, security weaknesses remain regarding the computation of $ECDSA$ signatures. This is due to the fact that this cryptographic scheme is particularly sensitive, even to the smallest leakages as they can be leveraged by mathematical attacks.

We thus started by demonstrating that these mathematical attacks are relatively easy to implement and exploit if partial information of the $ECDSA$ nonces are available and without requiring any extended mathematical background. Experimental results with a widely used elliptic curve were provided in order to understand this threat.

While existing side channel and fault countermeasures protect the EC scalar manipulation in overall, they can fail in specific cases. Thus, different side channel leakage sources that allow recovering information about the EC scalar have been described and demonstrated. While some leakages are already well known, such as the operation flow that is scalar dependent inside the double-and-add EC scalar algorithm, less obvious leakage have been presented.

-Demonstration was provided that leakages can be unintentionally inserted due to the choice of coordinates representation which result in having different ”+” operations that can be distinguished. Various leakages due to the use of the infinity point have

also been discussed and illustrated for different cases to recover information even when EC points are blinded. These leakages can be recovered either with power consumption or also with a basic timing attack. Finally, we have demonstrated that side channel collisions between signature generations may allows recovering the $ECDSA$ private key thanks to lattice attacks and brutforce without requiring any bits value.

-We have also demonstrated that some elliptic curve scalar algorithms are wrongly supposed to be safe-error resistant due to either the algorithm or to underlying computations. The Montgomery ladder and the coherency checking countermeasures illustrate this point. The problem of the Montgomery ladder is that, depending on the scalar value, operations may become useless to the computation of the correct result. Thus by injecting a fault, attackers can detect the particular value of the scalar. While it cannot be expected to fully recover the EC scalar value with this method, it allows to get couple of bits. The coherency checking algorithm faces another problem that is due to the scalar dependent update of a working register alongside the complexity of EC operation that creates fault injection opportunities. By using a memory safe-error attack on an EC input point of the point addition algorithm after it is used inside the computation, attackers know if the register update corrects the fault or not and thus can obtain the scalar bit value.

-The concept of dummy operand have been introduced due, for example, to the infinity point that can be used when this specific point is considered and manipulated as a normal point (E.g. with Edward curves). We have demonstrated that even if in the $ECDSA$ signature, the scalar represents a nonce that is refreshed for each signature, safe-error attacks are enough to recover the private key due to the fact that obtaining only a couple of bits per signature allow mathematical attacks.

We thus showed that safe-error attacks and consequences are underestimated in the case of $ECDSA$ computations. The EC scalar operation represents only the r part of $ECDSA$ signature. The s part of the signature can also be targeted.

-We thus showed that $ECDSA$ signatures generated from faulted private key can be used to recursively recover the key bits. We demonstrated that a slightly faulted private key used during the computation of the s part of the signature mays generate a small error that can be recovered from the faulted signature. Thanks to the knowledge of how the private key is represented in registers and the error, information about the key can be obtained. Indeed, the possible error range generated by a register of a given size depends on the size and the genuine value. By recursively generating signature with

faulted private key, it is possible to recover the error range and with the knowledge of the register size and how the secret is represented, the private key can be fully recovered.

-A similar method was also applied on nonces. While it is not possible to recursively attack a given nonce, signature can be selected according to the obtained error due to the fault. This allows to provide enough bits of information to be used within lattice attacks and then to recover the private key. We have also demonstrated that the error distribution allows attackers to understand the behavior of the injected fault and then to greatly improve the attack speed.

-A basic architecture of an $ECDSA$ system was described allowing to compute signatures. This allowed to understand the wide range of fault injection possibilities and opportunities for attackers and to understand how realistic the described fault injections attacks can be. This also demonstrated that countermeasures in the low level functional blocks are mandatory but unfortunately are not enough to ensure the security regarding fault injection.

The $ECDSA$ private key manipulation and use are sensitive to both side channel and fault attacks. Moreover, every single bit of $ECDSA$ signatures nonces are important. Unfortunately, small leakages or weaknesses can be used through side channel or fault to easily recover partial information. As existing countermeasures do not perfectly protect against these small leakages, new-ones were proposed.

-First, new methods to compute the EC scalar algorithm have been proposed. These new algorithms aim at completely avoiding dummy operations, even locally and also to avoid the use of the infinity point while not constraining the scalar. Some of the described methods have been specifically developed to efficiently compute the EC scalar operation with a blinded scalar. These methods also reduce the risk against small leakages as more bits of the scalar should be recovered. It also forces attackers to find security flaws in different scalar dependent parts of the algorithm and combining them. The overhead involved for the two most robust algorithms (scalar blinding with half of the scalar size of with the scalar size) is respectively about $\times 1.6$ in memory for $\times 0.5$ in computation time and $\times 1.6$ for $\times 1.08$ in computation time compared to a classic double-and-add algorithm with mixed jacobian-affine coordinates.

Then approaches to protect both the $ECDSA$ private key and the nonce while computing signatures have been described.

-As the identified private key threat requires attackers to recover the mathematical

error generated by a faulted private key register, the countermeasure simply consists in using different shares to represent the private key. By doing so, the mathematical error due to a faulted share is multiplied by the second one. As the search space become too large, attackers cannot recover the error and thus conclude about the initial register value.

-In order to protect the nonce, the EC scalar operation $k \cdot P$ is computed alongside a basic field scalar operation $k \cdot m$ with a random m. By doing so, it allows avoiding scalar evaluation mismatch between the r and s parts of the signature.

