UNIVERSITÉ DE BORDEAUX – ÉCOLE DOCTORALE DE MATHÉMATIQUES ET INFORMATIQUE

N° d'ordre : 4529

Vincent VERNEUIL

Thèse présentée pour obtenir le grade de Docteur
Spécialité : MATHÉMATIQUES PURES

Cryptographie à base de courbes elliptiques et sécurité de composants embarqués

Elliptic curve cryptography and security of embedded devices

Soutenue le 13 juin 2012 à l'Institut de Mathématiques de Bordeaux
devant la commission d'examen composée de :

Rapporteurs	**Louis GOUBIN**	Professeur, Université de Versailles
	Tanja LANGE	Professeur, Université d'Eindhoven
Président du jury	**Gilles ZÉMOR**	Professeur, Université Bordeaux 1
Directeur	**Karim BELABAS**	Professeur, Université Bordeaux 1
Co-directeur	**Christophe CLAVIER**	Professeur, Université de Limoges
Encadrant	**Benoit FEIX**	Ingénieur, Inside Secure
Examinateurs	**Éric BRIER**	Ingénieur, Ingenico
	Jean-Sébastien CORON	Assistant-professeur, Université du Luxembourg

Thèse réalisée d'avril 2009 à juin 2012 dans le cadre d'une Convention Industrielle de Formation par la REcherche (CIFRE) à l'Institut de Mathématiques de Bordeaux et à Inside Secure.

Institut de Mathématiques de Bordeaux - UMR 5251
Université Bordeaux 1
351, cours de la Libération
33405 TALENCE cedex

Inside Secure
41, parc du Golf
13856 Aix-en-Provence - cedex 3

Résumé

Les systèmes cryptographiques à base de courbes elliptiques sont aujourd'hui de plus en plus employés dans les protocoles utilisant la cryptographie à clef publique. Ceci est particulièrement vrai dans le monde de l'embarqué qui est soumis à de fortes contraintes de coût, de ressources et d'efficacité, car la cryptographie à base de courbes elliptiques permet de réduire significativement la taille des clefs utilisées par rapport à d'autres systèmes cryptographiques tels que RSA.

Les travaux qui suivent décrivent dans un premier temps l'implantation efficace et sécurisée de la cryptographie à base de courbes elliptiques sur des composants embarqués, en particulier sur des cartes à puce. La sécurisation de ces implantations nécessite de prendre en compte les attaques physiques dont un composant embarqué peut être la cible. Ces attaques incluent notamment les analyses par canaux auxiliaires qui consistent à étudier le comportement d'un composant qui manipule une clef secrête pour en déduire de l'information sur celle-ci, et les analyses par faute dans lesquelles un attaquant peut perturber le fonctionnement d'un composant dans le même but.

Dans la seconde partie de ce mémoire de thèse, nous étudions ces attaques et leurs conséquences concernant l'implantation des systèmes cryptographiques à clef publique les plus répandus. De nouvelles méthodes d'analyse et de nouvelles contre-mesures sont proposées pour ces systèmes cryptographiques, ainsi que des attaques spécifiques à l'algorithme de chiffrement par bloc AES.

Mots clefs : cryptographie à base de courbes elliptiques, analyse par canaux auxilliaires, RSA, multiplication scalaire, exponentiation.

Vincent Verneuil

—

Ph.D. Thesis

—

Elliptic curve cryptography and security of embedded devices

Abstract

Elliptic curve based cryptosystems are nowadays increasingly used in protocols involving public-key cryptography. This is particularly true in the context of embedded devices which are subject to strong cost, resources, and efficiency constraints, since elliptic curve cryptography requires significantly smaller key sizes compared to other cryptosystems such as RSA.

The following study focuses in the first part on secure and efficient implementation of elliptic curve cryptography in embedded devices, especially smart cards. Designing secure implementations requires to take into account physical attacks which can target embedded devices. These attacks include in particular side-channel analysis which may infer information on a secret key manipulated from a component by monitoring how it interacts with its environment, and fault analysis in which an adversary can disturb the normal functioning of a device with the same goal.

In the second part of this thesis, we study these attacks and their impact on the implementation of the most used public-key cryptosystems. In particular, we propose new analysis techniques and new countermeasures for these cryptosystems, together with specific attacks on the AES block cipher.

Keywords: elliptic curve cryptography, side-channel analysis, RSA, scalar multiplication, exponentiation.

Université de Bordeaux
Bordeaux, France, 2012.

Remerciements

Je remercie en tout premier lieu mes encadrants de thèse qui m'ont permis de mener à bien ces travaux. Mon directeur de thèse Karim Belabas m'a été d'un grand secours tout au long des ces trois années. Cette thèse doit beaucoup à ses conseils, à sa rigueur et aux lumières qu'il m'a apportées. Mon co-directeur de thèse Christophe Clavier, qui n'a pas son pareil pour faire de la recherche un jeu, m'a régulièrement prodigué ses bons conseils et a su exciter ma curiosité quand c'était nécessaire. Son calme et sa gentillesse m'ont été d'un grand réconfort. Enfin, Benoit Feix, mon responsable scientifique à Inside Secure, qui est à la fois un chef d'équipe formidable, un partenaire de recherche stimulant et un ami, m'a fait profiter de ses vastes connaissances et a soutenu jour après jour mes efforts dans l'environnement parfois complexe de l'innovation industrielle. Je n'imagine pas aujourd'hui avoir pu accomplir ces travaux de thèse sans l'assistance de ces trois personnes remarquables.

J'adresse également mes plus vifs remerciements aux membres du jury qui ont accepté d'évaluer mon travail, dont les rapporteurs Tanja Lange et Louis Goubin, ainsi qu'Eric Brier, Jean-Sébastien Coron et Gilles Zémor.

Je souhaite remercier sincèrement Mylène Roussellet et Georges Gagnerot, mes coéquipiers à Inside Secure. Au sein de cette équipe, emmenée par Benoit, et à laquelle Christophe a régulièrement prêté renfort, il a été si doux de travailler au rythme des petits-déjeuners improvisés que je n'ai pas vu passer ces trois années.

Merci à tous mes professeurs de Master et de Mastère spécialisé ; j'adresse une pensée particulière à Henri Cohen qui a enseigné les premiers rudiments de théorie des courbes elliptiques à notre promotion bordelaise. Merci à Christophe Giraud pour m'avoir mis le pied à l'étrier dans le monde de la carte à puce et avoir été un formidable co-auteur. Merci à Sean Commercial pour ses relectures attentives. Merci également à Andreas Enge et Gilles Zémor pour les bavettes taillées au détour d'un couloir de l'IMB.

Je remercie enfin tout particulièrement Lucie, pour nous avoir supportés et soutenus avec tendresse, ma thèse et moi, pendant tois ans, ainsi que ma famille et mes amis qui m'ont apporté un soutien sans faille.

à Lucie

Table of Contents

List of Algorithms

List of Tables

List of Figures

List of Acronyms

$\mathcal{A}$, $\mathcal{E}^e$, $\mathcal{H}$, $\mathcal{J}$, $\mathcal{J}^m$ Affine, extended twisted Edwards homogeneous, homogeneous, Jacobian, and modified Jacobian coordinates.

$\mathcal{J}/\mathcal{J}^m$ Mixed Jacobian and modified Jacobian coordinates strategy for right-to-left scalar multiplication algorithms.

$\mathcal{J}^m/\mathcal{J}$ Mixed Jacobian and modified Jacobian coordinates strategy for left-to-right scalar multiplication algorithms.

2ODPA Second Order Differential Power Analysis.

AES Advanced Encryption Standard.

ANSI American National Standards Institute.

BRIP Basic Random Initial Point.

CMOS Complementary Metal–Oxide–Semiconductor.

CoCo Collision-Correlation.

CPA Correlation Power Analysis.

CPU Central Processing Unit.

CRT Chinese Remainder Theorem.

DEMA Differential ElectroMagnetic Analysis.

DES Data Encryption Standard.

DFA Differential Fault Analysis.

DPA Differential Power Analysis.

DSA Digital Signature Algorithm.

DSCA Differential Side-Channel Analysis.

EAC Euclidean Addition Chain.

ECC Elliptic Curve Cryptography.

ECDSA Elliptic Curve Digital Signature Algorithm.

EMV Europay, MasterCard and VISA.

FIPS Federal Information Processing Standard.

HODPA Higher-Order Differential Power Analysis.

HW Hamming Weight.

ICAO International Civil Aviation Organization.

IEEE Institute of Electrical and Electronics Engineers.

ISO International Organization for Standardization.

MIA Mutual Information Analysis.

NAF Non-Adjacent Form.

NIST National Institute of Standards and Technology.

NSA National Security Agency.

RAM Random-Access Memory.

RISC Reduced Instruction Set Computing.

RSA Rivest-Shamir-Adleman.

SECG Standards for Efficient Cryptography Group.

SEMA Simple ElectroMagnetic Analysis.

SNR Signal-to-Noise Ratio.

SPA Simple Power Analysis.

SSCA Simple Side-Channel Analysis.

ZADDC Conjugate co-Z ADDition and Update.

ZADDC' $(X:Y)$-only Conjugate co-Z ADDition and Update.

ZADDU Co-Z ADDition and Update.

ZADDU' $(X:Y)$-only co-Z ADDition and Update.

Introduction

Motivation

Embedded devices are more and more present in everyday life and will probably become ubiquitous with the emergence of cloud computing. Numerous applications already rely on microchips for security features and to protect sensitive assets; for example we may cite e-payment, e-passport, pay TV, access control, trusted computing, e-purse, anti-counterfeiting, etc. Smart cards have been used to provide such services since the French *Télécarte* pay phone card was launched in 1983. While these first cards used very simple memory chips, the *Carte Bleue* debit card introduced in 1992 and the massive use of SIM cards in GSM mobile phones in the 1990s marked the beginning of a wide use of cards embedding a real microprocessor. Their features have substantially evolved since then, so that they have now significant processing capabilities and various form factors. Notably, most of them can store secret keys in tamper-proof areas, and high-end smart cards are able to use public-key cryptography involving computations over integers of hundreds of digits.

Currently, the most widespread public-key cryptosystem is the well-known Rivest-Shamir-Adleman (RSA) system which relies on the difficulty of factoring large integers. Since significant advances in this field have been made over the last two decades, keys of thousands of bits are required today. As a consequence, Elliptic Curve Cryptography (ECC) which is considered to provide an equivalent security with much shorter keys is gaining ground over RSA, especially in the context of embedded devices that have limited resources. The main operation of ECC is the *scalar multiplication* which is similar in many ways to the modular exponentiation used in RSA. Its implementation raises efficiency and security issues that have been covered by an abundant literature. Not all these works fit the context of embedded devices however.

Since their introduction, smart cards have been considered as tamper-proof devices — they are often referred to as *digital strong-boxes* — and security properties of cryptographic algorithms can be proven under mathematical assumptions. However, the emergence of *side-channel analysis* from 1996 onwards has shown that embedded systems provided with state-of-the-art cryptography may not be as secure as initially thought. An integrated circuit performing computations physically interacts with its environment so that various kinds of leakage can occur when it manipulates a secret key. As a consequence, straightforward implementations of cryptographic algorithms in which leakages are not taken into account are generally insecure when embedded in a device potentially owned by an adversary.

Fault analysis demonstrated another issue for the security of embedded devices: as it is not possible to control the environment in which these systems operate, they may be subject to voluntary perturbations. Many works have shown how corrupting computations performed by a device or data stored in its memories can help an adversary to extract secret keys out of it.

These physical attacks have initiated substantial research and nowadays constitute a full field of study known as *physical cryptanalysis*. This new area involves cryptology, computer science, statistics, signal processing and engineering. For 15 years, new attacks have been successively demonstrated as the review of existing implementations by the community progressed. In the same time, numerous countermeasures have been devised by embedded systems designers to prevent information leakages and to resist fault attacks. These countermeasures can take place either at hardware, software or mathematical levels; this thesis focuses on the last two.

Contributions and Outline

The aim of this thesis is twofold: first, Chapter 1 focuses on the implementation of ECC in existing smart cards with practical industrial constraints. We detail the options suitable for low-resources embedded devices, with regard to the standards used in real-life applications. Since most previous works do not fit this context, we estimate the cost of these solutions depending on devices characteristics. Finally, we propose a new method which is both efficient and resistant against known attacks.

Second, we detail in Chapter 2 how physical cryptanalysis threatens embedded systems that integrate cryptographic functionalities. Our study focuses on public-key cryptography but covers also some aspects of the implementation of the Advanced Encryption Standard (AES), the most used secret-key block cipher. We show that protecting devices against side-channel analysis requires both high-level and low-level implementations to be considered. In particular, we demonstrate how modular multiplication — the basic operation in both ECC and RSA — can be targeted by side-channel analysis, and how sound countermeasures are inefficient when improperly implemented. As a consequence, we give recommendations to implementors and propose new countermeasures at different implementation levels.

Specifically, key contributions of this thesis include:

- In Section 1.2, a comprehensive survey of the implementation of ECC scalar multiplication in embedded devices. In particular we evaluate the cost of available algorithms when taking field additions into account, which is relevant in the context of embedded devices. This survey addresses both algorithmic efficiency and resistance towards side-channel analysis.

- In Section 1.3, a new implementation of the atomicity countermeasure to protect scalar multiplication against side-channel analysis. Considering the context of embedded devices, this method brings the best efficiency to our knowledge in some realistic cases for general curves over large characteristic prime fields [GV10].

- In Section 2.5, new exponentiation algorithms using squarings only and implementing the atomicity principle. These new atomic algorithms are more resistant to attacks than previous atomic methods, while remaining faster than known regular algorithms. We also show that parallelizing squarings makes our method one of the most efficient to our knowledge [CFG+11g].

- In Section 2.6, a new side-channel analysis on an RSA exponentiation. It computes correlation estimates, which are generally used by side-channel analysis to infer information on a secret key from numerous exponentiation traces. Our method allows a secret exponent to be recovered from a single trace under realistic assumptions [CFG+10].

- In Section 2.7, new countermeasures operating at the multiplication implementation level. They are intended mainly for hardware designers and provide immunity against a wide range of side-channel analysis, including the one introduced in Section 2.6 [CFG+10].

- In Section 2.8, a new side-channel analysis on the AES using collisions between trace segments. This attack is shown to be successful against classical countermeasures when realistic security trade-offs are used to improve the efficiency of implementations. It may thus threaten devices used in practice [CFG+11b].

Chapter 1

Elliptic Curve Cryptography on Embedded Devices

Elliptic Curve Cryptography (ECC) provides public-key primitives using much shorter key lengths for a given security level than other cryptosystems such as RSA, Digital Signature Algorithm (DSA), or Diffie-Hellman. This is a decisive advantage in the context of embedded devices where resources (power, memory, frequency, bandwidth, etc.) are generally limited. Thus, many applications[1] are currently switching to ECC as security requirements increase over the years and traditional key lengths become prohibitive in the embedded context.

For cryptographic applications, elliptic curves are generally defined over prime fields $\mathbb{F}_p$, $p > 3$, or binary fields $\mathbb{F}_{2^n}$. In this study, we leave aside the latter case since elliptic curves over $\mathbb{F}_p$ are generally preferred in applications[2]. Section 1.1 recalls the basics of elliptic curve point arithmetic over prime fields and its implementation.

Scalar multiplication is the key operation for ECC. A substantial literature focuses on the implementation of this operation with respect to efficiency and security constraints. Section 1.2 recalls most of the proposed methods and evaluates their cost in the context of embedded devices. We emphasize that the classical assumptions found in the literature do not fit the context of embedded devices provided with a cryptographic coprocessor, such as most smart cards designed for public-key cryptography. Indeed field additions are generally neglected when estimating the cost of algorithms, whereas we demonstrate that it is worthwhile to consider them also, cf. Section 1.1.1.6. To our knowledge, we provide the most comprehensive comparison regarding this point.

In Section 1.3 we propose a new implementation of the scalar multiplication. Our algorithm is designed to resist classical side-channel analysis and exhibits some of the best performances of protected methods for general curves over $\mathbb{F}_p$ with regard to the survey provided in Section 1.2.

Finally, in Section 1.4 we consider Edwards curves, elliptic curves with a particular shape, that have been recently put under the spotlight due to the efficient arithmetic

[1] E.g. the EMV banking protocol [EMV07], e-passport ICAO specification [ICAO06], etc.

[2] Part of it is the fact that the use of ECC over binary fields is restricted by several patents in practice. Also the NSA selected the prime moduli option for the Suite B cryptographic set [NSA05].

and the nice implementation properties they provide. We discuss the opportunity to use Edwards model in the context of standard-compliant products.

1.1 Point Arithmetic

In this section, we recall the point arithmetic of elliptic curves over large characteristic fields and some of the numerous methods used to speed up the computation of point additions and doublings. We also evaluate the cost of such operations in the context of embedded devices.

Section 1.1.1 presents some general statements about elliptic curve arithmetic using the affine representation. Projective representations that are generally used in implementations are detailed in Section 1.1.2 with the corresponding operations formulas. Careful attention is paid to recall all known formulas, including those operating under particular conditions such as readditions, mixed additions, co-Z additions, etc. Finally, Section 1.1.3 briefly recalls some unified addition formulas of interest for cryptographic applications.

1.1.1 Generalities

We recall hereafter some properties of elliptic curves over large characteristic fields and basic facts concerning their use in cryptographic applications. Also, we discuss the actual cost of field operations when performed on embedded devices provided with a cryptographic coprocessor. This point yields a key observation for the sequel of this study.

1.1.1.1 Equation and Group Law

An elliptic curve over a field $\mathbb{K}$ of characteristic $\mathrm{char}(\mathbb{K}) \neq 2, 3$, is defined by an affine equation of the form:

$$y^2 = x^3 + ax + b \tag{1.1}$$

where a, b are elements of $\mathbb{K}$ such that $4a^3 + 27b^2 \neq 0$. Equation (1.1) is called *short Weierstraß* equation of the curve.

Let $\mathcal{E}(\mathbb{K})$ denote the set of projective points $(X:Y:Z)$ on the curve with coordinates in $\mathbb{K}$. It corresponds to the set of affine points (x, y) of $\mathbb{K}^2$ satisfying the equation of $\mathcal{E}$ together with the projective point $\mathcal{O} = (0:1:0)$, the *point at infinity*. The set $\mathcal{E}(\mathbb{K})$ has an abelian group structure considering the *chord and tangent* group law denoted additively and described hereafter:

- $\mathcal{O}$ is the neutral element for the group law,
- the opposite of $P = (x, y)$ is $-P = (x, -y)$,

- for $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ two affine points, $P_1 \neq \pm P_2$, the point $(x_3, y_3) = P_1 + P_2$ is given by:

$$\begin{cases} x_3 = m^2 - x_1 - x_2 \\ y_3 = m\,(x_1 - x_3) - y_1 \end{cases} \quad \text{with} \quad m = \frac{y_2 - y_1}{x_2 - x_1} \tag{1.2}$$

  this formula is referred to as the affine *addition* formula,

- for $P_1 = (x_1, y_1)$ with $y_1 \neq 0$, the point $(x_3, y_3) = P_1 + P_1$ is given by:

$$\begin{cases} x_3 = m^2 - 2x_1 \\ y_3 = m\,(x_1 - x_3) - y_1 \end{cases} \quad \text{with} \quad m = \frac{3{x_1}^2 + a}{2y_1} \tag{1.3}$$

  this formula is referred to as the affine *doubling* formula.

This group law, so called because of its nice geometric representation, is depicted in Figure 1.1 on the elliptic curve $y^2 = x^3 - x + 1$ defined over $\mathbb{R}$.

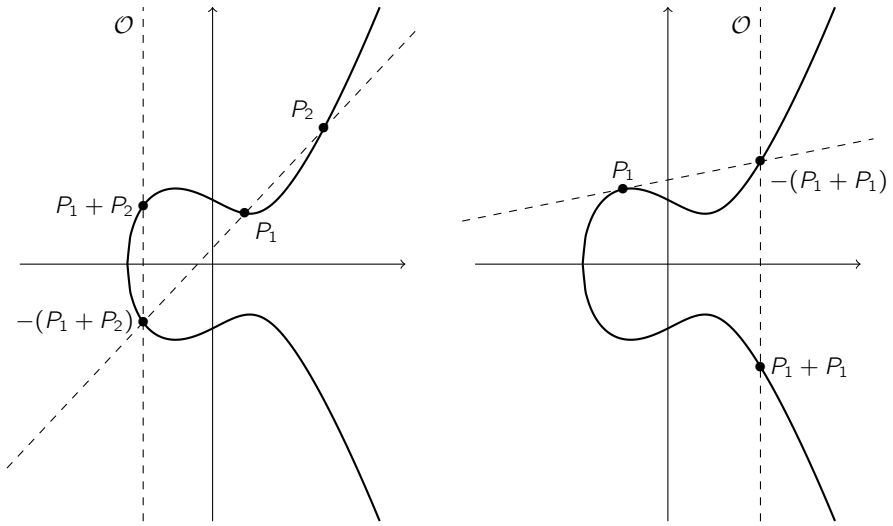


Figure 1.1: Geometric representation of the chord and tangent group law over $\mathbb{R}$

Choice of Parameters The groups of points of the curves given by $\mathcal{E}_1 : y^2 = x^3 + a_1 x + b_1$ and $\mathcal{E}_2 : y^2 = x^3 + a_2 x + b_2$ over $\mathbb{F}_q$ of characteristic $p > 3$ and $a_1, b_1 \neq 0$, are isomorphic if and only if there exists v in $\mathbb{F}_q^*$ such that $a_1 = v^4 a_2$ and $b_1 = v^6 b_2$. The corresponding isomorphism is then:

$$\begin{aligned} \mathcal{E}_1(\mathbb{F}_q) &\to \mathcal{E}_2(\mathbb{F}_q) \\ (x, y) &\mapsto \left(v^2 x, v^3 y\right) \end{aligned}$$

Therefore, if a_2 is a more convenient value for computing the doubling of a point, one can perform point operations on $\mathcal{E}_2$ instead of $\mathcal{E}_1$. This property is used in standards [NIST06; LM10] which restrict themselves to isomorphism classes containing a curve with $a = -3$.

1.1.1.2 Scalar Multiplication

Given P a point of $\mathcal{E}(\mathbb{F}_q)$ and $k \in \mathbb{N}^*$, let kP be the point of the subgroup generated by P defined by:

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

This operation is called the *scalar multiplication* of the point P by k. This definition extends naturally to $k \in \mathbb{Z}$ with $0P = \mathcal{O}$ and $(-k)P = k(-P)$.

In the sequel, we restrict ourselves to the implementation of elliptic curves over a field $\mathbb{F}_q$ of large characteristic p.

1.1.1.3 Security and Key Lengths

In the same way as ElGamal cryptosystem is based on the discrete logarithm problem in the multiplicative group of a finite field, ECC is based on the difficulty to compute discrete logarithms in the group of points of an elliptic curve. As the matter of comparison, RSA encryption relies on the difficulty of the problem of factoring large integers.

The discrete logarithm problem in a multiplicative group G is the following: given an element g of G and x an element of the subgroup of order n_g generated by g, find the integer d such that $0 \leq d \leq n_g - 1$ and $x = g^d$. Using the additive notation, the elliptic curve discrete logarithm problem consists in finding d such that $Q = dP$ for P a point of the curve, and Q a point of the subgroup generated by P.

Unlike the discrete logarithm on $\mathbb{F}_q$ or the factorization of integers, no known algorithm solves in sub-exponential time the discrete logarithm problem on an elliptic curve. The fastest known general method is the Pollard ρ algorithm, which heuristically uses $O(n^{\frac{1}{2}})$ group additions.

Consequently, it is considered that ECC provides an equivalent security level as ElGamal or RSA with much shorter keys. Table 1.1 gives the commonly accepted equivalences between these cryptosystems key lengths [HMV03]. Note however that the estimates vary a little depending on the source [Gir08]. A security level s is achieved when we estimate that solving the instance will require more that 2^s operations.

Table 1.1: Comparison of estimated equivalent key lengths between ECC and RSA for different security levels

Security level	80	112	128	192	256
ECC	160	224	256	384	512
RSA	1024	2048	3072	8192	15360

1.1.1.4 Cardinality

Let $n = \#\mathcal{E}(\mathbb{F}_q)$ denote the cardinality of the group of points $\mathcal{E}(\mathbb{F}_q)$. Define t, the value such that $n = q + 1 - t$, as the *trace* of the curve. Then, Hasse's theorem states that:

$$|t| \leq 2\sqrt{q} \tag{1.4}$$

In other words, the cardinality of the group of points is close to q and bounded by:

$$(\sqrt{q} - 1)^2 \leq n \leq (\sqrt{q} + 1)^2 \tag{1.5}$$

For cryptographic applications, it is required that the cardinality of the group of points of a curve has a large prime factor, otherwise the discrete logarithm problem becomes easier using Pohlig-Hellman algorithm [PH78; MOV97]. In practice, considered groups have cardinality $n = hm$, with m prime and $h \in \{1, 2, 3, 4\}$. The value h is called the *cofactor* of the group of points of the curve.

The curve is *supersingular* if $\text{char}(\mathbb{F}_q)$ divides t, and *ordinary* otherwise. Since the discrete logarithm problem on the group of points of supersingular elliptic curves succumbs to *pairing* attacks [CFA+05], these curves are avoided for the cryptographic applications considered in this work. Pairing based cryptography is another branch of ECC which uses some properties of supersingular curves to provide new cryptographic primitives such as identity based cryptography [CS11].

1.1.1.5 Standard Curves

There are currently two main standards defining elliptic curves for cryptography: the NIST standard FIPS 186-3 [NIST06] and the German Brainpool standard [LM10]. Indeed, other standards such as ANSI X9.62 [ANSI05], ISO 15946 [ISO02], IEEE P1363 [IEEE04], and SECG [Cer00] mainly provide pointers to NIST curves.

Fast and secure implementations of the elliptic curve scalar multiplication for industrial products are optimized for NIST and/or Brainpool curves. There are 5 NIST curves over prime fields whose parameters sizes are 192, 224, 256, 384, and 521 bits and 7 Brainpool curves over prime fields whose parameters sizes are 160, 192, 224, 256, 320, 384, and 512 bits. All theses curves have cofactor 1: n is prime.

1.1.1.6 Estimating Computations Cost

In this study, we are interested in estimating the cost of different scalar multiplication techniques over a prime field $\mathbb{F}_p$ on an embedded device such as a smart card. On such devices, most of the computation time of a scalar multiplication is spent in the point arithmetic, i.e. the computation of field operations. Thus, other operations (assignments, loop processing, conditional branchings, etc.) have a negligible cost.

Field operations include additions, subtractions, multiplications and inversions. The main one in term of cost and usage is the multiplication. Denoting M, resp. S, the cost of a modular multiplication, resp. modular squaring, it is generally considered that $S/M = 0.8$ for operands of a few hundreds of bits. The modular inversion is very

expensive compared to the multiplication: we have generally observed on smart cards that $I/M \approx 100$, I being the cost of a modular inversion.

The following study focuses on embedded devices provided with a dedicated arithmetic coprocessor. This is the case of most smart cards used for public-key applications, e.g. Inside Secure (ex-Atmel SMS) AT90SC, NXP SmartMX, or Infineon SLE66CX series.

In the literature, the cost of field additions and subtractions is generally neglected compared to the cost of multiplications. While this assumption is asymptotically correct, these operations are not as insignificant as predicted in the context of embedded devices. For example, the aforementioned smart cards provided with arithmetic coprocessors offer the following operations: addition, subtraction, multiplication, modular multiplication and sometimes modular squaring. To our knowledge, only one chip is currently equipped with a hardware modular addition[3]. On all other devices, modular additions, resp. subtractions, are therefore carried out using one regular addition, resp. subtraction, and one conditional subtraction, resp. addition[4]. In practice, this conditional operation should always be performed for side-channel analysis immunity — i.e. a dummy operation is performed half of the time. Moreover every operation performed by the coprocessor requires a constant extra software processing to configure and start the coprocessor.

It stems from these features that the cost of field additions/subtractions is in practice not negligible compared to field multiplications. Figure 1.2 is an electromagnetic leakage measurement during the execution on a SmartMX chip of a 192-bit modular multiplication followed by a modular addition. Large amplitude blocks represent the 32-bit arithmetic coprocessor activity while those with smaller amplitude are CPU processing only. On this figure, we denote by δ the time spend by CPU instructions for filling the coprocessor registers and triggering a multi-precision operation. In this case the timing ratio between modular multiplication and modular addition is approximately 0.3.

From experiments on the aforementioned devices, we estimated the average cost of modular additions/subtractions compared to modular multiplications. Our results are presented in Table 1.2. The average value of A/M for considered bit lengths is about 0.2.

An obvious hardware improvement for these devices would be to offer native modular additions and subtractions, which saves one software processing δ per modular addition/subtraction. We thus also consider in the sequel of this study the case $A/M = 0.1$. It can also be used for estimating the cost of computations using key lengths from 384 to 521 bits.

[3] The reference of this chip is not publicly available.

[4] On some devices, the reduction following an addition or subtraction can be delayed if the following operation is a multiplication or a squaring. Also, it is possible on some devices to compute an addition using the CPU during a squaring or a multiplication performed by the coprocessor, i.e. virtually for free. We do not consider such features in the following study as they vary a lot from a chip to another.

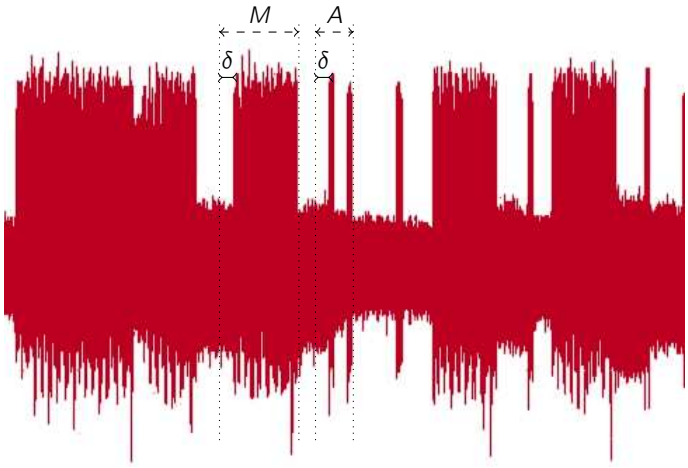


Figure 1.2: Comparison between modular multiplication and modular addition timings for 192-bit operands on a smart card designed for public-key cryptography

Table 1.2: Average A/M ratio observed on smart cards provided with an arithmetic coprocessor for standard ECC key lengths

Bit length	160	192	224	256	320	384	512	521
A/M	0.36	0.30	0.25	0.22	0.16	0.13	0.09	0.09

1.1.2 Projective Representations

The affine addition and doubling formulas (1.2) and (1.3) require the computation of an inverse in the base field $\mathbb{F}_q$. As discussed in Section 1.1.1.6, if q is prime, modular inversions are very expensive operations, compared to modular multiplications. For that reason, most implementations perform internal operations using a redundant projective representation which requires no inversion to add two points. As detailed hereafter, a scalar multiplication computed using projective coordinates requires to eventually perform a single inversion to output the result in affine representation.

We recall hereafter most of the projective formulas found in the literature. This survey makes intensive use of the usefull *Explicit Formula Database* [EFD] available at: `http://hyperelliptic.org/EFD/`.

1.1.2.1 On S–M Trade-Offs

Addition and doubling formulas presented hereafter are voluntarily not state-of-the-art [EFD]. Indeed, recent advances have provided projective formulas where some field multiplications have been traded for faster field squarings [Ber01; Lon07, Sec. 4.1]. These advances have been achieved by using the so-called *S–M trade-off* principle which is based on the fact that computing ab when a^2 and b^2 are known can be done as $2ab = (a+b)^2 - a^2 - b^2$. Thus, a squaring can replace a multiplication and the additional factor 2 can be handled by considering the representative $(2X : 2Y : 2Z)$ of the homogeneous coordinates coset[5] $(X : Y : Z)$.

Nevertheless such trade-offs not only replace field multiplications by field squarings but also add field additions. In the previous example, 3 extra additions have to be performed[6], thus the trade-off is profitable only if $A/M < (1 - S/M)/3$, i.e. $A/M < 0.067$ taking $S/M = 0.8$. We show in Section 1.1.1.6 that it is not the case for the considered devices. In practice, it also depends on the considered hardware capabilities which highly vary from a chip to another.

Therefore, we consider that these new formulas are generally not relevant in the context of embedded devices — especially smart cards —, but they would be of interest if modular additions could be performed at a lower cost.

1.1.2.2 Homogeneous Projective Coordinates

By denoting $x = X/Z$ and $y = Y/Z$, $Z \neq 0$, we obtain the homogeneous projective Weierstraß equation of the elliptic curve $\mathcal{E}$:

$$Y^2Z = X^3 + aXZ^2 + bZ^3 \tag{1.6}$$

Each affine point (x, y) is represented by homogeneous projective coordinates $(\lambda x : \lambda y : \lambda)$ with $\lambda \in \mathbb{F}_q^*$. Conversely, every point represented by $(X : Y : Z)$, $Z \neq 0$, has affine coordinates $(x, y) = (X/Z, Y/Z)$.

[5] In the case of Jacobian coordinates, the representative $(4X : 8Y : 2Z)$ is considered.

[6] If Jacobian coordinates are used, more additions may be required to compute $(4X : 8Y : 2Z)$. In such a case, the A/M threshold is even lower.

The opposite of a point $(X:Y:Z)$ is $(X:-Y:Z)$ and the point at infinity $\mathcal{O}$ is represented by $(0:\lambda:0)$, $\lambda \in \mathbb{F}_q^*$, the unique coset in $\mathcal{E}(\mathbb{F}_q)$ with $Z = 0$.

The sum of $P_1 = (X_1 : Y_1 : Z_1)$ and $P_2 = (X_2 : Y_2 : Z_2)$, with $Z_1, Z_2 \neq 0$ and $P_1 \neq \pm P_2$, is the point $(X_3:Y_3:Z_3)$ such that:

$$\begin{cases} X_3 = BC \\ Y_3 = A\left(B^2X_1Z_2 - C\right) - B^3Y_1Z_2 \\ Z_3 = B^3Z_1Z_2 \end{cases} \quad \text{with} \quad \begin{array}{l} A = Y_2Z_1 - Y_1Z_2 \\ B = X_2Z_1 - X_1Z_2 \\ C = A^2Z_1Z_2 - B^3 \\ \qquad -2B^2X_1Z_2 \end{array} \tag{1.7}$$

If P_1 is given in affine coordinates — i.e. $Z_1 = 1$ — we save three multiplications in (1.7). Such a case is referred to as *mixed affine-projective* addition.

The double of the point $(X_1:Y_1:Z_1)$ is the point $(X_2:Y_2:Z_2)$ such that:

$$\begin{cases} X_2 = EB \\ Y_2 = A(D - E) - 2C^2 \\ Z_2 = B^3 \end{cases} \quad \text{with} \quad \begin{array}{l} A = 3{X_1}^2 + a{Z_1}^2 \\ B = 2Y_1Z_1 \\ C = BY_1 \\ D = 2CX_1 \\ E = A^2 - 2D \end{array} \tag{1.8}$$

When curve parameter a is -3, a doubling can be carried out using:

$$A = 3(X_1 + Z_1)(X_1 - Z_1)$$

which saves two squarings in the above formula. We denote this operation by *fast doubling*.

Adding up field operations yields $12M + 2S + 7A$ for general addition, $9M + 2S + 7A$ for mixed addition, $7M + 5S + 10A$ for general doubling and $7M + 3S + 11A$ for fast doubling.

1.1.2.3 Jacobian Projective Coordinates

By denoting $x = X/Z^2$ and $y = Y/Z^3$, $Z \neq 0$, we obtain the Jacobian projective Weierstraß equation of the elliptic curve $\mathcal{E}$:

$$Y^2 = X^3 + aXZ^4 + bZ^6 \tag{1.9}$$

Each affine point (x, y) is represented by Jacobian projective coordinates $(\lambda^2 x : \lambda^3 y : \lambda)$ with $\lambda \in \mathbb{F}_q^*$. Conversely, every point represented by $(X : Y : Z)$, $Z \neq 0$, has affine coordinates $(x, y) = (X/Z^2, Y/Z^3)$.

The opposite of a point $(X:Y:Z)$ is $(X:-Y:Z)$ and the point at infinity $\mathcal{O}$ is represented by $(\lambda^2:\lambda^3:0)$, $\lambda \in \mathbb{F}_q^*$, the unique coset with $Z = 0$.

The sum of $P_1 = (X_1 : Y_1 : Z_1)$ and $P_2 = (X_2 : Y_2 : Z_2)$, with $Z_1, Z_2 \neq 0$ and $P_1 \neq \pm P_2$, is the point $(X_3 : Y_3 : Z_3)$ such that:

$$\begin{cases} X_3 = F^2 - E^3 - 2AE^2 \\ Y_3 = F\left(AE^2 - X_3\right) - CE^3 \\ Z_3 = Z_1 Z_2 E \end{cases} \quad \text{with} \quad \begin{array}{l} A = X_1 {Z_2}^2 \\ B = X_2 {Z_1}^2 \\ C = Y_1 {Z_2}^3 \\ D = Y_2 {Z_1}^3 \\ E = B - A \\ F = D - C \end{array} \tag{1.10}$$

If P_1 is given in affine coordinates (i.e. $Z_1 = 1$) we save one squaring and four multiplications in (1.10). Besides, if P has to be added several times, storing ${Z_1}^2$ and ${Z_1}^3$ saves one squaring and one multiplication in all succeeding additions involving P. This latter case is referred to as *readdition*.

The double of the point $(X_1 : Y_1 : Z_1)$ is the point $(X_2 : Y_2 : Z_2)$ such that:

$$\begin{cases} X_2 = C^2 - 2B \\ Y_2 = C\left(B - X_2\right) - 2A^2 \\ Z_2 = 2Y_1 Z_1 \end{cases} \quad \text{with} \quad \begin{array}{l} A = 2{Y_1}^2 \\ B = 2AX_1 \\ C = 3{X_1}^2 + a{Z_1}^4 \end{array} \tag{1.11}$$

When curve parameter a is -3, two squarings can be saved in formula (1.11) using:

$$C = 3\left(X_1 + {Z_1}^2\right)\left(X_1 - {Z_1}^2\right)$$

Adding up field operations yields $12M+4S+7A$ for general addition, $11M+3S+7A$ for readdition, $8M + 3S + 7A$ for mixed addition, $4M + 6S + 11A$ for general doubling formula and $4M + 4S + 12A$ for fast doubling.

Remark 1. Consecutive general doublings can be sped-up using the following trick: store the values $U \leftarrow a{Z_1}^4$ and $V \leftarrow 2A^2$ manipulated in the first doubling, then compute $a{Z_1}^4$ in the second doubling as $2UV$ which requires only one multiplication and one addition. Iterate this process to trade two squarings for one addition per doubling after the first one. This trick has been generalized with the introduction of the so-called *modified Jacobian coordinates*, see 1.1.2.4.

Co-Z Addition Meloni observes that two Jacobian points sharing the same Z coordinate can be added efficiently [Mel07]. Indeed, if $P_1 = (X_1 : Y_1 : Z)$ and $P_2 = (X_2 : Y_2 : Z)$, such that $X_1 \neq X_2$, the sum $P_1 + P_2 = P_3$, with $P_3 = (X_3 : Y_3 : Z_3)$ can be computed as follows:

$$\begin{cases} X_3 = D - B - C \\ Y_3 = (Y_2 - Y_1)(B - X_3) - Y_1(C - B) \\ Z_3 = Z\left(X_2 - X_1\right) \end{cases} \quad \text{with} \quad \begin{array}{l} A = (X_2 - X_1)^2 \\ B = X_1 A \\ C = X_2 A \\ D = (Y_2 - Y_1)^2 \end{array} \tag{1.12}$$

This addition formula, denoted Co-Z addition, requires only $5M + 2S + 7A$, which is even cheaper than the fast doubling formula.

An interesting property of this formula is that the input point P_1 can be converted for free to another representative $P_1' = (X_1' : Y_1' : Z_3)$ after the addition. Indeed observe that $Z_3 = Z(X_2 - X_1)$, which yields $X_1' = X_1(X_2 - X_1)^2 = X_1 A$ and $Y_1' = Y_1(X_2 - X_1)^3 = Y_1(C - B)$, both of which appear as subexpressions in (1.12).

In the following ZADDU (for co-Z ADDition and Update) denotes Meloni's addition formula, such that $\mathrm{ZADDU}(P_1, P_2) = (P_3, P_1')$ using the above notations.

Conjugate Co-Z Addition The ZADDU operation is extended to a *conjugate* addition by Goundar, Joye, and Miyaji [GJM10] by observing that the previous co-Z addition can also yield $P_1 - P_2 = (X_4 : Y_4 : Z_3)$ with:

$$\begin{cases} X_4 = (Y_2 + Y_1)^2 - B - C \\ Y_4 = (Y_2 + Y_1)(B - X_4) - Y_1(C - B) \end{cases} \tag{1.13}$$

This operation, referred to as $\mathrm{ZADDC}(P_1, P_2) = (P_3, P_4)$, with P_3 defined as previously and $P_4 = (X_4 : Y_4 : Z_3)$, costs $6M + 3S + 11A$.

Composite Operations A *composite* operation denotes several uses of the group law to output a single result. The most studied composite operations have the form $dP_1 + P_2$ for small values of $d \geq 2$.

Based on a work by Ciet, Joye, Lauter and Montgomery [CJLM06], Longa and Miri [LM08b] show that computing $2P_1 + P_2$ can be performed more efficiently as $P_1 + (P_1 + P_2)$ if the Co-Z formula is used for the second addition. Indeed, if $P_1 = (X_1 : Y_1 : Z_1)$ and $P_2 = (X_2 : Y_2 : Z_2)$, with $Z_1, Z_2 \neq 0$ and $P_1 \neq \pm P_2$, the sum $2P_1 + P_2 = (X_3 : Y_3 : Z_3)$ can be computed as follows [EPAF]:

$$X_3 = U^2 - T^3 - 2X_1'T^2, \quad Y_3 = U(X_3 - X_1'T^2) - Y_1'T^3, \quad Z_3 = Z_1'T \tag{1.14}$$

Where: $\alpha_1 = Y_2 {Z_1}^3$, $\alpha_2 = Y_1 {Z_2}^3$, $\beta_1 = X_2 {Z_1}^2$, $\beta_2 = X_1 {Z_2}^2$,
$A = \alpha_1 - \alpha_2$, $B = \beta_1 - \beta_2$,
$X_1' = \beta_2 B^2$, $Y_1' = \alpha_2 B^3$, $Z_1' = Z_1 Z_2 B$,
$T = A^2 - B^3 - 3X_1'$, and $U = AT + 2Y_1'$.

This scheme also allows the computation of $4P_1 + P_2$, resp. $8P_1 + P_2$, by applying $P_1 \leftarrow 2P_1$, resp. $P_1 \leftarrow 2(2P_1)$, prior to the previous formula. We give in Table 1.3 the cost of these composite operations depending on the context of the first addition: general case (Z_2 unknown), readdition (${Z_2}^2$, ${Z_2}^3$ known), or mixed addition ($Z_2 = 1$).

This scheme allows to compute $(2k + 1)P_1 + P_2$ for $k = 1, 2, \ldots$ by iterating the previous formulas. We do not detail these operations and their costs as they are not involved in the scalar multiplication algorithms considered in this study. The interested reader can find more details in the work by Longa and Miri [LM08b].

Table 1.3: Cost of Jacobian composite operation $dP_1 + P_2$, for $d = 2, 4, 8$, depending on the addition context

Operation	General case	Readdition	Mixed addition
$2P_1 + P_2$	$17M + 6S + 13A$	$16M + 5S + 13A$	$13M + 5S + 13A$
$4P_1 + P_2$	$21M + 12S + 24A$	$20M + 11S + 24A$	$17M + 11S + 24A$
$4P_1 + P_2$, $a = -3$	$21M + 10S + 25A$	$20M + 9S + 25A$	$17M + 9S + 25A$
$8P_1 + P_2$	$25M + 16S + 36A$	$24M + 15S + 36A$	$21M + 15S + 36A$
$8P_1 + P_2$, $a = -3$	$25M + 14S + 37A$	$24M + 13S + 37A$	$21M + 13S + 37A$

1.1.2.4 Modified Jacobian Projective Coordinates

This representation, introduced by Cohen, Miyaji, and Ono [CMO98], is derived from the Jacobian projective representation to which a fourth coordinate is added for computation convenience. In this representation, a point on the curve $\mathcal{E}$ is represented by $(X : Y : Z : aZ^4)$, where $(X : Y : Z)$ stands for the Jacobian representation.