-The nonce countermeasure has been extended to provide a control flow and computation integrity that ensures scalar operations are performed correctly. It allows to detect if operations are not computed and if the number of loop iterations are correctly performed is correct providing extended fault countermeasures. The performance overhead generated by this countermeasure is about $\times 1.13$ on the computation time. The memory overhead depends on the selected EC scalar algorithm to protect. In the basic case, it requires 5 more working register of the same size than the field.

The resulting algorithms thus provide a wide range of countermeasures against side channel and fault leakages. The scalar blinding techniques used aims at blind every bits of the scalar. Moreover it force attackers to face different scalar-bit selection methods reducing the risk of finding potential vulnerability. The overall overhead is contained as the two most advanced presented algorithms with all the countermeasures require respectively $\times 2$ in memory with $\times 1.22$ in computation time and $\times 2.3$ in memory with $\times 0.51$ in computation time compared to a classic double-and-add algorithm with mixed jacobian-affine coordinates. These numbers does not include the curve parameters which would reduce the memory overhead ratio.

Contributions

J. Dubeuf, D. Hely, and V. Beroulle, "Ecdsa passive attacks, leakage sources, and common design mistakes," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 21, pp. 31:1–31:24, Jan. 2016

J. Dubeuf, D. Hély, and V. Beroulle, "Enhanced elliptic curve scalar multiplication secure against side channel attacks and safe errors," in *Constructive Side-Channel Analysis and Secure Design - 8th International Workshop, COSADE 2017, Paris, France, April 13-14, 2017, Revised Selected Papers*, pp. 65–82, 2017

J. Dubeuf, F. Lhermet, and Y. LOISEL, "Systems and methods for operating secure elliptic curve cryptosystems," Sept. 22 2016. US Patent App. 14/744,927

Bibliography

[1] G. Martinez, H. Encinas, and S. Avila, "A survey of the elliptic curve integrated encryption scheme." Journal of computer science and engineering,2, 2 (2010), 7-13, 2010. http://hdl.handle.net/10261/32671/.

[2] "Information technology – Security techniques – Information security management systems – Overview and vocabulary," ISO/IEC 27000:2016, International Organization for Standardization, Geneva, CH, Feb. 2016.

[3] Y. CHERDANTSEVA and J. Hilton, "Information security and information assurance. the discussion about the meaning, scope and goals," 09 2013.

[4] C. National Research Council, K. W. Dam, and H. S. Lin, *Cryptography's Role in Securing the Information Society*. Washington, DC, USA: National Academy Press, 1996.

[5] P. C. Kocher, *Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems*, pp. 104–113. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996.

[6] P. C. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '99, (London, UK, UK), pp. 388–397, Springer-Verlag, 1999.

[7] K. Gandolfi, C. Mourtel, and F. Olivier, *Electromagnetic Analysis: Concrete Results*, pp. 251–261. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001.

[8] A. Schlösser, D. Nedospasov, J. Krämer, S. Orlic, and J.-P. Seifert, "Simple photonic emission analysis of aes," in *Cryptographic Hardware and Embedded*

Systems – CHES 2012 (E. Prouff and P. Schaumont, eds.), (Berlin, Heidelberg), pp. 41–57, Springer Berlin Heidelberg, 2012.

[9] R. Newman, "Visible light from a silicon $p-n$ junction," *Phys. Rev.*, vol. 100, pp. 700–703, Oct 1955.

[10] W. K. Chim, *Semiconductor Device and Failure Analysis : Using Photon Emission Microscopy.* Wiley, 2000.

[11] D. Genkin, A. Shamir, and E. Tromer, "Rsa key extraction via low-bandwidth acoustic cryptanalysis," in *Advances in Cryptology – CRYPTO 2014* (J. A. Garay and R. Gennaro, eds.), (Berlin, Heidelberg), pp. 444–461, Springer Berlin Heidelberg, 2014.

[12] D. J. Bernstein, "Cache-timing attacks on aes," tech. rep., 2005.

[13] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," *meltdownattack.com*, 2018.

[14] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown," *meltdownattack.com*, 2018.

[15] J. W. Bos, C. Hubain, W. Michiels, and P. Teuwen, "Differential computation analysis: Hiding your white-box designs is not enough," in *Cryptographic Hardware and Embedded Systems – CHES 2016* (B. Gierlichs and A. Y. Poschmann, eds.), (Berlin, Heidelberg), pp. 215–236, Springer Berlin Heidelberg, 2016.

[16] O. Faurax, A. Tria, L. Freund, and F. Bancel, "Robustness of circuits under delay-induced faults : test of aes with the pafi tool," in *13th IEEE International On-Line Testing Symposium (IOLTS 2007)*, pp. 185–186, July 2007.

[17] A. Barenghi, G. Bertoni, E. Parrinello, and G. Pelosi, "Low voltage fault attacks on the rsa cryptosystem," in *2009 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pp. 23–31, Sept 2009.

[18] R. Anderson and M. Kuhn, *Low cost attacks on tamper resistant devices*, pp. 125–136. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998.

[19] P. I. Breveglieri, I. Koren, F. D. A. T. In, H. Choukri, and M. Tunstall, "Round reduction using faults."

[20] J. Brouchier, T. Kean, C. Marsh, and D. Naccache, "Temperature attacks," *IEEE Security and Privacy*, vol. 7, pp. 79–82, Mar. 2009.

[21] A. Debhaoui, J.-M. Dutertre, B. Robisson, P. Orsatelli, P. Maurine, and A. Tria, "Injection of transient faults using electromagnetic pulses Practical results on a cryptographic system," 2012. Journal of Cryptology ePrint Archive: Report 2012/123.

[22] S. P. Skorobogatov and R. J. Anderson, *Optical Fault Induction Attacks*, pp. 2–12. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003.

[23] P. Maurine, K. Tobich, T. Ordas, and P. Y. Liardet, "Yet Another Fault Injection Technique : by Forward Body Biasing Injection," in *YACC'2012: Yet Another Conference on Cryptography*, (Porquerolles Island, France), Sept. 2012.

[24] N. Beringuier-Boher, M. Lacruche, D. El-Baze, J.-M. Dutertre, J.-B. Rigaud, and P. Maurine, "Body biasing injection attacks in practice," in *Proceedings of the Third Workshop on Cryptography and Security in Computing Systems*, CS2 '16, (New York, NY, USA), pp. 49–54, ACM, 2016.

[25] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pp. 361–372, June 2014.

[26] E. Sanfelix, C. Mune, and J. de Haas, "Unboxing the white-box," Blackhat Europe, 2015.

[27] N. Koblitz, "Elliptic curve cryptosystems," *Mathematics of Computation*, vol. 48, pp. 203–209, Jan. 1987.

[28] V. S. Miller, "Use of elliptic curves in cryptography," in *Advances in Cryptology — CRYPTO '85 Proceedings* (H. C. Williams, ed.), (Berlin, Heidelberg), pp. 417–426, Springer Berlin Heidelberg, 1986.

[29] D. Hankerson, A. J. Menezes, and S. Vanstone, *Guide to Elliptic Curve Cryptography*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2003.

[30] J. M. Pollard, "Monte Carlo methods for index computation mod p," *Mathematics of Computation*, vol. 32, pp. 918–924, 1978.

[31] S. Pohlig and M. Hellman, "An improved algorithm for computing logarithms over and its cryptographic significance (corresp.)," *IEEE Trans. Inf. Theor.*, vol. 24, pp. 106–110, Sept. 2006.

[32] T. Satoh and K. Araki, "Fermat quotients and the polynomial time discrete log algorithm for anomalous elliptic curves," *Commentarii Math. Univ. St. Pauli*, 1998.

[33] I. A. Semaev, "Evaluation of discrete logarithms in a group of p-torsion points of an elliptic curve in characteristic p," *Math. Comput.*, vol. 67, no. 221, pp. 353–356, 1998.

[34] N. P. Smart, "The discrete logarithm problem on elliptic curves of trace one," *Journal of Cryptology*, vol. 12, pp. 193–196, 1999.

[35] A. Menezes, S. Vanstone, and T. Okamoto, "Reducing elliptic curve logarithms to logarithms in a finite field," in *Proceedings of the Twenty-third Annual ACM Symposium on Theory of Computing*, STOC '91, (New York, NY, USA), pp. 80–89, ACM, 1991.

[36] G. Frey and H.-G. Rück, "A remark concerning m-divisibility and the discrete logarithm in the divisor class group of curves," *Math. Comput.*, vol. 62, pp. 865–874, Apr. 1994.

[37] NIST, "Digital signature standard (dss)." FIPS PUB 186., 2013. `http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf`.

[38] P. Q. Nguyen and I. Shparlinski, "The insecurity of the digital signature algorithm with partially known nonces," *J. Cryptology*, vol. 15, no. 3, pp. 151–176, 2002.

[39] B. B. Brumley and N. Tuveri, "Remote timing attacks are still practical," in *Computer Security - ESORICS 2011 - 16th European Symposium on Research in Computer Security, Leuven, Belgium, September 12-14, 2011. Proceedings*, pp. 355–371, 2011.

[40] L. A. L. L. Lenstra, H.W. jr., "Factoring polynomials with rational coefficients.," *Mathematische Annalen*, vol. 261, pp. 515–534, 1982.

[41] L. Babai, "On lovász' lattice reduction and the nearest lattice point problem," *Combinatorica*, vol. 6, Mar. 1986.

[42] D. Goudarzi, M. Rivain, and D. Vergnaud, "Lattice attacks against elliptic-curve signatures with blinded scalar multiplication," in *Selected Areas in Cryptography - SAC 2016 - 23rd International Conference, St. John's, NL, Canada, August 10-12, 2016, Revised Selected Papers*, pp. 120–139, 2016.

[43] D. Bleichenbacher, "On the generation of dss one-time keys." Preprint, 2001.

[44] NSA, "Tempest: A signal problem," in *Cryptologic Spectrum*, 1972.

[45] "Tempest certification program," Accessed: 2017-11-15. `https://www.iad.gov/iad/programs/iad-initiatives/tempest.cfm`.

[46] "Canadian emsec and tempest," Accessed: 2017-11-15. `https://www.cse-cst.gc.ca/en/publication/list/EMSEC-and-TEMPEST`.

[47] W. Van Eck, "Electromagnetic radiation from video display units: An eavesdropping risk?," *Computers & Security*, vol. 4, no. 4, pp. 269 – 286, 1985.

[48] J.-J. Quisquater and D. Samyde, *ElectroMagnetic Analysis (EMA): Measures and Counter-measures for Smart Cards*, pp. 200–210. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001.

[49] S. Mangard, E. Oswald, and T. Popp, *Power Analysis Attacks: Revealing the Secrets of Smart Cards (Advances in Information Security)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2007.

[50] J.-S. Coron, "Resistance against differential power analysis for elliptic curve cryptosystems," 1999.