Modified Jacobian projective coordinates provide a particularly efficient doubling formula. Indeed, the double of a point $(X_1 : Y_1 : Z_1 : W_1)$ is given by $(X_2 : Y_2 : Z_2 : W_2)$ such that:

$$\begin{cases} X_2 = A^2 - 2C \\ Y_2 = A(C - X_2) - D \\ Z_2 = 2Y_1Z_1 \\ W_2 = 2DW_1 \end{cases} \text{ with } \begin{array}{l} A = 3{X_1}^2 + W_1 \\ B = 2{Y_1}^2 \\ C = 2BX_1 \\ D = 2B^2 \end{array} \tag{1.15}$$

Hence, a doubling requires only $4M + 4S + 12A$ for all a values. On the other hand, addition is less efficient compared to Jacobian projective representation: by applying formula (1.10), we need to compute the fourth coordinate, thus adding an overhead of $1M + 2S$.

1.1.2.5 Comparison

Jacobian projective coordinates provide faster formulas than homogeneous ones, except in the case of the general addition of two points. On the other hand, modified Jacobian coordinates allow an even faster doubling than regular Jacobian coordinates, but suffer from a less efficient addition formula.

Tables 1.4 and 1.5 summarize the costs of addition, readdition, mixed affine-projective addition, doubling and fast doubling formulas for the different point representations. The affine coordinate representation is denoted by $\mathcal{A}$, the projective homogeneous coordinate by $\mathcal{H}$, the projective Jacobian coordinate by $\mathcal{J}$ and the modified Jacobian coordinate by $\mathcal{J}^m$.

Table 1.4: Cost of addition formulas depending on the point representation

Coord.	Addition	Readdition	Mixed addition
$\mathcal{A}$	$I + 2M + S + 6A$	$I + 2M + S + 6A$	–
$\mathcal{H}$	$12M + 2S + 7A$	$12M + 2S + 7A$	$9M + 2S + 7A$
$\mathcal{J}$	$12M + 4S + 7A$	$11M + 3S + 7A$	$8M + 3S + 7A$
$\mathcal{J}^m$	$13M + 6S + 7A$	$12M + 5S + 7A$	$9M + 5S + 7A$

Table 1.5: Cost of doubling formulas depending on the point representation

Coord.	Doubling	Fast doubling
$\mathcal{A}$	$I + 2M + 2S + 8A$	$I + 2M + 2S + 8A$
$\mathcal{H}$	$7M + 5S + 10A$	$7M + 3S + 11A$
$\mathcal{J}$	$4M + 6S + 11A$	$4M + 4S + 12A$
$\mathcal{J}^m$	$4M + 4S + 12A$	$4M + 4S + 12A$

1.1.3 Unified Addition Formulas

Classical point arithmetic on Weierstraß curves requires to distinguish point additions (i.e. $P_1 + P_2$, $P_1 \neq P_2$) from point doublings ($P_1 + P_1$) which may lead to side-channel leakages. In this section we explore the possibility of computing a scalar multiplication taking place on a standard curve using a *unified* addition formula, i.e. a formula handling both point additions and doublings, thus preventing side-channel leakages.

1.1.3.1 Weierstraß Curves

We recall hereafter the affine and homogeneous projective unified addition formulas presented by Billet and Joye [BJ03].

The sum of two points $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ given in affine representation, with $y_1 \neq -y_2$, is the point (x_3, y_3) such that:

$$\begin{cases} x_3 = \lambda - x_1 - x_2 \\ y_3 = \lambda(x_1 - x_3) - y_1 \end{cases} \quad \text{with} \quad \lambda = \frac{{x_1}^2 - x_1 x_2 - {x_2}^2 + a}{y_1 + y_2} \tag{1.16}$$

This formula requires $I + 2M + 2S + 8A$.

The sum of two points $P_1 = (X_1 : Y_1 : Z_1)$ and $P_2 = (X_2 : Y_2 : Z_2)$ given in homogeneous projective representation, with $Z_1, Z_2 \neq 0$ and $P_1 \neq -P_2$, is the point $(X_3 : Y_3 : Z_3)$ such that:

$$\begin{cases} X_3 = 2FI \\ Y_3 = E\,(H - 2I) - G^2 \\ Z_3 = 2F^3 \end{cases} \quad \text{with} \quad \begin{array}{l} A = X_1Z_2,\ B = X_2Z_1,\ C = A + B \\ D = Y_1Z_2 + Y_2Z_1 \\ E = C^2 - AB + a(Z_1Z_2)^2 \\ F = DZ_1Z_2,\ G = DF,\ H = CG \\ I = E^2 - H \end{array} \tag{1.17}$$

This formula requires $13M+5S+7A$ in the general case, $12M+5S+9A$ if $a = -3$, or $13M + 3S + 8A$ if $a = -1$ by setting $E = (C - Z_1Z_2)(C + Z_1Z_2) - AB$.

Note that no efficient unified addition formula using Jacobian coordinates on general Weierstraß curves has been reported so far.

1.1.3.2 Edwards Curves

Some particular classes of elliptic curves over prime fields provide more efficient unified addition formulas than general Weierstraß curves, for example Hessian curves[7] [JQ01] and extended Jacobi curves [LS01].

Recently, Edwards proposed a new class of curves [Edw07] provided with a unified addition formula among the fastest published in the literature [EFD]. Also, he provides an efficient doubling formula which can be used when side-channel indistinguishability is not required. Such curves being of the utmost interest for scalar multiplication implementors, an intensive survey has followed these publications.

We recall hereafter the best addition formulas proposed for Edwards curves. We study in Section 1.4 whether they can be used to perform scalar multiplication over general Weierstraß curves, in particular the standardized curves.

Equation and Group Law Edwards [Edw07] presents a unified addition law for elliptic curves over a field $\mathbb{K}$ of characteristic different from 2, given by an affine equation:

$$x^2 + y^2 = c^2(1 + x^2y^2) \tag{1.18}$$

where $c \in \mathbb{K}^*$, $c^4 \neq 1$. Bernstein and Lange generalize this law to a larger group of curves [BL07b], given by an equation:

$$x^2 + y^2 = c^2(1 + dx^2y^2) \tag{1.19}$$

where $c, d \in \mathbb{K}^*$, $dc^4 \neq 1$. These cover about 1/4 of all the elliptic curves over non-binary finite fields [BL07b].

[7] Contrary to the other unified formulas mentioned in this section, the Hessian unified addition [JQ01] is not *strongly unified*, i.e. it requires a particular processing to handle doublings.

The sum of (x_1, y_1) and (x_2, y_2) is the point (x_3, y_3) such that:

$$\begin{cases} x_3 &= \dfrac{x_1y_2 + y_1x_2}{c(1 + dx_1x_2y_1y_2)} \\ y_3 &= \dfrac{y_1y_2 - x_1x_2}{c(1 - dx_1x_2y_1y_2)} \end{cases} \tag{1.20}$$

This addition law is unified, it is moreover *complete* if d is not a square in $\mathbb{K}$ [BL07b]. Completeness means that denominators are never zero and there is thus no exceptional case. Note also that c can always be chosen equal to 1.

Projective Representation The point (x, y) is represented in projective coordinates by $(X:Y:Z)$ such that $x = X/Z$ and $y = Y/Z$, with $Z \neq 0$.

The sum of $(X_1:Y_1:Z_1)$ and $(X_2:Y_2:Z_2)$ is given by $(X_3:Y_3:Z_3)$ such that:

$$\begin{cases} X_3 = AF\left((X_1 + Y_1)(X_2 + Y_2) - C - D\right) \\ Y_3 = AG\left(D - C\right) \\ Z_3 = cFG \end{cases} \quad \text{with} \quad \begin{array}{l} A = Z_1Z_2,\ B = A^2 \\ C = X_1X_2,\ D = Y_1Y_2 \\ E = dCD \\ F = B - E,\ G = B + E \end{array} \tag{1.21}$$

This operation requires $10M + 1S + 1C^c + 1C^d + 7A$, where C^c, resp. C^d, denotes the cost of a multiplication by the constant c, resp. d, which can be chosen small. If $Z_2 = 1$, this cost drops to $9M + 1S + 1C^c + 1C^d + 7A$.

Inverted Coordinates As introduced by Bernstein and Lange [BL07c], the inverted coordinates of a point (x, y) are $(X : Y : Z)$ such that $x = Z/X$ and $y = Z/Y$, with $X, Y \neq 0$.

The sum of $(X_1:Y_1:Z_1)$ and $(X_2:Y_2:Z_2)$ is given by $(X_3:Y_3:Z_3)$ such that:

$$\begin{cases} X_3 = c(E + B)F \\ Y_3 = c(E - B)G \\ Z_3 = AFG \end{cases} \quad \text{with} \quad \begin{array}{l} A = Z_1Z_2,\ B = dA^2 \\ C = X_1X_2,\ D = Y_1Y_2 \\ E = CD,\ F = C - D \\ G = (X_1 + Y_1)(X_2 + Y_2) - C - D \end{array} \tag{1.22}$$

This operation requires $9M + 1S + 2C^c + 1C^d + 7A$. If $Z_2 = 1$, this cost boils down to $8M + 1S + 2C^c + 1C^d + 7A$. However, note that this addition formula is not complete contrary to the previous one.

1.1.3.3 Twisted Edwards Curves

Equation and Group Law Bernstein, Birkner, Joye, Lange, and Peters presented the twisted Edwards [BBJ+08] curves over a field $\mathbb{K}$ of characteristic different from 2 of equation:

$$ax^2 + y^2 = 1 + dx^2y^2 \tag{1.23}$$

where $a, d \in \mathbb{K}^*$, $a \neq d$. This equation covers more curves than Edwards curves and has a comparable addition cost.

The sum of $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ is the point $P + Q = (x_3, y_3)$ such that:

$$\begin{cases} x_3 &= \dfrac{x_1y_2 + y_1x_2}{1 + dx_1x_2y_1y_2} \\ y_3 &= \dfrac{y_1y_2 - ax_1x_2}{1 - dx_1x_2y_1y_2} \end{cases} \tag{1.24}$$

As with Edwards curves, this addition law is unified. It is complete if a is a square in $\mathbb{K}$ and d is not.

Projective Representation The point (x, y) is represented in projective coordinates by $(X : Y : Z)$ such that $x = X/Z$ and $y = Y/Z$, with $Z \neq 0$.

The sum of $(X_1 : Y_1 : Z_1)$ and $(X_2 : Y_2 : Z_2)$ is given by $(X_3 : Y_3 : Z_3)$ such that:

$$\begin{cases} X_3 = AF\left((X_1 + Y_1)(X_2 + Y_2) - C - D\right) \\ Y_3 = AG\left(D - aC\right) \\ Z_3 = FG \end{cases} \quad \text{with} \quad \begin{array}{l} A = Z_1Z_2,\ B = A^2 \\ C = X_1X_2,\ D = Y_1Y_2 \\ E = dCD \\ F = B - E,\ G = B + E \end{array} \tag{1.25}$$

Similarly to Edwards curves, this unified addition requires $10M+1S+1C^a+1C^d+7A$, where C^a, resp. C^d, denotes the cost of a multiplication by the constant a, resp. d, which can be chosen small. If $Z_2 = 1$, this cost drops to $9M + 1S + 1C^a + 1C^d + 7A$.

Inverted Coordinates As for Edwards curves, the inverted coordinates of a point (x, y) are $(X : Y : Z)$ such that $x = Z/X$ and $y = Z/Y$, with $X, Y \neq 0$.

The sum of $(X_1 : Y_1 : Z_1)$ and $(X_2 : Y_2 : Z_2)$ is given by $(X_3 : Y_3 : Z_3)$ such that:

$$\begin{cases} X_3 = (E + B)F \\ Y_3 = (E - B)G \\ Z_3 = AFG \end{cases} \quad \text{with} \quad \begin{array}{l} A = Z_1Z_2,\ B = dA^2 \\ C = X_1X_2,\ D = Y_1Y_2 \\ E = CD,\ F = C - aD \\ G = (X_1 + Y_1)(X_2 + Y_2) - C - D \end{array} \tag{1.26}$$

This operation requires $9M + 1S + 1C^a + 1C^d + 7A$. If $Z_2 = 1$, this cost boils down to $8M + 1S + 1C^a + 1C^d + 7A$.

Extended Coordinates This representation is introduced by Hisil, Wong, Carter, and Dawson [HWCD08]. The so-called extended coordinates of a point (x, y) are $(X : Y : T : Z)$ such that $x = X/Z$, $y = Y/Z$, and $xy = T/Z$, with $Z \neq 0$. In the following, this representation is denoted by $\mathcal{E}^e$.

The sum of $(X_1 : Y_1 : T_1 : Z_1)$ and $(X_2 : Y_2 : T_2 : Z_2)$ is given by $(X_3 : Y_3 : T_3 : Z_3)$ such that:

$$\begin{cases} X_3 = EF \\ Y_3 = GH \\ T_3 = EH \\ Z_3 = FG \end{cases} \quad \text{with} \quad \begin{array}{l} A = X_1X_2,\ B = Y_1Y_2 \\ C = dT_1T_2,\ D = Z_1Z_2 \\ E = (X_1 + Y_1)(X_2 + Y_2) - A - B \\ F = D - C,\ G = D + C \\ H = B - aA \end{array} \tag{1.27}$$

This operation requires $9M + 1C^a + 1C^d + 7A$. If $Z_2 = 1$, this cost boils down to $8M + 1C^a + 1C^d + 7A$. As previously, the formula is complete if a is a square and d is not.

In the special case $a = -1$, the formula may be rewritten as follows:

$$\begin{cases} X_3 = EF \\ Y_3 = GH \\ T_3 = EH \\ Z_3 = FG \end{cases} \quad \text{with} \quad \begin{array}{l} A = (Y_1 - X_1)(Y_2 - X_2) \\ B = (Y_1 + X_1)(Y_2 + X_2) \\ C = 2dT_1T_2,\ D = 2Z_1Z_2 \\ E = B - A,\ F = D - C \\ G = D + C,\ H = B + A \end{array} \tag{1.28}$$

This operation now requires $8M + 1C^{2d} + 9A$. If $Z_2 = 1$, this cost boils down to $7M + 1C^{2d} + 9A$.

1.2 Scalar Multiplication Algorithms

As already stated, scalar multiplication is the key operation of ECC. Therefore, strong efficiency and security requirements are attached to its implementation, especially in the context of embedded devices.

Scalar multiplication in additive groups is algorithmically analogous to exponentiation in multiplicative groups. However, some differences justify that scalar multiplication be studied on its own. For instance, the cheap point inversion within the group of points on an elliptic curve allows some algorithmic optimizations. Besides, the numerous point addition formulas require to consider both the scalar multiplication and the point arithmetic levels to design an efficient implementation. Section 1.2.1 presents classical algorithms and evaluates their efficiency depending on the underlying device.

Then, we take into account the threat of the simple side-channel analysis, cf. Section 2.2.1. Since its introduction, this technique has required to revisit the design of scalar multiplication schemes — and exponentiation schemes as well — to exhibit a regular structure to an adversary monitoring the sequence of performed operations. Section 1.2.2 recalls most of the techniques that have been proposed in this aim and evaluates their efficiency with regard to the device properties.

1.2.1 Efficient Algorithms for Embedded Devices

We present in the following classical algorithms that may be used in embedded devices to compute the scalar multiplication if the calculation is not threatened by side-channel attacks (e.g. public computation, secure environment, etc.)

We start with simple binary methods, also known as *double-and-add* algorithms. Then we recall how the use of a signed scalar representation such as the NAF can improve the efficiency of binary algorithms. Finally, we present the sliding window methods that use time-memory trade-offs to speed up the scalar multiplication if extra memory is available.

To assess the cost of algorithms presented in this section, it is generally assumed that the base point is provided in affine coordinates, which allows some improvements when using left-to-right algorithms. We deem this assumption reasonable in our context since affine coordinates are standard in most protocols.

Though recent works [DI06; LM08c; LG09] improve the theoretical cost of the scalar multiplication using various multi-base representations, we consider that they do not fit the embedded context due to the cost of precomputing a scalar representation in odd bases. Besides, it is not possible to store scalar precomputations for some protocols such as the Elliptic Curve Digital Signature Algorithm (ECDSA) [ANSI05] where the scalar is randomly generated before each scalar multiplication. For this reason such techniques are not mentioned in our study.

1.2.1.1 Simple Binary Algorithms

The two formulas (1.30) and (1.29) yield two ways to compute the scalar multiplication from the binary decomposition of the scalar. The corresponding algorithms are similar to the classical *square-and-multiply* exponentiation algorithms and are called *double-and-add* due to the additive group structure.

$$kP = k_0P + k_1 2P + \ldots + k_{l-1}2^{l-1}P \tag{1.29}$$
$$kP = k_0P + 2\left(k_1P + 2\left(\ldots + 2\left(k_{l-1}P\right)\ldots\right)\right) \tag{1.30}$$

Algorithm 1.1 scans the scalar bits from the most significant to the least significant and is thus said *left-to-right* while Algorithm 1.2 scans the scalar bits in the opposite direction and is said *right-to-left*.

Algorithm 1.1 Left-to-right double-and-add scalar multiplication

Input: $P \in \mathcal{E}(\mathbb{F}_q)$, $k = (k_{l-1}k_{l-2}\ldots k_0)_2$
Output: kP
Uses: P and Q

1: $Q \leftarrow \mathcal{O}$
2: **for** $i = l-1$ **to** 0 **do**
3: $\quad Q \leftarrow 2Q$ ▷ doubling of Q from the group law
4: $\quad$ **if** $k_i = 1$ **then**
5: $\quad\quad Q \leftarrow Q + P$ ▷ addition of P and Q from the group law
6: **return** Q

Algorithm 1.2 Right-to-left double-and-add scalar multiplication

Input: $P \in \mathcal{E}(\mathbb{F}_q)$, $k = (k_{l-1}k_{l-2}\ldots k_0)_2$
Output: kP
Uses: Q and R

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

Denoting $\mathfrak{A}$, resp. $\mathfrak{D}$, the cost (running time) of an addition, resp. a doubling, and considering an l-bit random scalar, the average cost of Algorithms 1.1 and 1.2 is:

$$\frac{l}{2}\mathfrak{A} + l\mathfrak{D}$$

Though the complexity of the two algorithms is identical in terms of point operations, they have different costs in terms of field multiplications. On one hand, the left-to-right variant allows to use the mixed affine-projective addition formula if the input point is provided in affine coordinates. On the other hand, with the right-to-left algorithm we can express the point Q in Jacobian coordinates and the point R in modified Jacobian coordinates in order to use the Jacobian addition formula at step 5 and the modified Jacobian doubling formula at step 6 as observed by Joye [Joy08]. This trick is called mixed Jacobian and modified Jacobian coordinates and denoted $\mathcal{J}/\mathcal{J}^m$ in the following.

Mixed Jacobian and modified Jacobian coordinates have been initially introduced to speed up the computation of consecutive doublings in left-to-right scalar multiplication algorithms [CMO98]. Cohen et al. distinguish three types of operations in left-to-right scalar multiplication: intermediate doublings, final doublings and additions. A final doubling is a doubling preceding an addition, all the other doublings being intermediate doublings. Observing that the fourth modified Jacobian coordinate is not used in additions, their method consists in using modified Jacobian coordinates for intermediate doublings, and Jacobian coordinates as output of final doublings, which saves $1M+1A$ in the modified Jacobian doubling formula. This way, a final doubling costs only $3M + 4S + 11A$. In the following, we denote this strategy by $\mathcal{J}^m/\mathcal{J}$.

The average computation cost of Algorithm 1.1, resp. Algorithm 1.2, is given in Table 1.6, resp. Table 1.7, depending on the cost of a modular multiplication (M), a modular squaring (S), and a modular addition or subtraction (A) for the different projective coordinate systems introduced previously.

Table 1.6: Average cost per scalar bit of Algorithm 1.1 depending on the point representation

Coord.	Cost for any a	Cost for $a = -3$
$\mathcal{H}$	$11.5M + 6S + 13.5A$	$11.5M + 4S + 14.5A$
$\mathcal{J}$	$8M + 7.5S + 14.5A$	$8M + 5.5S + 15.5A$
$\mathcal{J}^m/\mathcal{J}$	$8M + 6.5S + 15A$	

Table 1.7: Average cost per scalar bit of Algorithm 1.2 depending on the point representation

Coord.	Cost for any a	Cost for $a = -3$
$\mathcal{H}$	$13M + 6S + 13.5A$	$13M + 4S + 14.5A$
$\mathcal{J}/\mathcal{J}^m$	$10M + 6S + 15.5A$	

The detailed computation costs of Algorithms 1.1 and 1.2 are given in Table 1.16 (p. 35), depending on the ratios S/M and A/M for Jacobian and modified Jacobian representations. The homogeneous projective representation is discarded since its performances are always worse than the Jacobian ones.

Remark 2. The first loop iteration of Algorithm 1.1 can be saved if the most significant bit of k is 1. In this case, step 1 becomes $Q \leftarrow P$ and i is initialized to $l-2$ instead of $l-1$ at step 2. Similarly, the first addition of Algorithm 1.2 can be avoided by setting $Q \leftarrow k_0 P$ and $R \leftarrow 2P$ at steps 1 and 2, and initializing i to 1 in the **for** loop. Furthermore, the last doubling can be skipped if $k_{l-1} = 1$ by leaving the loop before i takes $l-1$ and adding a step $Q \leftarrow Q + R$ before the return statement. These tricks are useful in practice to save a few operations and may be applied to most scalar multiplication algorithms. However, we do not include them in most of the following descriptions for readability reasons.

Using Composite Operations One can note that in the left-to-right algorithm, the point Q is updated by $2Q + P$ when a 1 bit is processed. The main loop can thus be rewritten as:

for $i = l-1$ **to** 0 **do**
  if $k_i = 0$ **then**
    $Q \leftarrow 2Q$
  else
    $Q \leftarrow 2Q + P$

Using the Jacobian composite addition described in 1.1.2.3, the average cost of Algorithm 1.1 becomes $8.5M + 5.5S + 12A$ or $8.5M + 4.5S + 12.5A$ if $a = -3$. The corresponding detailed computation costs are given in Table 1.16 (p. 35) on the row labelled (c.o.). The average speed-up is about $9-11\,\%$ in the general case and about $4-7\,\%$ if $a = -3$.

1.2.1.2 Simple NAF Algorithms

Considering that the inversion operation $P \rightarrow -P$ is almost free, it is interesting to use signed representations in order to decrease the number of additions in the scalar multiplication.

A base b signed representation of k is $(k_{l_b-1} k_{l_b-2} \dots k_0)$ such that:

$$k = \sum_{i=0}^{l_b-1} k_i b^i \quad \text{with} \quad |k_i| < b$$

Among them the binary *Non-Adjacent Form (NAF)* is defined as follows. The NAF representation of an integer $k \in \mathbb{N}^*$ is $(k_{l-1} k_{l-2} \dots k_0)_{\text{NAF}}$ with $k_i \in \{-1, 0, 1\}$, $0 \leq i < l-1$ and $k_{l-1} = 1$ such that for all pairs of consecutive digits k_i and k_{i+1}, we have $k_i k_{i+1} = 0$.

The NAF representation has the following properties:

(i) Given $k \in \mathbb{N}^*$, $(k)_{\text{NAF}}$ is unique.

(ii) The length of the NAF representation of k is $\lfloor \log_2(k) \rfloor + 1$ or $\lfloor \log_2(k) \rfloor + 2$, i.e. it has the same length or one more digit than the binary representation of k.

(iii) The Hamming weight — the number of non-zero digits — of $(k)_{\text{NAF}}$ is always minimal among base 2 signed representations for a given k.

(iv) The average Hamming weight of l-digit NAF representations is approximately $l/3$.

The computation of the NAF representation of a positive integer is described in Algorithm 1.3. This algorithm runs in $O(l)$ and involves only cheap operations.

Algorithm 1.3 NAF representation computation

Input: $k \in \mathbb{N}^*$
Output: $(k)_{\text{NAF}}$
1: $i \leftarrow 0$
2: **while** $k \geq 1$ **do**
3: **if** $k \bmod 2 = 1$ **then**
4: $k_i \leftarrow 2 - (k \bmod 4)$
5: $k \leftarrow k - k_i$
6: **else**
7: $k_i \leftarrow 0$
8: $k \leftarrow k/2$
9: $i \leftarrow i + 1$
10: **return** $(k_{l-1}k_{l-2}\dots k_0)$

Algorithm 1.4 presents how to compute the left-to-right scalar multiplication given the NAF decomposition of the scalar. Algorithm 1.5, first presented by Joye [Joy08], is a right-to-left variant including on-the-fly computation of the scalar NAF representation. One may note that left-to-right on-the-fly computation of the NAF representation is possible as well using a lookup table [Mui04].

Algorithm 1.4 Left-to-right binary NAF scalar multiplication

Input: $P \in \mathcal{E}(\mathbb{F}_q)$, $k = (k_{l-1}k_{l-2}\dots k_0)_{\text{NAF}}$
Output: kP
Uses: P and Q
1: $Q \leftarrow \mathcal{O}$
2: **for** $i = l - 1$ **to** 0 **do**
3: $Q \leftarrow 2Q$
4: **if** $k_i = 1$ **then**
5: $Q \leftarrow Q + P$
6: **if** $k_i = -1$ **then**
7: $Q \leftarrow Q + (-P)$
8: **return** Q

Considering an l-bit random scalar, the average cost of Algorithms 1.4 and 1.5 is:

$$\frac{l}{3}\mathfrak{A} + l\mathfrak{D}$$

Algorithm 1.5 Right-to-left on-the-fly binary NAF scalar multiplication

Input: $P \in \mathcal{E}(\mathbb{F}_q)$, $k \in \mathbb{N}^*$
Output: kP
Uses: Q and R
1: $Q \leftarrow \mathcal{O}$
2: $R \leftarrow P$
3: **while** $k \geq 1$ **do**
4: **if** $k \bmod 2 = 1$ **then**
5: $u \leftarrow 2 - (k \bmod 4)$
6: $k \leftarrow k - u$
7: **if** $u = 1$ **then**
8: $Q \leftarrow Q + R$
9: **else**
10: $Q \leftarrow Q + (-R)$
11: $k \leftarrow k/2$
12: $R \leftarrow 2R$
13: **return** Q

For the same reasons as previously stated, the left-to-right and right-to-left variants have different costs. The average computation cost of Algorithm 1.4, resp. Algorithm 1.5, is given in Table 1.8, resp. Table 1.9.

Table 1.8: Average cost per scalar bit of Algorithm 1.4 depending on the point representation

Coord.	Cost for any a	Cost for $a = -3$
$\mathcal{H}$	$10M + 5.7S + 12.3A$	$10M + 3.7S + 13.3A$
$\mathcal{J}$	$6.7M + 7S + 13.3A$	$6.7M + 5S + 14.3A$
$\mathcal{J}^m/\mathcal{J}$	$6.7M + 5.7S + 14A$	

Table 1.9: Average cost per scalar bit of Algorithm 1.5 depending on the point representation

Coord.	Cost for any a	Cost for $a = -3$
$\mathcal{H}$	$11M + 5.7S + 12.3A$	$11M + 3.7S + 13.3A$
$\mathcal{J}/\mathcal{J}^m$	$8M + 5.3S + 14.3A$	

The detailed computation costs of Algorithms 1.4 and 1.5 are given in Table 1.16 (p. 35), depending on the ratios S/M and A/M for Jacobian and modified Jacobian representations.

Using Composite Operations As for the binary algorithm, Algorithm 1.4 can use the composite operation $2Q + P$ to speed-up the computations. In that case the main loop should be rewritten as:

for $i = l - 1$ **to** 0 **do**
  if $k_i = 0$ **then**
    $Q \leftarrow 2Q$
  else if $k_i = 1$ **then**
    $Q \leftarrow 2Q + P$
  else
    $Q \leftarrow 2Q + (-P)$

Using the Jacobian composite addition, the average cost of Algorithm 1.4 becomes $7M + 5.7S + 11.7A$ or $7M + 4.3S + 12.3A$ if $a = -3$. The average speed-up is about $6-8\,\%$ in the general case and about $3-5\,\%$ if $a = -3$, cf. Table 1.16 (p. 35).

1.2.1.3 Classical m-ary Algorithms

The double-and-add algorithms can easily be extended to radices $m > 2$, and are so-called m-ary scalar multiplication [Gor98]. In the context of embedded devices, m is generally a power of 2 for efficiency reasons. If $m = 2^t$ these algorithms scan t scalar bits at a time and require the precomputation of $2P, 3P, \ldots, (m-1)P$.

We do not detail this class of algorithms as they require to precompute more points than the window methods presented hereafter and have worse performances.

1.2.1.4 Window NAF Algorithms

It is possible to enhance the efficiency of scalar multiplication by precomputing some odd multiples of the input point. For example when processing bits 11, the left-to-right double-and-add algorithm performs $2(2Q + P) + P$. Observe now that an addition can be saved if $3P$ is known by computing $2(2Q) + 3P$. This idea can be generalized to larger blocks of bits — larger *window sizes* — and NAF representation [KT93; Gor98]. Moreover, if the base point is constant or known in advance these precomputations can be run off-line.

A comparable strategy is possible with right-to-left algorithms: in that case we store several intermediate results during the scalar multiplication and combine them in the end [Yao76; Möl03] — what we call postcomputations.

Two families of *window* algorithms — i.e. algorithms scanning several bits at a time — are described by Hankerson, Menezes, and Vanstone [HMV03]: the sliding window NAF and the window width-w NAF algorithms. Each of them can be implemented in left-to-right or right-to-left directions. Algorithm 1.6 hereafter is the left-to-right sliding window NAF method and takes the binary NAF representation of the scalar as input. Algorithm 1.7 presents window width-w NAF method using the right-to-left strategy and computes on-the-fly the width-w NAF representation of k.

Algorithm 1.6 Left-to-right sliding window NAF scalar multiplication

Input: $P \in \mathcal{E}(\mathbb{F}_q)$, $k = (k_{l-1}k_{l-2}\dots k_0)_{\text{NAF}}$, $w \geq 2$
Output: kP
Uses: Q, P_1, P_3, ..., and P_m with $m = 2(2^w - (-1)^w)/3 - 1$
1: $Q \leftarrow \mathcal{O}$
2: $i \leftarrow l - 1$
`Precomputations`
3: **for** $i = 1$ **to** m **by** 2 **do**
4: $\quad P_i \leftarrow iP$
`Main loop`
5: **while** $i \geq 0$ **do**
6: $\quad$ **if** $k_i = 0$ **then**
7: $\quad\quad Q \leftarrow 2Q$
8: $\quad\quad i \leftarrow i - 1$
9: $\quad$ **else**
10: $\quad\quad s \leftarrow \max(i - w + 1, 0)$
11: $\quad\quad$ **while** $k_s = 0$ **do**
12: $\quad\quad\quad s \leftarrow s + 1$
13: $\quad\quad u \leftarrow (k_i \dots k_s)_{\text{NAF}}$
14: $\quad\quad$ **for** $j = 1$ **to** $i - s + 1$ **do**
15: $\quad\quad\quad Q \leftarrow 2Q$
16: $\quad\quad$ **if** $u > 0$ **then**
17: $\quad\quad\quad Q \leftarrow Q + P_u$
18: $\quad\quad$ **if** $u < 0$ **then**
19: $\quad\quad\quad Q \leftarrow Q + (-P_{-u})$
20: $\quad\quad i \leftarrow s - 1$
21: **return** Q

Algorithm 1.7 Right-to-left on-the-fly window width-w NAF scalar multiplication

Input: $P \in \mathcal{E}(\mathbb{F}_q)$, $k = (k_{l-1}k_{l-2}\dots k_0)_2$, $w \geq 2$
Output: kP
Uses: R, Q_1, Q_3, ..., and Q_m with $m = 2^{w-1} - 1$

1: $R \leftarrow P$
2: $Q_1, Q_3, \dots, Q_m \leftarrow \mathcal{O}$
`Main loop`
3: **while** $k \geq 1$ **do**
4: **if** $k \bmod 2 = 1$ **then**
5: $t \leftarrow k \bmod 2^w$ ▷ where $k \bmod 2^w$ is in $[-2^{w-1}, 2^{w-1} - 1]$
6: **if** $t > 0$ **then**
7: $Q_t \leftarrow Q_t + R$
8: **if** $t < 0$ **then**
9: $Q_{-t} \leftarrow Q_{-t} - R$
10: $k \leftarrow k - t$
11: $R \leftarrow 2R$
12: $k \leftarrow k/2$
`Postcomputations`
13: **for** $i = 3$ **to** m **by** 2 **do**
14: $Q_1 \leftarrow Q_1 + iQ_i$
15: **return** Q_1

The cost of Algorithm 1.6, omitting the precomputations step, for an l-bit random scalar is [HMV03]:

$$\frac{l}{w + v(w)}\mathfrak{A} + l\mathfrak{D} \quad \text{with} \quad v(w) = \frac{4}{3} - \frac{(-1)^w}{3 \times 2^{w-2}}$$

The cost of Algorithm 1.7, omitting postcomputations step, for an l-bit random scalar is:

$$\frac{l}{w+1}\mathfrak{A} + l\mathfrak{D}$$

The sliding window NAF algorithm requires the storage of $(2^w - (-1)^w)/3 - 1$ extra points compared to the simple NAF method, while the window width-w NAF algorithm requires an additional storage of $2^{w-2} - 1$ points. Table 1.10 gives a comparison of the extra memory required to store points — i.e. $\#\{P_3, \dots, P_m\}$ in Algorithm 1.6 or $\#\{Q_3, \dots, Q_m\}$ in Algorithm 1.7 — for values of w up to 5.

These two window methods have a similar efficiency but allow different trade-offs between the number of saved additions and the cost of pre/postcomputations. In practice, the optimal strategy depends on the available memory — more memory allows larger window width — and on the scalar length — for a given a window width, the cost of pre/postcomputations should not cancel out the benefit of saved additions.

Table 1.10: Comparison of the number of extra points required by window algorithms for pre/postcomputations

w	2	3	4	5
Sliding window NAF (Algorithm 1.6)	0	2	4	10
Window width-w NAF (Algorithm 1.7)	0	1	3	7

As in the context of embedded devices a little memory is generally available, we restrict the following study to values of w — or windowing strategies — requiring at most 3 additional points to be stored. Namely, a windowing strategy of 1 extra point refers to the window width-w NAF algorithm with $w = 3$, a windowing strategy of 2 extra points refers to the sliding window NAF algorithm with $w = 3$, and a windowing strategy of 3 extra points refers to the window width-w NAF algorithm with $w = 4$.

Remark 3. Möller generalizes the sliding window and window NAF techniques with the *signed fractionnal window* representation [Möl03]. This representation allows the use of any set of consecutive odd digits $\{\pm 1, \pm 3, \pm 5, \ldots\}$ such that the exact amount of available storage can be used to optimize the efficiency of a scalar multiplication algorithm using a window width $w \geq 4$.

Remark 4. The trick described in Remark 2 can be used as well in Algorithms 1.6 and 1.7. In left-to-right window algorithms, the processing of the second scalar bit k_{l-2} can also be skipped since $2P$ and $3P$ are already computed. If 2 or 3 extra points are used, the processing of the third scalar bit k_{l-3} can be skipped or replaced by a single group addition or doubling, e.g. $6P = 2(3P)$. In some cases, one more bit may be processed using a single point operation, for instance $10P = 2(5P)$ but $11P$ requires two operations.

Comparing the Cost of Left-to-Right and Right-to-Left Algorithms

In Algorithm 1.6, mixed affine-projective addition formulas can be used if extra points are precomputed in affine coordinates. Thus, an inversion is required during the precomputation stage, which increases significantly its cost. An alternative solution is to use general addition or readdition formulas. Besides, if the input point is fixed and known in advance, then affine coordinates of $P_3, \ldots, P_m$ can be precomputed off-line.

The average computation cost of left-to-right window algorithms such as Algorithm 1.6 using mixed additions, resp. readditions, is given in Table 1.11, resp. Table 1.12, depending on the available memory. The average computation cost of right-to-left variants such as Algorithm 1.7 is given in Table 1.13. These costs omit pre/postcomputations.

The detailed computation costs of left-to-right window algorithms such as Algorithm 1.6 using mixed additions, resp. readditions, are given for representations $\mathcal{J}$ and $\mathcal{J}^m/\mathcal{J}$ in Table 1.17 (p. 35), resp. Table 1.18 (p. 36), depending on the ratios S/M and A/M and on the available memory. The detailed computation costs of right-to-left window NAF algorithms such as Algorithm 1.7 are given for $\mathcal{J}/\mathcal{J}^m$ in Table 1.19 (p. 36). All results presented in tables 1.17, 1.18, and 1.19 omit pre/postcomputations.

Table 1.11: Average cost per scalar bit of left-to-right window algorithms using mixed affine-projective additions depending on the point representation and on the window strategy (the number of extra points)

Extra pts.	Coord.	Cost for any a	Cost for $a = -3$
1	$\mathcal{J}$	$6M + 6.8S + 12.8A$	$6M + 4.8S + 13.8A$
	$\mathcal{J}$ (c.o.)	$6.3M + 5.8S + 11.5A$	$6.3M + 4.3S + 12.3A$
	$\mathcal{J}^m/\mathcal{J}$	$6M + 5.3S + 13.5A$	
2	$\mathcal{J}$	$5.8M + 6.7S + 12.6A$	$5.8M + 4.7S + 13.6A$
	$\mathcal{J}$ (c.o.)	$6M + 5.8S + 11.4A$	$6M + 4.2S + 12.2A$
	$\mathcal{J}^m/\mathcal{J}$	$5.8M + 5.1S + 13.3A$	
3	$\mathcal{J}$	$5.6M + 6.6S + 12.4A$	$5.6M + 4.6S + 13.4A$
	$\mathcal{J}$ (c.o.)	$5.8M + 5.8S + 11.4A$	$5.8M + 4.2S + 12.2A$
	$\mathcal{J}^m/\mathcal{J}$	$5.6M + 5S + 13.2A$	

Using Composite Operations As with previous left-to-right algorithms 1.1 and 1.4, the implementation of left-to-right window algorithms can be improved using composite operations if Jacobian coordinates are utilized. Experiments show that replacing the additions of Algorithm 1.6 with $w = 3$ by composite additions $2Q \pm P_{\pm u}$ brings a speed-up of about $4-6\,\%$ in the general case and about $2-4\,\%$ if $a = -3$. The corresponding costs are given in tables 1.11 and 1.12 on rows labelled (c.o.).

Left-to-right window algorithms could also take advantage of the Jacobian consecutive doubling trick, especially with large window sizes.

Precomputations Cost in Left-to-Right Algorithms In this section, we state the precise cost of precomputations for left-to-right algorithms using Jacobian or modified Jacobian coordinates.

Since the base point P is given in affine coordinates, $3P = 2P + P$ can be computed using a doubling with $Z = 1$, which costs $2M + 4S + 11A$, and a mixed affine-projective addition. Then, $5P = 3P + 2P$ requires an additional readdition, as well as $7P = 5P + 2P$.

To use mixed affine-projective additions during the scalar multiplication, these points have to be *scaled* to obtain $Z = 1$. Scaling one point from the Jacobian to the affine representation requires $I + 3M + S$. Using the "Montgomery's trick", each additional scaling costs $6M + S$ [BL07a]. On the other hand, if readditions are used during the scalar multiplication, Z^2 and Z^3 must be precomputed for the extra points.

We summarize in Table 1.14 the cost of these precomputations depending on the number of extra points.

Table 1.12: Average cost per scalar bit of left-to-right window algorithms using readditions depending on the point representation and on the windowing strategy

Extra pts.	Coord.	Cost for any a	Cost for $a = -3$
1	$\mathcal{J}$	$6.8M + 6.8S + 12.8A$	$6.8M + 4.8S + 13.8A$
	$\mathcal{J}$ (c.o.)	$7M + 5.8S + 11.5A$	$7M + 4.3S + 12.3A$
	$\mathcal{J}^m/\mathcal{J}$	$6.8M + 5.3S + 13.5A$	
2	$\mathcal{J}$	$6.4M + 6.7S + 12.6A$	$6.4M + 4.7S + 13.6A$
	$\mathcal{J}$ (c.o.)	$6.7M + 5.8S + 11.4A$	$6.7M + 4.2S + 12.2A$
	$\mathcal{J}^m/\mathcal{J}$	$6.4M + 5.1S + 13.3A$	
3	$\mathcal{J}$	$6.2M + 6.6S + 12.4A$	$6.2M + 4.6S + 13.4A$
	$\mathcal{J}$ (c.o.)	$6.4M + 5.8S + 11.4A$	$6.4M + 4.2S + 12.2A$
	$\mathcal{J}^m/\mathcal{J}$	$6.2M + 5S + 13.2A$	

Table 1.13: Average cost per scalar bit of right-to-left window algorithms depending on the point representation and on the windowing strategy

Extra pts.	Coord.	Cost
1	$\mathcal{J}/\mathcal{J}^m$	$7M + 5S + 13.8A$
2	$\mathcal{J}/\mathcal{J}^m$	$6.7M + 4.9S + 13.6A$
3	$\mathcal{J}/\mathcal{J}^m$	$6.4M + 4.8S + 13.4A$

Table 1.14: Cost of precomputations for left-to-right algorithms using Jacobian or modified Jacobian coordinates

Extra pts.	Additions cost	Optional scaling cost	Optional Z^2, Z^3
1	$10M + 7S + 18A$	$1I + 3M + 1S$	$1M + 1S$
2	$21M + 10S + 25A$	$1I + 9M + 2S$	$2M + 2S$
3	$32M + 13S + 32A$	$1I + 15M + 3S$	$3M + 3S$

Postcomputations Cost in Right-to-Left Algorithms In this section, we state the precise cost of postcomputations for right-to-left algorithms using mixed Jacobian and modified Jacobian coordinates.

The computation of $Q_1 \leftarrow Q_1 + 3Q_3$ requires a point tripling and a general addition. Longa and Miri [LM08a, Section 4.4] show how to compute efficiently a tripling using Jacobian coordinates. Unfolding S–M trade-offs, it costs $7M + 9S + 21A$ in the general case, or $7M + 7S + 22A$ if $a = -3$.

If two extra points are used, an additional $Q_1 \leftarrow Q_1 + 5Q_5$ is performed, which requires a point quintupling and a general addition. Again, Longa and Miri [LM08c, Appendix C] give an efficient formula to compute a quintupling using Jacobian coordinates. Unfolding S–M trade-offs, it costs $13M + 11S + 19A$ in the general case, or $13M + 9S + 20A$ if $a = -3$.

If three extra points are used, we further compute $Q_1 \leftarrow Q_1 + 7Q_7$, which requires a point septupling and a general addition. Applying Longa and Miri's methodology, point septupling can be computed as $7Q_7 = 2Q_7 + (2Q_7 + (2Q_7 + Q_7))$ where the three additions use the co-Z formula. With this method, point septupling costs $19M + 12S + 32A$ in the general case, or $19M + 10S + 33A$ if $a = -3$.

We sum up in Table 1.15 the cost of these postcomputations depending on the number of extra points.

Table 1.15: Cost of postcomputations for right-to-left algorithms using mixed Jacobian and modified Jacobian coordinates

Extra pts.	Cost for any a	Cost for $a = -3$
1	$19M + 13S + 28A$	$19M + 11S + 29A$
2	$44M + 28S + 54A$	$44M + 24S + 56A$
3	$75M + 44S + 93A$	$75M + 38S + 96A$

1.2.1.5 Detailed Cost Comparison

The following tables summarize the cost of the algorithms presented in this section depending on the cost of field squarings and additions. Label (c.o.) denote the use of Jacobian composite operations $dP_1 + P_2$.