[51] C. Rebeiro, D. Mukhopadhyay, and S. Bhattacharya, *Timing Channels in Cryptography*. Springer-Verlag New York, Inc., 2015.

[52] E. Brier, M. Joye, and T. E. D. Win, "Weierstraß elliptic curves and side-channel attacks," in *Public Key Cryptography – PKC 2002, volume 2274 of LNCS*, pp. 335–345, Springer–Verlag, 2002.

[53] M. Medwed and E. Oswald, "Template attacks on ecdsa.," in *WISA* (K.-I. Chung, K. Sohn, and M. Yung, eds.), vol. 5379 of *Lecture Notes in Computer Science*, pp. 14–27, Springer, 2008.

[54] L. Batina, L. Chmielewski, L. Papachristodoulou, P. Schwabe, and M. Tunstall, "Online template attacks," in *Progress in Cryptology - INDOCRYPT 2014 - 15th International Conference on Cryptology in India, New Delhi, India, December 14-17, 2014, Proceedings*, pp. 21–36, 2014.

[55] J.-L. Danger, S. Guilley, P. Hoogvorst, C. Murdica, and D. Naccache, "A synthesis of side-channel attacks on elliptic curve cryptography in smart-cards," *Journal of Cryptographic Engineering*, vol. 3, no. 4, pp. 241–265, 2013.

[56] H. Mamiya, A. Miyaji, and H. Morimoto, "Efficient countermeasures against rpa, dpa, and SPA," in *Cryptographic Hardware and Embedded Systems - CHES 2004: 6th International Workshop Cambridge, MA, USA, August 11-13, 2004. Proceedings*, pp. 343–356, 2004.

[57] G. L. Keister and H. V. Stewart, "The effect of nuclear radiation on selected semiconductor devices," *Proceedings of the IRE*, vol. 45, pp. 931–937, July 1957.

[58] R. S. Caldwell, D. S. Gage, and G. H. Hanson, "The transient behavior of transistors due to ionized radiation pulses," *Transactions of the American Institute of Electrical Engineers, Part I: Communication and Electronics*, vol. 81, pp. 483–491, Jan 1963.

[59] J. L. Wirth and S. C. Rogers, "The transient response of transistors and diodes to ionizing radiation," *IEEE Transactions on Nuclear Science*, vol. 11, pp. 24–38, Nov 1964.

[60] D. H. Habing, "The use of lasers to simulate radiation-induced transients in semiconductor devices and circuits," *IEEE Transactions on Nuclear Science*, vol. 12, pp. 91–100, Oct 1965.

[61] D. Binder, E. C. Smith, and A. B. Holman, "Satellite anomalies from galactic cosmic rays," *IEEE Transactions on Nuclear Science*, vol. 22, pp. 2675–2680, Dec 1975.

[62] T. C. May and M. H. Woods, "A new physical mechanism for soft errors in dynamic memories," in *16th International Reliability Physics Symposium*, pp. 33–40, April 1978.

[63] J. F. Ziegler and W. A. Lanford, "Effect of cosmic rays on computer memories," *Science*, vol. 206, no. 4420, pp. 776–788, 1979.

[64] D. Boneh, R. A. Demillo, and R. J. Lipton, "On the importance of eliminating errors in cryptographic computations," *Journal of Cryptology*, vol. 14, pp. 101–119, 2001.

[65] E. Biham and A. Shamir, *Differential fault analysis of secret key cryptosystems*, pp. 513–525. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997.

[66] C. Aumüller, P. Bier, W. Fischer, P. Hofreiter, and J.-P. Seifert, *Fault Attacks on RSA with CRT: Concrete Results and Practical Countermeasures*, pp. 260–275. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003.

[67] J. L. Danger, S. Guilley, S. Bhasin, and M. Nassar, "Overview of dual rail with precharge logic styles to thwart implementation-level attacks on hardware cryptoprocessors," in *2009 3rd International Conference on Signals, Circuits and Systems (SCS)*, pp. 1–8, Nov 2009.

[68] T. Fukunaga and J. Takahashi, "Practical fault attack on a cryptographic lsi with iso/iec 18033-3 block ciphers," in *2009 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pp. 84–92, Sept 2009.

[69] M. Agoyan, J.-M. Dutertre, D. Naccache, B. Robisson, and A. Tria, *When Clocks Fail: On Critical Paths and Clock Faults*, pp. 182–193. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010.

[70] K. Sakiyama, T. Yagi, and K. Ohta, "Fault analysis attack against an aes prototype chip using rsl," in *Proceedings of the The Cryptographers' Track at the RSA Conference 2009 on Topics in Cryptology*, CT-RSA '09, (Berlin, Heidelberg), pp. 429–443, Springer-Verlag, 2009.

[71] F. Khelil, M. Hamdi, S. Guilley, J. L. Danger, and N. Selmane, "Fault analysis attack on an fpga aes implementation," in *2008 New Technologies, Mobility and Security*, pp. 1–5, Nov 2008.

[72] N. Selmane, S. Guilley, and J. L. Danger, "Practical setup time violation attacks on aes," in *2008 Seventh European Dependable Computing Conference*, pp. 91–96, May 2008.

[73] J. M. Schmidt, M. Hutter, and T. Plos, "Optical fault attacks on aes: A threat in violet," in *2009 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pp. 13–22, Sept 2009.

[74] S. Skorobogatov, "Using optical emission analysis for estimating contribution to power analysis," in *2009 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pp. 111–119, Sept 2009.

[75] Y. Monnet, M. Renaudin, R. Leveugle, C. Clavier, and P. Moitrel, *Case Study of a Fault Attack on Asynchronous DES Crypto-Processors*, pp. 88–97. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006.

[76] Alphanov, "Focus and scan two laser spots through the field of the objective," Accessed: 2017-09-15. `http://www.alphanov.com/41-optoelectronics-systems-and-microscopy-double-laser-microscope-station.html`.

[77] A. Tang, S. Sethumadhavan, and S. Stolfo, "CLKSCREW: Exposing the perils of security-oblivious energy management," in *26th USENIX Security Symposium (USENIX Security 17)*, (Vancouver, BC), pp. 1057–1074, USENIX Association, 2017.

[78] "Soc and cpu system-wide approach to security," Accessed: 2017-11-12. `https://www.arm.com/products/security-on-arm/trustzone`.

[79] P. Q. Nguyen and M. Tibouchi, *Lattice-Based Fault Attacks on Signatures*. nformation Security and Cryptography, Springer, 2011.

[80] S.-M. Yen, S. Kim, S. Lim, and S. Moon, "A countermeasure against one physical cryptanalysis may benefit another attack," in *Proceedings of the 4th International Conference Seoul on Information Security and Cryptology*, ICISC '01, (London, UK, UK), pp. 414–427, Springer-Verlag, 2002.

[81] M. Joye and S.-M. Yen, "The montgomery powering ladder," in *Revised Papers from the 4th International Workshop on Cryptographic Hardware and Embedded Systems*, CHES '02, (London, UK, UK), pp. 291–302, Springer-Verlag, 2003.

[82] J. Fan and I. Verbauwhede, "An updated survey on secure ecc implementations: Attacks, countermeasures and cost," in cryptography," in *Cryptography and Security: From Theory to Applications* (D. Naccache, ed.), vol. 6805 of *Lecture Notes in Computer Science*, pp. 265–282, Springer Berlin Heidelberg, 2012.

[83] F. Rondepierre, "Revisiting atomic patterns for scalar multiplications on elliptic curves," in *Smart Card Research and Advanced Applications* (A. Francillon and P. Rohatgi, eds.), vol. 8419 of *Lecture Notes in Computer Science*, pp. 171–186, Springer International Publishing, 2013.

[84] H. Mamiya, A. Miyaji, and H. Morimoto, "Efficient countermeasures against rpa, dpa, and spa.," in *CHES* (M. Joye and J.-J. Quisquater, eds.), vol. 3156 of *Lecture Notes in Computer Science*, pp. 343–356, Springer, 2004.

[85] K. Itoh, T. Izu, and M. Takenaka, "Efficient countermeasures against power analysis for elliptic curve cryptosystems," in *Smart Card Research and Advanced Applications VI, IFIP 18th World Computer Congress, TC8/WG8.8 & TC11/WG11.2 Sixth International Conference on Smart Card Research and Advanced Applications (CARDIS), 22-27 August 2004, Toulouse, France*, pp. 99–113, 2004.

[86] L. Goubin, "A refined power-analysis attack on elliptic curve cryptosystems.," in *Public Key Cryptography* (Y. Desmedt, ed.), vol. 2567 of *Lecture Notes in Computer Science*, pp. 199–210, Springer, 2003.

[87] T. Akishita and T. Takagi, "Zero-value point attacks on elliptic curve cryptosystem," in *Information Security* (C. Boyd and W. Mao, eds.), vol. 2851 of *Lecture Notes in Computer Science*, pp. 218–233, Springer Berlin Heidelberg, 2003.

[88] J. Dubeuf, D. Hely, and V. Beroulle, "Ecdsa passive attacks, leakage sources, and common design mistakes," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 21, pp. 31:1–31:24, Jan. 2016.

[89] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B. Yang, "High-speed high-security signatures," *IACR Cryptology ePrint Archive*, vol. 2011, p. 368, 2011.

[90] D. J. Bernstein, P. Birkner, M. Joye, T. Lange, and C. Peters, "Twisted edwards curves." Cryptology ePrint Archive, Report 2008/013, 2008. `http://eprint.iacr.org/`.

[91] A. Alkhoraidly, A. Dominguez-Oviedo, and M. A. Hasan, "Fault attacks on elliptic curve cryptosystems," in *Fault Analysis in Cryptography*, pp. 137–155, 2012.

[92] A. Barenghi, G. Bertoni, A. Palomba, and R. Susella, "A novel fault attack against ECDSA," in *HOST 2011, Proceedings of the 2011 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST), 5-6 June 2011, San Diego, California, USA*, pp. 161–166, 2011.

[93] S. Ali, X. Guo, R. Karri, and D. Mukhopadhyay, *Fault Attacks on AES and Their Countermeasures*, pp. 163–208. Cham: Springer International Publishing, 2016.

[94] P.-A. Fouque, D. Réal, F. Valette, and M. Drissi, "The carry leakage on the randomized exponent countermeasure.," in *CHES* (E. Oswald and P. Rohatgi, eds.), vol. 5154 of *Lecture Notes in Computer Science*, pp. 198–213, Springer, 2008.

[95] K. Itoh, T. Izu, and M. Takenaka, "Address-bit differential power analysis of cryptographic schemes ok-ecdh and ok-ecdsa," in *Revised Papers from the 4th International Workshop on Cryptographic Hardware and Embedded Systems*, CHES '02, (London, UK, UK), pp. 129–143, Springer-Verlag, 2003.

[96] A. Bauer, É. Jaulmes, E. Prouff, and J. Wild, "Horizontal collision correlation attack on elliptic curves," in *Selected Areas in Cryptography - SAC 2013 -*

20th International Conference, Burnaby, BC, Canada, August 14-16, 2013, Revised Selected Papers, pp. 553–570, 2013.