Table 1.16: Detailed average cost of binary and NAF algorithms expressed as field multiplications per scalar bit

Algorithm	Coord.	a	$S/M = 1$		$S/M = 0.8$	
			$A/M = 0.2$	$A/M = 0.1$	$A/M = 0.2$	$A/M = 0.1$
1.1	$\mathcal{J}^m/\mathcal{J}$	any	17.5	16.0	16.2	14.7
1.1	$\mathcal{J}$ (c.o.)	any	16.4	15.2	15.3	14.1
1.2	$\mathcal{J}/\mathcal{J}^m$	any	19.1	17.6	17.9	16.4
1.4	$\mathcal{J}^m/\mathcal{J}$	any	15.1	**13.7**	14.0	**12.6**
1.4	$\mathcal{J}$ (c.o.)	any	**15.0**	13.8	**13.9**	12.7
1.5	$\mathcal{J}/\mathcal{J}^m$	any	16.2	14.7	15.1	13.7
1.1	$\mathcal{J}$ (c.o.)	-3	15.5	14.3	14.6	13.4
1.4	$\mathcal{J}$ (c.o.)	-3	**13.8**	**12.6**	**12.9**	**11.7**

Table 1.17: Detailed average cost of left-to-right window algorithms such as Algorithm 1.6 using mixed affine-projective additions expressed as field multiplications per scalar bit

Extra pts.	Coord.	a	$S/M = 1$		$S/M = 0.8$	
			$A/M = 0.2$	$A/M = 0.1$	$A/M = 0.2$	$A/M = 0.1$
1	$\mathcal{J}^m/\mathcal{J}$	any	14.0	12.6	12.9	11.6
	$\mathcal{J}$ (c.o.)	any	14.3	13.2	13.2	12.0
2	$\mathcal{J}^m/\mathcal{J}$	any	13.6	12.2	12.5	11.2
	$\mathcal{J}$ (c.o.)	any	14.1	12.9	12.9	11.8
3	$\mathcal{J}^m/\mathcal{J}$	any	**13.2**	**11.9**	**12.2**	**10.9**
	$\mathcal{J}$ (c.o.)	any	13.9	12.7	12.7	11.6
1	$\mathcal{J}$ (c.o.)	-3	13.0	11.7	12.1	10.9
2	$\mathcal{J}$ (c.o.)	-3	12.7	11.4	11.8	10.6
3	$\mathcal{J}$ (c.o.)	-3	**12.4**	**11.2**	**11.6**	**10.4**

Table 1.18: Detailed average cost of left-to-right window algorithms such as Algorithm 1.6 using readditions expressed as field multiplications per scalar bit

Extra pts.	Coord.	a	$S/M = 1$		$S/M = 0.8$	
			$A/M = 0.2$	$A/M = 0.1$	$A/M = 0.2$	$A/M = 0.1$
1	$\mathcal{J}^m/\mathcal{J}$	any	14.7	13.4	13.7	12.3
	$\mathcal{J}$ (c.o.)	any	15.1	13.9	13.9	12.8
2	$\mathcal{J}^m/\mathcal{J}$	any	14.2	12.9	13.2	11.9
	$\mathcal{J}$ (c.o.)	any	14.7	13.6	13.6	12.4
3	$\mathcal{J}^m/\mathcal{J}$	any	**13.8**	**12.5**	**12.8**	**11.5**
	$\mathcal{J}$ (c.o.)	any	14.5	13.3	13.3	12.2
1	$\mathcal{J}$ (c.o.)	-3	13.7	12.5	12.9	11.6
2	$\mathcal{J}$ (c.o.)	-3	13.3	12.1	12.5	11.3
3	$\mathcal{J}$ (c.o.)	-3	**13.0**	**11.8**	**12.2**	**11.0**

Table 1.19: Detailed average cost of right-to-left window algorithms such as Algorithm 1.7 using mixed coordinates expressed as field multiplications per scalar bit

Extra pts.	Coord.	a	$S/M = 1$		$S/M = 0.8$	
			$A/M = 0.2$	$A/M = 0.1$	$A/M = 0.2$	$A/M = 0.1$
1	$\mathcal{J}/\mathcal{J}^m$	any	14.8	13.4	13.8	12.4
2	$\mathcal{J}/\mathcal{J}^m$	any	14.3	12.9	13.3	11.9
3	$\mathcal{J}/\mathcal{J}^m$	any	**13.9**	**12.5**	**12.9**	**11.6**

Cost Comparison We compare hereafter the costs of the most efficient scalar multiplication techniques identified in the previous study. We include the left-to-right NAF algorithm as a matter of comparison. Costs are computed as:

$$\frac{(l - [\text{offset}]) \times [\text{cost per scalar bit}] + [\text{optional pre/postcomputation cost}]}{l}$$

The *offset* parameter refers to the tricks described in remarks 2 and 4. It is fixed to 1 for NAF and right-to-left window algorithms, to 2 for left-to-right window algorithms using one extra point and to 3 for left-to-right window algorithms using two or three extra points.

Considering $S/M = 0.8$, $A/M = 0.2$, and $I/M = 100$, Figure 1.3 compares the cost per scalar bit of these algorithms depending on the scalar length in the general case (any a). Figure 1.4 gives a similar comparison assuming $a = -3$. For a fair comparison between methods, remember that left-to-right window NAF algorithms using readditions require the storage of five coordinates X, Y, Z, Z^2, Z^3 per extra point (including the base point).

Both figures show that left-to-right window NAF algorithms outperform other methods under our assumptions.

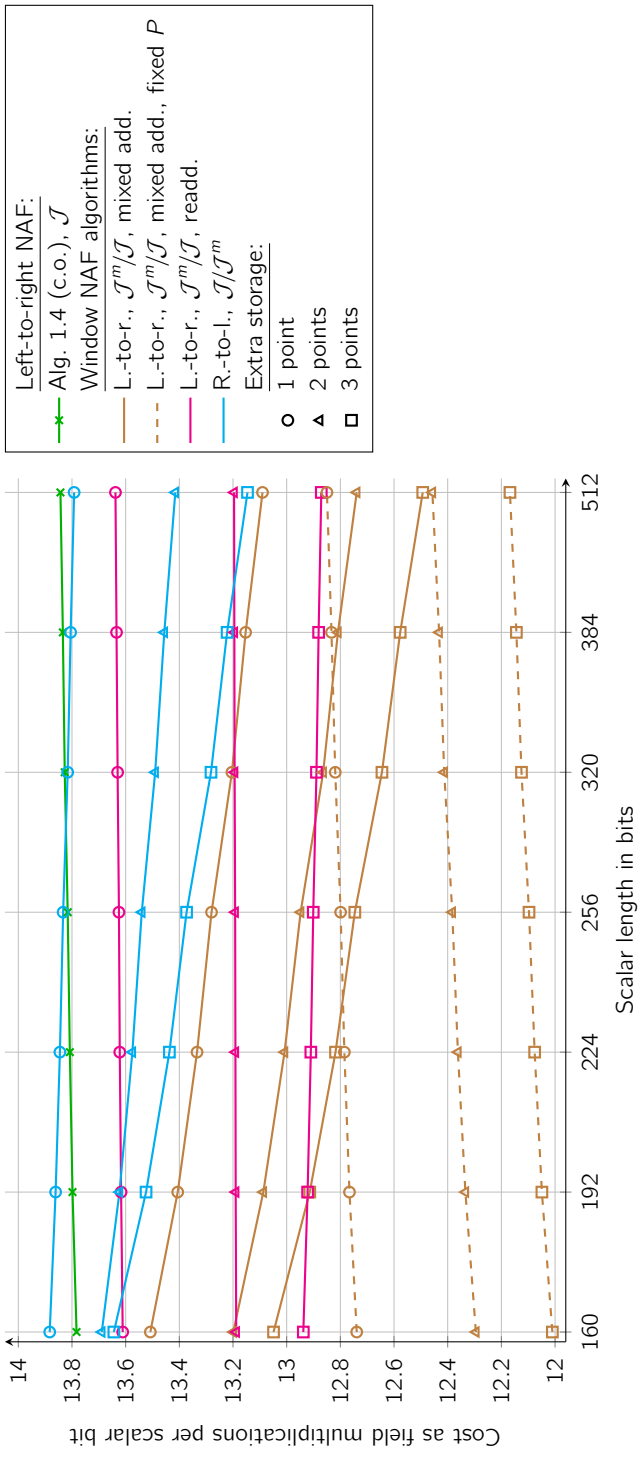


Figure 1.3: Comparison of scalar multiplication algorithms cost depending on scalar length for any a, assuming $I/M = 100$, $S/M = 0.8$, and $A/M = 0.2$

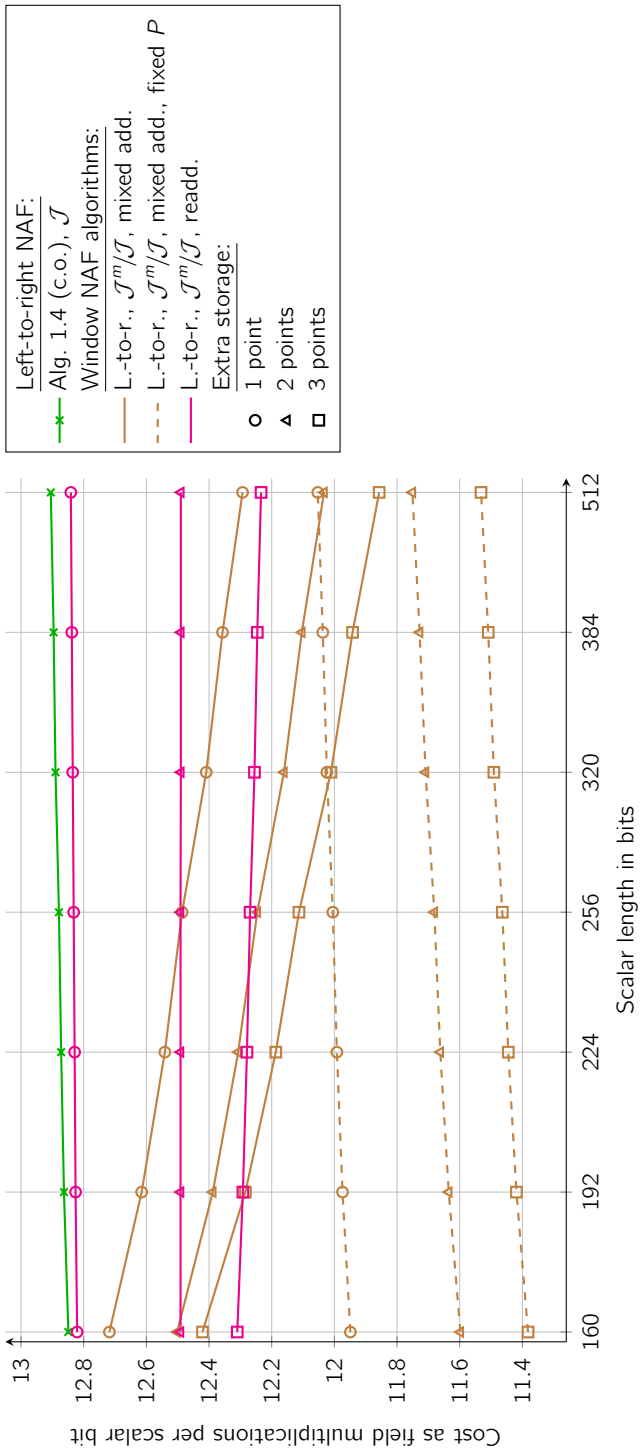


Figure 1.4: Comparison of scalar multiplication algorithms cost depending on scalar length for $a = -3$, assuming $I/M = 100$, $S/M = 0.8$, and $A/M = 0.2$

1.2.2 Algorithms Immune to Simple Side-Channel Analysis

Since the emergence of Simple Side-Channel Analysis (SSCA) (see Section 2.2.1) several countermeasures have been proposed. All of them intend to turn the execution of an algorithm into a regular succession of operations independent from the scalar bits. However, we see in the following that they differ in the level at which this regularity is applied.

As in the previous section, we assume in the following that the input base point of scalar multiplication algorithms is given in affine coordinates.

1.2.2.1 Regular Algorithms

Regular algorithms ensure the regularity property at the highest level: they yield a constant sequence of additions and doublings whatever the scalar.

Double-and-Add-Always The first countermeasure proposed against SSCA was to perform a dummy point addition in the double-and-add algorithm when 0 bits are processed [Cor99]. This method is called *double-and-add-always*, a left-to-right, resp. right-to-left, variant is presented in Algorithm 1.8, resp. Algorithm 1.9, where a dummy addition is added at step 7.

Algorithm 1.8 Left-to-right double-and-add-always scalar multiplication

Input: $P \in \mathcal{E}(\mathbb{F}_q)$, $k = (k_{l-1}k_{l-2}\dots k_0)_2$
Output: kP
Uses: P, Q, and T
1: $Q \leftarrow \mathcal{O}$
2: **for** $i = l-1$ **to** 0 **do**
3: $\quad Q \leftarrow 2Q$
4: $\quad$ **if** $k_i = 1$ **then**
5: $\quad\quad Q \leftarrow Q + P$
6: $\quad$ **else**
7: $\quad\quad T \leftarrow Q + P$
8: **return** Q

Considering an l-bit random scalar, the average cost of double-and-add-always algorithms is:

$$l\mathfrak{A} + l\mathfrak{D}$$

As previously, the left-to-right and right-to-left variants have a different cost. The average computation cost of the left-to-right, resp. right-to-left, double-and-add-always algorithms is given in Table 1.20, resp. Table 1.21. In Table 1.20, we see that modified Jacobian coordinates are always less attractive than Jacobian ones.

Note that Algorithm 1.8 cannot be optimized using the Jacobian composite operation $2Q+P$. Indeed, observe that a 1 scalar bit requires the computation of $2Q$ which is not computed using the composite operation $2Q+P$. Thus, the regularity requirement

Algorithm 1.9 Right-to-left double-and-add-always scalar multiplication

Input: $P \in \mathcal{E}(\mathbb{F}_q)$, $k = (k_{l-1}k_{l-2}\dots k_0)_2$
Output: kP
Uses: Q, R, and T
1: $Q, T \leftarrow \mathcal{O}$
2: $R \leftarrow P$
3: **for** $i = 0$ **to** $l - 1$ **do**
4: **if** $k_i = 1$ **then**
5: $Q \leftarrow Q + R$
6: **else**
7: $T \leftarrow T + R$
8: $R \leftarrow 2R$
9: **return** Q

of the double-and-add-always is incompatible with the use of a composite operation in the left-to-right algorithm. On the other hand, the mixed coordinates strategy $\mathcal{J}^m/\mathcal{J}$ used to speed-up left-to-right algorithms in the previous section cannot be applied either as no consecutive doublings occur in Algorithm 1.8.

Table 1.20: Cost per scalar bit of Algorithm 1.8 depending on the point representation

Coord.	Cost for any a	Cost for $a = -3$
$\mathcal{H}$	$16M + 7S + 17A$	$16M + 5S + 18A$
$\mathcal{J}$	$12M + 9S + 18A$	$12M + 7S + 19A$
$\mathcal{J}^m$	$13M + 9S + 19A$	

Table 1.21: Cost per scalar bit of Algorithm 1.9 depending on the point representation

Coord.	Cost for any a	Cost for $a = -3$
$\mathcal{H}$	$19M + 7S + 17A$	$19M + 5S + 18A$
$\mathcal{J}/\mathcal{J}^m$	$16M + 8S + 19A$	

The detailed computation costs of the double-and-add-always algorithms are given in Table 1.34 (p. 63) depending on the ratios S/M and A/M for Jacobian and modified Jacobian representations.

Remark 5. These algorithms are immune to SSCA but introduce another weakness: they become subject to safe-error attacks as detailed in Section 2.4.2. Indeed, if one is able to introduce a fault during a point addition (which can be performed at step 5 or 7 in both algorithms), it is possible to deduce the corresponding bit value by checking the output of the algorithms: a correct result implies that a dummy addition was performed, so that $k_i = 0$ while an erroneous result indicates that $k_i = 1$. However the right-to-left method provides a simple way to check for faults: at the end of the algorithm there

should be $Q = kP$, $T = (2^l - k - 1)P$, and $R = 2^l P$. Therefore an easy countermeasure is to check that $Q + T + P = R$.

Montgomery Ladder A second option with similar complexity to the double-and-add-always algorithm is the so-called *Montgomery ladder* [Mon87] presented in Algorithm 1.10.

Algorithm 1.10 Montgomery ladder scalar multiplication

Input: $P \in \mathcal{E}(\mathbb{F}_q)$, $k = (k_{l-1}k_{l-2}\dots k_0)_2$ with $k_{l-1} = 1$
Output: kP
Uses: Q_0 and Q_1
1: $Q_0 \leftarrow P$
2: $Q_1 \leftarrow 2P$
3: **for** $i = l - 2$ **to** 0 **do**
4: $\quad Q_{1-k_i} \leftarrow Q_0 + Q_1$
5: $\quad Q_{k_i} \leftarrow 2Q_{k_i}$
6: **return** Q_0

Although this algorithm has the same theoretical complexity as the double-and-add-always, it was originally introduced to speed-up the scalar multiplication on a specific class of curves [Mon87]. This result has been generalized to general elliptic curves over fields of large characteristic [BJ02; IT02; FGKS02]. This technique can be used to accelerate the scalar multiplication if the y coordinate of the result is not needed, which is the case in most cryptographic protocols (e.g. ECDSA).

Noticing that $Q_1 - Q_0 = P$ all along the scalar multiplication, one deduces the following formulas computing the homogeneous projective coordinates X and Z of points $Q_1 + Q_2$ and $2Q_1$ given X_{Q_1}, Z_{Q_1}, X_{Q_2}, Z_{Q_2}, curve parameters a, b, and x_P:

$$\begin{cases} X_{Q_1+Q_2} &= 2\left(X_{Q_1}Z_{Q_2} + X_{Q_2}Z_{Q_1}\right)\left(X_{Q_1}X_{Q_2} + aZ_{Q_1}Z_{Q_2}\right) + 4b{Z_{Q_1}}^2{Z_{Q_2}}^2 \\ & \quad - x_P\left(X_{Q_1}Z_{Q_2} - X_{Q_2}Z_{Q_1}\right)^2 \\ Z_{Q_1+Q_2} &= \left(X_{Q_1}Z_{Q_2} - X_{Q_2}Z_{Q_1}\right)^2 \\ X_{2Q_1} &= \left({X_{Q_1}}^2 - a{Z_{Q_1}}^2\right)^2 - 8bX_{Q_1}{Z_{Q_1}}^3 \\ Z_{2Q_1} &= 4\left(X_{Q_1}Z_{Q_1}\left({X_{Q_1}}^2 + a{Z_{Q_1}}^2\right) + b{Z_{Q_1}}^4\right) \end{cases}$$

As a result, steps 4 and 5 of Algorithm 1.10 require only $14M + 5S + 14A$ in the general case, or $12M + 5S + 18A$ (i) if $a = -3$ (the detailed operations scheme is available in the work by Fischer, Giraud, Knudsen, and Seifert [FGKS02]).

Another cost of $9M + 7S$ if $a = -3$ is reported by Goundar et al. [GJM10]. However, it is obtained using 33 field additions (counting 2 additions for a multiplication by a). Removing the S-M trade-offs (cf. Section 1.1.2.1) yields a cost of $11M + 5S + 24A$ (ii) which is better than our formula if $A/M = .1$, but worse if $A/M = .2$.

The detailed computation cost of Algorithm 1.10 is given in Table 1.34 (p. 63) depending on the ratios S/M and A/M for the homogeneous projective representation without recovering the y coordinate. We keep the lowest cost of the two formulas (i) and (ii) depending on A/M.

Remark 6. The Montgomery ladder is naturally immune to safe-error attacks as no dummy operation is performed. Besides, it provides a convenient way to detect faults by checking that $Q_1 - Q_0 = P$, possibly at each iteration of the main loop. However, this check — as the check that the output point belongs to the original curve described in Section 2.4.1 — cannot be performed in the x-only variant presented above.

Finally, it is worth noticing that Montgomery ladder algorithm is well suited for parallelization as demonstrated by Fischer et al. [FGKS02].

Joye Double-Add Ladder The following algorithm is presented by Joye [Joy07].

Algorithm 1.11 Double-add scalar multiplication

Input: $P \in \mathcal{E}(\mathbb{F}_q)$, $k = (k_{l-1}k_{l-2}\dots k_0)_2$
Output: kP
Uses: Q_0 and Q_1
1: $Q_0 \leftarrow \mathcal{O}$
2: $Q_1 \leftarrow P$
3: **for** $i = 0$ **to** $l - 1$ **do**
4: $\quad Q_{1-k_i} \leftarrow 2Q_{1-k_i} + Q_{k_i}$
5: **return** Q_0

To understand how this algorithm operates, observe that after each loop iteration i we have $Q_0 = \sum_{j=0}^{i} (k_j 2^j)P$ and $Q_1 = \left(2^{i+1} - \sum_{j=0}^{i} (k_j 2^j)\right) P$. This property is stable after the processing of the $(i+1)$-th bit.

As the double-and-add-always and Montgomery ladder, this algorithm performs a doubling and an addition for every scalar bit. Since right-to-left algorithms are generally preferable to left-to-right ones in regard to side-channel analysis, cf. Section 2.2.2, its interest is to provide a right-to-left method naturally immune to safe-errors.

The cost per scalar bit of Algorithm 1.11 depending on the point representation is detailed in Table 1.22. The Jacobian cost is computed assuming the use of the composite operation $2P_1 + P_2$ at step 4.

Table 1.22: Cost per scalar bit of Algorithm 1.11 depending on the point representation

Coord.	Cost for any a	Cost for $a = -3$
$\mathcal{H}$	$19M + 7S + 17A$	$19M + 5S + 18A$
$\mathcal{J}$ (c.o.)	$17M + 6S + 13A$	
$\mathcal{J}^m$	$17M + 10S + 19A$	

The detailed cost per scalar bit of Algorithm 1.11 for the Jacobian representation is detailed in Table 1.34 (p. 63).

Joye m-ary Ladders Joye presents m-ary regular algorithms [Joy09]. We recall in Algorithm 1.12, resp. Algorithm 1.13, the general description of left-to-right, resp. right-to-left, variants of these algorithms.

Algorithm 1.12 Regular left-to-right m-ary scalar multiplication

Input: $P \in \mathcal{E}(\mathbb{F}_q)$, $k = (k_{l_m-1}k_{l_m-2}\dots k_0)_m$ $(k > 0)$
Output: kP
Uses: Q, R_0, R_1, ..., and R_{m-1}
1: $Q \leftarrow -P$
`Precomputations`
2: **for** $i = 0$ **to** $m-1$ **do**
3: $\quad R_i \leftarrow (m+i-1)P$
`Main loop`
4: **for** $i = l_m - 1$ **to** 0 **do**
5: $\quad Q \leftarrow mQ + R_{k_i}$ ▷ t doublings and one addition if $m = 2^t$
`Final correction`
6: $Q \leftarrow Q + P$
7: **return** Q

Algorithm 1.13 Regular right-to-left m-ary scalar multiplication

Input: $P \in \mathcal{E}(\mathbb{F}_q)$, $k = (k_{l_m-1}k_{l_m-2}\dots k_0)_m$ $(k > 0)$
Output: kP
Uses: R, Q_0, Q_1, ..., and Q_{m-1}
1: $R \leftarrow P$
2: $Q_0, Q_1, \dots, Q_{m-1} \leftarrow \mathcal{O}$
`Main loop`
3: **for** $i = 0$ **to** $l_m - 1$ **do**
4: $\quad Q_{k_i} \leftarrow Q_{k_i} + R$
5: $\quad R \leftarrow mR$ ▷ t doublings if $m = 2^t$
6: $R \leftarrow -R$
`Postcomputations`
7: **for** $i = 0$ **to** $m-1$ **do**
8: $\quad R \leftarrow R + (m+i-1)Q_i$
`Final correction`
9: $R \leftarrow R + P$
10: **return** R

The cost of Algorithms 1.12 and 1.13 with $m = 2^t$, omitting the pre/postcomputations stage, for an l-bit scalar is:

$$\frac{l}{t}\mathfrak{A} + l\mathfrak{D}$$

In the following, we detail the performances of these algorithms for $m = 2$, 4, and 8. The computation cost of Algorithm 1.12, resp. Algorithm 1.13, is given in

Table 1.23, resp. Table 1.24, depending on the available memory. Though $m = 8$ requires a prohibitive amount of memory in the context of embedded devices, we include this configuration in the comparatives below.

These costs omit pre/postcomputations. In Table 1.23, readdition formulas are used as ${Z_{R_i}}^2$ and ${Z_{R_i}}^3$, $0 \leq i < m$ can be precomputed once for all. Also, Jacobian costs are computed using composite operations $mP_1 + P_2$ in Table 1.23.

Table 1.23: Cost per scalar bit of Algorithm 1.12 depending on the radix m and on the point representation

m	Coord.	Cost for any a	Cost for $a = -3$
2	$\mathcal{H}$	$19M + 7S + 17A$	$19M + 5S + 18A$
	$\mathcal{J}$ (c.o.)	$16M + 5S + 13A$	
	$\mathcal{J}^m$	$16M + 9S + 19A$	
4	$\mathcal{H}$	$13M + 6S + 13.5A$	$13M + 4S + 14.5A$
	$\mathcal{J}$ (c.o.)	$10M + 5.5S + 12A$	
	$\mathcal{J}^m$	$10M + 6.5S + 15.5A$	
8	$\mathcal{H}$	$11M + 5.7S + 12.3A$	$11M + 3.7S + 13.3A$
	$\mathcal{J}$ (c.o.)	$8M + 5S + 12A$	
	$\mathcal{J}^m$	$8M + 5.7S + 14.3A$	

Table 1.24: Cost per scalar bit of Algorithm 1.13 depending on the radix m and on the point representation

m	Coord.	Cost
2	$\mathcal{J}/\mathcal{J}^m$	$16M + 8S + 19A$
4	$\mathcal{J}/\mathcal{J}^m$	$10M + 6S + 15.5A$
8	$\mathcal{J}/\mathcal{J}^m$	$8M + 5.3S + 14.3A$

The detailed computation costs of Algorithm 1.12 and 1.13 are given in Table 1.35 (p. 63) depending on the ratios S/M and A/M and the available memory — the number m of points to be stored — for Jacobian and modified Jacobian representations. These costs omit pre/postcomputations.

Remark 7. It is worth noticing that Algorithm 1.12 and Algorithm 1.13 with $m = 2$ are very similar to Montgomery ladder and to Joye ladder [Joy09]. Therefore it makes sense to compare their performances: the x-only Montgomery ladder is the fastest method. Besides, Algorithm 1.12 with $m = 2$ has better performances than the double-add ladder, which is due to the use of readdition formulas. Finally, Algorithm 1.13 is slower than the other methods in spite of the use of mixed Jacobian and modified Jacobian coordinates allowed by its right-to-left structure.

Pre/postcomputations Cost Considering Algorithm 1.12 with $m = 2$, R_0 and R_1 are respectively set to P and $2P$. It requires only a doubling from affine to Jacobian coordinates, which costs $2M + 4S + 11A$. An extra $1M + 1S$ allows readditions of R_1.

If $m = 4$, then $R_0 = 3P$, $R_1 = 4P$, $R_2 = 5P$, and $R_3 = 6P$ must be computed, which requires a tripling from affine coordinates, which costs $6M+7S+21A$, and three mixed affine-projective additions. An extra $4M + 4S$ allows readditions of these four points.

Considering Algorithm 1.13 with $m = 2$, the postcomputations are $R + Q_0 + 2Q_1$, which requires a doubling and two additions. If $m = 4$, the algorithm perform $R+3Q_0+4Q_1 + 5Q_2 + 6Q_3$. Postcomputations cost amounts to three doublings, two triplings, one quintupling and four additions — a tripling followed by a doubling computes $6Q_3$. As stated in Section 1.2.1.4, a tripling costs $7M + 9S + 21A$ in the general case, or $7M + 7S + 22A$ if $a = -3$. A quintupling costs $13M + 11S + 19A$ in the general case, or $13M + 9S + 20A$ if $a = -3$.

We summarize in Table 1.25, resp. Table 1.26, the cost of these precomputations, resp. postcomputations, depending on m.

Table 1.25: Cost of precomputations for Algorithm 1.12 using $\mathcal{J}$ with $m = 2$ or 4

m	Cost
2	$3M + 5S + 11A$
4	$34M + 20S + 42A$

Table 1.26: Cost of postcomputations for Algorithm 1.13 using $\mathcal{J}/\mathcal{J}^m$ with $m = 2$ or 4

m	Cost for any a	Cost for $a = -3$
2	$28M + 14S + 25A$	$28M + 12S + 26A$
4	$87M + 63S + 122A$	$87M + 51S + 128A$

Co-Z Ladders A new scalar multiplication method based on the Jacobian co-Z arithmetic is presented by Goundar et al. [GJM10] one one hand, and Venelli and Dassance [VD10] on the other hand. These works have then be completed by other studies [Riv11; GJM+11].

Montgomery and Joye ladders can be rewritten in order to take advantage of the Jacobian co-Z addition formulas ZADDU and ZADDC. The former is presented below in Algorithm 1.14 and the latter in Algorithm 1.15.

Each iteration of the main loop of these algorithms costs $11M + 5S + 18A$ using the formulas presented in Section 1.1.2.3. Another formula mixing ZADDU and ZADDC [GJM+11] requires $9M + 7S + 25A$, which is worse than the former one whenever $A/M \geq 0.1$.

Algorithm 1.14 Montgomery ladder scalar multiplication using $\mathcal{J}$ with co-Z addition

Input: $P = (x, y) \in \mathcal{E}(\mathbb{F}_q)$, $k = (k_{l-1}k_{l-2}\dots k_0)_2$ with $k_{l-1} = 1$
Output: kP
Uses: Q_0 and Q_1

1: $Q_1 \leftarrow (X_{2P} : Y_{2P} : Z_{2P})$ ▷ Jacobian representation of $2P$
2: $Q_0 \leftarrow (xZ_{2P}{}^2 : yZ_{2P}{}^3 : Z_{2P})$ ▷ Jacobian representation of P
3: **for** $i = l - 2$ **to** 0 **do**
4: $\quad (Q_{1-k_i}, Q_{k_i}) \leftarrow \mathrm{ZADDC}(Q_{k_i}, Q_{1-k_i})$
5: $\quad (Q_{k_i}, Q_{1-k_i}) \leftarrow \mathrm{ZADDU}(Q_{1-k_i}, Q_{k_i})$
6: **return** Q_0

Algorithm 1.15 Joye's double-add ladder scalar multiplication using $\mathcal{J}$ with co-Z addition

Input: $P = (x, y) \in \mathcal{E}(\mathbb{F}_q)$, $k = (k_{l-1}k_{l-2}\dots k_0)_2$ with $k_0 = 1$
Output: kP
Uses: Q_0 and Q_1

1: $Q_{1-k_1} \leftarrow (X_{3P} : Y_{3P} : Z_{3P})$ ▷ Jacobian representation of $3P$
2: $Q_{k_1} \leftarrow (xZ_{3P}{}^2 : yZ_{3P}{}^3 : Z_{3P})$ ▷ Jacobian representation of P
3: **for** $i = 2$ **to** $l - 1$ **do**
4: $\quad (Q_{k_i}, Q_{1-k_i}) \leftarrow \mathrm{ZADDU}(Q_{1-k_i}, Q_{k_i})$
5: $\quad (Q_{1-k_i}, Q_{k_i}) \leftarrow \mathrm{ZADDC}(Q_{k_i}, Q_{1-k_i})$
6: **return** Q_0

Algorithm 1.16 Montgomery ladder scalar multiplication using $\mathcal{J}$ with $(X : Y)$-only co-Z addition

Input: $P = (x, y) \in \mathcal{E}(\mathbb{F}_q)$, $k = (k_{l-1}k_{l-2}\dots k_0)_2$ with $k_{l-1} = 1$
Output: kP
Uses: Q_0 and Q_1

1: $Q_1 \leftarrow (X_{2P} : Y_{2P})$
2: $Q_0 \leftarrow (xZ_{2P}{}^2 : yZ_{2P}{}^3)$
3: **for** $i = l - 2$ **to** 1 **do**
4: $\quad (Q_{1-k_i}, Q_{k_i}) \leftarrow \mathrm{ZADDC}'(Q_{k_i}, Q_{1-k_i})$
5: $\quad (Q_{k_i}, Q_{1-k_i}) \leftarrow \mathrm{ZADDU}'(Q_{1-k_i}, Q_{k_i})$
6: $(Q_{1-k_0}, Q_{k_0}) \leftarrow \mathrm{ZADDC}'(Q_{k_0}, Q_{1-k_0})$
7: $Z \leftarrow xY_{Q_{k_0}}(X_{Q_0} - X_{Q_1})$
8: $\lambda \leftarrow yX_{Q_{k_0}}$
9: $(Q_{k_0}, Q_{1-k_0}) \leftarrow \mathrm{ZADDU}'(Q_{1-k_0}, Q_{k_0})$
10: **return** $(\lambda^2 X_{Q_0} : \lambda^3 Y_{Q_0} : Z)$

Meloni observes that an $(X : Y)$-only formula of ZADDU, denoted ZADDU' in the following, can be used in scalar multiplication algorithms [Mel07]. Indeed, the Z coordinate can generally be recovered at the end with a few extra field operations. The same trick applies to the co-Z Montgomery ladder algorithm [VD10; GJM+11], presented in Algorithm 1.16. It uses an $(X : Y)$-only formula of operation ZADDC, denoted ZADDC'.

The cost of the operation ZADDU', resp. ZADDC', being $4M + 2S + 7A$, resp. $5M+3S+11A$, each iteration of the main loop of Algorithm 1.16 costs $9M+5S+18A$. Another formula mixing ZADDU' and ZADDC' [GJM+11] requires $8M+6S+33A$, which is worse than the former one whenever $A/M \geq 0.1$.

The detailed computation cost of Algorithms 1.14, 1.15, and 1.16 is given in Table 1.34 (p. 63), depending on the ratios S/M and A/M.

1.2.2.2 Atomicity

In a nutshell, the original SSCA is enabled by the difference of processing between 0 and 1 scalar bits (or group of bits considering window methods). This difference is mainly observable because of the dissimilar addition and doubling patterns that may reveal a consumption or radiation trace. Regular algorithms thus perform a regular sequence of operations regardless the scanned bits. However, classical scalar multiplication optimizations relying on performing as few point additions as possible — the number of doublings is fixed —, the efficiency of regular algorithms precisely suffers from this strategy.

The atomicity principle introduced by Chevallier-Mames, Ciet, and Joye [CMCJ04] tends to smooth the execution of non-regular algorithms in such a manner that the processing of bits of the scalar — or groups of bits — is indistinguishable. As a consequence, atomic algorithms produce a regular sequence of operations, but this regularity is achieved at a lower level than in the previous regular algorithms.

Double and Add Using Unified Formulas A first step towards SSCA immunity is to render an addition indistinguishable from a doubling. This property is easily achieved if a unified addition formula is available (see Section 1.1.3), since the same portion of code performs both operations. Furthermore, the bit processing of the main loop of the scalar multiplication algorithm must be rearranged to smoothen the processing of the scalar bits.

Algorithm 1.17 presents how the scalar multiplication can be performed in an SSCA resistant manner using a unified addition formula and an atomic loop processing. Using this algorithm, one may observe the number of operations but cannot differentiate between the processing of scalar bits, assuming that a doubling and an addition are indistinguishable. If the length of the scalar is known, an attacker can at most deduce its Hamming weight, which is not a threat for a random scalar of hundreds of bits.

Denoting $\mathfrak{A}$ the computation timing of the unified addition and considering an l-bit random scalar, the average cost of this algorithm is:

$$\frac{3}{2}l\mathfrak{A}$$

Algorithm 1.17 Left-to-right atomic double-and-add scalar multiplication using a unified addition

Input: $P \in \mathcal{E}(\mathbb{F}_q)$, $k = (k_{l-1}k_{l-2}\dots k_0)_2$
Output: kP
Uses: R_0 and R_1
1: $R_0 \leftarrow \mathcal{O}$
2: $R_1 \leftarrow P$
3: $i \leftarrow l-1$
4: $t \leftarrow 0$
5: **while** $i \geq 0$ **do**
6: $\quad R_0 \leftarrow R_0 + R_t$ ▷ addition using a unified formula
7: $\quad t \leftarrow t \oplus k_i$ ▷ $\oplus$ stands for the bitwise X-or
8: $\quad i \leftarrow i - 1 + t$
9: **return** R_0

The average computation cost of Algorithm 1.17 is given in Table 1.27, resp. Table 1.28, for a Weierstraß curve using the homogeneous projective addition formula, resp. a twisted Edwards curve using the extended coordinates ($\mathcal{E}^e$) addition formula, presented in Section 1.1.3.

Table 1.27: Average cost per scalar bit of Algorithm 1.17 for Weierstraß curves using homogeneous unified additions

Coord.	Cost for any a	Cost for $a = -3$
$\mathcal{H}$	$19.5M + 7.5S + 10.5A$	$18M + 7.5S + 13.5A$

Table 1.28: Average cost per scalar bit of Algorithm 1.17 for twisted Edwards curves using twisted Edwards unified additions

Coord.	Cost for any a, d	Cost for $a = -1$, any d
$\mathcal{E}^e$	$13.5M + 1.5C^a + 1.5C^d + 10.5A$	$12M + 1.5C^{2d} + 13.5A$

The detailed average computation costs of Algorithm 1.17 for Weierstraß curves, resp. twisted Edwards curves, using the homogeneous projective addition formula, resp. the extended coordinates addition formula, are given in Table 1.36 (p. 64).

These tables show that the unified addition formula of twisted Edwards curves provides better performances than the expensive Weierstraß unified addition.

Besides, we emphasize that Algorithm 1.17 can further be improved using NAF and window optimizations presented in Section 1.2.1.

Original Atomicity for ECC It is depicted in Algorithm 1.17 how atomicity can smooth the computation of non-regular algorithms to hide the processing of scalar bits by using unified addition formulas. However, Chevallier-Mames et al. [CMCJ04] propose a more efficient technique for smoothing the processing of classical addition and doubling formulas.

Indeed, if the addition and doubling formulas are decomposed in sequences of an identical *atomic pattern*, and if all the loop processing and conditional branching between the iterations of this pattern are performed in an atomic manner, any scalar multiplication algorithm can theoretically be made SSCA resistant using classical formulas.

The atomic pattern (1) proposed by Chevallier-Mames et al. [CMCJ04] is depicted below, with $R_{u(i)}$, $0 \leq i \leq 9$ denoting an arithmetic coprocessor register, and $(u(i))_{0 \leq i \leq 9}$ a vector defining a pattern iteration.

$$(1) \left[\begin{array}{l} R_{u(0)} \leftarrow R_{u(1)} \times R_{u(2)} \\ R_{u(3)} \leftarrow R_{u(4)} + R_{u(5)} \\ R_{u(6)} \leftarrow -R_{u(6)} \\ R_{u(7)} \leftarrow R_{u(8)} + R_{u(9)} \end{array} \right.$$

This choice relies on the observation that during the execution of point additions and doublings, no more than two additions and one negation are required between two multiplications. Atomicity consists then of expressing point additions and doublings as sequences of this pattern — as many as the required number of field multiplications (including squarings).

Algorithm 1.18 presents the original atomic scalar multiplication using atomic pattern (1) and Jacobian coordinates presented by Chevallier-Mames et al. From the execution of this algorithm, one may observe the number of performed atomic patterns but cannot distinguish the processing of scalar bits. As previously, if the length of the scalar is known, an attacker can read its Hamming weight, which is not an interesting leakage in practice.

The effective atomic pattern of Algorithm 1.18 includes the whole content of the **while** loop. However, steps 7, 8, and 13 consisting in the scalar bits and loop processing, we denote by *atomic pattern* the sequence of field operations only, i.e. steps 9 to 12, here performed over $\mathbb{F}_q$.

The point operations to be performed are not given in Algorithm 1.18, but are moved to the matrix M, where each row defines a pattern iteration. Note that rows 0 to 9 correspond to the point doubling $(R_1 : R_2 : R_3) \leftarrow 2(R_1 : R_2 : R_3)$, and that rows 10 to 25 correspond to the addition $(R_1 : R_2 : R_3) \leftarrow (R_1 : R_2 : R_3) + (R_7 : R_8 : R_9)$. In other words, a point doubling is carried out using 10 iterations of pattern (1) because of the 10 field multiplications to be performed in the Jacobian general doubling formula, and a point addition is computed using 16 iterations of pattern (1) because of the 16 field multiplications of the Jacobian general addition formula.

Algorithm 1.18 Left-to-right atomic double-and-add scalar multiplication using $\mathcal{J}$ and pattern (1)

Input: $P = (X_1 : Y_1 : Z_1) \in \mathcal{E}(\mathbb{F}_q)$, $k = (k_{l-1}k_{l-2}\dots k_0)_2$ with $k_{l-1} = 1$
Output: kP
Uses: 10 l-bit registers R_0 to R_9

1: $R_0 \leftarrow a$
2: $(R_1, R_2, R_3) \leftarrow (X_1, Y_1, Z_1)$
3: $(R_7, R_8, R_9) \leftarrow (X_1, Y_1, Z_1)$
4: $i \leftarrow l - 2$
5: $s \leftarrow 1$
6: **while** $i \geq 0$ **do**
7: $\quad t \leftarrow (\neg s)(t + 1)$ ▷ $\neg$ stands for the bitwise negation
8: $\quad s \leftarrow k_i(t \div 25) + (\neg k_i)(t \div 9)$ ▷ $\div$ stands for the Euclidean quotient
9: $\quad R_{M_{t,0}} \leftarrow R_{M_{t,1}} \times R_{M_{t,2}}$
10: $\quad R_{M_{t,3}} \leftarrow R_{M_{t,4}} + R_{M_{t,5}}$
11: $\quad R_{M_{t,6}} \leftarrow -R_{M_{t,6}}$
12: $\quad R_{M_{t,7}} \leftarrow R_{M_{t,8}} + R_{M_{t,9}}$
13: $\quad i \leftarrow i - s$
14: **return** $(R_1 : R_2 : R_3)$

$$M = \left(\begin{array}{cccccccccc}
4 & 1 & 1 & 5 & 4 & 4 & 3 & 4 & 4 & 5 \\
5 & 3 & 3 & 1 & 1 & 1 & 3 & 1 & 1 & 3 \\
5 & 5 & 5 & 1 & 1 & 3 & 3 & 1 & 1 & 3 \\
5 & 0 & 5 & 4 & 4 & 5 & 3 & 5 & 2 & 2 \\
3 & 3 & 5 & 1 & 1 & 3 & 3 & 1 & 1 & 3 \\
2 & 2 & 2 & 2 & 2 & 2 & 4 & 1 & 1 & 3 \\
5 & 1 & 2 & 1 & 1 & 5 & 5 & 1 & 1 & 5 \\
1 & 4 & 4 & 1 & 1 & 5 & 4 & 1 & 1 & 5 \\
2 & 2 & 2 & 2 & 2 & 2 & 3 & 5 & 1 & 5 \\
4 & 4 & 5 & 2 & 2 & 4 & 2 & 4 & 4 & 5 \\
\hline
4 & 9 & 9 & 5 & 1 & 5 & 5 & 5 & 1 & 5 \\
1 & 1 & 4 & 5 & 1 & 5 & 5 & 5 & 1 & 5 \\
4 & 4 & 9 & 5 & 1 & 5 & 5 & 5 & 1 & 5 \\
2 & 2 & 4 & 5 & 1 & 5 & 5 & 5 & 1 & 5 \\
4 & 3 & 3 & 5 & 1 & 5 & 5 & 5 & 1 & 5 \\
5 & 4 & 7 & 2 & 2 & 5 & 5 & 5 & 1 & 5 \\
4 & 3 & 4 & 2 & 2 & 5 & 6 & 6 & 5 & 6 \\
4 & 4 & 8 & 6 & 5 & 6 & 4 & 4 & 2 & 4 \\
3 & 3 & 9 & 6 & 5 & 6 & 6 & 6 & 5 & 6 \\
3 & 3 & 5 & 6 & 5 & 6 & 6 & 6 & 5 & 6 \\
6 & 5 & 5 & 6 & 3 & 6 & 3 & 6 & 3 & 6 \\
1 & 1 & 6 & 1 & 1 & 4 & 4 & 1 & 1 & 4 \\
5 & 5 & 6 & 6 & 1 & 2 & 2 & 6 & 2 & 6 \\
1 & 4 & 4 & 1 & 1 & 5 & 6 & 1 & 1 & 6 \\
2 & 2 & 5 & 1 & 1 & 6 & 3 & 6 & 1 & 6 \\
4 & 4 & 6 & 2 & 2 & 4 & 6 & 6 & 1 & 6
\end{array}\right)$$

$R_4 \leftarrow R_1 \times R_1$; $R_5 \leftarrow R_4 + R_4$; $R_3 \leftarrow -R_3$; $R_4 \leftarrow R_4 + R_5$
$R_5 \leftarrow R_3 \times R_3$; $R_1 \leftarrow R_1 + R_1$; $R_3 \leftarrow -R_3$; $R_1 \leftarrow R_1 + R_3$
$R_5 \leftarrow R_5 \times R_5$; $R_1 \leftarrow R_1 + R_3$; $R_3 \leftarrow -R_3$; $R_1 \leftarrow R_1 + R_3$

$\vdots$

$R_4 \leftarrow R_9 \times R_9$; $R_5 \leftarrow R_1 + R_5$; $R_5 \leftarrow -R_5$; $R_5 \leftarrow R_1 + R_5$
$R_1 \leftarrow R_1 \times R_4$; $R_5 \leftarrow R_1 + R_5$; $R_5 \leftarrow -R_5$; $R_5 \leftarrow R_1 + R_5$
$R_4 \leftarrow R_4 \times R_9$; $R_5 \leftarrow R_1 + R_5$; $R_5 \leftarrow -R_5$; $R_5 \leftarrow R_1 + R_5$

$\vdots$

This algorithm induces two kinds of cost overhead:

(i) field squarings are performed as field multiplications. This approach is costly on embedded devices provided with a dedicated hardware modular squaring operation, i.e. when $S/M < 1$, cf. Section 1.1.1.6,

(ii) dummy additions and negations are introduced. Although their cost is generally neglected, we have seen in Section 1.1.1.6 that it must be taken into account in our context.