[97] R. R. Goundar, M. Joye, A. Miyaji, M. Rivain, and A. Venelli, "Scalar multiplication on weierstraß elliptic curves from co-z arithmetic.," *J. Cryptographic Engineering*, vol. 1, no. 2, pp. 161–176, 2011.

[98] M. Hedabou, P. Pinel, and L. Bénéteau, "Countermeasures for preventing comb method against sca attacks," in *Proceedings of the First International Conference on Information Security Practice and Experience*, ISPEC'05, (Berlin, Heidelberg), pp. 85–96, Springer-Verlag, 2005.

[99] F. Muller and F. Valette, "High-order attacks against the exponent splitting protection," in *Public Key Cryptography*, pp. 315–329, 2006.

[100] B. Feix, M. Roussellet, and A. Venelli, "Side-channel analysis on blinded regular scalar multiplications." Cryptology ePrint Archive, Report 2014/191, 2014. `http://eprint.iacr.org/`.

[101] K. Okeya and K. Sakurai, "Power analysis breaks elliptic curve cryptosystems even secure against the timing attack," in *Progress in Cryptology - INDOCRYPT 2000, First International Conference in Cryptology in India, Calcutta, India, December 10-13, 2000, Proceedings*, pp. 178–190, 2000.

[102] P.-A. Fouque and F. Valette, "The doubling attack - why upwards is better than downwards.," in *CHES* (C. D. Walter, e. K. Koç, and C. Paar, eds.), vol. 2779 of *Lecture Notes in Computer Science*, pp. 269–280, Springer, 2003.

[103] F. Courbon, *Partial hardware reverse engineering applied to fine grained laser fault injection and efficient hardware trojans detection*. Theses, Ecole Nationale Supérieure des Mines de Saint-Etienne, Sept. 2015.

[104] M. Hutter, M. Medwed, D. Hein, and J. Wolkerstorfer, *Attacking ECDSA-Enabled RFID Devices*, pp. 519–534. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009.

[105] M. Repka, M. Varchola, and M. Drutarovský, "Improving cpa attack against dsa and ecdsa," vol. 66, pp. 159–163, 06 2015.

[106] J. Schmidt and M. Medwed, "A fault attack on ECDSA," in *FDTC*, pp. 93–99, IEEE Computer Society, 2009.

[107] J. Dubeuf, D. Hély, and V. Beroulle, "Enhanced elliptic curve scalar multiplication secure against side channel attacks and safe errors," in *Constructive Side-Channel Analysis and Secure Design - 8th International Workshop, COSADE 2017, Paris, France, April 13-14, 2017, Revised Selected Papers*, pp. 65–82, 2017.

[108] J. Dubeuf, F. Lhermet, and Y. LOISEL, "Systems and methods for operating secure elliptic curve cryptosystems," Sept. 22 2016. US Patent App. 14/744,927.

Résumé substantiel

Ce document de thèse vise tout d'abord à soulever un certain nombre de vulnérabilités liées à l'utilisation de l'$ECDSA$ dans un environnement permettant les analyses par canaux-cachés ou par attaques en fautes. Il rappelle que l'$ECDSA$ a pour particularité d'être extrêmement sensible en cas de fuite partielle d'informations sur les secrets manipulés, ce qui est dû aux attaques à base de réseau euclidien. Nous constatons à travers la thèse que toutes les contremesures contre les attaques par canaux-cachés et attaques en faute ne protègent pas intégralement les secrets manipulés. Il devient alors possible de récupérer partiellement de l'information sur ces secrets et ainsi d'utiliser des outils de cryptanalyse mathématique afin de complètement récupérer les secrets. De nouvelles contremesures sont donc indispensables afin de garantir qu'aucune information permettant une cryptanalyse mathématique ne fuite du système.

La sécurité de l'information est l'un des enjeux majeurs du monde actuel. La cryptographie y joue un rôle particulier puisqu'elle permet, via une assurance mathématique ou physique, d'apporter confidentialité, intégrité et authenticité aux systèmes. Les implémentations cryptographiques dépendent d'un support physique qui est bien souvent un circuit intégré. Ces circuits peuvent être la cible d'attaques diverses et variées. Les attaques par canaux-cachés ainsi que les attaques en fautes posent un problème particulier de par le faible coût de mise en œuvre et l'accessibilité qu'elles permettent. En effet, les attaques par canaux-cachés visent simplement à observer le système durant l'exécution d'opération critique afin d'obtenir de l'information. Les observations se portent sur des éléments tels que le temps de calcul, la consommation ou même le rayonnement électromagnétique du composant, de façon totalement passive. Les attaques en fautes visent à influencer le composant durant l'exécution d'opération critique afin de les altérer. L'attaquant analyse ensuite le résultat produit pour en retirer de l'information. Pour se faire, différents moyens sont possibles. Tout d'abord, l'attaquant peut essayer de faire fonctionner le composant au-delà des spécifications de fonctionnement fournies par le fabricant. Il est également possible d'améliorer le contrôle de la faute injectée, en ouvrant mécaniquement ou chimiquement le composant afin d'exposer certaines zones du circuit logique à un faisceau laser, de telle sorte à apporter localement de l'énergie.

L'algorithme $ECDSA$ est un algorithme cryptographique à clef publique à base de courbe elliptique qui permet de valider qu'une entité est porteuse d'une clef privée à partir d'une clef publique via un mécanisme de signature et vérification. Cet algorithme est très utilisé à travers une multitude de protocoles afin de garantir l'authenticité d'un message. Trois phases sont distinctes. Tout d'abord la génération de clef vise à générer

une clef publique à partir d'une clef privée grâce à l'utilisation d'un scalaire elliptique. Cette paire de clefs correspond à une identité. La clef publique peut être connue de tous alors que la clef privée permet à son porteur d'être le seul à pouvoir effectuer certaines opérations, telle que la génération de signature. Cette dernière vise à générer une signature $(r; s)$ d'un message grâce à la clef privée et à un nombre aléatoirement généré (nonce). La vérification de l'$ECDSA$ permet de valider que la signature $(r; s)$ d'un message a bien été générée par quelqu'un connaissant la clef privée. L'étape de vérification nécessite uniquement la clef publique, le message, ainsi que la signature.

Garantir la sécurité de l'implémentation de la signature de l'$ECDSA$ est donc primordial pour éviter le vol d'identité numérique. Ce schéma est cependant particulièrement vulnérable à cause d'outils de cryptanalyse tels que les réseaux euclidiens qui permettent de récupérer la clef privée à partir de signatures dont on connaît des fragments d'information sur les nonces. Il en résulte qu'une implémentation légèrement imparfaite peut être fatale à la sécurité globale du système. Nous avons donc expérimentalement testé les attaques à base de réseaux euclidiens et constaté que 70 signatures dont on connaît 9 bits consécutifs du nonce suffisent pour extraire la clef privée d'un système utilisant une courbe elliptique sur 256 bits. Cette constatation est dérangeante puisque le nombre de bits requis est extrêmement faible en comparaison à la taille de la courbe. De plus, notre implémentation de l'attaque est loin de représenter l'état de l'art en la matière.

L'étape suivante du manuscrit vise donc à évaluer les algorithmes de scalaire elliptique existants, afin de vérifier s'ils permettent de garantir la confidentialité de tous les bits du scalaire manipulés face aux attaques par canaux cachés et en faute. Bien que ces sujets ne soient pas inconnus, l'utilisation du scalaire elliptique dans le cadre de l'$ECDSA$ le soumet à des risques de sécurité particuliers qui n'ont, à notre goût, pas suffisamment été abordés par le passé, du fait qu'une fuite partielle permet à un attaquant de récupérer intégralement la clef privée. Par exemple, concernant les canaux cachés, nous avons constaté que l'observation de calcul de différentes signatures pouvait dans certains cas permettre à un attaquant d'identifier des signatures générées à partir de nonces qui possèdent un groupe de bits avec la même valeur. Bien que la valeur reste inconnue de l'attaquant, celui-ci peut néanmoins mener une cryptanalyse à base de réseau euclidien tout en émettant des hypothèses sur la valeur du groupe de bits. A partir de cette constatation, un attaquant peut donc observer les traces de consommation générées lors de plusieurs signatures ; trouver des traces qui ont les mêmes bits de poids fort et ensuite retrouver la clef privée grâce à une cryptanalyse à base de réseau euclidien. En réalisant ceci à partir de cinq cent mille traces, l'attaquant peut espérer obtenir de façon probabiliste trente signatures ayant les même quatorze bits de poids fort. Ce qui, au vu de nos expérimentations, est largement suffisant pour extraire la clef privée d'un

système utilisant une courbe elliptique sur 256 bits.

Après une courte introduction sur la manière dont peut être insérée une faute dans une puce électronique, les attaques en fautes visant l'$ECDSA$ sont traitées en deux points. Tout d'abord, les attaques sur le scalaire elliptique ; ensuite sur le reste de la signature. Dans la première partie, nous montrons entre autres qu'il est possible d'obtenir de l'information sur quelques bits du scalaire elliptique grâce à une faute. Nous approfondissons notamment l'utilisation des safe-error et démontrons leur efficacité sur différents algorithmes. Il en ressort que la quasi-totalité des algorithmes y sont vulnérables lorsque le scalaire n'est pas masqué. Un exemple d'algorithme généralement reconnu pour être résistant aux safe-errors est traité. Il s'agit du scalaire de Montgomery. Cependant nous avons constaté que dans le cas d'un scalaire nul, cet algorithme se transforme de telle sorte qu'une partie des opérations exécutées n'influent pas sur le résultat final. Nous avons donc montré qu'il est possible, en fautant ces opérations, de déterminer si les bits les moins significatifs d' un scalaire aléatoire ont une valeur nulle ou non avec le principe de safe-error.

D'autres exemples d'utilisation des safe-errors sont donnés dans le document visant les différents niveaux arithmétiques des calculs. La notion de "dummy operand", également introduite, permet de détecter des valeurs particulières d'opérandes à partir du principe de safe-error. Il est également mentionné que le masquage du scalaire peut dans quelques cas ne pas suffire car certains masquages ne protègent pas l'ensemble des bits du scalaire. La deuxième partie concernant les attaque en faute sur l'$ECDSA$ traite des fautes lors de la génération de signature en dehors du scalaire elliptique. De nouvelles attaques sont démontrées tirant parti de signatures générées à partir d'une clef privée ou d'un nonce erroné. Il est montré qu'à partir d'une signature générée avec une clef fautée de quelques bits, un attaquant peut retrouver l'erreur mathématique générée. En ayant connaissance de la façon dont est représentée la clef dans le matériel (e.g. représentation binaire), l'attaquant peut obtenir de l'information sur la clef privée. En attaquant récursivement, l'ensemble de la clef privée peut ainsi être obtenue. De façon similaire, si, due à une faute, les parties r et s d'une signature sont calculées à partir d'un nonce légèrement différent de quelque bits, l'attaquant peut retrouver l'erreur générée et obtenir quelques bits d'information sur le nonce utilisé. En cumulant ce genre de signature, il devient possible à l'attaquant de récupérer la clef privée grâce aux attaques à base de réseau euclidien. Nous montrons dans le document qu'il est également possible avec la même approche d'extraire de l'information en ciblant des calculs intermédiaires.