Denoting $\mathfrak{P}^{(1)}$ the computation timing of pattern (1) and considering an l-bit random scalar, the average cost of this algorithm is:

$$\left(\frac{16}{2} + 10\right) l\mathfrak{P}^{(1)} = 18l\mathfrak{P}^{(1)}$$

The average cost per bit of Algorithm 1.18 is thus $18M + 36A + 18N$, N being the cost of a field negation. The detailed average computation cost is then $27.0M$, resp. $23.4M$, per scalar bit if $A/M = 0.2$, resp. $A/M = 0.1$, assuming $N/M = 0.1$.

We know that using mixed affine-projective additions improves the efficiency of classical left-to-right algorithms. To apply this optimization here, rows 10 to 25 of matrix M have to we replaced by 11 rows performing a Jacobian mixed affine-projective addition [Lon07, App. B3]. In this case, the average cost of Algorithm 1.18 becomes $15.5l\mathfrak{P}^{(1)}$, i.e. $15.5M + 31A + 15.5N$ per scalar bit. If $a = -3$, the fast doubling formula can be expressed [Lon07, App. B1] using 8 instances of pattern (1) — which replace rows 0 to 9 of matrix M. The average cost of the scalar multiplication drops to $13.5l\mathfrak{P}^{(1)}$, i.e. $13.5M + 27A + 13.5N$ per scalar bit.

Though we do not detail the algorithm, a NAF variant of Algorithm 1.18 reduces the cost per bit of a scalar multiplication. In addition, windowing strategy can be used to further improve efficiency as shown in Table 1.29. The detailed computation costs of these methods are given in Table 1.37 (p. 64) depending on A/M and assuming $N/M = 0.1$.

Table 1.29: Average cost per scalar bit of left-to-right window NAF atomic algorithms using pattern (1) and mixed affine-projective additions depending on the windowing strategy

Extra pts.	Coord.	Cost for any a	Cost for $a = -3$
NAF	$\mathcal{J}$	$13.7M + 27.3A + 13.7N$	$11.7M + 23.3A + 11.7N$
1	$\mathcal{J}$	$12.8M + 25.5A + 12.8N$	$10.8M + 21.5A + 10.8N$
2	$\mathcal{J}$	$12.4M + 24.9A + 12.4N$	$10.4M + 20.9A + 10.4N$
3	$\mathcal{J}$	$12.2M + 24.4A + 12.2N$	$10.2M + 20.4A + 10.2N$

Another option is to use the Jacobian readdition formula to avoid costly precomputations. The decomposition of this formula using 14 iterations of the atomic pattern (1) is given in Figure 1.5. Under these assumptions, the average cost per bit of a window NAF atomic algorithm using Jacobian coordinates is given in Table 1.30 depending on the windowing strategy. The detailed computation cost of such a method is given in Table 1.38 (p. 65) depending on A/M and assuming $N/M = 0.1$.

The cost of precomputations is the same as described in Section 1.2.1.4.

Input: $(X_1 : Y_1 : Z_1 : {Z_1}^2 : {Z_1}^3)$, $(X_2 : Y_2 : Z_2)$
Output: $(X_3 : Y_3 : Z_3)$
Uses: 7 registers R_1 to R_7

1
$R_1 \leftarrow Y_2 \times {Z_1}^3$
$\star$
$\star$
$\star$

2
$R_2 \leftarrow Y_1 \times Z_2$
$\star$
$\star$
$\star$

3
$R_3 \leftarrow Z_2 \times Z_2$
$\star$
$\star$
$\star$

4
$R_4 \leftarrow R_2 \times R_3$
$\star$
$R_1 \leftarrow -R_1$
$R_4 \leftarrow R_4 + R_1$

5
$R_2 \leftarrow X_2 \times {Z_1}^2$
$\star$
$R_1 \leftarrow -R_1$
$\star$

6
$R_3 \leftarrow X_1 \times R_3$
$\star$
$R_2 \leftarrow -R_2$
$R_3 \leftarrow R_3 + R_2$

7
$R_5 \leftarrow R_3 \times R_3$
$\star$
$\star$
$\star$

8
$R_2 \leftarrow R_2 \times R_5$
$R_6 \leftarrow R_2 + R_2$
$\star$
$\star$

9
$R_5 \leftarrow R_5 \times R_3$
$\star$
$\star$
$\star$

10
$R_3 \leftarrow Z_2 \times R_3$
$\star$
$\star$
$\star$

11
$R_7 \leftarrow R_4 \times R_4$
$\star$
$\star$
$\star$

12
$Z_3 \leftarrow R_3 \times Z_1$
$R_7 \leftarrow R_7 + R_6$
$R_5 \leftarrow -R_5$
$X_3 \leftarrow R_7 + R_5$

13
$R_3 \leftarrow R_1 \times R_5$
$R_2 \leftarrow R_2 + X_3$
$R_2 \leftarrow -R_2$
$\star$

14
$R_1 \leftarrow R_2 \times R_4$
$Y_3 \leftarrow R_1 + R_3$
$\star$
$\star$

Figure 1.5: Decomposition of a readdition using $\mathcal{J}$ and atomic pattern (1)

Table 1.30: Average cost per scalar bit of left-to-right window NAF atomic algorithms using pattern (1) and readditions depending on the windowing strategy

Extra pts.	Coord.	Cost for any a	Cost for $a = -3$
NAF	$\mathcal{J}$	$14.7M + 29.3A + 14.7N$	$12.7M + 25.3A + 12.7N$
1	$\mathcal{J}$	$13.5M + 27A + 13.5N$	$11.5M + 23A + 11.5N$
2	$\mathcal{J}$	$13.1M + 26.2A + 13.1N$	$11.1M + 22.2A + 11.1N$
3	$\mathcal{J}$	$12.8M + 25.6A + 12.8N$	$10.8M + 21.6A + 10.8N$

Table 1.31: Average cost per scalar bit of right-to-left window NAF atomic algorithms implemented using pattern (1) and mixed $\mathcal{J}/\mathcal{J}^m$ coordinates, depending on the windowing strategy

Extra pts.	Coord.	Cost for any a
NAF	$\mathcal{J}/\mathcal{J}^m$	$13.3M + 26.7A + 13.3N$
1	$\mathcal{J}/\mathcal{J}^m$	$12M + 24A + 12N$
2	$\mathcal{J}/\mathcal{J}^m$	$11.6M + 23.1A + 11.6N$
3	$\mathcal{J}/\mathcal{J}^m$	$11.2M + 22.4A + 11.2N$

Right-to-Left Variant As suggested by Joye [Joy08], Algorithm 1.5 — the right-to-left NAF scalar multiplication using mixed Jacobian and modified Jacobian coordinates — can be protected against SSCA using atomic pattern (1). Note in this case that NAF digits computation must also be performed atomically. We present in Figure 1.6 the modified Jacobian doubling formula expressed using 8 atomic patterns (1). Dummy operations are denoted by $\star$.

The average cost of this protected algorithm is $13.3M/\mathfrak{P}^{(1)}$. Applying a window method yields the average cost per scalar bit detailed in Table 1.31 whatever the value of a. The corresponding detailed average cost per bit is given in Table 1.39 (p. 65) depending on A/M and assuming $N/M = 0.1$.

The cost of postcomputations is the same as described in Section 1.2.1.4.

Longa's Atomic Patterns To reduce the costs induced by Algorithm 1.18, Longa proposes in his Master's thesis [Lon07, Chap. 5] the following two atomic patterns in the context of Jacobian coordinates:

$$(2)\begin{bmatrix} R_{u(0)} \leftarrow R_{u(1)} \cdot R_{u(2)} \\ R_{u(3)} \leftarrow -R_{u(4)} \\ R_{u(5)} \leftarrow R_{u(6)} + R_{u(7)} \\ R_{u(8)} \leftarrow R_{u(9)} \cdot R_{u(10)} \\ R_{u(11)} \leftarrow -R_{u(12)} \\ R_{u(13)} \leftarrow R_{u(14)} + R_{u(15)} \\ R_{u(16)} \leftarrow R_{u(17)} + R_{u(18)} \end{bmatrix} \quad (3)\begin{bmatrix} R_{u(0)} \leftarrow {R_{u(1)}}^2 \\ R_{u(2)} \leftarrow -R_{u(3)} \\ R_{u(4)} \leftarrow R_{u(5)} + R_{u(6)} \\ R_{u(7)} \leftarrow R_{u(8)} \cdot R_{u(9)} \\ R_{u(10)} \leftarrow -R_{u(11)} \\ R_{u(12)} \leftarrow R_{u(13)} + R_{u(14)} \\ R_{u(15)} \leftarrow R_{u(16)} + R_{u(17)} \end{bmatrix}$$

Input: $(X_1 : Y_1 : Z_1)$
Output: $(X_2 : Y_2 : Z_2)$
Uses: 6 registers R_1 to R_6

$$
\begin{array}{ll}
1 \left[\begin{array}{l} R_1 \leftarrow X_1 \cdot X_1 \\ R_2 \leftarrow Y_1 + Y_1 \\ \star \\ \star \end{array}\right. & 5 \left[\begin{array}{l} R_3 \leftarrow R_1 \cdot R_1 \\ \star \\ \star \\ \star \end{array}\right. \\
2 \left[\begin{array}{l} Z_2 \leftarrow R_2 \cdot Z_1 \\ R_4 \leftarrow R_1 + R_1 \\ \star \\ \star \end{array}\right. & 6 \left[\begin{array}{l} R_4 \leftarrow R_6 \cdot X_1 \\ R_5 \leftarrow W_1 + W_1 \\ R_4 \leftarrow -R_4 \\ R_3 \leftarrow R_3 + R_4 \end{array}\right. \\
3 \left[\begin{array}{l} R_3 \leftarrow R_2 \cdot Y_1 \\ R_6 \leftarrow R_3 + R_3 \\ \star \\ \star \end{array}\right. & 7 \left[\begin{array}{l} W_2 \leftarrow R_2 \cdot R_5 \\ X_2 \leftarrow R_3 + R_4 \\ R_2 \leftarrow -R_2 \\ R_6 \leftarrow R_4 + X_2 \end{array}\right. \\
4 \left[\begin{array}{l} R_2 \leftarrow R_6 \cdot R_3 \\ R_1 \leftarrow R_4 + R_1 \\ \star \\ R_1 \leftarrow R_1 + W_1 \end{array}\right. & 8 \left[\begin{array}{l} R_4 \leftarrow R_6 \cdot R_1 \\ \star \\ R_4 \leftarrow -R_4 \\ Y_2 \leftarrow R_4 + R_2 \end{array}\right.
\end{array}
$$

Figure 1.6: Decomposition of a doubling using $\mathcal{J}^m$ and atomic pattern (1)

The atomic pattern (2) slightly reduces the number of field additions — gain of one addition every two multiplications — compared to pattern (1). Moreover, using atomic pattern (3) instead of (2) in a scalar multiplication implementation allows to replace one multiplication out of two by a squaring. This option should be considered on devices providing a dedicated squaring operation.

Longa expresses [Lon07, Appendices] mixed affine-Jacobian addition formula as 6 atomic patterns (2) or 6 patterns (3) and fast doubling formula as 4 atomic patterns (2) or 4 patterns (3). Since Longa does not investigate general doubling nor readdition, we present in Figure 1.7, resp. Figure 1.8, the decomposition of Jacobian readdition using 7 atomic patterns (2), resp. Jacobian doubling using 5 atomic patterns (2) or (3). It allows left-to-right scalar multiplication using either mixed affine-projective Jacobian additions with atomic pattern (2) or (3), or Jacobian readdition with atomic pattern (2). General or fast doubling can be used in both cases depending on a.

The average computation cost per bit of a left-to-right window NAF scalar multiplication using mixed affine-projective additions and pattern (2) or (3) is detailed in Table 1.32. The corresponding detailed computation costs are given in Table 1.37 (p. 64) depending on A/M and assuming $N/M = 0.1$.

Using the Jacobian readdition formula, the average cost per bit of a window NAF atomic algorithm using pattern (2) is given in Table 1.33 depending on the windowing strategy. The detailed computation costs of this method are given in Table 1.38 (p. 65) depending on A/M and assuming $N/M = 0.1$.

Input: $(X_1 : Y_1 : Z_1 : {Z_1}^2 : {Z_1}^3)$, $(X_2 : Y_2 : Z_2)$
Output: $(X_3 : Y_3 : Z_3)$
Uses: 7 registers R_1 to R_7

$$
\begin{array}{ll}
1 \left[\begin{array}{l} R_1 \leftarrow Y_2 \times {Z_1}^3 \\ \star \\ \star \\ R_2 \leftarrow Y_1 \times Z_2 \\ \star \\ \star \\ \star \end{array}\right. &
5 \left[\begin{array}{l} R_5 \leftarrow R_5 \times R_3 \\ \star \\ \star \\ R_3 \leftarrow Z_2 \times R_3 \\ \star \\ \star \\ \star \end{array}\right. \\
2 \left[\begin{array}{l} R_3 \leftarrow Z_2 \times Z_2 \\ \star \\ \star \\ R_4 \leftarrow R_2 \times R_3 \\ R_5 \leftarrow -R_1 \\ R_4 \leftarrow R_4 + R_5 \\ \star \end{array}\right. &
6 \left[\begin{array}{l} R_7 \leftarrow R_4 \times R_4 \\ \star \\ \star \\ Z_3 \leftarrow R_3 \times Z_1 \\ R_5 \leftarrow -R_5 \\ R_7 \leftarrow R_7 + R_6 \\ X_3 \leftarrow R_7 + R_5 \end{array}\right. \\
3 \left[\begin{array}{l} R_2 \leftarrow X_2 \times {Z_1}^2 \\ \star \\ \star \\ R_3 \leftarrow X_1 \times R_3 \\ R_5 \leftarrow -R_2 \\ R_3 \leftarrow R_3 + R_5 \\ \star \end{array}\right. &
7 \left[\begin{array}{l} R_3 \leftarrow R_1 \times R_5 \\ R_1 \leftarrow -X_3 \\ R_2 \leftarrow R_2 + R_1 \\ R_1 \leftarrow R_2 \times R_4 \\ \star \\ Y_3 \leftarrow R_1 + R_3 \\ \star \end{array}\right. \\
4 \left[\begin{array}{l} R_5 \leftarrow R_3 \times R_3 \\ \star \\ \star \\ R_2 \leftarrow R_2 \times R_5 \\ R_6 \leftarrow -R_2 \\ R_6 \leftarrow R_6 + R_6 \\ \star \end{array}\right. &
\end{array}
$$

Figure 1.7: Decomposition of a readdition using $\mathcal{J}$ and atomic pattern (2)

Input: $(X_1 : Y_1 : Z_1)$
Output: $(X_2 : Y_2 : Z_2)$
Uses: 4 registers R_1 to R_4

$$
1\left\lceil\begin{array}{l} R_1 \leftarrow {X_1}^2 \\ \star \\ R_3 \leftarrow R_1 + R_1 \\ R_2 \leftarrow Z_1 \times Z_1 \\ \star \\ R_1 \leftarrow R_1 + R_3 \\ R_4 \leftarrow X_1 + X_1 \end{array}\right.
\qquad
4\left\lceil\begin{array}{l} R_2 \leftarrow {R_1}^2 \\ \star \\ \star \\ R_4 \leftarrow R_4 \times R_3 \\ R_4 \leftarrow -R_4 \\ R_2 \leftarrow R_2 + R_4 \\ X_2 \leftarrow R_2 + R_4 \end{array}\right.
$$

$$
2\left\lceil\begin{array}{l} R_2 \leftarrow {R_2}^2 \\ \star \\ \star \\ R_2 \leftarrow a \times R_2 \\ \star \\ R_1 \leftarrow R_1 + R_2 \\ R_2 \leftarrow Y_1 + Y_1 \end{array}\right.
\qquad
5\left\lceil\begin{array}{l} R_3 \leftarrow {R_3}^2 \\ R_1 \leftarrow -R_1 \\ R_4 \leftarrow X_2 + R_4 \\ R_1 \leftarrow R_1 \times R_4 \\ R_3 \leftarrow -R_3 \\ R_3 \leftarrow R_3 + R_3 \\ Y_2 \leftarrow R_3 + R_1 \end{array}\right.
$$

$$
3\left\lceil\begin{array}{l} R_3 \leftarrow {Y_1}^2 \\ \star \\ \star \\ Z_2 \leftarrow Z_1 \times R_2 \\ \star \\ R_3 \leftarrow R_3 + R_3 \\ \star \end{array}\right.
$$

Figure 1.8: Decomposition of a general doubling using $\mathcal{J}$ and atomic pattern (3) — or (2) by replacing squarings by multiplications

Table 1.32: Average cost per scalar bit of left-to-right window NAF atomic algorithms using pattern (2) or (3) and mixed affine-projective Jacobian additions, depending on the windowing strategy

Extra pts.	Pattern	Cost for any a	Cost for $a = -3$
NAF	(2)	$14M + 21A + 14N$	$12M + 18A + 12N$
	(3)	$7M + 7S + 21A + 14N$	$6M + 6S + 18A + 12N$
1	(2)	$13M + 19.5A + 13N$	$11M + 16.5A + 11N$
	(3)	$6.5M + 6.5S + 19.5A + 13N$	$5.5M + 5.5S + 16.5A + 11N$
2	(2)	$12.7M + 19A + 12.7N$	$10.7M + 16A + 10.7N$
	(3)	$6.3M + 6.3S + 19A + 12.7N$	$5.3M + 5.3S + 16A + 10.7N$
3	(2)	$12.4M + 18.6A + 12.4N$	$10.4M + 15.6A + 10.4N$
	(3)	$6.2M + 6.2S + 18.6A + 12.4N$	$5.2M + 5.2S + 15.6A + 10.4N$

Table 1.33: Average cost per scalar bit of left-to-right window NAF atomic algorithms using pattern (2) and Jacobian readditions depending on the windowing strategy

Extra pts.	Pattern	Cost for any a	Cost for $a = -3$
NAF	(2)	$14.7M + 22A + 14.7N$	$12.7M + 19A + 12.7N$
1	(2)	$13.5M + 20.3A + 13.5N$	$11.5M + 17.3A + 11.5N$
2	(2)	$13.1M + 19.7A + 13.1N$	$11.1M + 16.7A + 11.1N$
3	(2)	$12.8M + 19.2A + 12.8N$	$10.8M + 16.2A + 10.8N$

The efficiency improvement brought by Longa's patterns (2) and (3) over pattern (1) can be seen in tables 1.37 and 1.38 in Section 1.2.2.4. We present in Section 1.3 a new atomic pattern for the right-to-left atomic algorithm that brings better performances than those presented above in most cases.

1.2.2.3 Euclidean Addition Chains

Let us consider a binary sequence $c = (c_i)_{1 \leq i \leq m}$, $c_i \in \{0, 1\}$, and the sequence $v = (v_i)_{-2 \leq i \leq m}$ defined by $v_{-2} = 1, v_{-1} = 2, v_0 = 3$ and for $1 \leq i \leq m$:

$$v_i = v_{i-1} + v_{j_i} \text{ with } \begin{cases} j_i = i - 2 & \text{if } c_i = 0 \text{ (\textit{big step})} \\ j_i = j_{i-1} & \text{if } c_i = 1 \text{ (\textit{small step})} \\ j_0 = -2. & \end{cases}$$

The sequence v is called an *Euclidean Addition Chain (EAC)* and is completely defined by the binary sequence c. As the purpose of c and v is to compute v_m, we consider that c is a (non unique) representation of v_m, and denote $v_m = \chi(c)$. By abuse of language, we will say that c is an EAC computing v_m. We will see in the following that every integer greater than 3 can be computed by an EAC.

For example, let $c = (0110100)$ be an EAC, one can compute $\chi(c) = 62$ as follows:

i				1	2	3	4	5	6	7
v_i	1	2	3	5	7	9	16	23	39	**62**
c_i				0	1	1	0	1	0	0
v_{j_i}				2	2	2	7	7	16	23

Remark that by construction of the sequence v, computing $v_1, v_2, \ldots, v_m$ involves only additions of different terms. As a consequence, the sequence $v_1P, v_2P, \ldots, v_mP$ can be computed using point additions only, assuming that v_m is less than[8] the order of P. Meloni observes that a scalar multiplication algorithm evaluating an EAC can perform all point additions using the co-Z Jacobian formula [Mel07], as presented in Algorithm 1.19.

[8]Note that, even if v_m is greater than $n = \text{ord}(P)$, the probability that $v_i \equiv v_j \pmod{n}$, with $0 < i, j < m$, and $i \neq j$, and thus that a doubling possibly occurs in the addition chain, is negligible in cryptographic applications where n is large.

Algorithm 1.19 Euclidean addition chain scalar multiplication using ZADDU

Input: $P = (x, y) \in \mathcal{E}(\mathbb{F}_q)$, $c = (c_i)_{1 \leq i \leq m}$ a binary sequence with $\chi(c) = k$
Output: kP
Uses: U_0 and U_1

1: $U_0 \leftarrow (X_{2P} : Y_{2P} : Z_{2P})$ ▷ Jacobian representation of $2P$
2: $U_1 \leftarrow (xZ_{2P}{}^2 : yZ_{2P}{}^3 : Z_{2P})$ ▷ Jacobian representation of P
3: **for** $i = 1$ **to** m **do**
4: $\quad (U_0, U_1) \leftarrow \text{ZADDU}(U_{c_i}, U_{\neg c_i})$
5: **return** $U_0 + U_1$

This method both provides SSCA immunity and benefits of the efficiency of the co-Z Jacobian addition. Moreover, Algorithm 1.19 is easy to implement and fits the memory constraints of embedded devices.

Meloni [Mel07] observes that an x-only variant of this algorithm can be devised using ZADDU' operation as presented in Algorithm 1.20. This method saves one field multiplication per loop iteration. The overhead of the final x coordinate recovery is $11M + 4S + 8A$.

Algorithm 1.20 Euclidean addition chain $(X : Y)$-only scalar multiplication using ZADDU′

Input: $P = (x, y) \in \mathcal{E}(\mathbb{F}_q)$, $c = (c_i)_{1 \leq i \leq m}$ a binary sequence with $\chi(c) = k$, $\lambda = 2b/a$
Output: x coordinate of kP
Uses: U_0, U_1, and R

1: $U_0 \leftarrow (X_{2P} : Y_{2P})$
2: $U_1 \leftarrow (xZ_{2P}{}^2 : yZ_{2P}{}^3)$
3: **for** $i = 1$ **to** m **do**
4: $\quad (U_0, U_1) \leftarrow \text{ZADDU}'(U_{c_i}, U_{\neg c_i})$
5: $(R, T) \leftarrow \text{ZADDU}'(U_0, U_1)$ ▷ T is discarded
6: $A \leftarrow Y_0{}^2 - Y_1{}^2 + X_1{}^3 - X_0{}^3$ ▷ with $U_0 = (X_0 : Y_0)$ and $U_1 = (X_1 : Y_1)$
7: $B \leftarrow X_R + 2Y_0Y_1 - X_0X_1(X_0 + X_1)$
8: $C \leftarrow B(X_0 - X_1) - A(X_0 + X_1)$
9: $x \leftarrow \lambda X_R A(C(X_1 - X_0)^2)^{-1}$ ▷ $(X_1 - X_0)^2$ is computed at step 5
10: **return** x

Using algorithms 1.19 or 1.20 to compute a scalar multiplication kP requires the knowledge of an EAC computing k. This question is discussed by Meloni [Mel07], we just recall hereafter some key points.

EAC Arithmetic First, note that an EAC v of length $m \geq 1$ can be recovered from the last two terms v_m and v_{m-1} using Euclid's algorithm, for instance:

$$\begin{aligned} \mathbf{62} - 1 \times 39 &= 23 \\ \mathbf{39} - 1 \times 23 &= 16 \\ \mathbf{23} - 1 \times 16 &= 7 \\ \mathbf{16} - 2 \times 7 &= 2 \\ \mathbf{7} - 3 \times 2 &= 1 \\ \mathbf{2} - 1 \times 1 &= 1 \\ \mathbf{1} - 1 \times 1 &= 0 \end{aligned}$$

Let q_0, q_1, ..., q_s denote the successive quotients of Euclid's algorithm as presented above. We define $\kappa'(v_m, v_{m-1})_i$ the binary sequence such that:

$$\kappa'(v_m, v_{m-1}) = \underbrace{11\ldots1}_{q_s-1}0\underbrace{11\ldots1}_{q_{s-1}-1}\ \ldots\ 0\underbrace{11\ldots1}_{q_0-1}$$

Then, let $\kappa(v_m, v_{m-1})$ the sequence obtained by dropping the first and last bits of $\kappa'(v_m, v_{m-1})$. Given $v_m > v_{m-1} \geq 3$ and $\gcd(v_m, v_{m-1}) = 1$, the sequence $\kappa(v_m, v_{m-1})$ is the binary EAC computing v_m with penultimate term v_{m-1}. Alternatively, the bits of $\kappa(v_m, v_{m-1})$ can be computed one by one using the subtractive form of Euclid's algorithm [Mel07].

As a consequence, the EACs evaluating an integer k can be enumerated by browsing integers $g < k$ coprime to k. Remark that $\kappa(k, g_1) = \kappa(k, g_2)$ if and only if $g_1 = g_2$ or $g_1 = k - g_2$. Assume $k > 2$, since $g_1 = k - g_2$ implies $g_1 \neq g_2$ — otherwise $k = 2g_2$, and $\gcd(k, g_2) = 1$ implies $k = 2$ —, we have Theorem 1.

Theorem 1. *Let $k > 2$ be an integer, there are $\varphi(k)/2$ different EACs evaluating k, where φ is Euler's totient.*

Now, let us call the reverse of an EAC $c = (c_i)_{1 \leq i \leq m}$ the EAC $\overline{c} = (\overline{c}_i)_{1 \leq i \leq m}$ such that $c_i = \overline{c}_{n-i+1}$ for $1 \leq i \leq n$. As stated by Theorem 2, it is remarkable that two reversed EACs evaluate the same integer, e.g.:

i				1	2	3	4	5	6	7
v_i	1	2	3	5	8	11	19	27	35	**62**
c_i				0	0	1	0	1	1	0
v_{j_i}				2	3	3	8	8	8	27

Theorem 2. *Let c be an EAC, then $\chi(c) = \chi(\overline{c})$. Besides, if $c = \kappa(k, g)$ then $\overline{c} = \kappa(k, h)$ with $h = \pm g^{-1} \bmod k$.*

Proof. Let us consider two integers k and g, and the binary sequence $c = \kappa(k, g)$, such that $k > g \geq 3$, and $\gcd(k, g) = 1$. Furthermore, let q_0, q_1, ..., q_s denote the successive quotients of Euclid's algorithm run on k and g. We have:

$$\begin{pmatrix} q_0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} q_1 & 1 \\ 1 & 0 \end{pmatrix} \cdots \begin{pmatrix} q_s & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} k & u \\ g & v \end{pmatrix}$$

where u and v are Bézout coefficients of (k, g) such that $gu - kv = (-1)^s$. If we transpose both sides of this equation, we have:

$$\begin{pmatrix} q_s & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} q_{s-1} & 1 \\ 1 & 0 \end{pmatrix} \cdots \begin{pmatrix} q_0 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} k & g \\ u & v \end{pmatrix}$$

Thus $\kappa(k, u) = \overline{c}$, with $u = (-1)^s g^{-1} \bmod k$, and, since $\chi(\kappa(k, u)) = \chi(\kappa(k, g)) = k$ by definition, then $\chi(c) = \chi(\overline{c})$. □

Considering $k = (k_{l-1} k_{l-2} \ldots k_0)_2$ Meloni explores [Mel07] ways to find a *short* EAC evaluating k, where short means a length $m \leq 2l$. A basic method consists in picking at random g's coprime to k and applying Euclid's algorithm. For $l = 160$, Meloni reports that a few tens of g should be tried to find an EAC of length 320, and thousands of g's should be tried to find a very short EAC of length 270.

Obviously, such computations are prohibitive for a use in the context of embedded devices using a random scalar — as required for example in the ECDSA protocol. This conclusion has led other works to study the use of random EACs in scalar multiplication as presented in the rest of this section.

Random EAC Generation A different approach from finding a small EAC evaluating a random integer k consists in randomly generating a small length EAC c and using it directly as input of Algorithm 1.19 — note that $\chi(c)$ is inexpensive to compute. Such a method raises two points: (i) what is the expected length of $\chi(c)$, and (ii) is the distribution of $\chi(c)$ compatible with key generation security requirements regarding the discrete logarithm problem ?

Two families of EACs with good properties towards these two points are studied by Herbaut, Liardet, Meloni, Teglia, and Véron [HLM+10].

- Family $\mathcal{M}_l^0$ Let $\mathcal{M}_l^0$ denote the set of EAC of length $m = 2l$ beginning with l zeros. First, the authors prove that the restriction of χ to $\mathcal{M}_l^0$ is injective, and second prove sharp bounds:

$$\chi(\mathcal{M}_l^0) \subset [(l+1)F_{l+2} + F_{l+3}, F_{2l+4}]$$

where F_i is the i^{th} Fibonacci number. Moreover, the mean value is $\left(\frac{3}{2}\right)^l F_{l+4}$.

Thus, 2^l distinct scalars can be generated using chains from $\mathcal{M}_l^0$. However, since $F_{2l+4} > 2^l$, with $l \in \mathbb{N}$, if we consider a random EAC $c \in \mathcal{M}_l^0$ to be used for scalar multiplication $\kappa(c)P$ on an l-bit curve where $\#\langle P \rangle = n_P \approx 2^l$, the output is $(\kappa(c) \bmod n_P)P$, and $c \to \kappa(c) \bmod n_P$ is not injective in general.

Nevertheless, experiments presented by Herbaut et al. [HLM+10] seem to show that the average distribution of $\kappa(c) \bmod n$, $c \in \mathcal{M}_l^0$, $n \approx 2^l$ into $[0, n]$ is close enough to uniform distribution to be used for cryptographic purposes.

Under this assumption, we compute the cost of using this family of chains for scalar multiplication on an l-bit curve. One co-Z addition costs $5M + 2S + 7A$ and the main loop of Algorithm 1.19 involves $2l$ of them if $c \in \mathcal{M}_l^0$, which yields an equivalent cost per scalar bit of $10M + 4S + 14A$. This cost drops to $8M + 4S + 14A$ using Algorithm 1.20. The corresponding detailed costs are given in Table 1.40 (p. 66), depending on ratios S/M and A/M.

- **Family** $\mathcal{M}_{\frac{3l}{2},\frac{l}{2}}$ Let $\mathcal{M}_{\frac{3l}{2},\frac{l}{2}}$ denote the set of EACs of length $m = \frac{3l}{2}$ of Hamming weight $\frac{l}{2}$. This choice is motivated by the fact that, given a chain length, EACs with a low Hamming weight have more big steps and thus produce bigger integers.

This family does not have the injectivity property of $\mathcal{M}_l^0$. Nevertheless, Herbaut et al. conjecture that the cardinality of $\chi(\mathcal{M}_{\frac{3l}{2},\frac{l}{2}})$ is close to 2^l. Thus, under the same assumption as previously — i.e. the distribution of $\chi(c) \bmod n_P$ is close enough to uniform distribution — this family of chains would lead to faster scalar multiplication than $\mathcal{M}_l^0$.

Under these assumptions, we compute the cost of using this family of chains for scalar multiplication on an l-bit curve. These results are similar to family $\mathcal{M}_l^0$ with a factor $\frac{3}{4}$ corresponding to the length ratio. The corresponding detailed costs are given in Table 1.40 (p. 66).

1.2.2.4 Detailed Cost Comparison

The following tables summarize the cost of the algorithms presented in this section depending on the cost of field squarings and additions. Label (c.o.) denote the use of Jacobian composite operations $dP_1 + P_2$.

Table 1.34: Detailed cost of regular algorithms expressed as field multiplications per scalar bit

Algorithm	Coord.	a	$S/M = 1$		$S/M = 0.8$	
			$A/M = 0.2$	$A/M = 0.1$	$A/M = 0.2$	$A/M = 0.1$
1.8	$\mathcal{J}$	any	24.6	22.8	22.8	21.0
1.9	$\mathcal{J}/\mathcal{J}^m$	any	27.8	25.9	26.2	24.3
1.10	$\mathcal{H}$, x-only	any	21.8	20.4	20.8	19.4
1.11	$\mathcal{J}$ (c.o.)	any	25.6	24.3	24.4	23.1
1.14 1.15	$\mathcal{J}$	any	**19.6**	**17.8**	**18.6**	**16.8**
1.16	$\mathcal{J}$, $(X:Y)$-only	any	**17.6**	**15.8**	**16.6**	**14.8**
1.8	$\mathcal{J}$	-3	22.8	20.9	21.4	19.5
1.10	$\mathcal{H}$, x-only	-3	20.6 (i)	18.4 (ii)	19.6 (i)	17.4 (ii)

Table 1.35: Detailed cost of regular m-ary ladders* depending on the radix m, expressed as field multiplications per scalar bit

Algorithm	m	Coord.	a	$S/M = 1$		$S/M = 0.8$	
				$A/M = 0.2$	$A/M = 0.1$	$A/M = 0.2$	$A/M = 0.1$
1.12	2	$\mathcal{J}$ (c.o.)	any	23.6	22.3	22.6	21.3
1.13	2	$\mathcal{J}/\mathcal{J}^m$	any	27.8	25.9	26.2	24.3
1.12	4	$\mathcal{J}$ (c.o.)	any	**17.9**	**16.7**	**16.8**	**15.6**
1.13	4	$\mathcal{J}/\mathcal{J}^m$	any	19.1	17.6	17.9	16.4
1.12	8	$\mathcal{J}$ (c.o.)	any	15.4	14.2	14.4	13.2
1.13	8	$\mathcal{J}/\mathcal{J}^m$	any	16.2	14.8	15.1	13.7

* Radix $m = 8$ requires a prohibitive amount of memory in our context.

Table 1.36: Detailed average cost of Algorithm 1.17 using Weierstraß and twisted Edwards unified additions expressed as field multiplications per scalar bit

Model	Coord.	Param.	$S/M = 1$		$S/M = 0.8$	
			$A/M = 0.2$	$A/M = 0.1$	$A/M = 0.2$	$A/M = 0.1$
Weierstraß	$\mathcal{H}$	any a	29.1	28.1	27.6	26.6
Weierstraß	$\mathcal{H}$	$a = -3$	28.2	26.9	26.7	25.4
Twisted Ed.	$\mathcal{E}^e$	any a, d	18.6	17.6	18.6	17.6
Twisted Ed.	$\mathcal{E}^e$	$a = -1$, any d	**16.2**	**14.9**	**16.2**	**14.9**

Table 1.37: Detailed average cost of left-to-right window NAF atomic algorithms using pattern (1), (2) or (3) and mixed affine-projective additions, expressed as field multiplications per scalar bit and assuming $N/M = 0.1$

Extra pts.	Coord.	Pattern	a	$S/M = 1$		$S/M = 0.8$	
				$A/M = 0.2$	$A/M = 0.1$	$A/M = 0.2$	$A/M = 0.1$
NAF	$\mathcal{J}$	(1)	any	20.5	17.8	-	-
	$\mathcal{J}$	(2) or (3)	any	19.6	17.5	18.2*	16.1*
1	$\mathcal{J}$	(1)	any	19.1	16.6	-	-
	$\mathcal{J}$	(2) or (3)	any	18.2	16.3	16.9*	15.0*
2	$\mathcal{J}$	(1)	any	18.7	16.2	-	-
	$\mathcal{J}$	(2) or (3)	any	17.7	15.8	16.5*	14.6*
3	$\mathcal{J}$	(1)	any	18.3	15.9	-	-
	$\mathcal{J}$	(2) or (3)	any	**17.4**	**15.5**	**16.1***	**14.3***
NAF	$\mathcal{J}$	(1)	-3	17.5	15.2	-	-
	$\mathcal{J}$	(2) or (3)	-3	16.8	15.0	15.6*	13.8*
1	$\mathcal{J}$	(1)	-3	16.1	14.0	-	-
	$\mathcal{J}$	(2) or (3)	-3	15.4	13.8	14.3*	12.7*
2	$\mathcal{J}$	(1)	-3	15.7	13.6	-	-
	$\mathcal{J}$	(2) or (3)	-3	14.9	13.3	13.9*	12.3*
3	$\mathcal{J}$	(1)	-3	15.3	13.3	-	-
	$\mathcal{J}$	(2) or (3)	-3	**14.6**	**13.0**	**13.5***	**12.0***

* Using pattern (3)

Table 1.38: Detailed average cost of left-to-right window NAF atomic algorithms using pattern (1) or (2) and readditions, expressed as field multiplications per scalar bit and assuming $N/M = 0.1$

Extra pts.	Coord.	Pattern	a	$A/M = 0.2$	$A/M = 0.1$
NAF	$\mathcal{J}$	(1)	any	22.0	19.1
	$\mathcal{J}$	(2)	any	20.5	18.3
1	$\mathcal{J}$	(1)	any	20.3	17.6
	$\mathcal{J}$	(2)	any	18.9	16.9
2	$\mathcal{J}$	(1)	any	19.7	17.0
	$\mathcal{J}$	(2)	any	18.4	16.4
3	$\mathcal{J}$	(1)	any	19.2	16.6
	$\mathcal{J}$	(2)	any	**17.9**	**16.0**
NAF	$\mathcal{J}$	(1)	-3	19.0	16.5
	$\mathcal{J}$	(2)	-3	17.7	15.8
1	$\mathcal{J}$	(1)	-3	17.3	15.0
	$\mathcal{J}$	(2)	-3	16.1	14.4
2	$\mathcal{J}$	(1)	-3	16.7	14.4
	$\mathcal{J}$	(2)	-3	15.6	13.9
3	$\mathcal{J}$	(1)	-3	16.2	14.0
	$\mathcal{J}$	(2)	-3	**15.1**	**13.5**

Table 1.39: Detailed average cost of right-to-left window NAF atomic algorithms using pattern (1) and mixed $\mathcal{J}/\mathcal{J}^m$ coordinates, expressed as field multiplications per scalar bit and assuming $N/M = 0.1$

Extra pts.	Coord.	Pattern	a	$A/M = 0.2$	$A/M = 0.1$
NAF	$\mathcal{J}/\mathcal{J}^m$	(1)	any	20.0	17.3
1	$\mathcal{J}/\mathcal{J}^m$	(1)	any	18.0	15.6
2	$\mathcal{J}/\mathcal{J}^m$	(1)	any	17.3	15.0
3	$\mathcal{J}/\mathcal{J}^m$	(1)	any	**16.8**	**14.6**

Table 1.40: Detailed average cost of algorithms 1.19 and 1.20 using EAC families $\mathcal{M}_l^0$ and $\mathcal{M}_{\frac{3l}{2},\frac{l}{2}}$, expressed as field multiplications per bit of an equivalent l-bit scalar

Algorithm	EAC family	Coord.	a	$S/M = 1$		$S/M = 0.8$	
				$A/M = 0.2$	$A/M = 0.1$	$A/M = 0.2$	$A/M = 0.1$
1.19	$\mathcal{M}_l^0$	$\mathcal{J}$	any	16.8	15.4	16.0	14.6
1.20	$\mathcal{M}_l^0$	$\mathcal{J}$, x-only	any	**14.8**	**13.4**	**14.0**	**12.6**
1.19	$\mathcal{M}_{\frac{3l}{2},\frac{l}{2}}{}^*$	$\mathcal{J}$	any	12.6	11.6	12.0	11.0
1.20	$\mathcal{M}_{\frac{3l}{2},\frac{l}{2}}{}^*$	$\mathcal{J}$, x-only	any	11.1	10.1	10.5	9.5

* The distribution of scalars generated using EAC family $\mathcal{M}_{\frac{3l}{2},\frac{l}{2}}$ is not proven to be safe for cryptographic applications.

Cost Comparison We compare hereafter the costs of the most efficient scalar multiplication techniques immune to SSCA presented above. As previously, costs are computed as:

$$\frac{(l - [\text{offset}]) \times [\text{cost per scalar bit}] + [\text{optional pre/postcomputation cost}]}{l}$$

Offset is fixed to 1 for most algorithms, except those listed below:

- Algorithms 1.11 and 1.15 have offset set to 2 using the tripling trick described in Algorithm 1.15. The cost of a tripling from affine to Jacobian coordinates is therefore added to the final cost. This trick assumes the scalar least significant bit to be 1, which requires an addition with $-P$ at the end of the scalar multiplication one time out of two, thus the cost of a mixed affine-projective addition is added to the final cost.
- Left-to-right window NAF algorithms using 1 extra point have offset set to 2.
- Left-to-right window NAF algorithms using 2 or 3 extra points have offset set to 3.
- Algorithm 1.19 used for EAC evaluation has offset set to 0 and an overhead of one fast doubling from affine coordinates and one co-Z addition. Algorithm 1.20 has the same properties with an additional overhead of $11M + 4S + 8A$.

Considering $S/M = 0.8$, $A/M = 0.2$, and $I/M = 100$, Figure 1.9 compares the cost per scalar bit of these algorithms depending on the scalar length in the general case (any a). Figure 1.10 gives a similar comparison assuming $a = -3$. For a correct comparison between methods, remember that left-to-right window NAF algorithms using readditions require the storage of five coordinates X, Y, Z, Z^2, Z^3 per extra point (including the base point).

Considering general elliptic curves and our cost assumptions, both figures show that:

i) $(X:Y)$-only co-Z Montgomery ladder algorithm (1.16) and x-only EAC-based scalar multiplication algorithm (1.20) are the most efficient regular algorithms,

ii) left-to-right atomic window NAF scalar multiplication using mixed affine-projective additions and pattern (3) is the most efficient atomic technique.

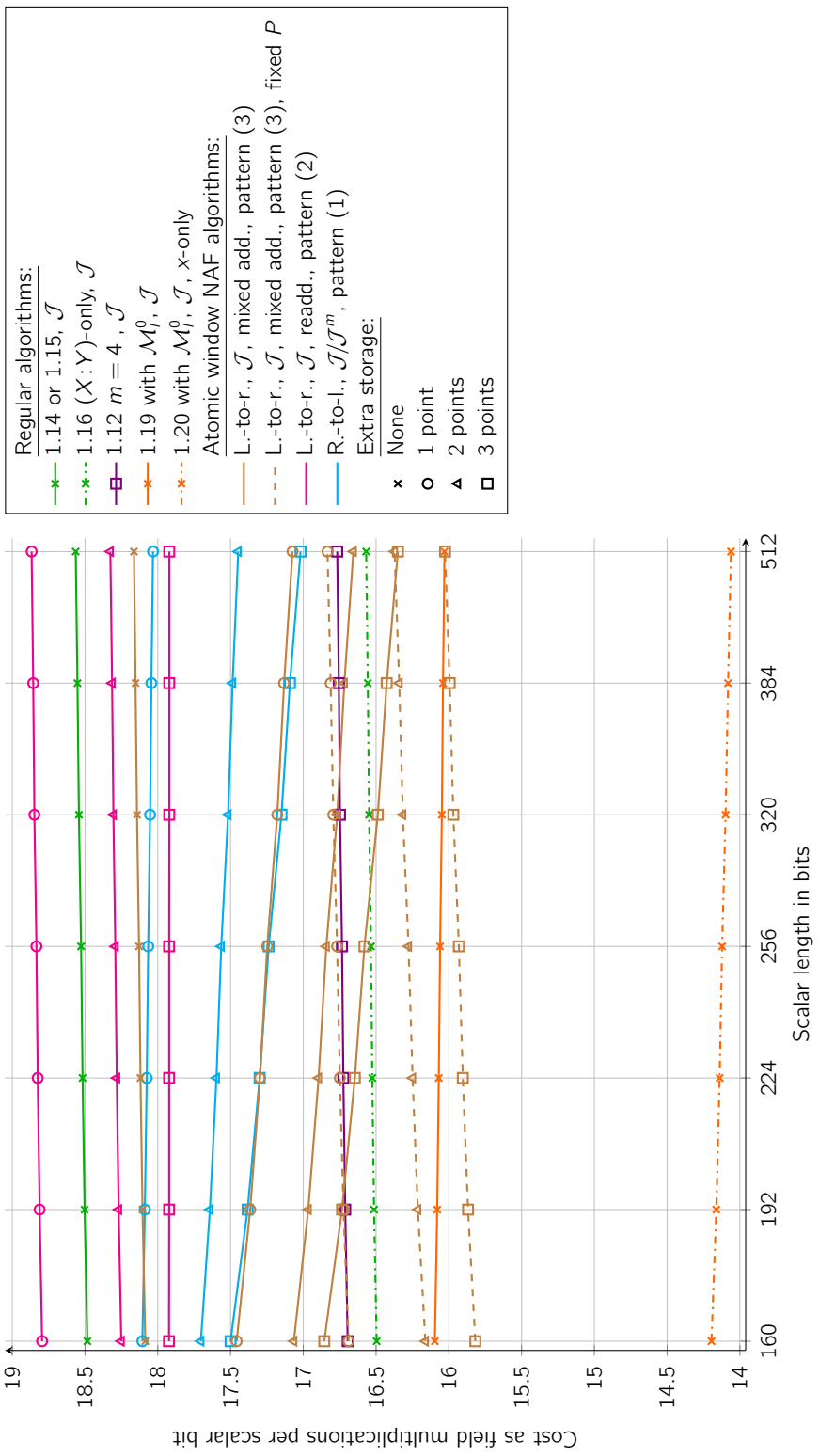


Figure 1.9: Comparison of SSCA-protected scalar multiplication algorithms cost depending on scalar length for any a, assuming $I/M = 100$, $S/M = 0.8$, and $A/M = 0.2$

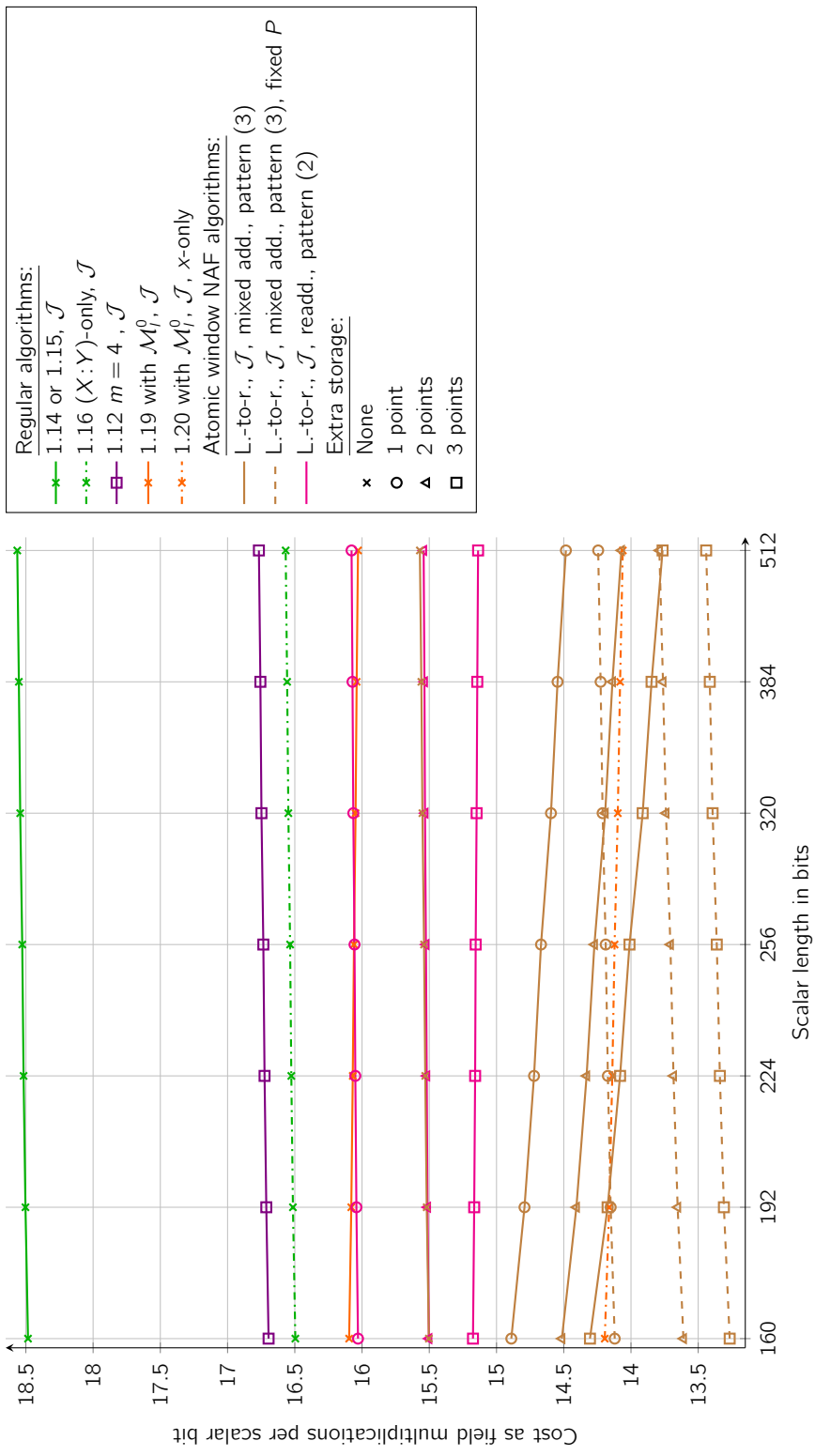


Figure 1.10: Comparison of SSCA-protected scalar multiplication algorithms cost depending on scalar length for $a = -3$, assuming $I/M = 100$, $S/M = 0.8$, and $A/M = 0.2$

1.3 Improved Atomic Pattern for Right-to-Left Algorithms

In this section, we present our contribution to the implementation of the atomicity principle [GV10]. Namely, we propose a new atomic pattern for Algorithm 1.5 using mixed Jacobian and modified Jacobian representations.

We propose hereafter a twofold atomicity improvement method. First, we show in Section 1.3.1 how to take advantage of the efficiency of a dedicated squaring when it is available. Secondly, in Section 1.3.2 we reduce the number of additions and negations used in atomic patterns.

In Section 1.3.3 we evaluate the theoretical improvement provided by our technique and we present some practical results in Section 1.3.4. Interestingly, our method provides some of the best efficiency results of all the techniques presented so far. Note that this method has been the subject of a patent deposit while at Oberthur Technologies [GV09].

1.3.1 Atomic Pattern Extension

Let O_1 and O_2 be two atomically written operations such that they require m_1 and m_2 atomic patterns respectively. Let us assume that a sub-operation o_1 from the atomic pattern could sometimes be replaced by another preferred sub-operation o_2. Let us eventually assume that O_1 requires at least m'_1 sub-operations o_1 (along with $m_1 - m'_1$ sub-operations o_2) and that O_2 requires at least m'_2 sub-operations o_1 (along with $m_2 - m'_2$ sub-operations o_2).

Then, if $d = \gcd(m_1, m_2) > 1$, let e be the greatest positive integer satisfying:

$$e \cdot \frac{m_1}{d} \leq m_1 - m'_1 \quad \text{and} \quad e \cdot \frac{m_2}{d} \leq m_2 - m'_2 \ . \tag{1.31}$$

If $e > 0$ we can now apply the following method. Let a new pattern be defined with $d - e$ original atomic patterns followed by e atomic patterns with o_2 replacing o_1 — the order can be modified at convenience.

It is now possible to express operations O_1 and O_2 with m_1/d and m_2/d new patterns respectively. Using the new pattern in O_1, resp. O_2, instead of the old one allows replacing $e \cdot m_1/d$, resp. $e \cdot m_2/d$, sub-operations o_1 by o_2.

Applying this method to Algorithm 1.5 yields the following result: O_1 being the Jacobian projective addition, O_2 the modified Jacobian projective doubling, o_1 the field multiplication and o_2 the field squaring, then m_1=16, m'_1=11, $m_2 = 8$, $m'_2 = 3$, $d = 8$ and $e = 2$. Therefore we define a new temporary atomic pattern composed of 8 patterns (1) where 2 multiplications are replaced by squarings. We thus have one fourth of the field multiplications carried out as field squarings. This extended pattern would have to be repeated twice for an addition and once for a doubling.

We applied this new approach in Figure 1.11 where atomic general Jacobian addition and modified Jacobian doubling are rewritten in order to take advantage of the squarings. As previously, the dummy field additions and negations are denoted by $\star$.

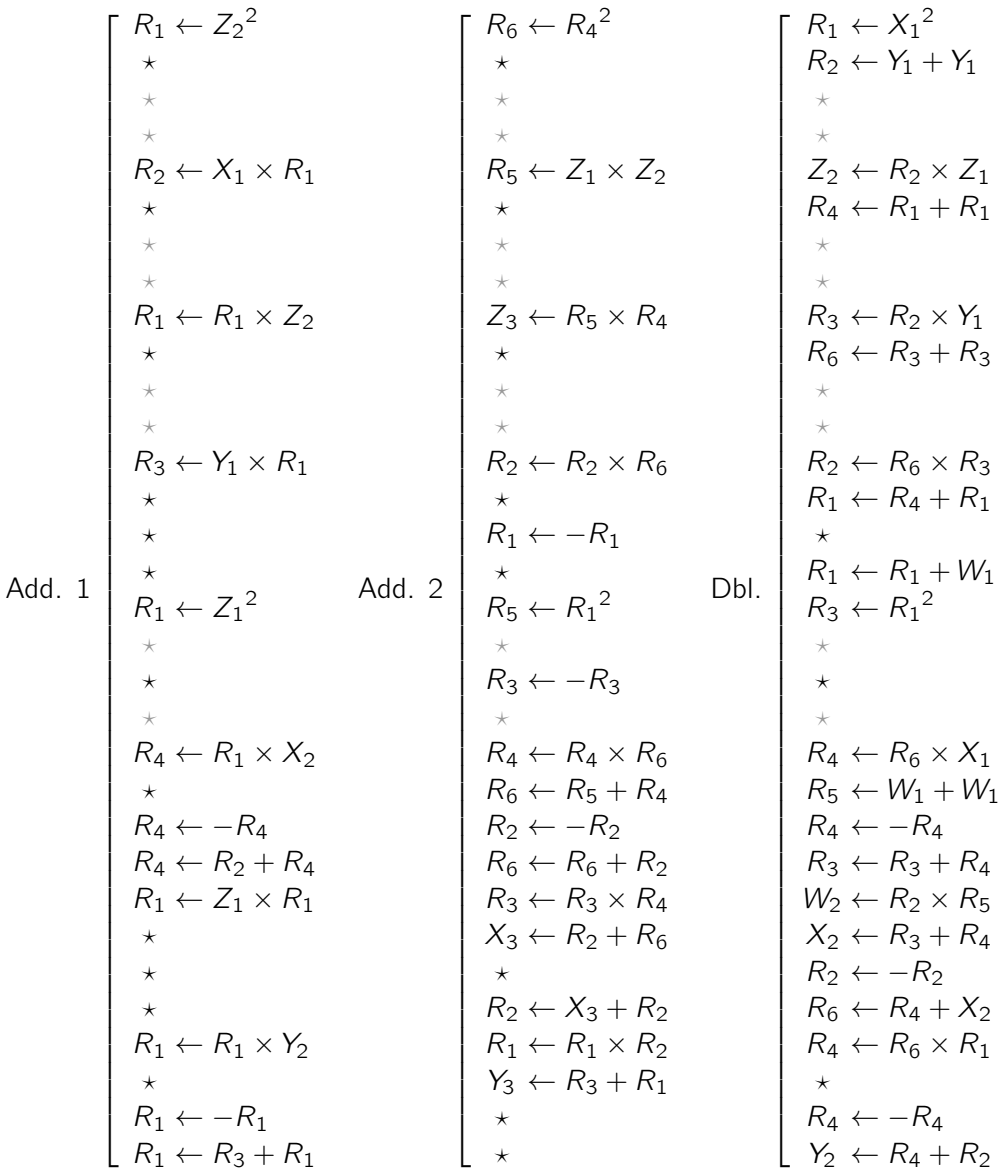


Figure 1.11: Extended atomic pattern applied to $\mathcal{J}$ addition and $\mathcal{J}^m$ doubling

1.3.2 Atomic Pattern Cleanup

In a second step we aim at reducing the number of dummy field operations. In Figure 1.11, we identified by $\star$ the operations that are never used in Add.1, Add.2, and Dbl. These field operations may then be removed saving up 5 field additions and 3 field negations per pattern occurrence.

However, we found out that field operations could be rearranged in order to maximize the number of rows over the three columns composed of dummy operations only. We then merge negations and additions into subtractions when possible. This improvement is depicted in Figure 1.12.

This final optimization saves 6 field additions and removes the 8 field negations per pattern occurrence. One may note that no more dummy operation remains in modified Jacobian doubling. We thus believe that our atomic pattern (4) is optimal for this algorithm:

$$(4)\quad \left[\begin{array}{l} R_{u(0)} \leftarrow {R_{u(1)}}^2 \\ R_{u(2)} \leftarrow R_{u(3)} + R_{u(4)} \\ R_{u(5)} \leftarrow R_{u(6)} \times R_{u(7)} \\ R_{u(8)} \leftarrow R_{u(9)} + R_{u(10)} \\ R_{u(11)} \leftarrow R_{u(12)} \times R_{u(13)} \\ R_{u(14)} \leftarrow R_{u(15)} + R_{u(16)} \\ R_{u(17)} \leftarrow R_{u(18)} \times R_{u(19)} \\ R_{u(20)} \leftarrow R_{u(21)} + R_{u(22)} \\ R_{u(23)} \leftarrow R_{u(24)} + R_{u(25)} \\ R_{u(26)} \leftarrow {R_{u(27)}}^2 \\ R_{u(28)} \leftarrow R_{u(29)} \times R_{u(30)} \\ R_{u(31)} \leftarrow R_{u(32)} + R_{u(33)} \\ R_{u(34)} \leftarrow R_{u(35)} - R_{u(36)} \\ R_{u(37)} \leftarrow R_{u(38)} \times R_{u(39)} \\ R_{u(40)} \leftarrow R_{u(41)} - R_{u(42)} \\ R_{u(43)} \leftarrow R_{u(44)} - R_{u(45)} \\ R_{u(46)} \leftarrow R_{u(47)} \times R_{u(48)} \\ R_{u(49)} \leftarrow R_{u(50)} - R_{u(51)} \end{array}\right.$$

1.3.3 Results

We give in Table 1.41 the average cost of Algorithm 1.5 and of similar algorithms using window NAF methods protected with atomic pattern (4) given the amount of available extra memory. Table 1.42 presents the corresponding detailed average costs depending on ratios S/M and A/M, postcomputations are not taken into account.

The gain of our atomic pattern (4) over the original pattern (1) is constant whatever the window size and ranges from 13.5 % to 20.0 %[9] (cf. Table 1.39).

[9] Assuming $S/M = 0.8$, the precise theoretical gain is 20.0 %, resp. 17.3 %, if A/M is 0.2, resp. 0.1. Assuming $S/M = 1$, the gain is 16.7 %, resp. 13.5 %.

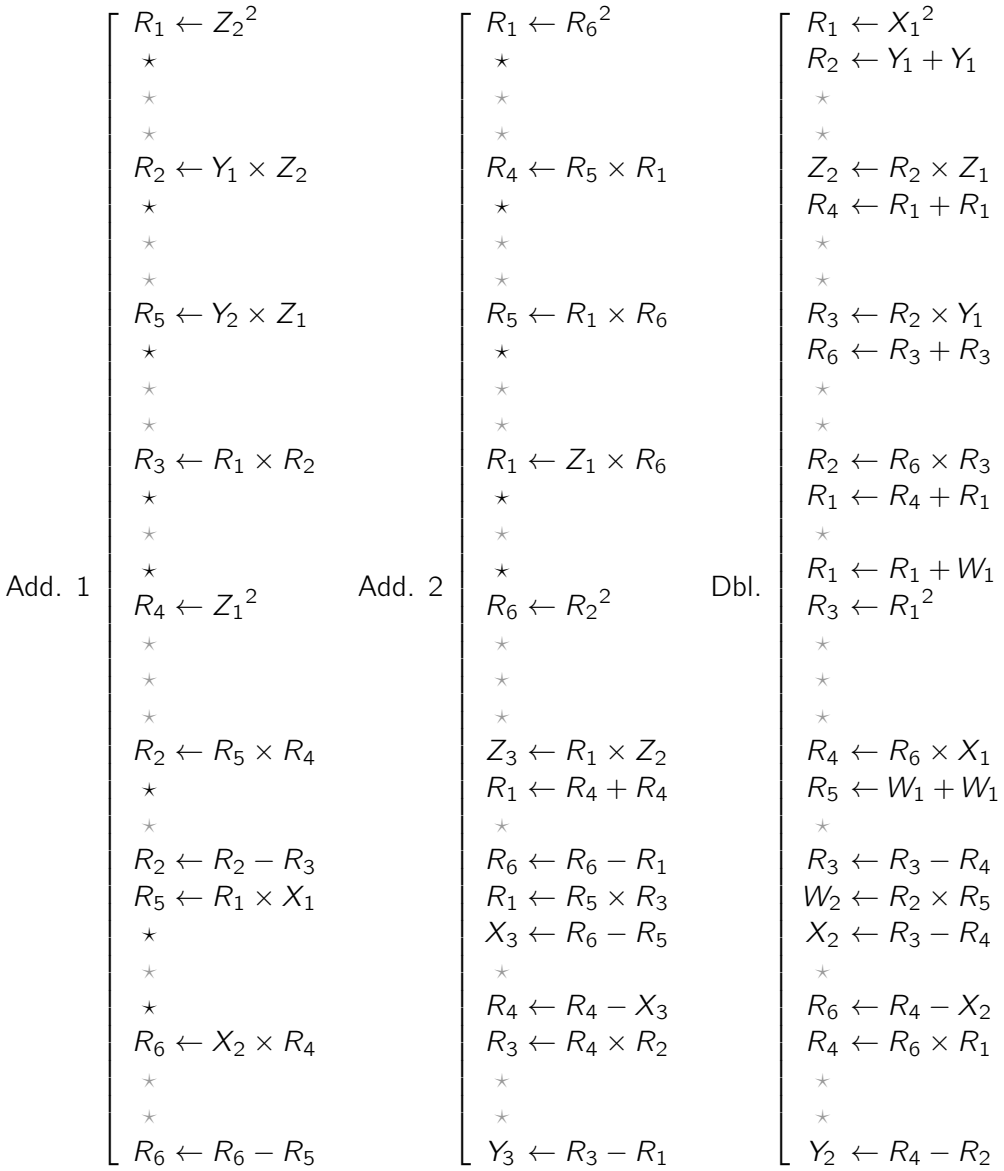


Figure 1.12: Improved arrangement of field operations in extended atomic pattern from Figure 1.11

Table 1.41: Average cost per scalar bit of right-to-left window NAF atomic algorithms using pattern (4) and mixed $\mathcal{J}/\mathcal{J}^m$ coordinates, depending on the windowing strategy

Extra pts.	Coord.	Cost for any a
0 (NAF)	$\mathcal{J}/\mathcal{J}^m$	$10M + 3.3S + 16.7A$
1	$\mathcal{J}/\mathcal{J}^m$	$9M + 3S + 15A$
2	$\mathcal{J}/\mathcal{J}^m$	$8.7M + 2.9S + 14.4A$
3	$\mathcal{J}/\mathcal{J}^m$	$8.4M + 2.8S + 14A$

Table 1.42: Detailed average cost of right-to-left window NAF atomic algorithms using pattern (4) and mixed $\mathcal{J}/\mathcal{J}^m$ coordinates, as field multiplications per scalar bit

Extra pts.	Coord.	Pat.	a	$S/M = 1$		$S/M = 0.8$	
				$A/M = 0.2$	$A/M = 0.1$	$A/M = 0.2$	$A/M = 0.1$
0 (NAF)	$\mathcal{J}/\mathcal{J}^m$	(4)	any	16.7	15.0	16.0	14.3
1	$\mathcal{J}/\mathcal{J}^m$	(4)	any	15.0	13.5	14.4	12.9
2	$\mathcal{J}/\mathcal{J}^m$	(4)	any	14.4	13.0	13.9	12.4
3	$\mathcal{J}/\mathcal{J}^m$	(4)	any	**14.0**	**12.6**	**13.4**	**12.0**

Figures 1.14 and 1.15 depicts the comparison of the fastest scalar multiplication techniques protected against SSCA updated with our new atomic pattern (4) and taking account of pre/postcomputations. The fixed point strategy for the left-to-right algorithm is not presented in these comparisons, since Differential Side-Channel Analysis (DSCA) countermeasures generally require to randomize the input point or its coordinates.

For the considered scalar lengths, our method provides the best efficiency results of all methods in the general case (any a). Using the three extra points windowing strategy, it even outperforms the x-only EAC scalar multiplication algorithm using $\mathcal{M}_l^0$. If $a = -3$, the left-to-right algorithm using mixed affine-projective additions and pattern (3) is faster than our method if no extra point is used[10]. If extra points are used, our method outperforms the left-to-right algorithm due to the costly inversion in required in precomputations.

1.3.4 Implementation

We have implemented Algorithm 1.5 — without any window method — protected with the atomic pattern (1) on one hand and with our improved atomic pattern (4) on the other hand. We used a smart card provided with an 8-bit CPU running at 30 MHz

[10]This comparison assumes that no inversion is required in the left-to-right algorithm; if it were the case — e.g. using input point projective coordinates randomization — an overhead of I should be added to the cost of the left-to-right method.

and a 32-bit arithmetic coprocessor running at 50 MHz. In particular, this coprocessor provides a dedicated modular squaring, i.e. $S/M \approx 0.8$, and the modular addition cost ratio A/M for the considered key size is about 0.2. The two implementations required a comparable amount of code (3–4 kB) and RAM (about 400 B).

On the NIST P-192 curve [NIST06] we obtained a practical speed-up of about 14.5 % to be compared to the predicted 20 %. This difference can be explained by the extra software processing required in the scalar multiplication loop management, especially the on-the-fly NAF decomposition of the scalar in an SSCA-resistant way.

When observing the electromagnetic leakage of our implementation we obtained the signal presented in Figure 1.13. The atomic pattern iterations comprising 8 modular multiplications and several additions/subtractions can easily be identified.

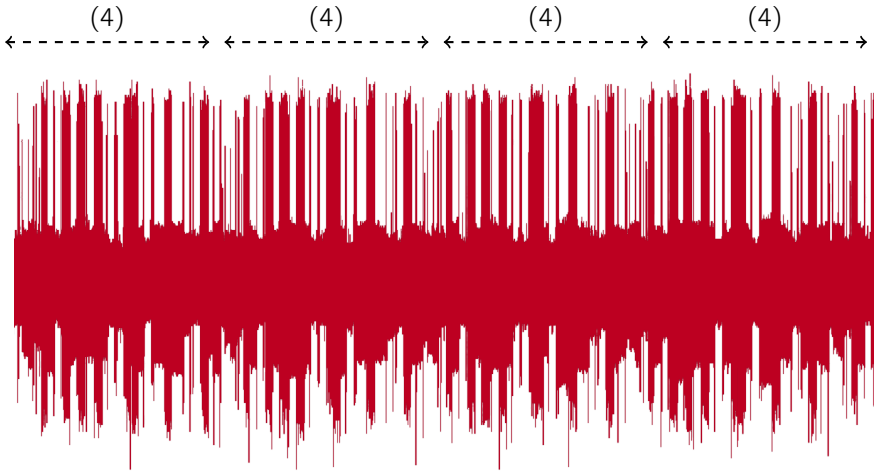


Figure 1.13: Side-channel leakage observed during the execution of our scalar multiplication implementation showing a sequence of iterations of the atomic pattern (4)

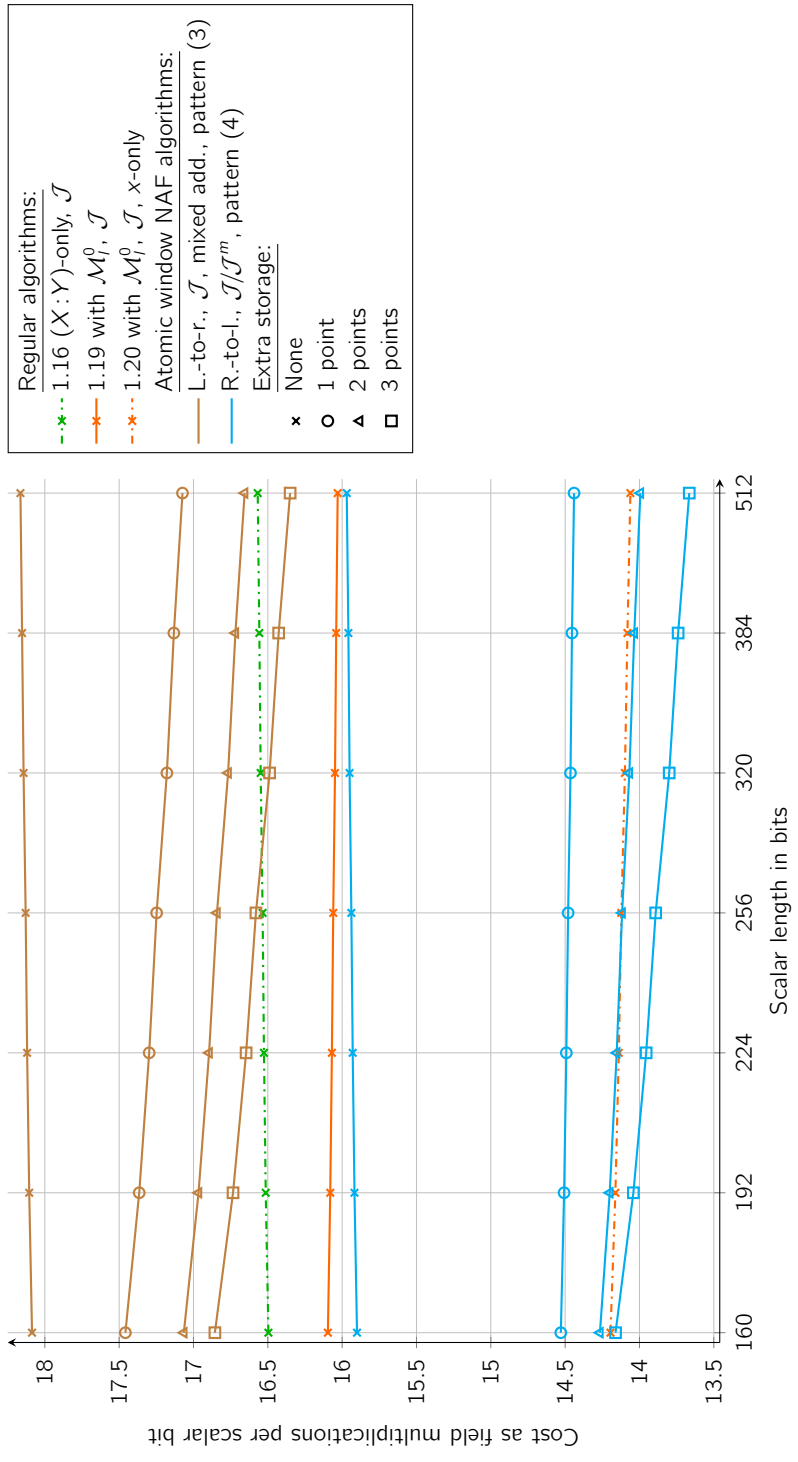


Figure 1.14: Comparison of SSCA-protected scalar multiplication algorithms cost including the new atomic pattern (4) depending on scalar length for any a, assuming $I/M = 100$, $S/M = 0.8$, and $A/M = 0.2$

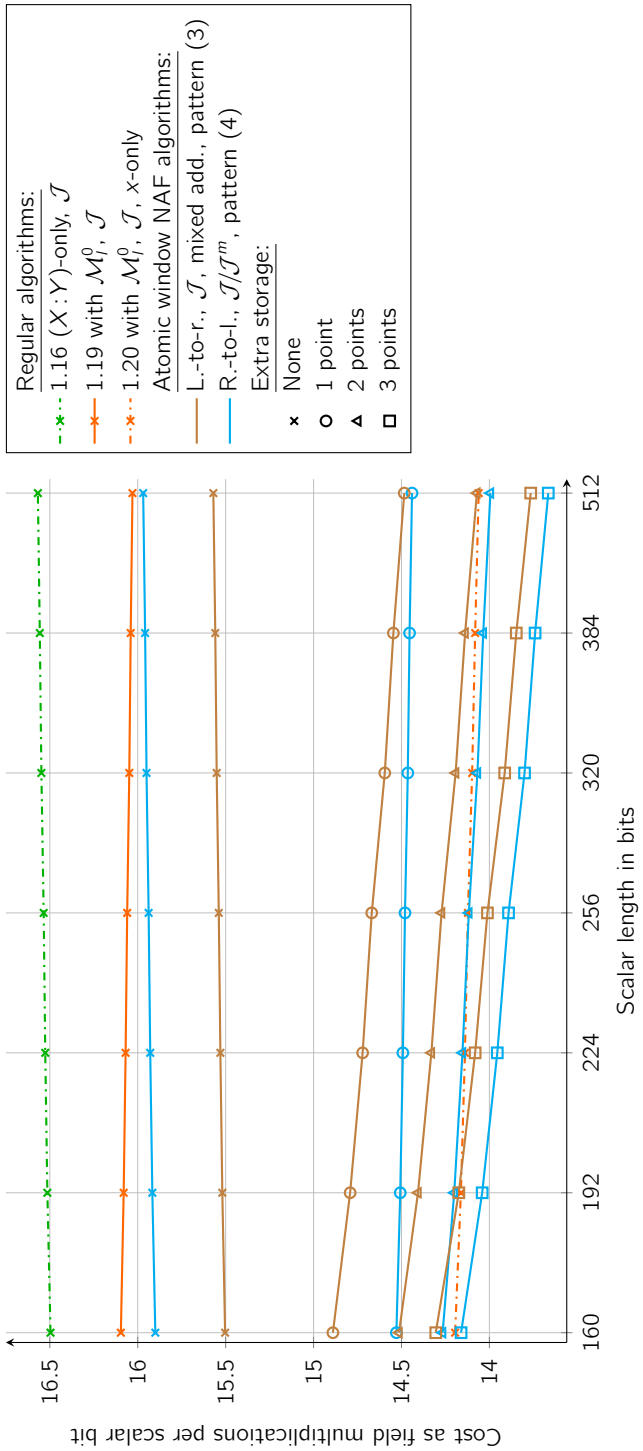


Figure 1.15: Comparison of SSCA-protected scalar multiplication algorithms cost including the new atomic pattern (4) depending on scalar length for $a = -3$, assuming $I/M = 100$, $S/M = 0.8$, and $A/M = 0.2$

1.4 Using Edwards Addition Law on Standard Curves

The efficiency of Edwards curves unified addition and doubling formulas makes them particularly interesting for cryptographic purpose in embedded devices. However, it is generally required in industrial applications to use standardized curves, and all of them have a generic Weierstraß form, cf. Section 1.1.1.5. Therefore, we study in this section the possibility of using Edwards addition law on general curves considering Theorem 3.

Theorem 3. *Let $\mathcal{E}$ be an elliptic curve defined over a field $\mathbb{F}_q$ of odd characteristic, then:*

(i) $\mathcal{E}$ is birationally equivalent to an Edwards curve over $\mathbb{F}_q$ if and only if $\mathcal{E}(\mathbb{F}_q)$ has a point of order 4,

(ii) if $\mathcal{E}(\mathbb{F}_q)$ has three points of order 2, then $\mathcal{E}(\mathbb{F}_q)$ is isogenous to an Edwards curve.

Proof. Statement *(i)* is proven by Bernstein et al. [BBJ+08, Theorem 3.3] and Morain proves [Mor09, Theorem 17] *(ii)*. □

In the following, Section 1.4.1 describes a method to switch from a standard curve to an Edwards curve considering an extension of the base field following Bernstein and Lange's construction. Similarly, Section 1.4.2 shows that an isogeny may be used to switch from a standard curve to a twisted Edwards curve. Finally, Section 1.4.3 discusses the minimal extension field degree to build isogenous Edwards curves over an odd characteristic field.

1.4.1 Standard $\mathbb{F}_p$ Curves to Edwards Form

Since all NIST and Brainpool curves over a prime field $\mathbb{F}_p$ have cofactor 1, building a birationally equivalent Edwards curve requires to find a point of order 4 over an extension of $\mathbb{F}_p$.

Assume that a point $P = (x_1, y_1)$ lies on a standard curve $\mathcal{E}_1/\mathbb{F}_p$ such that $\mathcal{E}_1(\mathbb{F}_p)$ has prime cardinality $n > 2$ and:

$$\mathcal{E}_1 : y^2 = x^3 + ax + b$$

Theorem 4. *The minimal extension field to find a point of order 4 is $\mathbb{F}_{p^6}$ for all NIST and Brainpool curves over $\mathbb{F}_p$.*

Proof. The roots of the polynomial $\psi_4^* = x^6 + 5ax^4 + 20bx^3 - 5a^2x^2 - 4abx - a^3 - 8b^2$ give the x coordinates of the points of order 4 of $\mathcal{E}_1$. Since this polynomial has no root in $\mathbb{F}_p$ for the standard curves, we need to consider an extension field $\mathbb{F}_{p^e}$ to find them. The degree of ψ_4^* ensures that $e \leq 6$. More precisely if ψ_4^* is irreducible over $\mathbb{F}_p$ then $e = 6$. This is the case for all NIST and Brainpool curves, except NIST P-224 for which ψ_4^* splits in two polynomials of degree 3.

Consider an irreducible polynomial $A(t)$ of degree e and r a root of ψ_4^* in $\mathbb{F}_p[t]/A(t)$. Notice that ψ_4^* is a suitable choice for A in most cases, in which case we can take $r = t \bmod \psi_4^*$. Then compute s a square root of $r^3 + ar + b$. If $s \notin \mathbb{F}_{p^e}$ it is necessary to consider an extension field of degree $2e$ instead of e, this is the case with the curve NIST P-224. Thus the required point (r, s) of order 4 is in $\mathbb{F}_{p^6}$ for all NIST and Brainpool curves. □

We recall in the following the construction given by Bernstein and Lange [BL07b, Section 2] and completed by Bernstein et al. [BBJ+08, Section 3] to obtain an Edwards model from a Weierstraß form. Let $(r_2, 0)$ denote the double of (r, s); it has order 2. We translate the curve $\mathcal{E}_1$ to ensure that the point $(0, 0)$ belongs to it. This is done by considering the isomorphic curve obtained with the map $(x, y) \mapsto (x - r_2, y)$. This gives:

$$\mathcal{E}_2 : y^2 = x^3 + \lambda_1 x^2 + \lambda_2 x$$

on which lies $(r_1, s_1) = (r - r_2, s)$ the point of order 4, and $(x_2, y_2) = (x_1 - r_2, y_1)$ the point corresponding to P. We have $\lambda_1 = {s_1}^2/{r_1}^2 - 2r_1$ and $\lambda_2 = r_1^2$.

Let $d = 1 - 4{r_1}^3/{s_1}^2$ and consider the Edwards curve $\mathcal{E}_3$:

$$\mathcal{E}_3 : x^2 + y^2 = 1 + dx^2y^2$$

Bernstein et al. show that $\mathcal{E}_2$ — and therefore $\mathcal{E}_1$ — is birationally equivalent to $\mathcal{E}_3$. The point corresponding to P is given by $(x_3, y_3) = (s_1x_2/r_1y_2, (x_2 - r_1)/(x_2 + r_1))$.

Let k be a positive integer. It is now possible to compute $(u_3, v_3) = k(x_3, y_3)$ using Edwards addition formula in $\mathbb{F}_{p^6}$. Note that d is a square for all NIST and Brainpool curves, thus Edwards addition law is not complete.

The corresponding result on $\mathcal{E}_2$ is obtained as:

$$(u_2, v_2) \quad = \quad \left(\frac{r_1(1 + v_3)}{1 - v_3}, \frac{s_1(1 + v_3)}{u_3(1 - v_3)} \right)$$

Finally $(u_2 + r_2, v_2) = kP$.

This method allows to perform scalar multiplications on the 5 NIST curves and the 7 Brainpool curves defined over $\mathbb{F}_p$ using Edwards addition law. However it implies handling coordinates in $\mathbb{F}_{p^6}$ during the scalar multiplication and thus increases operations cost by a factor at least 6 — which is optimistic. On the other hand, the speed-up provided by Edwards formula depends on the exact context but cannot reasonably exceed 50%. Consequently the cost of handling coordinates in $\mathbb{F}_{p^6}$ clearly overtakes the gain of using Edwards addition law.

1.4.2 Standard $\mathbb{F}_p$ Curves to Twisted Edwards Form

Using the same notations $\mathcal{E}_1$, a, b, P, k as previously, we aim at building, for each of the aforementioned standard curves, a 2-isogenous twisted Edwards curve. We use in the following the construction presented by Bernstein et al. [BBJ+08].

Theorem 5. *The minimal extension field to find three points of order 2 is $\mathbb{F}_{p^3}$ for all NIST and Brainpool curves over $\mathbb{F}_p$.*

Proof. The required points of order 2 are the $(r_i, 0)$, where r_0, r_1, r_2 the roots of the polynomial $x^3 + ax + b$. As this polynomial is irreducible over $\mathbb{F}_p$ for all standard curves, r_0, r_1, r_2 all generate $\mathbb{F}_{p^3}$. □

As previously we translate the curve $\mathcal{E}_1$ to ensure that the point of order 2 $(0, 0)$ belongs to it. Again, this is done by considering the isomorphic curve obtained by applying $(x, y) \mapsto (x - r_0, y)$. This yields

$$\mathcal{E}_2 : y^2 = x^3 - (r_1 + r_2)x^2 + (r_1 r_2)x$$

on which lies $(x_2, y_2) = (x_1 - r_0, y_1)$ the point corresponding to P.

Then we consider the isogenous curve $\mathcal{E}_3$ given by

$$\mathcal{E}_3 : y^2 = x^3 + 2(r_1 + r_2)x^2 + (r_1 - r_2)^2 x$$

on which lies $(x_3, y_3) = ({y_2}^2/{x_2}^2, y_2(r_1 r_2 - {x_2}^2)/{x_2}^2)$.

Then we consider the Montgomery curve $\mathcal{E}_4$, isomorphic to $\mathcal{E}_3$, given by:

$$\mathcal{E}_4 : \frac{1}{r_1 - r_2} y^2 = x^3 + \frac{2(r_1 + r_2)}{r_1 - r_2} x^2 + x$$

on which lies $(x_4, y_4) = (x_3/(r_1 - r_2), y_3/(r_1 - r_2))$.

Bernstein et al. have shown that every Montgomery curve is birationally equivalent to a twisted Edwards curve, in particular $\mathcal{E}_4$ is birationally equivalent to

$$\mathcal{E}_5 : 4r_1 x^2 + y^2 = 1 + 4r_2 x^2 y^2$$

on which lies $(x_5, y_5) = (x_4/y_4, (x_4 - 1)/(x_4 + 1))$ the point corresponding to P.

The scalar multiplication kP on $\mathcal{E}_5$ becomes $(u_5, v_5) = (k \times 2^{-1} \bmod n)(x_5, y_5)$. The twisted Edwards addition law is complete if r_1 is a square in $\mathbb{F}_{p^3}$ but not r_2. It means that among the three roots of $x^3 + ax + b$ at least one has to be a square and at least one a non-square. This property is always fulfilled except for NIST P-224.

The corresponding results on $\mathcal{E}_4$, $\mathcal{E}_3$ and $\mathcal{E}_2$ are respectively recovered as:

$$\begin{aligned}
(u_4, v_4) &= \left(\frac{1 + v_5}{1 - v_5}, \frac{1 + v_5}{u_5(1 - v_5)}\right) \\
(u_3, v_3) &= (u_4(r_1 - r_2), v_4(r_1 - r_2)) \\
(u_2, v_2) &= \left(\frac{{v_3}^2}{4{u_3}^2}, \frac{v_3((r_1 - r_2)^2 - {u_3}^2)}{8{u_3}^2}\right)
\end{aligned}$$

Finally $(u_2 + r_0, v_2) = kP$.

This method allows to perform scalar multiplications on the 12 NIST and Brainpool curves over $\mathbb{F}_p$ using the twisted Edwards addition law. However, it implies computations in $\mathbb{F}_{p^3}$, whose cost still overtakes the gain of using twisted Edwards formulas.

1.4.3 General Case of Elliptic Curves over $\mathbb{F}_q$

Let $\mathcal{E}$ be an elliptic curve over an odd characteristic field $\mathbb{F}_q$. The following theorem generalizes the preceding observations on standard curves.

Theorem 6. *If $\mathcal{E}(\mathbb{F}_q)$ has prime cardinality $n > 2$, there is an Edwards curve isogenous to $\mathcal{E}$ over $\mathbb{F}_{q^3}$. The field $\mathbb{F}_{q^3}$ is the extension of minimal degree with this property.*

Proof. Let t be the trace of $\mathcal{E}$ such that $n = q + 1 - t$. Now consider $\tau \in \mathbb{C}$ such that $\tau + \bar{\tau} = t$ and $\tau\bar{\tau} = q$. From Hasse-Weil theorem, we have:

$$\begin{aligned}\#\mathcal{E}(\mathbb{F}_{q^2}) &= q^2 + 1 - (\tau^2 + \bar{\tau}^2) \\ &= q^2 + 1 - (t^2 - 2q) \\ &\equiv -t^2 \pmod 4\end{aligned}$$

since q is odd, and $(q+1)^2 \equiv 0 \pmod 4$. Furthermore, since n and q are odd, then t is odd too, thus $\#\mathcal{E}(\mathbb{F}_{q^2}) \equiv 3 \pmod 4$. Therefore, no elliptic curve isogenous to $\mathcal{E}$ over $\mathbb{F}_{q^2}$ can have a subgroup of order 4.

Now, consider the cardinality of $\mathcal{E}$ on $\mathbb{F}_{q^3}$:

$$\begin{aligned}\#\mathcal{E}(\mathbb{F}_{q^3}) &= q^3 + 1 - (\tau^3 + \bar{\tau}^3) \\ &= q^3 + 1 - (\tau + \bar{\tau})\left((\tau + \bar{\tau})^2 - 3\tau\bar{\tau}\right) \\ &= q^3 + 1 - t(t^2 - 3q) \\ &\equiv (q+1)(1-t) \pmod 4 \\ &\equiv 0 \pmod 4\end{aligned}$$

since for any odd integer x, we have $x^2 \equiv 1 \pmod 4$. It follows that there is an elliptic curve isogenous to $\mathcal{E}$ over $\mathbb{F}_{q^3}$ having a subgroup of order 4. Thus, by Theorem 3, there is an Edwards curve isogenous to $\mathcal{E}$ over $\mathbb{F}_{q^3}$. □

Theorem 7. *If $\mathcal{E}$ has a single 2 torsion rational point on $\mathbb{F}_q$, then $\mathcal{E}$ has an isogenous Edwards curve over $\mathbb{F}_{q^2}$.*

Proof. As previously we have $\#\mathcal{E}(\mathbb{F}_{q^2}) \equiv -t^2 \pmod 4$. If n is even, then t is even, and consequently $\#\mathcal{E}(\mathbb{F}_{q^2}) \equiv 0 \pmod 4$.

In other words, there is an elliptic curve isogenous to $\mathcal{E}$ over $\mathbb{F}_{q^2}$ having a subgroup of order 4. Thus, by Theorem 3, there is an Edwards curve isogenous to $\mathcal{E}$ over $\mathbb{F}_{p^2}$. □

Chapter 2

Physical Cryptanalysis and Countermeasures on Embedded Devices

The side-channel attacks where first put in the spotlight in 1996 when Kocher showed that state-of-the-art public-key cryptosystem implementations where prone to *timing attacks* [Koc96]. These attacks are possible when execution timings of operations involving secret bits depend on their value, thereby leaking information. Two years later, Kocher, Jaffe, and Jun presented the *simple* and *differential* side-channel analysis [KJJ98; KJJ99], which can exploit various kinds of leakages such as the power consumption or the electromagnetical radiation. These attacks require a stronger physical access to the targeted device than timing attacks but can be devastating.

Since most of the literature dealing with side-channel analysis on public-key algorithms target RSA/DSA/Diffie-Hellman modular exponentiation, we recall in Section 2.1 how this operation is implemented on embedded devices.

Although timing attacks are nowadays circumvented using constant time implementations, simple and differential side-channel analysis have been refined year after year to bypass the first countermeasures in an exciting security race between implementors and attackers. In the following, Section 2.2 focuses on simple side-channel analysis, and Section 2.3 puts emphasis on differential techniques.

On the other hand, *fault attacks* belong to the family of active attacks as they consist in physically tampering with the processing of an operation. They were introduced in 1997 by Boneh, DeMillo, and Lipton [BDL97; BDL01]; many other works have followed since then. Some refinements even include a combination of side-channel analysis and fault attacks as described in Section 2.4.

All these attacks can be performed using various access and control levels to the targeted devices. Accordingly, attacks are said to be non-invasive, semi-invasive, or invasive. Non-invasive attacks are limited to monitoring the external behavior of a device and use its regular inputs and outputs. On the contrary, invasive techniques provide physical access to the different components of a chip and allow powerful reverse engineering of its design, but require high skill and expensive equipment. Since these

experimentation matters are not the subject of our study, we refer the interested reader to the numerous works published on the matter [KK99; Nohl; Tar10].

Exponentiation algorithms designed to resist the simple side-channel analysis generally perform a regular sequence of multiplications and squarings, or use multiplications only. We found out that using squarings only in an exponentiation scheme brings a new side-channel analysis countermeasure. The so-called *square always* exponentiation is detailed in Section 2.5.