Une analyse d'architecture classique de composants permettant la génération de signatures d'$ECDSA$ ainsi que des opportunités pour insérer les fautes discutées, est fournie. Elle met en évidence les nombreuses possibilités offertes aux attaquants pour récupérer la clef privée du système. Elle montre également l'importance de sécuriser

l'ensemble du système et non juste l'implémentation du scalaire elliptique.

Les contremesures développées dans le document visent à protéger l'ensemble de la génération de signature $ECDSA$ et sont présentées dans un chapitre dédié. Dans un premier temps, de nouveaux algorithmes permettant le calcul du scalaire elliptique sont fournis. Ils visent à résister aux différentes attaques en fautes et en canaux-cachés discutées. Différents algorithmes effectuant plusieurs types de calculs sur les courbes elliptique sont détaillés. Ces algorithmes sont ensuite modifiés afin de converger vers un algorithme permettant un masquage efficace et total du scalaire utilisé. Grâce à ce masquage, l'extraction de fragments d'information relatifs au scalaire, ne permet pas à l'attaquant d'obtenir la clef privée du système via la cryptanalyse à base de réseau euclidien.

Dans un second temps, des contremesures visant le reste du calcul de signature de l'$ECDSA$ sont détaillées. Ces dernières contremesures visent à protéger à la fois l'utilisation de la clef privée, le nonce, ainsi que les calculs intermédiaires des attaques en faute et par canaux-cachés. Le nonce est protégé en modifiant le schéma de signature et l'algorithme du scalaire elliptique de telle sorte qu'il soit évalué une seul fois durant la génération de la signature. La méthode présentée permet également de valider l'intégrité du flot d'opérations, tout en protégeant la signature via des calculs infectieux qui rendent les signatures modifiées par les attaquants inutilisables. Afin de protéger la clef privée, un schéma de masquage avec mise à jour est également proposé. Le surcout généré par l'ensemble de ces contremesures est contenu puisque le temps de calcul est augmenté de seulement 22% comparé à l'algorithme de référence "always-double-and-add" et ceci tout en doublant le nombre de registres de travail.

Studies and Implementation of Hardware Countermeasures for ECDSA Cryptosystems

Key Topics

- ∗ Elliptic curve arithmetic
- ∗ Lattice attack
- ∗ Side channel leakages of few EC scalar bits
- ∗ Safe-error on EC scalar to recover few secret bits
- ∗ Fault attack against the $ECDSA$
- ∗ Example of vulnerable architecture
- ∗ Countermeasures

Information security heavily relies on integrated circuits (ICs). Unfortunately, ICs face a lot of threats such as side channel or fault attacks. This work focuses on small vulnerabilities and countermeasures for the Elliptic Curve Digital Signature Algorithm ($ECDSA$). The motivation is that leakage sources may be used in different attack scenarios. By fixing the leakage, existing attacks are prevented but also undiscovered or non-disclosed attacks based on the leakage. Moreover, while the elliptic curve scalar algorithm is at the heart of the security of all elliptic curve related cryptographic schemes, all the $ECDSA$ system needs security. A small leakage of few secret bits may conduct to fully disclose the private key and thus should be avoided. The $ECDSA$ can be implemented in different flavors such as in a software that runs on a microcontroller or as a hardware self-contained block or also as a mix between software and hardware accelerator. Thus, a wide range of architectures is possible to implement an $ECDSA$ system. For this reason, this work mainly focuses on algorithmic countermeasures as they allow being compliant with different kinds of implementations.

Contenue clef

- ∗ Arithmétique des courbes elliptiques
- ∗ Attaque par réseau mathématique
- ∗ Canaux cachés visant quelques bits du secret
- ∗ Safe-error et récupération de quelques bits secrets
- ∗ $ECDSA$ et attaques par faute
- ∗ Exemple d'architecture vulnérable
- ∗ Contremesures

La sécurité de l'information repose étroitement sur les circuits intégrés (CI). Malheureusement, les CIs sont soumis à de nombreuses menaces telles que les attaques par canaux cachés ou par injection de fautes. Ce travail se concentre sur les petites vulnérabilités et les contremesures liées à l'algorithme ECDSA. La motivation est qu'une source de vulnérabilité peut être utilisée dans différents scénarios d'attaque. En corrigeant la vulnérabilité, les attaques existantes sont évitées mais également les attaques non découvertes ou non publiées utilisant la vulnérabilité en question. De plus, bien que le scalaire sur courbe elliptique soit au cœur de la sécurité de tous les schémas cryptographiques à base de courbe elliptique, l'ensemble du système a besoin d'être sécurisé. Une vulnérabilité concernant simplement quelques bits de secret peut suffire à récupérer la clef privée et donc doit être évitée.
L'ECDSA peut être implémenté de différentes façons, en logiciel ou via du matériel dédié ou un mix des deux. De nombreuses architectures différentes sont donc possibles pour implémenter un système à base d'ECDSA. Pour cette raison, ces travaux se concentrent principalement sur les contremesures algorithmiques.