Considering the multi-precision multiplication implementation, we propose a new side-channel attack in Section 2.6. We call it *horizontal* analysis since it may succeed using a single side-channel trace whereas most previous attacks require acquiring several traces. This work brings new recommendations for implementors to design secure exponentiation or scalar multiplication implementations.

Since our horizontal analysis operates at the modular multiplication level, we also design a new countermeasure consisting in blinding and shuffling internal operands of the multi-precision multiplication. Section 2.7 details how this may be implemented in classical algorithms.

Eventually, we show in Section 2.8 that side-channel collision analysis can threaten implementations considered secure until now. This work is presented on two different protected implementations of the AES block cipher.

2.1 Background on Exponentiation

Although ECC is gaining ground over RSA due to the shorter key lengths, the latter is still one of the most used public-key cryptosystems in security devices. DSA [NIST06] and Diffie-Hellman key agreement protocol [DH76] are also present in many applications, and, as RSA, require the computation of modular exponentiations with a secret exponent. Thus the problem of protecting the exponentiation against side-channel analysis in an efficient manner is worth of interest.

Since many of the side-channel analysis and countermeasures presented in the rest of this study target RSA/DSA/Diffie-Hellman exponentiations, we recall in Section 2.1.1 some classical exponentiation algorithms. Considering that the underlying implementation of the modular multiplication is also a key point to assess the security of exponentiation methods towards some side-channel analysis, we recall some basics about modular multiplication in Section 2.1.2.

2.1.1 Exponentiation Algorithms for Embedded Devices

We first recall in this section the well-known *square-and-multiply* algorithms — multiplicative equivalents of the double-and-add used for the elliptic curve scalar multiplication. Then we present some classical algorithms protected against the original SSCA. Many exponentiation algorithms have been proposed in the literature, among the numer-

ous references an interested reader can refer for instance to the *Handbook of Applied Cryptography* [MOV97] or the work by Bernstein [Ber02] on Pippenger's algorithm for details.

We emphasize that exponentiation algorithms and countermeasures are very similar to scalar multiplication. Regarding the state-of-the-art presented in Section 1.2, the most significant difference between these two contexts relates to signed representations which are not used in the modular exponentiation context due to the cost of the inversion. We briefly present the simplest exponentiation algorithms and refer the reader to Section 1.2 for algorithmic optimizations such as the various windowing methods.

2.1.1.1 Square-and-Multiply Algorithms

Algorithms 2.1 and 2.2 are two variants of the square-and-multiply method:

Algorithm 2.1 Left-to-right square-and-multiply exponentiation

Input: $m, n \in \mathbb{N}$, $m < n$, $d = (d_{k-1} d_{k-2} \dots d_0)_2$
Output: $m^d \bmod n$
1: $a \leftarrow 1$
2: **for** $i = k - 1$ **to** 0 **do**
3: $\quad a \leftarrow a^2 \bmod n$
4: $\quad$ **if** $d_i = 1$ **then**
5: $\quad\quad a \leftarrow a \times m \bmod n$
6: **return** a

Algorithm 2.2 Right-to-left square-and-multiply exponentiation

Input: $m, n \in \mathbb{N}$, $m < n$, $d = (d_{k-1} d_{k-2} \dots d_0)_2$
Output: $m^d \bmod n$
1: $a \leftarrow 1$
2: $b \leftarrow m$
3: **for** $i = 0$ **to** $k - 1$ **do**
4: $\quad$ **if** $d_i = 1$ **then**
5: $\quad\quad a \leftarrow a \times b \bmod n$
6: $\quad b \leftarrow b^2 \bmod n$
7: **return** a

These algorithms require on average $1S + 0.5M$ per bit of exponent to perform an exponentiation.

As double-and-add, these algorithms are no longer used in embedded devices for security applications since the emergence of side-channel analysis. Countermeasures consist of use regular algorithms or apply the atomicity principle, as detailed hereafter.

2.1.1.2 Regular Exponentiation Algorithms

Regular exponentiation algorithms include the well-known *square-and-multiply-always* and Montgomery ladder [Mon87; JY03]. Square-and-multiply-always uses dummy operations similarly to double-and-add-always. A multiplicative variant of Montgomery ladder is presented below in Algorithm 2.3. It is generally preferred over square-and-multiply-always since it does not involve dummy multiplications which makes it naturally immune to safe-error analysis, cf. Section 2.4.2.

Algorithm 2.3 Montgomery ladder exponentiation

Input: $m, n \in \mathbb{N}$, $m < n$, $d = (d_{k-1} d_{k-2} \dots d_0)_2$
Output: $m^d \bmod n$
Uses: R_0 and R_1
1: $R_0 \leftarrow 1$
2: $R_1 \leftarrow m$
3: **for** $i = k - 1$ **to** 0 **do**
4: $\quad R_{1-d_i} \leftarrow R_0 \times R_1 \bmod n$
5: $\quad R_{d_i} \leftarrow {R_{d_i}}^2 \bmod n$
6: **return** R_0

Such regular algorithms perform one squaring and one multiplication at every iteration and thus require $1M + 1S$ per exponent bit.

2.1.1.3 Exponentiation Using the Atomicity Principle

This method, presented by Chevallier-Mames et al. [CMCJ04], protects square-and-multiply algorithms against SSCA. It yields a *multiply-always* algorithm, where all squarings are performed as classical multiplications. We present left-to-right multiply-always in Algorithm 2.4.

Algorithm 2.4 Left-to-right multiply-always exponentiation

Input: $m, n \in \mathbb{N}$, $m < n$, $d = (d_{k-1} d_{k-2} \dots d_0)_2$
Output: $m^d \bmod n$
Uses: R_0 and R_1
1: $R_0 \leftarrow 1$
2: $R_1 \leftarrow m$
3: $i \leftarrow k - 1$
4: $t \leftarrow 0$
5: **while** $i \geq 0$ **do**
6: $\quad R_0 \leftarrow R_0 \times R_t \bmod n$
7: $\quad t \leftarrow t \oplus d_i$
8: $\quad i \leftarrow i - 1 + t$
9: **return** R_0

The multiply-always algorithm is faster than regular ones: it performs an exponentiation using on average $1.5M$ per exponent bit.

2.1.2 Modular Multiplication Implementation

All of the exponentiation algorithms presented above perform a sequence of modular long-integer multiplications. Heavy efficiency constraints thus lie on this operation, especially in the context of embedded devices. Several methods have been devised to avoid costly modular reductions (i.e. integer divisions): Montgomery, Barrett, Quisquater, etc. A detailed review of these techniques is provided by Dhem in his PhD thesis [Dhe98]. A modular multiplication $x \times y \bmod n$ generally consists in two steps: first the regular multiplication $x \times y$, and second the reduction modulo n according to one of the aforementioned methods. These two steps may also be interleaved for memory savings.

2.1.2.1 Independent Multiplication and Reduction

In embedded devices, the long-integer multiplication $x \times y$ generally involves a multi-precision multiplication implemented using the schoolbook method. This method is described in Algorithm 2.5 with one (or more) smaller hardware multiplier(s) operating on t-bit words — i.e. giving a $2t$-bit result. The decomposition of an l-bit integer x in t-bit words is denoted by $x = (x_{l_b-1} x_{l_b-2} \dots x_0)_b$ with $b = 2^t$ and $l_b = \lfloor \log_b(x) \rfloor + 1$. Other long-integer multiplications techniques may also be used such as Comba [Com90] and Karatsuba [KO62] algorithms.

Algorithm 2.5 Schoolbook long-integer multiplication

Input: $x = (x_{l_b-1} x_{l_b-2} \dots x_0)_b$, $y = (y_{l_b-1} y_{l_b-2} \dots y_0)_b$
Output: $x \times y$
Uses: $w = (w_{2l_b-1} w_{2l_b-2} \dots w_0)$

1: $w \leftarrow (00 \dots 0)$
2: **for** $i = 0$ **to** $l_b - 1$ **do**
3: $\quad c \leftarrow 0$
4: $\quad$ **for** $j = 0$ **to** $l_b - 1$ **do** $\quad \triangleright\ w \leftarrow w + b^i x_i \times y$
5: $\quad\quad (uv)_b \leftarrow w_{i+j} + x_i \times y_j + c$
6: $\quad\quad w_{i+j} \leftarrow v$
7: $\quad\quad c \leftarrow u$
8: $\quad w_{i+l_b} \leftarrow c$
9: **return** w

The $2l$-bit result of the long-integer multiplication is then reduced modulo n, depending on the chosen method. For instance, Montgomery reduction [Mon85] replaces divisions by shifts. Considering an odd modulus n, take $R = 2^i$ such that $R > n$ (in general $i = l$). Given $0 \leq w < 2^{2i}$, Montgomery reduction computes $0 \leq r < n$ such that $r \equiv wR^{-1} \pmod{n}$. Now, for an operand $0 \leq u < n$, let the residue $uR \bmod n$ be called the Montgomery representation of u with respect to R and n. The computation of $uv \bmod n$ thus requires to perform the long-integer multiplication $x \times y$ with $x = uR \bmod n$ by $y = vR \bmod n$ which yields uvR^2. Montgomery reduction is applied to compute $uvR \bmod n$, the Montgomery representation of uv with respect to R and n. Montgomery reduction is outlined in Algorithm 2.6 and a detailed implementation using single-precision multiplications only is presented in Algorithm 2.7.

Algorithm 2.6 Montgomery modular reduction (high level)

Input: $w < 2^{2l}$, $n < 2^l$, $R = 2^l$, $n' = -n^{-1} \bmod R$ with $\gcd(R, n) = 1$ and $w < nR$
Output: $wR^{-1} \bmod n$

1: $s \leftarrow w \bmod R$ ▷ ignore bits from the l-th one upwards
2: $q \leftarrow n' \times s \bmod R$
3: $r \leftarrow (w + q \times n)/R$ ▷ the division is an l-bit shift
4: **if** $r \geq n$ **then**
5: $\quad r \leftarrow r - n$
6: **return** r

Algorithm 2.7 Montgomery modular reduction implementation

Input: $w = (w_{2l_b-1} w_{2l_b-2} \ldots w_0)_b$, $n = (n_{l_b-1} n_{l_b-2} \ldots n_0)_b$, with $\gcd(2^l, n) = 1$ and $w < n2^l$
Output: $w2^{-l} \bmod n$

1: $n'_0 \leftarrow -{n_0}^{-1} \bmod b$
`Main loop`
2: **for** $i = 0$ **to** $l_b - 1$ **do**
3: $\quad q \leftarrow w_0 \times n'_0 \bmod b$ ▷ $q \leftarrow -w_0/n_0$
4: $\quad c \leftarrow 0$
5: $\quad$ **for** $j = 0$ **to** $l_b - 1$ **do** ▷ $w \leftarrow w + q \times n$
6: $\quad\quad (uv)_b \leftarrow w_j + q \times n_j + c$
7: $\quad\quad w_j \leftarrow v$
8: $\quad\quad c \leftarrow u$
9: $\quad$ **for** $j = l_b$ **to** $2l_b - i - 1$ **do** ▷ carry propagation
10: $\quad\quad (uv)_b \leftarrow w_j + c$
11: $\quad\quad w_j \leftarrow v$
12: $\quad\quad c \leftarrow u$
13: $\quad w_{2l_b-i} \leftarrow c$
14: $\quad (w_{2l_b} w_{2l_b-1} \ldots w_0)_b \leftarrow (0\, w_{2l_b} w_{2l_b-1} \ldots w_1)_b$ ▷ $(w)_b \leftarrow (w)_b \gg 1$
`Final subtraction`
15: **if** $w > n$ **then**
16: $\quad w \leftarrow w - n$
17: **return** w

All variables are manipulated using the Montgomery representation until all modular computations are performed. Eventually, an extra reduction step removes the R factor and outputs the final result. Such a modular multiplication method is efficient when many successive modular operations have to be computed, as in modular exponentiation algorithms. An interested reader can refer to the article by Koç and Acar comparing different implementations of the Montgomery multiplication [KA96].

Remark 8. Side-channel analysis can target the final subtraction step of Montgomery reduction [Sch00; WT01]. For example, Walter and Thompson report that it occurs with different probability depending on whether the multiplication being performed is a square or not. To address this issue, Walter shows that final subtractions can be avoided in implementations if enough multiplications are iterated [Wal99; Wal02]. We refer the reader to these works for details.

2.1.2.2 Interleaved Multiplication and Reduction

Performing long-integer multiplication and modular reduction as two independent steps has a drawback: the intermediate result $x \times y$ has to be stored using $2l$ bits. Therefore, many embedded implementations make use of algorithms interleaving multiplication and reduction steps. Indeed, interleaved algorithms perform a reduction of each $t + l$ bits intermediate result of the inner loop of the multiplication algorithm — i.e. every $t \times l$ multiplication $x_i \times y$ is reduced to l bits before the following multiplication $x_{i+1} \times y$. This method thus never requires more than $t + l$ bits for the storage of intermediate multiplication results. Algorithm 2.8 is the interleaved Montgomery multiplication of x and y modulo n using a t-bit multiplier.

Algorithm 2.8 Interleaved Montgomery modular multiplication and reduction

Input: $x = (x_{l_b-1}x_{l_b-2}\dots x_0)_b$, $y = (y_{l_b-1}y_{l_b-2}\dots y_0)_b$, $n = (n_{l_b-1}n_{l_b-2}\dots n_0)_b$
Output: $xy2^{-l} \bmod n$
Uses: $w = (w_{l_b}w_{l_b-1}\dots w_0)$

1: $n'_0 \leftarrow -{n_0}^{-1} \bmod b$
2: $w \leftarrow (00\dots0)$
`Main loop`
3: **for** $i = 0$ **to** $l_b - 1$ **do**
4: $\quad c \leftarrow 0$
5: $\quad$ **for** $j = 0$ **to** $l_b - 1$ **do** $\quad \triangleright\ w \leftarrow w + x_i \times y$
6: $\quad\quad (uv)_b \leftarrow w_j + x_i \times y_j + c$
7: $\quad\quad w_j \leftarrow v$
8: $\quad\quad c \leftarrow u$
9: $\quad w_{l_b} \leftarrow c$
10: $\quad q \leftarrow w_0 \times n'_0 \bmod b$ $\quad \triangleright\ q \leftarrow -w_0/n_0$
11: $\quad c \leftarrow 0$
12: $\quad$ **for** $j = 0$ **to** $l_b - 1$ **do** $\quad \triangleright\ w \leftarrow w + q \times n$
13: $\quad\quad (uv)_b \leftarrow w_j + q \times n_j + c$
14: $\quad\quad w_j \leftarrow v$
15: $\quad\quad c \leftarrow u$
16: $\quad w_{l_b} \leftarrow w_{l_b} + c$
17: $\quad (w_{l_b}w_{l_b-1}\dots w_0)_b \leftarrow (0\, w_{l_b}w_{l_b-1}\dots w_1)_b$ $\quad \triangleright\ (w)_b \leftarrow (w)_b \gg 1$
`Final subtraction`
18: **if** $w > n$ **then**
19: $\quad w \leftarrow w - n$
20: **return** w

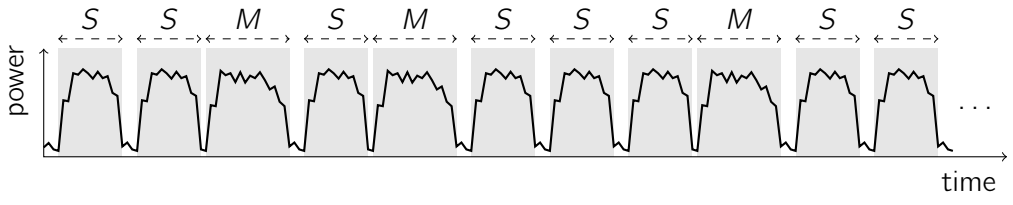


Figure 2.1: Square-and-multiply side-channel leakage

2.2 Simple Side-Channel Analysis

Simple and *differential* side-channel analysis rely on the following physical property: a microprocessor is physically made of thousands of logical gates switching differently depending on the executed operations and on the manipulated data. Therefore power consumption and electromagnetic radiation reflect and may leak information on both instructions and data. By monitoring a device performing cryptographic operations, an observer may infer information on the implementation of the program executed and on the secret data involved.

Simple Side-Channel Analysis (SSCA) include *Simple Power Analysis (SPA)*, resp. *Simple ElectroMagnetic Analysis (SEMA)*, if the measured leakage is the power consumption, resp. electromagnetic radiation. However, SPA is often used as a synonym for SSCA by abuse of language.

In the following, Section 2.2.1 presents the original SPA, Section 2.2.2 recalls further refinements when the input message can be chosen by an adversary, and Section 2.2.3 briefly describes some attacks requiring a preliminary characterization phase.

2.2.1 Original Simple Side-Channel Analysis

The simple side-channel analysis, introduced by Kocher et al. [KJJ99], consists in monitoring and acquiring a trace of some side-channel activity of a device performing a cryptographic operation or any computation involving sensitive data. Then, the attacker uses this measurement to observe the performed operations. If conditional branching in the code depends on secret data and can be detected, the secret value can be recovered. Such a leakage can possibly lead to the recovery of the whole key with a single trace and a single execution.

This attack assumes that the attacker has full knowledge of the performed algorithms. Note that even if this were not the case, a side-channel leakage can be used first to perform reverse engineering and guess how an operation is implemented.

SPA was first presented on an RSA square-and-multiply exponentiation. This algorithm is particularly prone to SSCA when implemented on a device where a modular multiplication and a modular squaring have different patterns that can be observed on some side-channel leakage. Devices provided with an arithmetic coprocessor using a fast dedicated squaring fall into this category since the coprocessor activity is generally easy to spot on power or electromagnetic leakage traces. In this case, the duration (the size on the trace) of each operation indicates if it is a squaring (short pattern) or a multiplication (longer pattern) as depicted on Figure 2.1.

Recovering the secret exponent bits from the sequence of operations is straightforward: considering the left-to-right implementation (cf. Algorithm 2.1), a squaring followed by a multiplication corresponds to a 1 bit, and a squaring followed by another squaring corresponds to a 0 bit. For instance, the sequence of operations of Figure 2.1 corresponds to the sequence of exponent bits 0110010. Two countermeasures preventing SSCA on the exponentiation are presented in Section 2.1.1: regular algorithms and atomic algorithms.

Regular algorithms perform a sequence of operations that do not depend on the exponent. To achieve this property, the maximal number of operations that can be required per exponent bit — one squaring and one multiplication in our case — is always performed. Square-and-multiply-always and Montgomery ladder (cf. Algorithm 2.3) belong to this category. Figure 2.2 depicts the typical side-channel leakage of such algorithms.

On the other hand, atomic algorithms also perform a regular sequence of operations, but apply the opposite strategy: they always perform the minimal number of operations that may be required within a loop iteration — one multiplication here. Consequently several loop iterations may be necessary to the processing of an exponent bit. The most used atomic method is multiply-always (cf. Algorithm 2.4) which perform one loop iterations for a 0 exponent bit and two for a 1. Figure 2.3 depicts the typical side-channel leakage of this algorithm. We present in Section 2.5 another atomic exponentiation algorithm using squarings only.

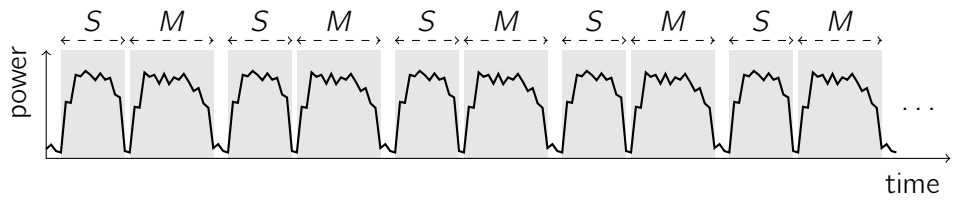


Figure 2.2: Square-and-multiply-always or Montgomery ladder side-channel leakage

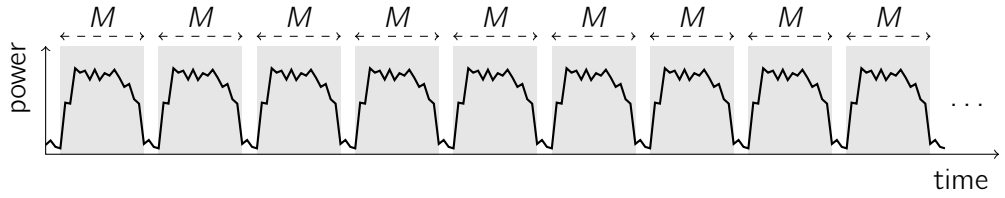


Figure 2.3: Atomic multiply-always side-channel leakage

The same applies to double-and-add scalar multiplications methods: algorithms 1.1 and 1.2 are subject to SSCA if an addition can be distinguished from a doubling using some side-channel leakage. For instance, Figure 2.4, resp. 2.5, presents electromagnetic measurements obtained for the execution of a point doubling, resp. a point addition, on a SmartMX chip provided with an arithmetic coprocessor. Large amplitude blocks show the coprocessor activity, which exposes the sequence of arithmetic operations. The two patterns are easily distinguishable (identifying the patterns is even easier when looking at the lower part of the traces). The sequence of performed operations on Figure 2.6 is "doubling, addition, doubling, doubling, addition", which yields three scalar bits 1, 0, and 1 if Algorithm 1.1 is used.

Countermeasures preventing the SSCA on elliptic curve scalar multiplication — which are similar to those presented for exponentiation — are detailed in Section 1.2.2.

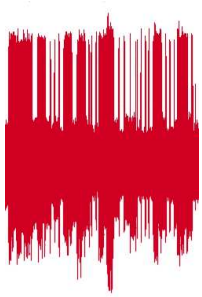


Figure 2.4: Doubling electromagnetic leakage trace

Figure 2.5: Addition electromagnetic leakage trace

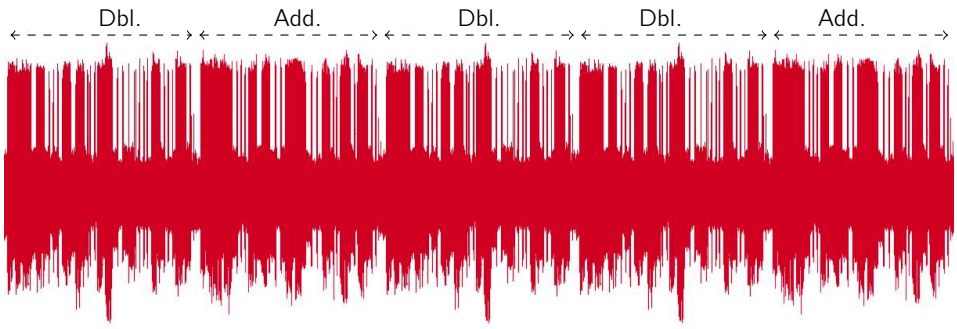


Figure 2.6: Scalar multiplication electromagnetic leakage trace extract

2.2.2 Chosen Message Simple Side-Channel Analysis

Intense attention has been paid to the original SSCA since its introduction, as evidenced by the numerous countermeasures published in the literature, cf. sections 1.2.2 and 2.5. However, other attacks than the simple analysis of conditional branchings may threaten protected implementations, as evidenced by Mayer-Sommer [MS00]. This is even more manifest when the attacker is able to control some input of the cryptographic algorithms. The following attacks are considered to belong to SSCA category since they use a single — or a very few number of — side-channel traces to recover information on the secret key.

2.2.2.1 Collision Analysis

In public-key cryptography, basic operations are generally arithmetic operations involving long operands. For instance, both in ECC and RSA, the base operation is the modular multiplication between operands of hundreds or thousands of bits. Since these operations are performed in most embedded devices using t-bit multipliers, $t \leq 64$, many clock cycles are needed to perform each of them.

Moreover, on most devices — for instance those using the CMOS technology — power consumption depends on manipulated data, which makes it possible to detect when an operation is performed twice with same operands, what we call a collision. We recall hereafter some attacks based on this observation.

Doubling Attack Fouque and Valette present an attack on regular left-to-right ECC scalar multiplication and RSA exponentiation algorithms [FV03]. In the most favorable case, only two power traces with chosen inputs yield the whole secret key.

This attack relies on the assumption that an adversary can detect collisions of side-channel trace segments if a same operation is performed twice by the device. Note that these segments can come from a single or more traces, and that the attacker does not need to identify which values are manipulated. For instance, an adversary is supposed to distinguish a collision between the processing of $2 \times P$ and $2 \times Q$ if $P = Q$, but not to guess any coordinate of P. The practical validity of this assumption is demonstrated by experiments by Fouque and Valette.

Suppose that a scalar multiplication is performed using the left-to-right double-and-add-always with an input point P and a scalar $k = 151 = (10010111)_2$. We show in Table 2.1 the sequence of operations performed by the algorithm and the sequence of operations performed with input point $2P$. One can observe that, when a 0 scalar bit is processed at step i, the doubling of the scalar multiplication $k(2P)$ and the doubling of kP at the next step $(i-1)$ are the same operation. Thus, under the previous assumption, an attacker can distinguish 0 bits from 1 bits using this leakage if he is able to control the input of the scalar multiplication, at least for the second execution.

A similar attack is possible on Montgomery ladder (cf. Algorithm 1.10) and other left-to-right regular algorithms. On Montgomery ladder for instance, a collision between a doubling at step i in the processing of $k(2P)$ and a doubling at step $i-1$ in the

processing of kP indicates that $k_i = k_{i-1}$, as shown in Table 2.2. When no collision occurs, an attacker infers that $k_i \neq k_{i-1}$. In the end, this yields a relation between all scalar bits, thus only two scalar values have to be checked by the attacker.

Table 2.1: Doubling attack on the left-to-right double-and-add-always algorithm with scalar $k = 151 = (10010111)_2$

i	k_i	Computation of kP	Computation of $k(2P)$
7	1	$2 \times \mathcal{O}$ $\mathcal{O} + P$	$2 \times \mathcal{O}$ $\mathcal{O} + 2P$
6	**0**	$2 \times P$ $2P + P$	$\mathbf{2 \times 2P}$ $4P + 2P$
5	**0**	$\mathbf{2 \times 2P}$ $4P + P$	$\mathbf{2 \times 4P}$ $8P + 2P$
4	1	$\mathbf{2 \times 4P}$ $8P + P$	$2 \times 8P$ $16P + 2P$
3	**0**	$2 \times 9P$ $18P + P$	$\mathbf{2 \times 18P}$ $36P + 2P$
2	1	$\mathbf{2 \times 18P}$ $36P + P$	$2 \times 36P$ $72P + 2P$
1	1	$2 \times 37P$ $74P + P$	$2 \times 74P$ $148P + 2P$
0	1	$2 \times 75P$ $150P + P$ $= 151P$	$2 \times 150P$ $300P + 2P$ $= 151(2P)$

Defending against the doubling attack requires either the use of right-to-left algorithms which do not present this leakage, or scalar or point randomization countermeasures, cf. Section 2.3.2. However, the efficiency of such countermeasures heavily depends on their practical implementation [FV03].

Indeed, scalar randomization prevents the previous attack to succeed with only 2 queries, but if the adversary can perform a large number of execution for both inputs P and $2P$, he can still look for a collision between two traces. If the scalar randomization process uses 2α random bits, a collision should occur after about 2^{α} many queries due to the birthday paradox. The practicality of the attack against this countermeasure thus depends on the number of available queries and the number of random bits used in the randomization process.

Table 2.2: Doubling attack on the Montgomery ladder algorithm with scalar $k = 151 = (10010111)_2$

i	k_i	Computation of kP	Computation of $k(2P)$
6	**0**	$P + 2P$ $2 \times P$	$2P + 4P$ $\mathbf{2} \times \mathbf{2P}$
5	**0**	$2P + 3P$ $\mathbf{2} \times \mathbf{2P}$	$4P + 6P$ $2 \times 4P$
4	1	$4P + 5P$ $2 \times 5P$	$8P + 10P$ $2 \times 10P$
3	0	$9P + 10P$ $2 \times 9P$	$18P + 20P$ $2 \times 18P$
2	**1**	$18P + 19P$ $2 \times 19P$	$36P + 38P$ $\mathbf{2} \times \mathbf{38P}$
1	**1**	$37P + 38P$ $\mathbf{2} \times \mathbf{38P}$	$74P + 76P$ $\mathbf{2} \times \mathbf{76P}$
0	**1**	$75P + 76P$ $\mathbf{2} \times \mathbf{76P}$ $\leftarrow 151P$	$150P + 152P$ $2 \times 152P$ $\leftarrow 151(2P)$

Regarding the input point randomization countermeasure, this method is expensive unless the input and output blinding points are generated (or updated) in an efficient manner. The scheme proposed to this purpose by Coron [Cor99] (see Section 2.3.2) is vulnerable to the doubling attack as it updates the input blinding point R by $2R$ one time out of two — in other words, the doubling attack applies exactly as presented above half the time.

The doubling attack is presented above on the point scalar multiplication, but also applies to similar exponentiation algorithms used, for instance, in RSA and ElGamal cryptosystems.

Exponentiation Internal Collisions A refinement of the doubling attack on RSA is proposed by Yen, Lien, Moon, and Ha [YLMH05]. The authors observe that if the input message of a left-to-right exponentiation is set to $m = n - 1$, with $n = p \times q$ being the RSA modulus, then the algorithm handles only two different values depending on exponent bits. Indeed, as $(n-1)^2 = 1 \bmod n$, each squaring of a left-to-right square-and-multiply algorithm outputs a 1, and each multiplication outputs $n - 1$. Thus any squaring after the first one is 1^2 if the previous exponent bit was 0 or $(n-1)^2$ otherwise.

Assuming that an adversary is able to distinguish the waveforms corresponding to the computation of $1^2 \bmod n$ from those corresponding to the computation of $(n-1)^2 \bmod n$, he can recover the scalar exponent bits using a single trace. Moreover, the authors show that the BRIP countermeasure presented in Section 2.3.2 is of no effect against this analysis.

An obvious countermeasure to this attack is to check that input messages are different from $n-1$. In this case, Yen et al. show that the analysis can yet be performed using two traces: one obtained with any input message m and the second obtained with the input message $m(n-1) \bmod n = n - m$. This refinement is then very similar to the doubling attack.

A similar attack applies to scalar multiplications performed on an elliptic curve $\mathcal{E}$ over $\mathbb{F}_q$ if $\mathcal{E}(\mathbb{F}_q)$ has elements of order 2, for instance on an Edwards curve. In this case, implementations must verify that input points belong to the required subgroup of $\mathcal{E}(\mathbb{F}_q)$. Note that this issue is not present when using current standard $\mathbb{F}_p$ curves which all have a prime order.

Generating Arbitrary Collisions Homma, Miyamoto, Aoki, Satoh, and Shamir generalize the two previous attacks by showing that squaring collisions depending on an unknown exponent bit d_i can be generated using well chosen input messages [HMA+08]. Their attack consists in finding two messages m_1 and m_2 such that ${m_1}^{\alpha} = {m_2}^{\beta} \bmod n$, for arbitrary pairs (α, β) depending on the known exponent bits $d_{l-1}d_{l-2}\ldots d_{i+1}$, resp. $d_{i-1}d_{i-2}\ldots d_0$, considering a left-to-right, resp. right-to-left, exponentiation algorithm. A similar attack can be devised on the scalar multiplication in an additive group.

This analysis can thus target any exponentiation or scalar multiplication algorithm, including m-ary, sliding window and ladder methods. Note however that it requires two traces with chosen inputs to be acquired per key bit whereas the doubling and Yen et al. attacks recovers the whole secret key with two traces in total.

2.2.2.2 Power Signature Analysis

Since the consumption/radiation of embedded devices depends on the manipulated data, it is sometimes possible to distinguish the computation or the manipulation of particular values on a side-channel trace. It is generally considered that power consumption of processors is correlated with the Hamming weight of manipulated variables or the Hamming distance of successive variables. This is due to the energy consumption required to flip the processors internal registers bits. Thus, values with particularly low or high Hamming weight can sometimes be directly identified on side-channel traces.

For instance, in the attack from Yen et al. presented above, operands computations 1^2 and $(n-1)^2$ may be directly identified on a trace due to the very low Hamming weight of the former. Indeed the handling of the value 1 by a multi-precision multiplication algorithm using a t-bit multiplier, $t \in \{8, 16, 32\}$ involves the manipulation of many null t-bit words. These manipulations have generally a lower consumption than others and may be identified on a power trace.

Courrege, Feix, and Roussellet [CFR10] explore this analysis and point out that the chosen message model is not always necessary. Indeed, if a single null t-bit word can be distinguished on a side-channel trace, the random message model is enough to mount the attack by triggering exponentiation execution until a message with a null word is manipulated. It is interesting to notice that the longer the exponent and the smaller the word size t, the higher is the attack success probability. Courrege et al. note that the message/modulus randomization technique — i.e. computing $(m + r_1 n)^d \bmod r_2 n$, with r_1, r_2 two random values such that $r_1 < r_2$ — is not efficient against their attack in the random message model, and may even provide the required message variability.

Goubin presents a differential chosen message attack [Gou02] against the coordinates randomization countermeasure, cf. Section 2.3.2. Though it is presented as a differential analysis, this attack uses an input point with one null coordinate to defeat the multiplicative blinding. As a consequence, this attack may be combined with power signature analysis to threaten an implementation using a left-to-right double-and-add scalar multiplication algorithm and projective coordinates randomization.

Remark 9. Note also that, as all attacks performed on a single trace, power signature analysis is not prevented by scalar or exponent randomization techniques (cf. Section 2.3.2). Indeed, a randomized private scalar or exponent, if recovered, can be used for signature and decryption as well as the original one.

2.2.3 Template Analysis

Template attacks, introduced by Chari, Rao, and Rohatgi [CRR03], form a particular class of attacks requiring that the targeted device (or a similar one) can be studied in a *white box* context prior to the attack.

This first phase allows a fine characterization of the chip and generally targets a particular operation or sequence of operations. In order to identify during the attack phase a value manipulated during the targeted computation, the attacker acquires one (or several) side-channel trace(s) per possible value. By doing this, the attacker con-

structs a *dictionary*, i.e. a set of template side-channel traces corresponding to possible manipulated values. Obviously, averaging as many traces as possible for each value can be used to reduce the influence of the device and experimental noise.

The attack phase then consists in acquiring a side-channel trace of the device when the secret data is manipulated by the targeted operation. The attacker uses his dictionary to find the template that matches best with the acquired trace. The value associated to the template reveals the secret manipulated with a probability depending on the device and experimental conditions.

Since this class of attacks only applies in a very specific context — having a full access and control on an experimental device identical to the attacked one —, our study does not focus on this threat. The interested reader can refer, for instance, to the following references [CRR03; ARRS05; APSQ06]. Nevertheless, we briefly recall hereafter a powerful template analysis presented in [MO09] by Medwed and Oswald on the ECDSA protocol, as we think that developers should keep it in mind when implementing this protocol.

A Template Attack on ECDSA Though most of the literature about template analysis deals with symmetric ciphers, Medwed and Oswald present [MO09] a withering attack on an implementation of the ECDSA [ANSI05] protocol protected against conventional SSCA by using the double-and-add-always scalar multiplication.

We recall the ECDSA signature computation in Algorithm 2.9 and the verification process in Algorithm 2.10. Recovering the ephemeral scalar k from the signature algorithm yields the private key d by computing $(sk - H(m))/r$. However, since a fresh random scalar k is generated for each signature (step 1), it seems that a differential side-channel analysis cannot be mounted against the scalar multiplication (step 2). It would thus be natural not to apply the randomization countermeasures described in Section 2.3.2.

Algorithm 2.9 ECDSA signature

Input: Public point $P \in \mathcal{E}(\mathbb{F}_q)$, $n_P = \mathrm{ord}_{\mathcal{E}}(P)$, private key d, hash function H, and message m
Output: Signature (r, s)
1: Pick at random k in $[1, n_P - 1]$
2: $P_1 \leftarrow kP$
3: $r \leftarrow x_1$ ▷ with $P_1 = (x_1, y_1)$
4: **if** $r \bmod n_P = 0$ **then**
5: **go to** step 1
6: $s \leftarrow k^{-1}(H(m) + dr) \bmod n_P$
7: **if** $s \bmod n_P = 0$ **then**
8: **go to** step 1
9: **return** (r, s)

Algorithm 2.10 ECDSA verification

Input: Public point $P \in \mathcal{E}(\mathbb{F}_q)$, $n_P = \text{ord}_{\mathcal{E}}(P)$, public key $Q = dP$, hash function H, message m, and signature (r, s)
Output: `valid` or `not valid`

1: **if** r or $s \notin [1, n_P - 1]$ **then**
2: **return** `not valid`
3: $s' \leftarrow s^{-1} \bmod n_P$
4: $P_1 \leftarrow (s'H(m))P + (s'r)Q$
5: **if** $P_1 = \mathcal{O}$ **then**
6: **return** `not valid`
7: **if** $x_{P_1} \not\equiv r \pmod{n_P}$ **then**
8: **return** `not valid`
9: **return** `valid`

Medwed and Oswald observe that:

(i) as demonstrated by Nguyen and Shparlinski [NS03], recovering completely the scalar k from a single execution is not necessary to retrieve the private key: a few bits of several ephemeral scalars and the associated ECDSA signatures are sufficient to determine the secret key via lattice attacks,

(ii) the base point involved in the scalar multiplication is fixed, thus the number of different intermediate points computed after a few bits of the scalar is small and a template attack can be used to identify them.

Therefore the attack proceeds as follows: to identify the first s bits of the random scalars, a dictionary of the traces associated to the 2^s first multiples of the base point P is build. Then traces are acquired from the targeted device and the corresponding signatures are stored. Matching the templates with the recordings enables to recover the first s bits of each ephemeral scalar, and finally an off-line lattice reduction extracts the private key d.

Furthermore, Medwed and Oswald notice that not all randomization countermeasures are efficient against this attack. For example, the scalar randomization only increases the size of the lattice, and point blinding may be targeted by a template analysis as well. The best countermeasures are the ones providing the randomization of the coordinates of the input point, assuming that large enough random numbers are used in this aim.

2.3 Differential Side-Channel Analysis

Introduced along with SSCA by Kocher et al. [KJJ98; KJJ99], Differential Side-Channel Analysis (DSCA) exploits the slight consumption leakages of a device depending on the data values it manipulates. Due to the loss of information of used side-channels, the experimental noise, hardware countermeasures, etc. practical attacks exploiting such a leakage require the acquisition of many execution traces — from hundreds to millions —, and possibly some sophisticated signal and statistical processing.

Similarly to simple analysis, DSCA are called *Differential Power Analysis (DPA)*, resp. *Differential ElectroMagnetic Analysis (DEMA)*, if the measured leakage is the power consumption, resp. the electromagnetic radiation. As well, DPA is sometimes used in the general meaning by abuse of language.

The original DPA has known several refinements among which the well-known Correlation Power Analysis (CPA) and Mutual Information Analysis (MIA) which are detailed in Section 2.3.1. Many countermeasures have been studied as exposed hereafter, it is generally considered that the more efficient of them rely on blinding techniques. We present some of those designed for the scalar multiplication and the exponentiation in Section 2.3.2.

2.3.1 Differential Side-Channel Analysis: From DPA to MIA

Differential side-channel analysis basically consists in acquiring many leakage traces $T_1, \ldots, T_N$ of executions in which a constant secret key is manipulated, but some other known input is variable — say, the message to sign or decrypt m_i, with $1 \leq i \leq N$. These traces are stored along with the known input value. Then, a guess $\mathcal{G}$ is made on some bits of the secret key, and a statistical processing between some temporary value x_i computed according to $\mathcal{G}$ and m_i, on one hand, and the traces T_i, on the other hand, is performed. Reproducing this processing for all possible guesses reveals the correct one, i.e. the value of the targeted key bits, if N is big enough with regard to the leakage model and experimental conditions. This process is eventually iterated on the remaining unknown key bits until all of them are recovered, or until the last ones can be found by exhaustive search.

Note that the previously said known input m_i is only necessary to compute the internal temporary value x_i manipulated by the algorithm, that depends on both m_i and the targeted secret key bits. Thus, in the context of an attack against a symmetric block cipher, resp. in the context of ECC/RSA, x_i is an internal variable of the first round, resp. an internal variable depending on the first scanned scalar/exponent bits.

By the way, the output of a symmetric algorithm can be used instead of its input. In this case, it is used to reverse the last operations performed by the algorithm until some key bits are involved in the computation of a variable x_i. This strategy is also possible with ECC by reversing the process of a scalar multiplication algorithm, but it does not apply to RSA since square roots modulo n cannot be extracted efficiently when the factorization of n is unknown.

As a consequence, first and last rounds of symmetric algorithms are more subject to differential analysis than others. This is different in the case of the scalar multiplication/exponentiation since only one scalar or exponent bit (or a few ones) is used at a time, which enable an adversary to target these key bits one after another.

2.3.1.1 Original Differential Power Analysis

The original differential power analysis presented by Kocher et al. [KJJ98; KJJ99] proceeds as follows: the attacker chooses a binary decision function D such that $D(x_i) \in \{0,1\}$ related to the leakage model used for the device. Usual choices for D include returning a bit of x_i, or returning 1 if the Hamming weight of (a group of bits of) x_i reaches a given threshold and 0 otherwise. Such decision functions assume that the device leakage follows a Hamming weight model. The Hamming distance model between successive values stored in registers is sometimes also used.

Then, for a given guess $\mathcal{G}$, the traces $T_1, \ldots, T_N$ are sorted in two sets S_0 and S_1 such that $S_b = \{T_i \mid D(x_i) = b\}$, for $b \in \{0,1\}$. Finally, the attacker computes the differential trace corresponding to $\mathcal{G}$ as $\Delta_{\mathcal{G}} = \langle S_0 \rangle - \langle S_1 \rangle$, where $\langle \cdot \rangle$ denotes the average function. A differential trace is computed for each possible $\mathcal{G}$. If the decision function D is well chosen and N is big enough, among all differential traces one presents some *differential spikes*, while the others show only noise. The former one corresponds to the correct guess.

The reason why this processing reveals the correct guess is the following. If the leakage model used for building the decision function is close to reality, $D(x_i)$ is correlated to the leakage if $\mathcal{G}$ is correct and uncorrelated otherwise. Hence, D outputs the correct bit with probability $P \approx 1/2$ if the guess $\mathcal{G}$ is incorrect. As a consequence, $\{S_0, S_1\}$ is a random partitioning of the set of traces, and $\Delta_{\mathcal{G}}$ approaches zero as N tends to infinity. Besides, if $\mathcal{G}$ is correct, x_i is actually manipulated by the device, and the partitioning of the traces corresponds to a real difference of consumption between the averages of the traces in S_0 and in S_1, at the moments when x_i is manipulated. Thus, the differential trace $\Delta_{\mathcal{G}}$ shows noise, except when x_i is manipulated which is revealed by the spikes.

Kocher et al. [KJJ99] present a successful DPA on a DES implementation using the following parameters: the targeted value x_i is an output bit of a given S-box in the last round, and the decision function D returns the bit of L_{15} which corresponds to x_i after the permutation P, cf. Figure 2.7. The value $D(x_i)$ can be computed as the X-or between x_i and a known ciphertext bit. Since x_i depends on the input of the considered S-box, x_i depends on 6 bits of the last round subkey K_{15} and on the right input R_{15}. From the ciphertext, an adversary deduces R_{15} and makes a guess $\mathcal{G}$ on the six involved subkey bits, i.e. $0 \leq \mathcal{G} \leq 63$. The correct guess reveals 6 bits of K_{15}. Iterating this process on the 8 S-boxes yields the whole value of K_{15}, i.e. 48 bits of the actual DES key. The remaining 8 key bits can be recovered using exhaustive search — or possibly by analyzing the second last round. The number of used traces is $N = 10^3$ or $N = 10^4$ but highly depends on the targeted device and experimental conditions. The reader interested to learn more about DSCA may refer to the following references [MDS99a; CCD00; GMO01].

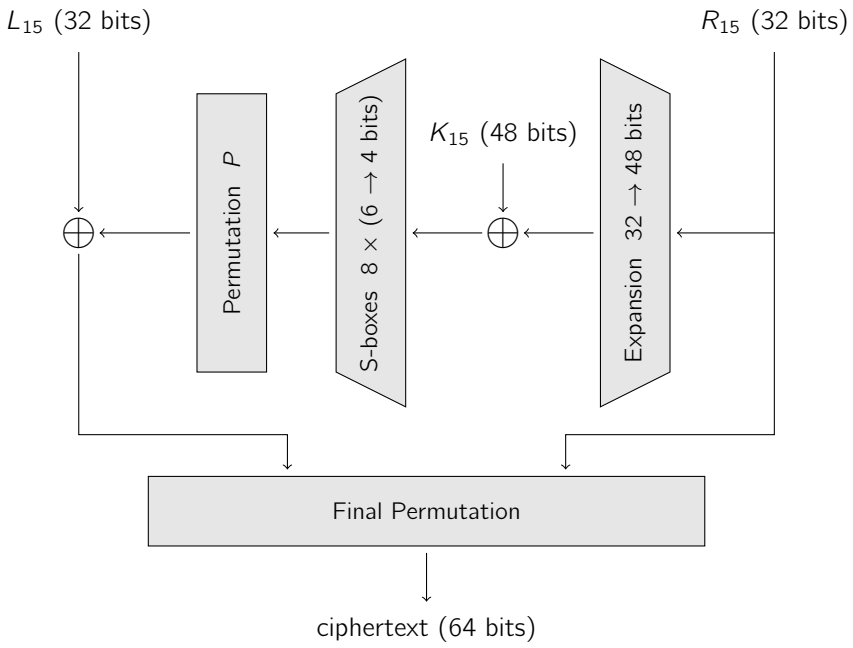


Figure 2.7: Outline of DES final round

The first differential analysis on a public-key algorithm was published by Messerges, Dabbish, and Sloan [MDS99b]. They attack an RSA exponentiation using the left-to-right square-and-multiply (cf. Algorithm 2.1). In this case, key bits are recovered one by one, from the most to the least significant bit. For each iteration of the main loop, the targeted value is the result of the modular multiplication $a \times m \bmod n$ of step 5. The decision function returns 0 if a given byte of x_i has a low Hamming weight and 1 otherwise. If the differential trace shows spikes, an adversary infers that the multiplication result is actually handled, thus the exponent bit k_i is a 1, otherwise k_i is a 0.

Messerges et al. also study the cases of chosen message and chosen exponent attacks which are even more practical than the one detailed above, since less knowledge on the implementation is required. They report that a few hundreds of traces per exponent bit are enough for the attack to be successful. An attack against RSA with Chinese Remainder Theorem (CRT) implementation, targeting the preliminary reduction step, has been presented by den Boer, Lemke, and Wicke [BLW03].

DSCA on ECC scalar multiplication can be mounted in a similar way. Note that the smaller number of key bits to be recovered and the larger number of intermediate results that may be targeted by the analysis tend to make it easier. However, a better knowledge of the implementation is required, since many more algorithmic choices can affect the manipulated data, e.g. the internal point representation, the way addition formulas are implemented, and the various possible optimizations presented in Section 1.2.

2.3.1.2 The Big Mac Attack

We present hereafter Walter's *Big Mac attack*, which may be seen as a *live* template analysis and lays the foundations of our horizontal analysis presented is Section 2.6.

Walter presents an attack [Wal01] against left-to-right sliding window and m-ary exponentiation schemes. This technique is at the border between SSCA, DSCA, and template analysis. Though it may ideally reveal information on the secret key using a single power trace as SSCA, it performs some statistical processing between different segments, which may be regarded as a DSCA operating on trace segments instead of whole traces. However, it does not use guesses on unknown key bits, contrary to a DSCA, but computes templates from a few power trace segments where known operations are performed. These templates are then compared to other operations manipulating unknown operands to identify them, which can be seen as a template analysis with a live characterization phase.

Walter observes that, during the precomputation step of a sliding window exponentiation, the computation of the first powers of m, say $m_i = m^i \bmod n$, can be easily identified and may be used to identify the operands of the multiplications performed in the exponentiation loop. Besides, these multi-precision multiplications are generally computed using the schoolbook method described in Algorithm 2.5. Thus, using notations from this algorithm, a multiplication $x \times y$ involves ${l_b}^2$ successive t-bit multiplications $x_i \times y_j$, with $0 \leq i, j \leq l_b - 1$, which can be identified on a side-channel trace T. Walter's idea consists in averaging the trace segments corresponding to the processing of $x_i \times y_j$ for a given i, i.e. computing $C(x_i) = \frac{1}{l_b} \sum_{j=0}^{l_b-1} T_{i,j}$, where $T_{i,j}$ is the segment of T corresponding to the single-precision multiplication $x_i \times y_j$. Assuming that y is a random operand (first assumption) and that l_b is big enough (second assumption), $C(x_i)$ provides an average pattern of the t-bit multiplication by x_i. An adversary then computes $C(x) = C(x_0) \mid C(x_1) \mid \cdots \mid C(x_{l_b-1})$, the average pattern of the long-integer multiplication by x, where $\mid$ stands for the concatenation.

Let us denote by $C_i = C(m_i)$ the template corresponding to the multiplication by m_i computed from the precomputation step or from a known multiplication. Walter assumes in his analysis that squarings are already identified using SSCA. Then, considering an unknown multiplication $a \times x$, where a is the algorithm accumulator and x is one of the m_i, the Big Mac attack proceeds by computing the Euclidean distance between $C(x)$ and the templates C_i. The smallest distance reveals the corresponding digit i. If squarings cannot be easily distinguished using SSCA, this technique may be used to identify them, fixing an experimental distance threshold below which a multiplication is supposed to be a squaring.

Walter notes that the pseudo-random behavior of the accumulator a during the execution of a left-to-right exponentiation such as Algorithm 2.1 fulfills the first assumption. The second assumption requires l_b to be large enough, which depends on both the targeted device and cryptographic parameters. Indeed, the efficiency of the attack increases with the key length l and decreases with the multiplier size t. Besides, the attack defeats many blinding countermeasures since (i) it succeeds using a single power trace, thus exponent blinding does not help, and (ii) it does not require to know any manipulated intermediate value, which circumvents some message and modulus blinding schemes.

Finally, Walter observes that, due to message variability, the attack may have a variable success for different exponentiation executions. This is not an issue if an adversary can acquire many exponentiation traces. In this case it is enough that the attack succeeds on one of them.

2.3.1.3 Correlation Power Analysis

Den Boer et al. [BLW03] are the first to compute a correlation coefficient between sample values and key hypothesis to improve the efficiency of the original DPA. This idea is generalized with the introduction of *Correlation Power Analysis (CPA)* by Brier, Clavier, and Olivier [BCO03; BCO04]. In this refined DSCA, the Pearson correlation coefficient is used instead of a binary decision function to compare the traces to the predicted leakages.

Brier et al. assume a Hamming distance leakage model, such that the power consumption W of a device manipulating a data D is:

$$W = a\,\mathrm{HW}(D \oplus R) + b \tag{2.1}$$

where R is the prior state of the register in which D is stored, $\mathrm{HW}(D \oplus R)$ is the Hamming distance between D and R, the factor a is a gain coefficient, and b encloses the independent power consumption of the device and the noise.

Hence, assuming that the leakage model is valid, W should fit the leakage traces for a correct guess on $D \oplus R$ and differ otherwise. When D is known — for instance it may be the output of an algorithm —, this is equivalent to guessing R. An attacker may thus target an R depending on a few key bits. Finally, taking $R = 0$ yields the Hamming weight model, which is often used in practice.

The Pearson correlation factor ρ_{WH} between W and $H = \mathrm{HW}(D \oplus R)$, considering H and W as random variables, measures the linear interdependence between H and W and is given by:

$$\rho_{WH} = \frac{\mathrm{cov}(W, H)}{\sigma_W \sigma_H} = \frac{\mathrm{E}(WH) - \mathrm{E}(W)\,\mathrm{E}(H)}{\sigma_W \sigma_H}$$

Note that the correlation factor is bounded by: $-1 \leq \rho_{WH} \leq 1$ and tends to ± 1 as the noise variance tends to zero [BCO04].

Given a set of N power traces T_i and N corresponding internal variables x_i computed according to a guess $\mathcal{G}$ on some key bits, let us denote $H_i = \mathrm{HW}(x_i \oplus R)$ the predicted Hamming distance between x_i and a reference state R. An estimate $\hat{\rho}_{WH}$ of the correlation factor ρ_{WH} is given by:

$$\hat{\rho}_{WH}(\mathcal{G}) = \frac{N \sum_{i=1}^{N} T_i H_i - \sum_{i=1}^{N} T_i \sum_{i=1}^{N} H_i}{\sqrt{N \sum_{i=1}^{N} {T_i}^2 - (\sum_{i=1}^{N} T_i)^2}\sqrt{N \sum_{i=1}^{N} {H_i}^2 - (\sum_{i=1}^{N} H_i)^2}}$$

Computing $\hat{\rho}_{WH}(\mathcal{G})$ for all possible guesses $\mathcal{G}$ yields a set of correlation traces among which one should show higher correlation values than the others — i.e. correlation values close to ± 1. These correlation peaks correspond to the correct guess and indicate the time samples where x_i is manipulated. Note that the number of traces required to distinguish the correct guess increases with the processor word size and the noise level.

Brier et al. present CPA results on a DES implementation and show that a hundred traces are enough for the attack to succeed. Amiel, Feix, and Villegas [AFV07] report the first CPA on public-key algorithms by attacking an RSA exponentiation and an ECDSA multiplication.

Many studies report that CPA significantly reduces the number of traces required by the original DPA. This is due to the use of the linear correlation factor which extracts more information from the leakage traces than the difference of means. Note however that CPA involves a leakage model, which may require a preliminary characterization step. Though it may argue in favor of DPA which does not assume any specific model, the classical Hamming distance model seems relevant when considering CMOS devices.

Note also that, though the Pearson correlation factor is mostly used in practice, some works use other correlation formulas. For instance, Spearman and Kendall coefficients have been proposed [BGLR08].

2.3.1.4 Higher-Order Differential Power Analysis

Higher-Order Differential Power Analysis (HODPA) is defined by Kocher et al. [KJJ99] to be a DPA that combines multiple samples from within a trace. In other words, an HODPA targets sensitive information split up into several time samples, generally by masking techniques, and combines the leakages of the different samples to infer information about the targeted variable. It is called *Second Order Differential Power Analysis (2ODPA)* if it makes use of two different samples per trace, or, more generally, *d-th Order* DPA if d time samples are combined.

The implementation of a 2ODPA has been investigated first by Messerges [Mes00] and many other works have followed. As it is not the core subject of our study, we only detail a few aspects of HODPA in the following. The interested reader can refer for instance to the following references [WW04; JPS05; SPQ05; CPR07; PRB09].

Considering a variable x_i masked by a random variable u_i, the classical strategy of a 2ODPA is to target two time samples corresponding to the manipulations of the masked variable $x'_i = x_i \odot u_i$ and of the mask u_i only, where $\odot$ denotes a group operation, e.g. the X-or or the modular addition. The analysis thus requires the use of a *combining function* f such that $f(x'_i, u_i)$ is correlated to x_i. A classical DSCA is then performed using $f(L(x'_i), L(u_i))$ instead of a real leakage on x_i, where $L(t)$ denotes a leakage corresponding to the manipulation of t. A general d-th order DPA follows the same method, except that a d-th order secret sharing scheme $x_i \odot u_i^{(1)} \odot \cdots \odot u_i^{(d-1)}$ requires to combine d different time samples.

A first issue when implementing an HODPA is to determine the best combining function f in such a way that the output is highly correlated to the sensitive variable. Among the various possibilities, the product and the absolute difference are mostly used as combining functions. Prouff, Rivain, and Bévan prove that, under the Hamming weight leakage model, the normalized product is optimal among all combining functions[PRB09].

Another issue when dealing with higher-order analysis is to find the time samples corresponding to the handling of the different secret shares on the power traces. Indeed, while the first-order DPA or CPA can be performed without knowing exactly where sensitive variables are manipulated, using a combining function between two samples from within a trace requires to identify them precisely.

Finally, the effect of the device and experimental noise, and thus the difficulty of an HODPA, increases exponentially with its order. Therefore, d-th order analysis for $d > 3$ are generally considered to be infeasible in practice, in particular in the context of black box attacks because of the difficulty of identifying the time samples of interest.

2.3.1.5 Mutual Information Analysis

We denote by *distinguisher* of a DSCA the statistical processing that allows the ranking of key guesses depending on the measured leakages and known data. For instance, we have seen that the distinguisher of the original DPA is the difference of means and that the distinguisher of the CPA is the Pearson correlation estimation.

Gierlichs, Batina, Tuyls, and Preneel introduce *Mutual Information Analysis (MIA)* which is a generic DSCA based on an information-theoretic distinguisher [GBTP08]. This more generic distinguisher estimates the mutual information between leakage measurements and hypothetical manipulated values. The main motivation leading to this new analysis is to avoid the leakage and measurement assumptions, on which CPA highly depends. Indeed, their information-based distinguisher can exploit arbitrary relationships between leakages and measurements — that is, other noise models than the classical Gaussian one —, and can detect non-linear relationships between manipulated data and leakages contrary to CPA.

MIA is intended to provide analysis tools for attack contexts differing significantly from the usual assumptions, for instance attacks on devices designed to resist DSCA using the dual rail precharge logic, and more generally devices not using the standard CMOS technology. As a consequence, when applied to CMOS devices, experiments show that MIA gives worse results than CPA in terms of number of required traces.

We do not detail MIA in this study as we restrict ourselves to the classical context of CMOS devices, and focus more on identifying leakages in algorithms than on the attack experimental setup. We refer the interested reader to the abundant literature covering this subject [VCS09; PR09; Ven11; Ven10].

2.3.1.6 Distinguishing Squarings from Multiplications

Amiel, Feix, Tunstall, Whelan, and Marnane [AFT+09] present a specific differential analysis aimed at distinguishing multiplications from squarings in the multiply-always algorithm.

They observe that the average Hamming weight of the output of a multiplication $x \times y$ has a different distribution whether:

- the operation is a squaring performed using the multiplication routine, i.e. $x = y$, x uniformly distributed in $[0, 2^l - 1]$,
- or the operation is an *actual* multiplication, i.e. x and y independent and uniformly distributed in $[0, 2^l - 1]$.

Thus, considering a device with a Hamming weight side-channel leakage and a single long-integer multiplication routine used to perform either $x \times x$ or $x \times y$, a set of N multiplication traces with random operands can be distinguished from a set of N squaring traces, if N is big enough with respect to the SNR. This attack can thus target an atomic implementation such as Algorithm 2.4 where many exponentiations with random messages and a fixed exponent can be executed.

The attack by Amiel et al. proceeds as follows. First, many exponentiation traces using a fixed exponent but variable data are acquired and averaged. Then, considering the average trace, an adversary aims at revealing if two consecutive operations are identical — i.e. two squarings — or different — i.e. a squaring and a multiplication. Remember that, as in the classical SSCA, two consecutive squarings reveal that a 0 bit has been manipulated whereas a squaring followed by a multiplication reveals a 1 bit. This information is obtained using the above-mentioned leakage by subtracting the parts of the average trace corresponding to two consecutive operations: peaks occur if one is a squaring and the other is a multiplication while subtracting two squarings produces only noise. It is worth noticing that no particular knowledge on the underlying hardware implementation is needed, which increases the practicality of this analysis.

A classical countermeasure against this attack is to randomize the exponent, cf. Section 2.3.2. Notice however that randomizing either the message and/or the modulus has no effect on this attack, and even makes it easier by providing the required data variability.

2.3.2 Countermeasures for Elliptic Curve Scalar Multiplication

In this section, we present some classical countermeasures used for protecting scalar multiplication implementations against DSCA. All of them consist in blinding, i.e. randomizing, the internal variables of the computation using arithmetic properties at different levels: group arithmetic, point representation, and modular arithmetic. This generally breaks the repeatability requirement of the classical DSCA framework. Along with the following ECC countermeasures, we provide, when they exist, the corresponding techniques used to protect an RSA exponentiation.

Also, we detail the cases when some of these countermeasures can be circumvented and thus require the implementation of a combination of them. In the following, we consider only first-order masking countermeasures, since HODPA seems less applicable to ECC and RSA computations than to symmetric block ciphers. This is mainly due to the fact that secret shares obtained with multi-precision arithmetic blinding countermeasures are more difficult to combine than shares resulting from logical or arithmetic blinding on single words[1].

We use in the sequel of this section the following notations: a scalar multiplication $Q = kP$ takes place on an elliptic curve $\mathcal{E}$ defined by the equation $y^2 = x^3 + ax + b$ over a field $\mathbb{F}_q$ of characteristic $p > 3$. The affine coordinates of the base point P are denoted by (x, y), and the cardinality of $\mathcal{E}(\mathbb{F}_q)$ is denoted by n.

2.3.2.1 Scalar Blinding

This countermeasure is introduced by Coron [Cor99]. It consists in computing the scalar multiplication $k'P$ instead of kP where $k' = k + rn$ for some α-bit random value r. From the group law we have $k'P = kP$.

The cost of this countermeasure is the consequence of the extension of the scalar. It thus depends on α and on the key size as shown in Table 2.3.

Table 2.3: Theoretical cost of the scalar blinding countermeasure for common ECC key lengths

α (in bits)	Cost for $l = 192$	Cost for $l = 256$	Cost for $l = 384$	Cost for $l = 512$
24	12.5 %	9.4 %	6.3 %	4.7 %
32	16.7 %	12.5 %	8.3 %	6.3 %
48	25 %	18.8 %	12.5 %	9.4 %
64	33.3 %	25 %	16.7 %	12.5 %

While this countermeasure is efficient in the general case, Ciet shows that when p is a pseudo-Mersenne prime number, this countermeasure should be used with caution [Cie03]. Indeed, pseudo-Mersenne numbers of the form $2^l - \epsilon$ where $\epsilon < 2^{l/2}$ is sometimes chosen because it allows a fast modular reduction. For instance, we recall below the prime numbers selected for the NIST standard curves [NIST06]:

$\mathrm{P192} = 2^{192} - 2^{64} - 1$
$\mathrm{P224} = 2^{224} - 2^{96} + 1$
$\mathrm{P256} = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$
$\mathrm{P384} = 2^{384} - 2^{128} - 2^{96} + 2^{32} - 1$
$\mathrm{P521} = 2^{521} - 1$

[1] To our knowledge, the only reported HODPA on ECC scalar multiplication is an attack from Joye [Joy04] on $\mathbb{F}_{2^n}$ elliptic curves. This second-order analysis exploits a flaw of an additive randomization countermeasure in $\mathbb{F}_{2^n}$

According to Hasse bounds, considering an elliptic curve over $\mathbb{F}_p$, we have:

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

Thus, denoting l the bit length of p, the order n of the group has its $l/2$ most significant bits identical with p. Due to the construction of the NIST prime numbers, these bits are all 1, except for P256.

Construction of N192, N224, N384, and N521 :

$$\underbrace{11111111111111\ \ldots\ 1}_{l/2}\underbrace{\ \ldots\ \textit{miscellaneous}\ \ldots}_{l/2}$$

It follows that with an r of length $\alpha < l/2$, the product rn has the structure :

$$\underbrace{[r-1]}_{\alpha}\underbrace{11111111111\ \ldots\ 1}_{l/2-\alpha}\underbrace{\ \ldots\ \textit{miscellaneous}\ \ldots}_{l/2+\alpha}$$

Therefore, the blinded scalar $k' = k + rn$ has $l/2 - \alpha$ bits complementary to those of k whatever the value of r. Thus a DSCA can recover almost half of a private key if α is small.

Remark 10. This countermeasure is similar to the RSA exponent blinding: $m^{d'}$ mod n with $d' = d + r\varphi(n)$ and $\varphi(n) = (p-1)(q-1)$.

2.3.2.2 Projective Coordinates Blinding

This countermeasure is known as the third Coron's countermeasure [Cor99]. It consists in applying a multiplicative blinding to the point coordinates using the projective representation redundancy.

We mentioned in Section 1.1.2 that a point represented by $(X : Y : Z)$ in homogeneous coordinates, resp. Jacobian coordinates, is also represented by $(\lambda X : \lambda Y : \lambda Z)$, resp. $(\lambda^2 X : \lambda^3 Y : \lambda Z)$, for every $\lambda \neq 0$. Thus, at the beginning of the scalar multiplication, one can pick at random a scalar λ and apply a multiplicative blinding to the input point, and possibly to other points manipulated by the algorithm. Furthermore, this blinding can be performed again during the computation of the scalar multiplication.

This countermeasure has a very low cost since only a few multiplications are required — $3M$ for the homogeneous representation and $4M+1S$ for the Jacobian representation. Due to the final inversion, the blinding is naturally removed at the end of the scalar multiplication.

However, Goubin [Gou02] shows that, since null coordinates are not affected by a multiplicative blinding, a DSCA can be mounted in spite of this countermeasure if the input point can be chosen by an adversary. Indeed, most standard curves have a special point with a null coordinate which is not affected by the blinding. Goubin's differential attack thus consists in choosing the input point such that the special point is computed at some point of the algorithm depending on a few unknown key bits. This attack is refined by Akishita and Takagi [AT03] by considering that null intermediate results of doubling and addition formulas can also be detected. The so-called *zero-value point*

attack does not require that a point with a null coordinate is computed during the scalar multiplication process, but only that a register used in the computation is set to zero at some point of the computation. This condition is less restrictive than the previous one and enables the attack in most cases, yet it requires a detailed knowledge of the targeted implementation.

The scalar or input point blinding countermeasures are recommended in contexts where zero-value point attacks can be applied.

2.3.2.3 Random Curve Isomorphism

This countermeasure is introduced by Joye and Tymen [JT01]. It consists in performing the scalar multiplication on a curve isomorphic to $\mathcal{E}$.

Algorithm 2.11 details how to implement this countermeasure using projective homogeneous coordinates. Note that, if Jacobian coordinates are used, step 6 should compute the affine point $(X_{Q'} \times (\alpha Z_{Q'})^{-2}, Y_{Q'} \times (\alpha Z_{Q'})^{-3})$.

Algorithm 2.11 Random curve isomorphism countermeasure using $\mathcal{H}$

Input: $P \in \mathcal{E}(\mathbb{F}_q)$, a, b, $k \in \mathbb{N}^*$
Output: kP

1: Pick at random $\alpha \in {\mathbb{F}_q}^*$
2: $P' \leftarrow (\alpha^2 x : \alpha^3 y : 1)$
3: $a' \leftarrow \alpha^4 a$
4: $b' \leftarrow \alpha^6 b$ ▷ optional: b is not used in most cases
5: $Q' \leftarrow (X_{Q'} : Y_{Q'} : Z_{Q'}) = kP'$
6: $Q \leftarrow (X_{Q'} \times (\alpha^2 Z_{Q'})^{-1}, Y_{Q'} \times (\alpha^3 Z_{Q'})^{-1})$
7: **return** Q

In practice, this countermeasure has a moderate cost and an effect comparable to the projective coordinate randomization, with some slight differences however. First, the randomization of the input point applies to the affine coordinates, thus its Z coordinate is not randomized. This allows using mixed affine-projective addition formulas in left-to-right scalar multiplication algorithms, contrary to the projective coordinates blinding countermeasure. Second, the curve parameter a is randomized, thus doubling formulas will be randomized also, which may provide a better DSCA protection. However, this property also implies that fast doubling formulas for $a = -3$ cannot be used.

Finally, this countermeasure has the same weakness as the projective coordinates blinding against zero-value point attacks [Gou02; AT03].

2.3.2.4 Input Point Blinding

This countermeasure is known as the second Coron's countermeasure [Cor99]. Since most DSCA require that input points are known or chosen by the adversary, this countermeasure consists in blinding the input point by computing $k(P + R) - kR$ where R is a randomly generated point.

Obviously, applying the countermeasure this way is very costly since two scalar multiplications must be performed instead of one. To address this issue, Coron shows that it may be implemented efficiently if k is fixed: the system is initialized by choosing a random point $R \in \mathcal{E}(\mathbb{F}_q)$ and $S = kR$ is computed and stored. Then, each scalar multiplication is computed according to Algorithm 2.12.

Algorithm 2.12 Coron's input point blinding

Input: $P \in \mathcal{E}(\mathbb{F}_q)$, ($R$, k, and $S = kR$ stored)
Output: kP
1: $P \leftarrow P + R$
2: $Q \leftarrow kP$
3: $Q \leftarrow Q - S$
4: Pick at random $\alpha \in \{0, 1\}$
5: $R \leftarrow (-1)^\alpha 2R$ ▷ store the new R
6: $S \leftarrow (-1)^\alpha 2S$ ▷ store the new S
7: **return** Q

Note however that this implementation has some weaknesses due to the basic update process. For instance, it does not prevent the doubling attack, cf. Section 2.2.2.

Another implementation of this countermeasure is known as *Basic Random Initial Point (BRIP)* [MMM04]. It consists in computing $kP + R = kP + 1\bar{1}\bar{1}\ldots\bar{1}R$ using Algorithm 2.13 before subtracting R.

The cost of BRIP algorithm is comparable to the double-and-add-always algorithm. Note however that point readdition formulas should be used instead of mixed affine-projective formulas at steps 8 and 10 since Q_2 is not likely to be computed in affine coordinates. The cost per bit of Algorithm 2.13 using Jacobian coordinates is thus $15M + 9S + 18A$ in the general case or $15M + 7S + 19A$ if $a = -3$. The corresponding detailed cost with respect to S/M and A/M is given in Table 2.4.

Algorithm 2.13 BRIP scalar multiplication

Input: $P \in \mathcal{E}(\mathbb{F}_q)$, $k = (k_{l-1}k_{l-2}\ldots k_0)_2$
Output: kP
Uses: Q_0, Q_1, and Q_2
1: Pick at random $R \in \mathcal{E}(\mathbb{F}_q)$
2: $Q_0 \leftarrow R$
3: $Q_1 \leftarrow -R$
4: $Q_2 \leftarrow P - R$
5: **for** $i = l - 1$ **to** 0 **do**
6: $\quad Q_0 \leftarrow 2Q_0$
7: $\quad$ **if** $k_i = 0$ **then**
8: $\quad\quad Q_0 \leftarrow Q_0 + Q_1$
9: $\quad$ **else**
10: $\quad\quad Q_0 \leftarrow Q_0 + Q_2$
11: $Q_0 \leftarrow Q_0 + Q_1$
12: **return** Q_0

Table 2.4: Detailed cost of BRIP algorithm using $\mathcal{J}$ expressed as field multiplications per scalar bit, label (c.o.) denote the use of composite operations

Algorithm	Coord.	a	$S/M = 1$		$S/M = 0.8$	
			$A/M = 0.2$	$A/M = 0.1$	$A/M = 0.2$	$A/M = 0.1$
2.13	$\mathcal{J}$	any	27.6	25.8	25.8	24.0
2.13	$\mathcal{J}$	-3	25.8	23.9	24.4	22.5
2.13 (c.o.)	$\mathcal{J}$	any	23.6	22.3	22.6	21.3

Using Composite Operations The main loop of Algorithm 2.13 can be rewritten to make use of the composite operation $2P_1 + P_2$ as:

for $i = l - 1$ **to** 0 **do**
  $Q_0 \leftarrow 2Q_0 + Q_{1+k_i}$

Considering the Jacobian composite addition formula described in 1.1.2.3 using readditions of Q_0, the average cost of the algorithm becomes $16M + 5S + 13A$. The corresponding detailed computation costs are given in Table 2.4.

This countermeasure may be generalized to the RSA exponentiation. Note however that an attack against the BRIP exponentiation is presented by Yen et al. [YLMH05], cf. Section 2.2.2.

Remark 11. The RSA countermeasure corresponding to input point blinding is generally known as *message blinding*. Given that an RSA public exponent e is chosen small, it consists in a multiplicative randomization: $m' = r^e m \bmod n$ where r is a random value and m is the message to sign or the ciphertext. At the end of the exponentiation, the result is recovered as $m^d \bmod n = r^{-1}(m')^d \bmod n$.

Alternatively, message blinding can be achieved using an extended modulus, that is computing: $(m + rn)^d \bmod kn$ where k is a constant and r is a random number from $[0, k - 1]$. The result is reduced modulo n at the end of the exponentiation.

2.3.2.5 Field Arithmetic Blinding

While the previous countermeasures perform blinding at the curve arithmetic or point representation level, a randomization of the computation is possible at the field arithmetic level. For instance, Joye and Tymen show how to use random field isomorphisms [JT01] to implement a DSCA resistant scalar multiplication over $\mathbb{F}_{2^n}$.

Another technique is proposed considering the underlying implementation of modular arithmetic [DV11]. As we mentioned in Section 2.1.2, modular multiplications are generally implemented using one of the following methods: Montgomery, Barrett, Quisquater, etc. We briefly present hereafter the blinded Montgomery method using notations from Section 2.1.2.

First of all, Montgomery representation of operands u and v are blinded by adding a small multiple of the modulus, say $x = uR + r_1 n \bmod cn$ and $y = vR + r_2 n \bmod cn$ where $2^{i-1} \leq c < 2^i$, and r_1, r_2 are random values from $[1, c - 1]$. Considering the

new constant $R' = 2^{l+2i}$, they define the modified Montgomery reduction presented in Algorithm 2.14. This algorithm adds a random multiple of the modulus to the result while maintaining its size to $l+i$ bits. A final normalization step at the end of the scalar multiplication or exponentiation yields a result modulo n.

Algorithm 2.14 Blinded Montgomery modular reduction

Input: $w < 2^{2l+2i}$, $n < 2^l$, $R' = 2^{l+2i}$, $n' = -n^{-1} \bmod R'$ with $\gcd(R', n) = 1$ and $w < nR'$
Output: $r < (wR'^{-1} \bmod n) + (k+1)n$, s.t. $r \bmod n = xR^{-1} \bmod n$
1: Pick at random $k \in [0, 2^i - 1]$
2: $s \leftarrow w \bmod R'$
3: $q \leftarrow n' \times s \bmod R'$
4: $q \leftarrow q + k \times R'$
5: $r \leftarrow (w + q \times m)/R'$
6: **return** r

The cost of this countermeasure depends on the size i allowed for the randomization process. For instance, $i = 16$, 24, or 32 may be used depending on the targeted security level. The cost of the blinding process thus depends on the operands expansion factor $(l+i)/l$ and on the implementation of the random generation of step 1.

2.4 Fault Analysis

Fault analysis is a class of physical attacks relying on a stronger assumption regarding the adversary: he is now able to perturb the targeted device during instruction processing. Thus, fault analysis are said to be *active* attacks.

Two kinds of perturbations are generally considered: faults induced in computations and faults induced in memories. The former are said to be *transient*, since they impact a single computation result, whereas the latter permanently modify data in memory — or modify data for the duration of the current session considering faults induced in volatile memories.

Boneh, DeMillo, and Lipton [BDL96] on one hand, and Anderson and Kuhn [AK96] on the other hand, report that various means can be used to induce faults on embedded devices such as smart cards. For instance, power spikes or glitches which consist in very short perturbations on the power supply of the device may lead to wrong operations. Similarly, external clock glitches can affect the circuitry of smart cards and make the CPU execute wrong instructions or cause errors in memories. More invasive techniques require the depackaging of the chip through mechanical and/or chemical means. This allows using a wide range of physical perturbations directly on the chip instead of tampering with its inputs. Focusing radiations, such as UV light, on the chip is an interesting solution to generate faults [BECN+06; GT04; Sko05]. Nowadays, most security evaluation labs use lasers which enable a precise control of where to induce a fault. Such techniques makes it possible to target a particular area of a chip: CPU, coprocessors, memories, buses, etc.

Different fault models may be considered when studying fault analysis depending on the presumed ability of the adversary to control the effect of his perturbation. For instance, it may be assumed that an attacker can tamper with a computation so that it returns a random result. With a stronger adversarial model, it may be assumed that an attacker can modify some data bytes or bits only. Besides, the effect on the targeted bits may be random, fixed, or causing bit flips. This latter point requires generally to take into consideration the targeted chip design and technology.

The remainder of this section is organized as follows. Section 2.4.1 presents classical fault analysis on ECC algorithms. Then Section 2.4.2 focuses on a particular class of fault analysis: the safe-errors. Finally, the recent combined attacks which use both faults and classical side-channel analysis are presented in Section 2.4.3.

2.4.1 Fault Analysis on Elliptic Curve Cryptosystems

The first practical fault attack on a cryptosystem is presented in 1997 by Boneh, DeMillo, and Lipton [BDL97]. They recover the private exponent of an RSA exponentiation using the CRT implementation with a single faulty signature. The same year, Biham and Shamir [BS97] show another attack known as *Differential Fault Analysis (DFA)* on the DES block cipher that may potentially apply to any secret key cryptosystem. Their attack requires a few tens of pairs of correct and faulty ciphertexts and analyses the differences induced by the fault in each pair — therefore the term *differential*.

The first fault analysis on an ECC implementation is reported by Biehl, Meyer, and Müller [BMM00]. The two different kinds of attacks described in this paper are presented hereafter.

2.4.1.1 Weak Curve Fault Attack

First of all, Biehl et al. present an attack on the EC ElGamal cryptosystem [BMM00]. The EC ElGamal encryption, resp. decryption, is described in Algorithm 2.15, resp. Algorithm 2.16, where d is a private key such that $1 \leq d < n$ and $Q = dP$ is the corresponding public key.

Algorithm 2.15 EC ElGamal encryption

Input: Public point $P \in \mathcal{E}(\mathbb{F}_q)$, $n = \text{ord}_{\mathcal{E}}(P)$, public key Q, and message m
Output: Ciphertext (R, c)
1: Pick at random k in $[1, n-1]$
2: $P_k \leftarrow kP$
3: $Q_k \leftarrow kQ$
4: $c \leftarrow x_{Q_k} \oplus m$
5: **return** (P_k, c)

Algorithm 2.16 EC ElGamal decryption

Input: Private key d and ciphertext (R, c)
Output: Plaintext m
1: $Q_k \leftarrow dR$
2: $m \leftarrow x_{Q_k} \oplus c$
3: **return** m

Biehl et al. observe that the EC Elgamal decryption routine may be seen as a blackbox function taking as input any point P and returning the x coordinate of dP. Obviously, implementations should first check that the input point belong to the curve $\mathcal{E}$. Considering a "buggy" implementation that does not perform this verification, Biehl et al. show a very simple attack. One first observes that the curve parameter b is not involved in point addition and doubling formulas. Therefore it is possible to find a point P belonging to a curve $\mathcal{E}' : y^2 = x^3 + ax + b'$ such that $r = \text{ord}_{\mathcal{E}'}(P)$ is small enough to be able to compute discrete logarithms in the subgroup of $\mathcal{E}'(\mathbb{F}_q)$ generated by P.

The remainder of the attack is straightforward: the attacker inputs such a point P to the decryption function and recovers the x coordinate of dP computed on $\mathcal{E}'$. Then he computes the discrete logarithm to get $d \bmod r$ — note that recovering the y coordinate of dP, if necessary, demands a guess on a single bit knowing a and b' —, and repeats these operations for other points with different small orders until having enough relations to compute d using the CRT.

This simple attack highlights how checking the validity of inputs of cryptographic algorithms may be important. In a context sensitive to fault analysis, Biehl et al. show that checking the validity of outputs is important as well. Indeed, if an implementation of Algorithm 2.16 checks the input point R, but not the result of the scalar multiplication

dR, an adversary can use fault injection to tamper with the value of R at the very beginning of the scalar multiplication and conduct an attack similar to the previous one. However, the attack is significantly more difficult in this case since neither b' nor the y coordinate of the result are known to the attacker. Nevertheless, Biehl et al. prove that the attack remains possible if one can use faults to modify the value of single bits. They show also that a similar fault analysis is possible on the ECDSA signature algorithm.

Ciet and Joye improve this attack [CJ05a] by considering random errors and fault injection in curve parameters. Both of these works therefore recommend to check the result of scalar multiplications involving a sensitive scalar before returning or using it.

Remark 12. Fouque, Lercier, Réal, and Valette [FLRV08] show that Montgomery ladder (Algorithm 1.10) x-only implementation — i.e. without y coordinate computation — is unsafe towards fault analysis. Indeed, since the y coordinate is not computed, the check that a point (x, y) belongs to a curve of parameters a and b consists only in verifying that $x^3 + ax + b$ is a square in the base field. Fouque et al. prove that this check is too weak to prevent fault analysis.

Remark 13. Bernstein [Ber06] include in the design of Curve25519 for high-speed Diffie-Hellman the security requirement that the curve is *twist-secure*. It consists in choosing the curve such that a change moving a point of the curve to the quadratic twist does not succeed in breaking the discrete logarithm problem.

2.4.1.2 Differential Fault Analysis on ECC

Biehl et al. [BMM00] also present a DFA on a fixed-scalar multiplication such as the one from Algorithm 2.16 when no result check is implemented. We sketch this attack on the right-to-left double-and-add method. Note however that the principle of this differential analysis applies to any kind of scalar multiplication algorithm, regular or not. In the following, let us denote by Q_i the value of register Q at the end of the i-th loop iteration of Algorithm 1.2, $0 \leq i < l$.

Being a differential analysis, this attack requires both a correct and an erroneous result. We assume that an adversary is able to induce a 1-bit fault on the x coordinate of Q_i, for $l - s \leq i < l$, i.e. during the processing of the s last bits of the scalar, where s is be small enough to exhaust 2^s computations. We denote $\tilde{Q}_i$ the corresponding altered value of Q_i. Given $\tilde{Q}_{l-1}$ the faulty output of the scalar multiplication and Q_{l-1} the correct one, let define $Q_i^{(u)} = Q_{l-1} - u2^i P$ and $\tilde{Q}_i^{(u)} = \tilde{Q}_{l-1} - u2^i P$, where $0 \leq u < 2^s$ and P is the base point of the scalar multiplication. An exhaustive search for u — i.e. on the s most significant bits of the scalar — leads to at least one — and only one with high probability — value of u for which $Q_i^{(u)}$ and $\tilde{Q}_i^{(u)}$ differ from one bit only. Thus the s most significant bits of the scalar are revealed, and the attack can be repeated on the following s bits, and so on until the whole scalar is recovered.

An obvious countermeasure against this kind of analysis is again to check for consistency the output of every scalar multiplication involving a sensitive scalar, namely to check that coordinates of the result are valid with respect to the curve equation.

Nevertheless, Blömer, Otto, and Seifert have proven that such countermeasures are circumvented by a so-called *sign change attack* [BOS06]. This attack consists in inducing a fault in the sign of the Y projective coordinate of a register during the computation of the scalar multiplication. It allows a similar analysis than previously but is undetectable by checking that point coordinates verify the curve equation. Indeed, on an elliptic curve in short Weierstraß form, changing the sign of the Y coordinate results in replacing a point by its negative, which is still on the curve. Blömer et al. show that algorithms taking advantage of the signed NAF representation are particularly prone to the sign change attack.

Countermeasures As presented by Rivain on the RSA exponentiation [Riv09], two main directions have been investigated to prevent DFA.

First, some countermeasures use an *extended modulus* to add redundancy in the computation and check the consistency of data at the end of an exponentiation. This is the case with Shamir's countermeasure [Sha98; ABF+03; CJ05b] and Vigilent's scheme [Vig08]. In the ECC scalar multiplication context, Blömer et al. [BOS06] and Baek and Vasyltsov [BV07] propose to perform computations in the ring $\mathbb{Z}/rp\mathbb{Z}$ instead of $\mathbb{F}_p$, with r a random number whose size depends on the targeted security level. This way, results may be checked modulo r and intermediate values are randomized which also prevents DSCA.

Second, other proposals consist in using *self-secure* algorithms, i.e. algorithms provided with consistency properties that may be checked during or at the end of computations. For instance, Giraud [Gir06] proposes to check the invariant provided by Montgomery ladder, namely $Q_1 - Q_0 = P$ using notations from Algorithm 1.10. Similarly, Boscher, Naciri, and Prouff [BNP07] propose to verify that $Q + T + P = R$ at the end of the right-to-left double-and-add-always, see Algorithm 1.9. We note also that the Joye ladder, cf. Algorithm 1.11, has the invariant $Q_0 + Q_1 = 2^{i+1}P$ after processing the i-th scalar bit. Finally, in RSA exponentiation context, Rivain [Riv09] uses a multi-exponentiation algorithm to compute both m^d and $m^{\varphi(n)-d}$ and checks that their product equals 1 modulo n. Considering ECC, an equivalent solution is to compute both kP and $(n-k)P$, which sum to $\mathcal{O}$. Rivain shows that this method is more efficient than other self-secure algorithms.

Yen, Kim, Lim, and Moon note that all these proposals rely in the end on a comparison which is itself prone to faults [YKLM01b]. The publication by Kim and Quisquater of a successful second-order fault analysis [KQ07] demonstrates the importance of this issue. Therefore, Yen et al. devise a third strategy to circumvent DFA that does not rely on a comparison: the *infective computation* technique which intrinsically corrupts the result of a computation if an error occurs. More details about this solution are given in Section 2.4.3.

2.4.2 Safe-Error Analysis

Safe-error analysis denotes a particular class of fault attacks in which an adversary induces a fault on a device and observes if it has a consequence on its output. Most of

the time, the exact effect of the fault does not matter much, the only used leakage is the correctness of the result.

Safe-error analysis has been introduced by Yen and Joye [YJ00] on an RSA square-and-multiply-always exponentiation. It straightforwardly extends to ECC double-and-add-always algorithms. For instance, considering Algorithm 1.9, the attack consists in perturbing the computation of a point addition to identify if it corresponds to the step 5 or to the step 7. Indeed, if correctly implemented, these two steps are indistinguishable from an SSCA point of view. However, perturbing the computation of step 5 yields an incorrect result at the end of the algorithm, whereas tampering with the computation of step 7 has no influence on the output. This is due to the use of *dummy* additions in the double-and-add-always algorithm. By definition, the result of a dummy operation is not used for computing the output of an algorithm, contrary to other operations. Thus, any operation that is either a regular one or a dummy one depending on a secret key bit may be targeted by a safe-error.

The literature [YKLM01a; JY03] distinguishes between two different kinds of safe-errors: *C* and *M safe-errors*. C safe-errors, also called transient safe-errors, perturb the result of a targeted operation only. This is the kind of error considered in the above attack description. M safe-errors are faults injected on a value in the memory. This model is stronger than the C safe-errors model. Observe for example that an adversarial M safe-error model allows at least the same attacks as the C safe-error model since each error induced in a computation may also be induced in the memory where its result is stored.

The previous countermeasures consisting in checking outputs and intermediate results are ineffective here. However, note that performing such an analysis reveals scalar bits one by one and therefore requires that the scalar be fixed. As a consequence, a possible countermeasure for the double-and-add-always algorithm is the scalar blinding presented in Section 2.3.2. A more general guidance is to avoid algorithms involving dummy operations. For instance Montgomery and Joye ladders should be preferred to double-and-add-always algorithms.

2.4.3 Combined Fault and Side-Channel Analysis

Recently, Amiel, Feix, Marcel, and Villegas [AFMV07] have introduced the principle of *Passive and Active Combined Attacks* (*PACA* or *combined attacks* for short) using both faults and classical side-channel analysis. They show that using these two techniques together can break an RSA implementation provided with countermeasures against these threats considered separately. The same year, Robisson and Manet [RM07] have presented the differential behavioral analysis which combines classical DPA and safe-errors.

Amiel et al. present a simple attack on an atomic RSA exponentiation using Algorithm 2.4. A fault attack is used to bypass step 2 such that register R_1 is left uninitialized, which is expected to give $R_1 = 0$ in most cases. Equivalently, the fault may tamper with the value of R_1 or the value of R_1 pointer to yield a similar result. Then, a power signature analysis (cf. Section 2.2.2.2) is conducted on the power trace assuming that the manipulation of a whole zero register can be distinguished from the

manipulation of a balanced one. This attack straightforwardly extends to ECC scalar multiplication [STA+10].

Classical side-channel and fault analysis countermeasures have no effect against this attack since the detection of an erroneous result happens at the end of the computation, which is too late. Indeed, in this attack an adversary needs the power trace only, not the faulty output. Thus, countermeasures implemented independently against side-channel and fault analysis do not prevent the combined analysis in general.

Amiel et al. propose a so-called *detect and derive* countermeasure based on the principle of infective computation introduced by Yen et al. [YKLM01b]. This solution consists in masking the exponent/scalar using a *check value* computed on the inputs to be protected against faults. Then, the exponent/scalar is unmasked on the fly during the computation, such that a fault induced on one of the inputs yields a wrong check value and consequently corrupts the exponent/scalar.

This countermeasure is refined by Schmidt, Tunstall, Avanzi, Kizhvatov, Kasper, and Oswald [STA+10]. They combine the use of an extended modulus rn for RSA, or computations in a ring $\mathbb{Z}/rp\mathbb{Z}$ in the case of ECC over $\mathbb{F}_p$, and the infective computation strategy, such that the check value is an intermediate value modulo r. The size of r is thus a security parameter on which depends the probability of a fault to be detected on one hand and the cost of the countermeasure on the other hand.

2.5 Square-Always Exponentiation

An exponentiation is generally computed using a sequence of multiplications, some of them having different operands and some of them being squarings. We presented in Section 2.1.1 the two main options for implementing the exponentiation in an SSCA resistant manner: the regular algorithms where the sequence of multiplications and squarings does not depend on the secret exponent, and the atomic multiply-always algorithm which performs multiplications only and smoothens their processing in order to hide the processing of exponent bits.

Amiel et al. [AFT+09] show that an intrinsic difference between multiplications and squarings can provide exploitable side-channel leakages to an attacker, cf. Section 2.3.1.6. This leakage enables an attack on the atomic implementation. In spite of the possibility to apply the exponent randomization, this attack brings into light an intrinsic flaw of the multiply-always algorithm: the fact that at some instant a multiplication performs a squaring ($x \times x$) or not ($x \times y$) depending on the exponent. On the other hand, regular exponentiation algorithms are naturally immune to this kind of analysis, but have a higher cost.

Our contribution is to propose a new exponentiation scheme using squarings only, which removes the former leakage and is faster than using regular algorithms. The principle of our method and several algorithms are presented in Section 2.5.1. Also, we introduce in Section 2.5.2 new algorithms having a particularly low cost when two squarings can be parallelized. Finally, some practical results are given in Section 2.5.3. This research has been published [CFG+11g] and has been the subject of a patent application [CFG+11d].

2.5.1 Square-Always Countermeasure

We present in this section new exponentiation algorithms which simultaneously benefit from efficiency of the atomicity principle and immunity against the aforementioned weakness of the multiply-always method.

2.5.1.1 Principle

As far as we know, all atomic exponentiation algorithms proposed in the literature use multiplications only to perform both squarings and "actual" multiplications. It is well known however that a multiplication can be computed using two or more squarings, for instance using expression (2.2) or (2.3). Therefore, we propose in the following new atomic algorithms using squarings only to perform all multiplications in an exponentiation.

In characteristic different from 2, we have :

$$x \times y = \frac{(x+y)^2 - x^2 - y^2}{2} \tag{2.2}$$

$$x \times y = \left(\frac{x+y}{2}\right)^2 - \left(\frac{x-y}{2}\right)^2 \tag{2.3}$$

Our countermeasure completely prevents the attack described in Section 2.3.1.6 since only squarings are performed. Moreover, applying the atomicity principle prevents SSCA as well. As a matter of comparison with other SSCA countermeasures (see Section 2.2.1), Figure 2.8 depicts the side-channel leakage of an atomic algorithm using squaring only.

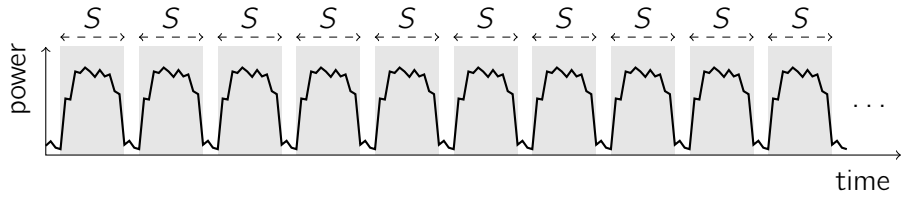


Figure 2.8: Atomic square-always side-channel leakage

At first glance, expression (2.2) requires three squarings to perform a multiplication whereas expression (2.3) requires only two. Further analysis reveals however that using (2.2) or (2.3) in algorithms 2.1 and 2.2 has always the cost of replacing multiplications by twice more squarings. Indeed, in the multiplication $a \leftarrow a \times m$ of Algorithm 2.1 m is a constant operand. Therefore, we need to compute $m^2 \bmod n$ only once at the beginning of the exponentiation, then (2.2) computes $a \times m$ using only two squarings.

This trick does not apply to Algorithm 2.2 since no operand is constant in step 5. However $b \leftarrow b^2$ is the following operation. Using equation (2.2) in Algorithm 2.2 we can store $t \leftarrow y^2$ and save the following squaring: $b \leftarrow t$. The resulting cost is again equivalent to trading one multiplication for two squarings.

Remark 14. In our context, (2.2) or (2.3) refer to operations modulo n. Notice however that divisions by 2 in these equations require neither inversion nor multiplication, only a shift. For example, assuming that n is odd, we recommend computing $z/2 \bmod n$ in the following atomic way:

$t_0 \leftarrow z$
$t_1 \leftarrow z + n$
$\alpha \leftarrow z \bmod 2$
return $t_\alpha/2$

2.5.1.2 Atomic Algorithms

Trading multiplications for squarings in Algorithms 2.1 and 2.2 just requires to apply formula (2.2) or (2.3) at step 5 in Algorithm 2.1 or step 5 in Algorithm 2.2. However the resulting algorithms would still present a leakage since different operations would be performed when processing a 0 or 1 bit. Hence it is necessary to apply the atomicity principle on these algorithms.

This step is achieved by identifying a minimal pattern of operations to be performed on each loop iteration, then rewriting the algorithms using this pattern. For the considered algorithms, the minimal pattern should obviously contain a single squaring since it is the only operation required by the processing of a 0 bit and performing dummy

squarings would slow down the algorithm. An addition, a subtraction and a division by 2 should also be present to compute formulas (2.2) or (2.3). Finally some additional operations are required to manage the loop counter and the pointer on exponent bits.

Algorithm 2.17 presented hereafter details how to implement atomically the square-always method in a left-to-right exponentiation using equation (2.2). As Chevallier-Mames et al. [CMCJ04], we use a matrix for a more readable and efficient implementation:

$$M = \begin{pmatrix} 1 & 1 & 1 & 0 & 2 & 1 & 1 & 1 & 2 & 1 \\ 2 & 0 & 1 & 2 & 2 & 2 & 2 & 2 & 3 & 0 \\ 1 & 1 & 3 & 0 & 0 & 0 & 0 & 2 & 0 & 0 \\ 3 & 3 & 3 & 0 & 3 & 3 & 1 & 1 & 3 & 1 \end{pmatrix}$$

Algorithm 2.17 Left-to-right atomic square-always exponentiation with (2.2)

Input: $m, n \in \mathbb{N}$, $m < n$, $d = (d_{k-1} d_{k-2} \dots d_0)_2$
Output: $m^d \bmod n$
Uses: R_0, R_1, R_2, and R_3
1: $R_0, R_2 \leftarrow 1$
2: $R_1 \leftarrow m$
3: $R_3 \leftarrow m^2/2 \bmod n$
4: $j \leftarrow 0$
5: $i \leftarrow k - 1$
6: **while** $i \geq 0$ **do**
7: $\quad R_{M_{j,0}} \leftarrow R_{M_{j,1}} + R_{M_{j,2}} \bmod n$
8: $\quad R_{M_{j,3}} \leftarrow {R_{M_{j,3}}}^2 \bmod n$
9: $\quad R_{M_{j,4}} \leftarrow R_{M_{j,5}}/2 \bmod n$
10: $\quad R_{M_{j,6}} \leftarrow R_{M_{j,7}} - R_{M_{j,8}} \bmod n$
11: $\quad j \leftarrow d_i(1 + (j \bmod 3))$
12: $\quad i \leftarrow i - M_{j,9}$
13: **return** R_0

The main loop of Algorithm 2.17 can be viewed as a four state machine where each row j of M defines the operands of the atomic pattern, cf. Figure 2.11. The atomic pattern itself is given by the content of the loop, i.e. steps 7 to 12. An exponent bit d_i is processed by the state $j = 0$ (resp. $j = 3$) if the previous bit d_{i+1} is a 0 (resp. a 1). This state is followed by the processing of the next bit if $d_i = 0$, or by the states $j = 1$ and $j = 2$ if $d_i = 1$. For the sake of clarity, we present below the four sequences of operations corresponding to each state. The dummy operations are identified by a $\star$.

$j = 0$ ($d_i = 0$ or 1)	
$R_1 \leftarrow R_1 + R_1 \bmod n$	$\star$
$R_0 \leftarrow {R_0}^2 \bmod n$	
$R_2 \leftarrow R_1/2 \bmod n$	$\star$
$R_1 \leftarrow R_1 - R_2 \bmod n$	$\star$
$j \leftarrow d_i$	[$\star$ if $d_i = 0$]
$i \leftarrow i - (1 - d_i)$	[$\star$ if $d_i = 1$]

$j = 2$ ($d_i = 1$)	
$R_1 \leftarrow R_1 + R_3 \bmod n$	$\star$
$R_0 \leftarrow {R_0}^2 \bmod n$	
$R_0 \leftarrow R_0/2 \bmod n$	
$R_0 \leftarrow R_2 - R_0 \bmod n$	
$j \leftarrow 3$	
$i \leftarrow i - 1$	

$j = 1$ ($d_i = 1$)	
$R_2 \leftarrow R_0 + R_1 \bmod n$	
$R_2 \leftarrow {R_2}^2 \bmod n$	
$R_2 \leftarrow R_2/2 \bmod n$	
$R_2 \leftarrow R_2 - R_3 \bmod n$	
$j \leftarrow 2$	
$i \leftarrow i$	$\star$

$j = 3$ ($d_i = 0$ or 1)	
$R_3 \leftarrow R_3 + R_3 \bmod n$	$\star$
$R_0 \leftarrow {R_0}^2 \bmod n$	
$R_3 \leftarrow R_3/2 \bmod n$	$\star$
$R_1 \leftarrow R_1 - R_3 \bmod n$	$\star$
$j \leftarrow d_i$	
$i \leftarrow i - (1 - d_i)$	[$\star$ if $d_i = 1$]

Figure 2.9: Detailed states of Algorithm 2.17

We also present in Algorithm 2.18 a right-to-left variant of the square-always exponentiation using equation (2.3). This algorithm uses the following matrix:

$$M = \begin{pmatrix} 0 & 0 & 2 & 0 & 0 & 0 & 2 & 1 \\ 2 & 1 & 2 & 2 & 1 & 0 & 1 & 0 \\ 0 & 2 & 1 & 1 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 1 & 2 & 1 & 1 \end{pmatrix}$$

Algorithm 2.18 Right-to-left atomic square-always exponentiation with (2.3)

Input: $m, n \in \mathbb{N}$, $m < n$, $d = (d_{k-1} d_{k-2} \ldots d_0)_2$
Output: $m^d \bmod n$
Uses: R_0, R_1, and R_2

1: $R_0 \leftarrow m$
2: $R_1, R_2 \leftarrow 1$
3: $i, j \leftarrow 0$
4: **while** $i \leq k - 1$ **do**
5: $\quad j \leftarrow d_i(1 + (j \bmod 3))$
6: $\quad R_{M_{j,0}} \leftarrow R_{M_{j,1}} + R_0 \bmod n$
7: $\quad R_{M_{j,2}} \leftarrow R_{M_{j,3}}/2 \bmod n$
8: $\quad R_{M_{j,4}} \leftarrow R_{M_{j,5}} - R_{M_{j,6}} \bmod n$
9: $\quad R_{M_{j,3}} \leftarrow {R_{M_{j,3}}}^2 \bmod n$
10: $\quad i \leftarrow i + M_{j,7}$
11: **return** R_1

$j = 0$ ($d_i = 0$)	
$j \leftarrow 0$	[$\star$ if j was 0]
$R_0 \leftarrow R_0 + R_0 \bmod n$	$\star$
$R_2 \leftarrow R_0/2 \bmod n$	$\star$
$R_0 \leftarrow R_0 - R_2 \bmod n$	$\star$
$R_0 \leftarrow {R_0}^2 \bmod n$	
$i \leftarrow i + 1$	

$j = 2$ ($d_i = 1$)	
$j \leftarrow 2$	
$R_0 \leftarrow R_2 + R_0 \bmod n$	$\star$
$R_1 \leftarrow R_1/2 \bmod n$	
$R_0 \leftarrow R_0 - R_2 \bmod n$	$\star$
$R_1 \leftarrow {R_1}^2 \bmod n$	
$i \leftarrow i$	$\star$

$j = 1$ ($d_i = 1$)	
$j \leftarrow 1$	
$R_2 \leftarrow R_1 + R_0 \bmod n$	
$R_2 \leftarrow R_2/2 \bmod n$	
$R_1 \leftarrow R_0 - R_1 \bmod n$	
$R_2 \leftarrow {R_2}^2 \bmod n$	
$i \leftarrow i$	$\star$

$j = 3$ ($d_i = 1$)	
$j \leftarrow 3$	
$R_0 \leftarrow R_0 + R_0 \bmod n$	$\star$
$R_0 \leftarrow R_0/2 \bmod n$	$\star$
$R_1 \leftarrow R_2 - R_1 \bmod n$	
$R_0 \leftarrow {R_0}^2 \bmod n$	
$i \leftarrow i + 1$	

Figure 2.10: Detailed states of Algorithm 2.18

As for the previous algorithm, the main loop of Algorithm 2.18 has four states. Here, the state $j = 0$ corresponds to the processing of a 0 bit and the sequence $j = 1$, $j = 2$, and $j = 3$ corresponds to the processing of a 1 bit, as detailed below. The state machine is thus similar to the one of Algorithm 2.17, cf. Figure 2.11.

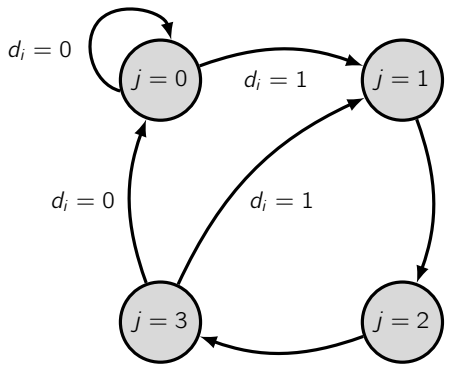


Figure 2.11: State machine of Algorithm 2.17

2.5.1.3 Performance Analysis

Algorithms 2.17 and 2.18 are mostly equivalent in terms of operations realized in a single loop. The number of dummy operations (additions, subtractions and halvings) introduced to fill the atomic blocks are the same in the two versions — we considered that the cost of these operations is negligible compared to multiplications and squarings for operands of thousands of bits. Both algorithms require $2S$ per exponent bit on average or $1.6M$ if $S/M = 0.8$ which represents a theoretical 11.1% speed-up over Algorithm 2.3 which is the fastest known regular algorithm immune to the attack by Amiel

et al. [AFT+09]. Table 2.5 compares the efficiency of the multiply-always, Montgomery ladder, and square-always algorithms when $S = M$ and $S/M = 0.8$.

In addition, our algorithms can be enhanced using the window techniques presented in Section 1.2.1.4 for the scalar multiplication, while the Montgomery ladder cannot. These techniques provide a substantial speed-up over Algorithm 2.4 when extra memory is available. Though we did not investigate this path, we believe that a comparable time-memory trade-off can be expected.

Table 2.5: Comparison of the expected cost per exponent bit of SSCA protected exponentiation algorithms (* multiply-always is not immune to the attack by Amiel et al. [AFT+09])

Algorithm	Cost	$S = M$	$S/M = 0.8$	Nb. regs
Multiply-always* (2.4)	$1.5M$	$1.5M$	$1.5M$	2
Montgomery ladder (2.3)	$1M + 1S$	$2M$	$1.8M$	2
L.-to-r. Square-always (2.17)	$2S$	$2M$	**1.6M**	4
R.-to-l. Square-always (2.18)	$2S$	$2M$	**1.6M**	3

2.5.1.4 Security Considerations

Our algorithms are protected against SSCA by the implementation of the atomicity principle. The analysis by Amiel et al. [AFT+09] cannot apply either since only squarings are involved. As a matter of comparison, notice that the exponent blinding countermeasure does not fundamentally remove the leakage but only renders this attack practically infeasible. Embedded implementations should also be protected against the differential analysis, cf. Section 2.3. This can be achieved using classical DSCA countermeasures, like exponent or modulus randomization.

We recommend implementing Algorithm 2.18 rather than Algorithm 2.17 since left-to-right algorithms are vulnerable to the chosen message SPA and *doubling attack* as discussed in Section 2.2.2, and more subject to combined attacks [AFMV07]. Besides, Algorithm 2.18 requires one less register than Algorithm 2.17.

It is well known that algorithms using dummy operations generally succumb to safe-error attacks, cf. Section 2.4.2. Immunity to C and M safe-errors can be easily obtained by applying the exponent randomization technique, which also prevents DSCA. Nevertheless, special care has been taken in our algorithms to ensure that inducing a fault in any of the dummy operations would produce an erroneous result. For instance, in the following sequence of dummy operations from Algorithm 2.18 ($j = 0$), no operation can be tampered with without corrupting R_0 and thus the result of the exponentiation:

$$R_0 \leftarrow R_0 + R_0 \bmod n$$
$$R_2 \leftarrow R_0/2 \bmod n$$
$$R_0 \leftarrow R_0 - R_2 \bmod n$$

Only operations $i \leftarrow i$ and $j \leftarrow 0$, appearing in some instances of algorithms 2.17 and 2.18 patterns, have not been protected for readability reasons. It is easy to fix these points: perform $i \leftarrow i \pm M_{j,\cdot} + \alpha$ instead of $i \leftarrow i \pm M_{j,\cdot}$ in these algorithms and add a step $i \leftarrow i - \alpha$ in the loop. The $j \leftarrow d_i(1 + \dots)$ operation should be protected in the same manner. In the end, our algorithms are immune to C safe-error attacks.

Further work may focus on implementing on our algorithms the *infective computation* strategy presented by Schmidt et al. [STA+10] in order to counter combined attacks.

2.5.2 Parallelization

It is well known that Montgomery ladder algorithm is well suited for parallelization. It is thus natural to ask if the square-always algorithms have the same property. For example the two squarings needed to perform a classical multiplication using equation (2.3) are independent and can therefore be performed simultaneously. The same strategy applies for equation (2.2).

We believe that the interest of this section extends beyond the scope of embedded systems. Nowadays most computers, and even recent mobile devices, are provided with several processors which enable parallelizing algorithms to speed-up computations.

2.5.2.1 Parallelized Algorithms

We noticed that right-to-left exponentiations are more suited for parallelization than their left-to-right counterpart since more operations are independent. For example in Algorithm 2.2 one can first perform all squarings (step 6), store all values corresponding to a $d_i = 1$, and then perform the remaining multiplications. We present in Algorithm 2.19 a right-to-left square-always algorithm using (2.3) and two parallel squaring blocks (i.e. two 1-operand multipliers). For a better readability, this algorithm is not atomic. In the following, two operations o_1 and o_2 performed simultaneously are denoted $o_1 \mid\mid o_2$.

Algorithm 2.20 is an atomic version of Algorithm 2.19. It requires two extra registers compared to the non atomic one and the following matrices:

$$M = \begin{pmatrix} 1 & 1 & 5 & 6 & 5 & 5 & 5 & 0 & 1 \\ 0 & 6 & 4 & 3 & 0 & 1 & 3 & 1 & 1 \\ 2 & 5 & 3 & 1 & 5 & 5 & 5 & 0 & 0 \\ 2 & 5 & 0 & 6 & 0 & 1 & 5 & 0 & 1 \end{pmatrix} \qquad N = \begin{pmatrix} 1 & 1 & 0 \\ 5 & 2 & 2 \end{pmatrix}$$

It is possible to further enhance the efficiency of these algorithms if more memory is available by storing more free squarings when 1's sequences are processed. This observation yields Algorithm 2.21 which allows the storage of *extramax* simultaneous precomputed squarings using extra registers R_3, R_4, $\dots R_{extramax+2}$. Algorithm 2.21 with $extramax = 1$ is thus equivalent to algorithms 2.19 and 2.20. Though Algorithm 2.21 is not atomic for readability reasons and because of the difficulty to write an atomic algorithm depending on a variable (here *extramax*), it should be possible to write an atomic version for each *extramax* value in the same way as we proceeded for Algorithm 2.19.

Algorithm 2.19 Right-to-left parallel square-always exponentiation with (2.3)

Input: $m, n \in \mathbb{N}$, $m < n$, $d = (d_{k-1} \dots d_0)_2$
Output: $m^d \bmod n$
Uses: 5 k-bit registers a, b, R_0, R_1, R_2

1: $a \leftarrow 1$
2: $b \leftarrow m$
3: $extra \leftarrow 0$
4: **for** $i = 0$ **to** $k - 1$ **do**
5: **if** $d_i = 1$ **then**
6: **if** $extra = 0$ **then**
7: $R_0 \leftarrow (a - b)^2 \bmod n$ || $R_1 \leftarrow b^2 \bmod n$
8: $a \leftarrow (a + b)^2 \bmod n$ || $R_2 \leftarrow {R_1}^2 \bmod n$
9: $a \leftarrow (a - R_0)/4 \bmod n$
10: $b \leftarrow R_1$
11: $R_1 \leftarrow R_2$
12: $extra \leftarrow 1$
13: **else**
14: $R_0 \leftarrow (a - b)^2 \bmod n$ || $a \leftarrow (a + b)^2 \bmod n$
15: $a \leftarrow (a - R_0)/4 \bmod n$
16: $b \leftarrow R_1$
17: $extra \leftarrow 0$
18: **else**
19: **if** $extra = 0$ **then**
20: $b \leftarrow b^2 \bmod n$
21: **else**
22: $b \leftarrow R_1$
23: $extra \leftarrow 0$
24: **return** a

Algorithm 2.20 Right-to-left atomic parallel square-always exp. with (2.3)

Input: $m, n \in \mathbb{N}$, $m < n$, $d = (d_{k-1} d_{k-2} \dots d_0)_2$
Output: $m^d \bmod n$
Uses: 7 k-bit registers R_0 to R_6

1: $R_0 \leftarrow 1$
2: $R_1 \leftarrow m$
3: $v \leftarrow (0, 0, 0)$ ▷ v_0 is i and v_1 is *extra* from Alg. 2.19
4: $u \leftarrow 1$
5: **while** $v_0 \leq k - 1$ **do**
6: $j \leftarrow d_{v_0}(v_1 + u + 1)$
7: $R_5 \leftarrow (R_0 - R_1)/2 \bmod n$
8: $R_6 \leftarrow (R_0 + R_1)/2 \bmod n$
9: $R_{M_{j,0}} \leftarrow {R_{M_{j,1}}}^2 \bmod n \quad || \quad R_{M_{j,2}} \leftarrow {R_{M_{j,3}}}^2 \bmod n$
10: $R_{M_{j,4}} \leftarrow R_0 - R_2 \bmod n$
11: $R_{M_{j,5}} \leftarrow R_3$
12: $R_{M_{j,6}} \leftarrow R_4$
13: $v_1 \leftarrow M_{j,7}$
14: $u \leftarrow M_{j,8}$
15: $t \leftarrow 1 - v_1(1 - d_{v_0+1})$
16: $R_{N_{t,0}} \leftarrow R_3$
17: $v_{N_{t,1}} \leftarrow 0$
18: $v_{N_{t,2}} \leftarrow v_{N_{t,2}} + 1$
19: $v_0 \leftarrow v_0 + u$
20: **return** R_0

Algorithm 2.21 Right-to-left generalized parallel square-always exp. with (2.3)

Input: $m, n \in \mathbb{N}$, $m < n$, $d = (d_{k-1} d_{k-2} \dots d_0)_2$, $extramax \in \mathbb{N}^*$
Output: $m^d \bmod n$
Uses: $extramax+4$ k-bit registers a, R_0, R_1, $\dots R_{extramax+2}$

1: $a \leftarrow 1$
2: $R_1 \leftarrow m$
3: $extra \leftarrow 0$
4: **for** $i = 0$ **to** $k-1$ **do**
5: **if** $d_i = 1$ **then**
6: **if** $extra < extramax$ **then**
7: $R_0 \leftarrow (a - R_1)^2 \bmod n$ || $R_{extra+2} \leftarrow {R_{extra+1}}^2 \bmod n$
8: $a \leftarrow (a + R_1)^2 \bmod n$ || $R_{extra+3} \leftarrow {R_{extra+2}}^2 \bmod n$
9: $a \leftarrow (a - R_0)/4 \bmod n$
10: $(R_1, R_2, \dots R_{extramax+1}) \leftarrow (R_2, R_3, \dots R_{extramax+2})$
11: $extra \leftarrow extra+1$
12: **else**
13: $R_0 \leftarrow (a - R_1)^2 \bmod n$ || $a \leftarrow (a + R_1)^2 \bmod n$
14: $a \leftarrow (a - R_0)/4 \bmod n$
15: $(R_1, R_2, \dots R_{extramax+1}) \leftarrow (R_2, R_3, \dots R_{extramax+2})$
16: $extra \leftarrow extra-1$
17: **else**
18: **if** $extra = 0$ **then**
19: $R_1 \leftarrow {R_1}^2 \bmod n$
20: **else**
21: $(R_1, R_2, \dots R_{extramax+1}) \leftarrow (R_2, R_3, \dots R_{extramax+2})$
22: $extra \leftarrow extra-1$
23: **return** a

Remark 15. Notice that multiple assignments of steps 10, 15, and 21 may be traded for a cheap index increment if registers R_1, R_2, ..., $R_{extramax+2}$ are managed as a cyclic buffer.

2.5.2.2 Cost of Parallelized Algorithms

We demonstrate hereafter that, as the length of the exponent tends to infinity, the cost per exponent bit of Algorithm 2.21 tends to:

$$\left(1 + \frac{1}{4\,extramax+2}\right) S$$

It yields a cost of $7S/6$ for algorithms 2.19, 2.20, and 2.21 with $extramax = 1$, $11S/10$ for $extramax = 2$, $15S/14$ for $extramax = 3$, etc. The difference between this limit and costs actually observed in our simulations is negligible for 1024-bit or longer exponents.

It is remarkable that if $S/M = 0.8$ these costs become respectively $0.93M$, $0.88M$, $0.86M$, etc. per exponent bit. We believe that such performances cannot be achieved by binary algorithms using two parallelized 2-operands multiplication blocks. Indeed at least k multiplications have to be performed sequentially, which requires at least $1M$ per exponent bit. Moreover when *extramax* tends to infinity, the cost of Algorithm 2.21 tends to $1S$, which we believe to be the optimal cost of an exponentiation algorithm based on the binary decomposition of the exponent since k squarings at least have to be performed sequentially.

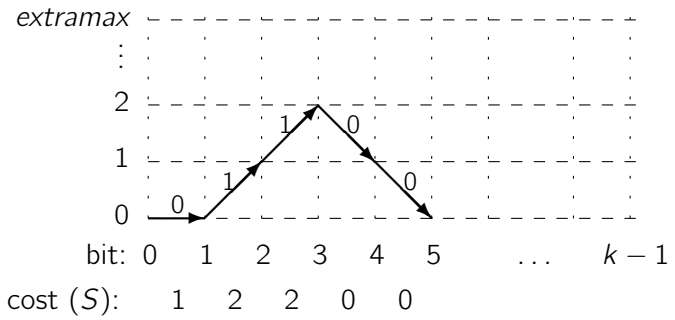
Table 2.6 summarizes the theoretical cost of parallelized algorithms cited in this study.

Table 2.6: Comparison of the expected cost per exponent bit of parallelized exponentiation algorithms (exponent length tending to infinity)

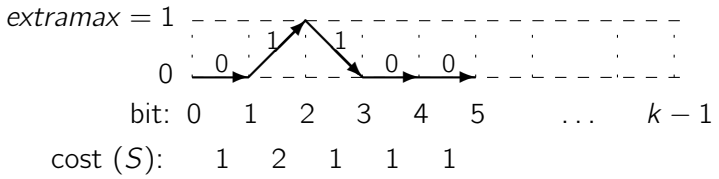
Algorithm	General cost	$S/M = 1$	$S/M = 0.8$
Parallelized Montgomery ladder	$1M$	$1M$	$1M$
Alg. 2.19, 2.20, and 2.21 with *extramax* $= 1$	$7S/6$	$1.17M$	**0.93M**
Alg. 2.21 with *extramax* $= 2$	$11S/10$	$1.10M$	**0.88M**
Alg. 2.21 with *extramax* $= 3$	$15S/14$	$1.07M$	**0.86M**
$\vdots$	$\vdots$	$\vdots$	$\vdots$
Alg. 2.21 with *extramax* $\to \infty$	$1S$	$1M$	**0.8M**

Proof. We first recall the principle of Algorithm 2.21: since 3 squarings are required to process a 1 bit, a fourth squaring slot is available at the same cost ($2S$). Thus, the algorithm scans the exponent from the right to the left and computes one squaring in advance at each 1 bit ($\nearrow$ in the following), within the limit of *extramax*. Then, as 0's are processed, the free squarings are consumed ($\searrow$) at null cost ($0S$). Two other cases may happen: first, a 1 bit can be processed but *extramax* squarings are already stored in registers, then one free squaring is consumed ($\searrow$) and $1S$ is enough to perform the two other squarings. Second, a 0 bit can be processed with no free squaring in registers (*extra* $= 0$). Only in this latter case one squaring is performed at the cost of $1S$ and the parallel squaring slot is wasted ($\rightarrow$).

We can consider the evolution of *extra* as exponent bits are processed using a diagram as below. For example, we have represented here the evolution of *extra* for the 5 first bits of an exponent $d = (d_{k-1} \ldots 00110)_2$ with *extramax* ≥ 2. The cost of the first 0 bit is $1S$ since *extra* $= 0$ at the beginning of the exponentiation, the cost of two next 1 bits is $2S$ each and *extra* is incremented, finally the two last 0 bits have cost $0S$ and *extra* is decremented. The total cost of the 5 bits is $5S$.



Observe now that the same bits have a higher cost if $extramax = 1$: as previously the two first bits 01 cost $1S$ and $2S$ respectively. However, the next 1 bit cannot lead to the computation of a second free squaring since $extramax = 1$. So the bit is processed at the cost of $1S$ and the free squaring is lost. Finally, the two last 0's cost $1S$ each since no free squaring is stored anymore. The cost of the sequence is $6S$.



For a given exponent and *extramax*, let's call a *c-cycle* a sequence of bits starting with $extra = c$, ending with $extra = c$, and inside which $extra > c$. In particular, we can decompose any exponent as a sequence of 0-cycles, except that the last one may be unterminated with $extra > 0$.

Then, let B_c^e stand for the expected number of bits of a c-cycle when $extramax = e$ and C_c^e its expected cost.

- $extramax = 1$ For a random exponent and $extramax = 1$, a 0-cycle is "0" with probability $1/2$ and "1x", $x \in \{0, 1\}$ otherwise. The cost of a 0-cycle "0" is $1S$ and the cost of a 0-cycle "1x" is $2S$ if $x = 0$ which happens with probability $1/2$, or $3S$ if $x = 1$.

$$B_0^1 = 1/2 \times 1 + 1/2 \times 2 = 3/2$$

$$C_0^1 = 1/2 \times 1S + 1/2 \times (1/2 \times 2S + 1/2 \times 3S) = 7S/4$$

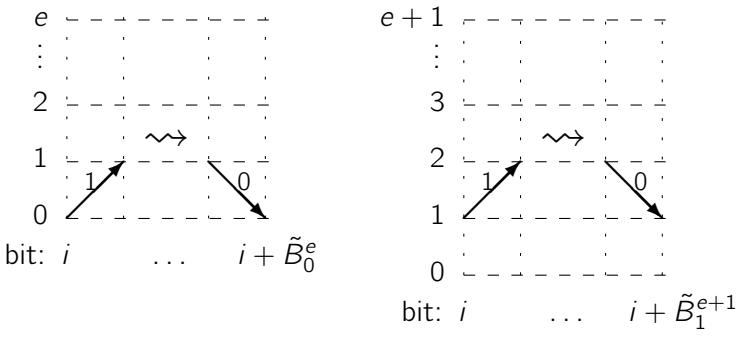
The expected cost of a 0-cycle with $extramax = 1$ is then $C_0^1/B_0^1 = 7S/6$ per bit. As the length of the exponent tends to infinity, the contribution of the possibly unterminated last 0-cycle becomes negligible. Therefore the cost per bit of a random exponent tends to the cost per bit of a 0-cycle as its length tends to infinity. So we can approximate the cost of algorithms 2.19, 2.20 and 2.21 to $7S/6$ for exponents of thousands of bits.

- *extramax* $= e > 1$ A 0-cycle starts with a 0 with probability 1/2 and with a 1 otherwise. In the first case its cost is $1S$ as previously. Let $\tilde{B}_c^e$, respectively $\tilde{C}_c^e$, denote the expected length, respectively the expected cost, of a c-cycle starting with a 1 bit when *extramax* $= e$.

$$B_0^e = 1/2 \times 1 + 1/2 \times \tilde{B}_0^e \tag{2.4}$$

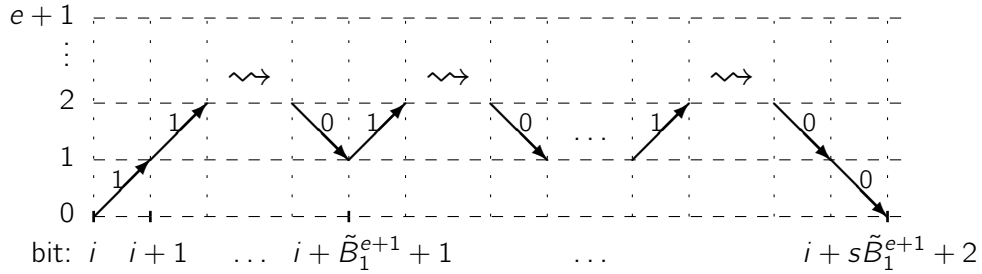
$$C_0^e = 1/2 \times 1S + 1/2 \times \tilde{C}_0^e \tag{2.5}$$

First we demonstrate that $\tilde{B}_0^e = 2e$. As depicted below, one can observe that $\tilde{B}_0^e = \tilde{B}_1^{e+1}$.



As depicted hereafter, the length $\tilde{B}_0^{e+1}$ of a 0-cycle with *extramax* $= e + 1$ and starting by a 1 bit is $s\tilde{B}_1^{e+1} + 2$ where s is the number of inner 1-cycles starting by a 1 bit. Note also that $s = i$ with probability $2^{-(i+1)}$, which gives:

$$\tilde{B}_0^{e+1} = 2 + \sum_{i=0}^{\infty} \frac{i\tilde{B}_1^{e+1}}{2^{(i+1)}} = 2 + \tilde{B}_1^{e+1} = 2 + \tilde{B}_0^e$$



$(\tilde{B}_0^e)_{e\geq 1}$ is thus an arithmetic progression with common difference 2 and $\tilde{B}_0^1 = 2$. This yields $\tilde{B}_0^e = 2e$.

In a same manner, we can observe that:

$$\tilde{C}_0^{e+1} = 2S + \sum_{i=0}^{\infty} \frac{i\tilde{C}_1^{e+1}}{2^{(i+1)}} = 2S + \tilde{C}_1^{e+1} = 2S + \tilde{C}_0^e$$

Since $\tilde{C}_0^1 = 5S/2$ we obtain that $\tilde{C}_0^e = (1/2 + 2e)S$.

Using the above results in equations (2.4) and (2.5), we obtain finally:

$$B_0^e = 1/2 \times 1 + 1/2 \times 2e = 1/2 + e$$

$$\text{and} \quad C_0^e = 1/2 \times 1S + 1/2 \times (1/2 + 2e)S = (3/4 + e)S$$

The expectation of the cost per bit of a 0-cycle is then:

$$\frac{C_0^e}{B_0^e} = \left(\frac{3/4 + e}{1/2 + e}\right) S = \left(1 + \frac{1}{4e + 2}\right) S$$

Therefore the expectation of the cost of Algorithm 2.21 with $extramax = e$ tends to $(1 + \frac{1}{4e+2})S$ as the length of the exponent tends to infinity. □

2.5.3 Practical Results

We briefly present below practical implementation results of the non-parallelized square-always algorithm. As discussed previously we focus on the right-to-left version.

We implemented this algorithm and Montgomery ladder on an Atmel AT90SC smart card chip. This component is provided with an 8-bit AVR core and the AdvX coprocessor dedicated to long-integer arithmetic. We used Barrett reduction [Bar87] to implement modular arithmetic.

We present in Table 2.7 the memory (code and RAM) and timing figures obtained with the chip and the AdvX running at 30 MHz. The observed speed-up of the square-always algorithm over the Montgomery ladder is 5% on average. This is less than the predicted 11% but the difference can be explained by the neglected operations of the atomic pattern. Keep in mind that such results highly depend on the considered device and its hardware capabilities.

Table 2.7: On-chip comparison of Montgomery ladder and square-always

Algorithm	Key length (b)	Code (B)	RAM (B)	Timing (ms)
Montgomery ladder (Alg. 2.3)	512	360	128	30
	1024	360	256	200
	2048	360	512	1840
Square-always (Alg. 2.18)	512	510	192	28
	1024	510	384	190
	2048	510	768	1740

We performed careful SPA on both implementations and observed no leakage on power traces.

2.6 Horizontal Correlation Analysis on Exponentiation

In this section, we introduce a correlation analysis using only one execution side-channel trace during an RSA exponentiation to recover the whole secret exponent manipulated by a chip. It uses the fact that long-integer multiplications are performed using multi-precision algorithms, and consequently that the side-channel leakage of only one of them can provide enough information to an adversary to mount a correlation analysis. Therefore, similarly to the Big Mac attack, longer keys facilitate this attack and its success depends on the arithmetic coprocessor characteristics.

Our technique uses a single exponentiation trace and thus cannot be prevented by exponent blinding. Moreover, contrarily to the Big Mac attack, it applies even in the case of regular implementations such as the square-and-multiply-always and Joye or Montgomery ladders. We also point out that DSA and Diffie-Hellman exponentiations are no more immune against the differential analysis. This research has been published [CFG+10] and led to patent registration [CFG+11f; CFG+11e].

In the following, we introduce the idea of our attack in Section 2.6.1. Section 2.6.2 presents its theory and evaluates the efficiency of classical countermeasures. Some practical successful results on an embedded device are given in Section 2.6.3, and the consequences of our work for implementation of common cryptosystems is discussed in Section 2.6.4.

2.6.1 Introduction to the Horizontal Analysis

Our *horizontal* analysis is inspired by many prior works. First of all, in their seminal paper [MDS99b], Messerges et al. were the first to use cross-correlation techniques between trace segments to identify the performed operations. However, their experiments were not successful. Then, as already stated, the Big Mac attack is the first successful attack using trace segments where known operations are used to identify others. Finally, a key observation from Amiel et al. [AFV07] aroused our curiosity.

In this latter paper, a CPA is performed to recover the secret exponent of public-key implementations. In accordance with what one may expect, their practical results show that the number of traces necessary for an attack is much lower compared to DPA: less than one hundred traces are sufficient. More interestingly, Amiel et al. observe that the correlation is highest when computed on t bits, t being the bit size of the device multiplier. They show the details [AFV07, Fig. 8] of the correlation factor obtained for every multiplicand t-bit word x_i during the squaring operation $x \times x$ using a hardware multiplier. It is worthwhile noticing that a correlation peak occurs for $\mathrm{HW}(x_i)$ each time a word x_i is involved in a multiplication $x_i \times x_j$.

The horizontal correlation analysis presented in the next section takes advantage of this observation.

Horizontal vs. Vertical Analysis We refer to the techniques analyzing time samples corresponding to a same moment in several execution traces as *vertical* side-channel analysis. Such analysis is depicted in Figure 2.12 where an operation starting at instant

t and of length ω is targeted on N execution traces. The classical DPA and CPA techniques thus fall into this category. We also categorize here the various collision analyses presented in Section 2.2.2.1 since two traces at least are needed in these attacks. Vertical analysis is classically circumvented using arithmetic blinding countermeasures, such as exponent/scalar randomization which breaks the basic requirement of these attacks that several executions with a fixed exponent/scalar may be monitored.

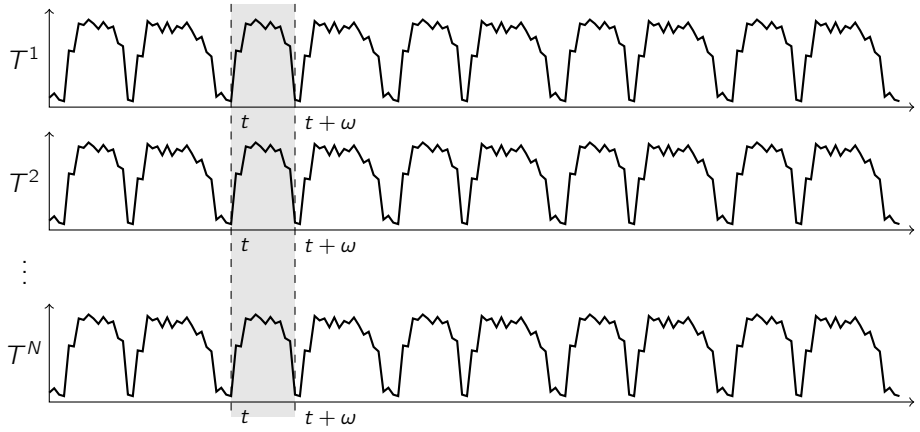


Figure 2.12: Vertical side-channel analysis on exponentiation

We call *horizontal* side-channel analysis the attack techniques using a single trace. In this respect, the first known horizontal power analysis is the original SSCA. Single trace cross-correlation and Big Mac attacks are also horizontal techniques. Our attack computes the correlation factor between many trace segments extracted from within a single consumption/radiation trace as depicted in Figure 2.13, therefore we call it *horizontal correlation analysis*. It contrasts with classical DSCA which targets a particular instant of the execution in several traces.

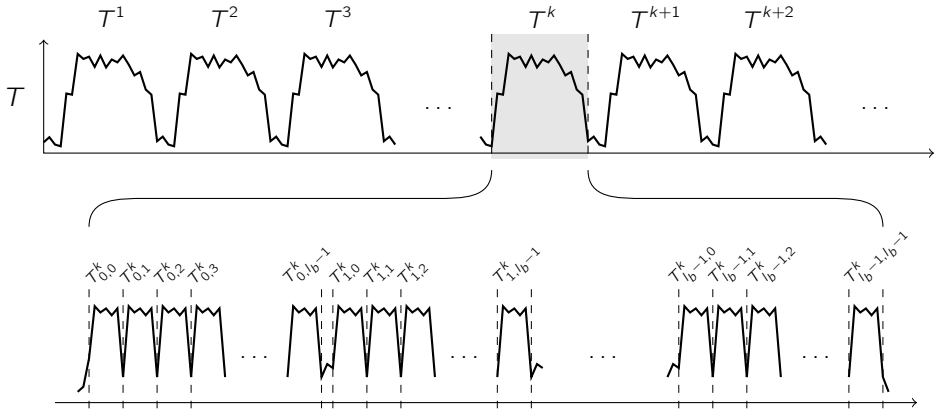


Figure 2.13: Horizontal side-channel analysis on exponentiation

Remark 16. The exponent/scalar blinding countermeasure is not efficient against horizontal attacks since, once recovered, blinded exponents/scalars may be used by an adversary to sign or decrypt messages.

2.6.2 Horizontal Correlation Analysis

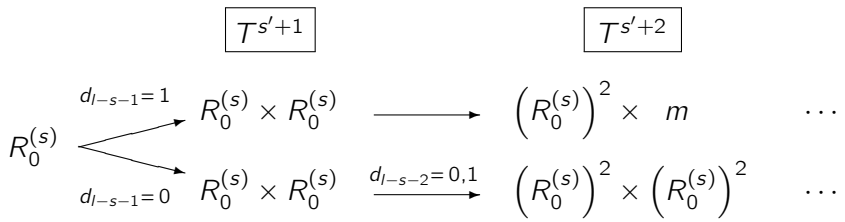
We present hereafter our attack on an atomically protected RSA exponentiation where the modular arithmetic is performed using the schoolbook multi-precision multiplication and a distinct modular reduction such as the Montgomery method presented in Section 2.1.2.

2.6.2.1 Recovering a Secret Exponent Using a Single Known Message Exponentiation Trace

As in vertical DPA and CPA on modular exponentiation, the horizontal correlation analysis reveals the bits of the private exponent d one after another. Considering an exponentiation implemented using Algorithm 2.4, each exponent bit d_i is recovered by determining whether the processing of this bit involves a multiplication by m or not. The difference with classical vertical analysis lies in the way we build and check our hypothesis test.

Given an exponentiation trace T, let T^k denote the portion of the waveform corresponding to the k-th long-integer multiplication. Computing $x \times y$ using Algorithm 2.5 requires ${l_b}^2$ t-bit multiplier calls. We denote $T^k_{i,j}$ the trace segment corresponding to the internal multiplication $x_i \times y_j$ in T^k. As illustrated in Figure 2.13, these ${l_b}^2$ segments are generally easy to identify by an attacker on a leakage trace with some knowledge of the underlying hardware, in particular the bitsize t and the implemented multi-precision multiplication algorithm.

Assuming that the first s bits $d_{l-1}d_{l-2}\ldots d_{l-s}$ of the exponent are already known, an adversary is able to compute the value $R_0^{(s)}$ of the accumulator in Algorithm 2.4 after processing the s-th bit. Observe that the processing of the first s bits corresponds to the first s' long-integer multiplications with $s' = s + \mathrm{HW}(d_{l-1}d_{l-2}\ldots d_{l-s})$ since 0 bits involve a single loop iteration, 1 bits require two loop iterations, and each loop iteration performs a multiplication. Therefore, the value of the unknown $(s+1)$-th exponent bit is 1 if and only if the $(s'+2)$-th long-integer multiplication is $\left(R_0^{(s)}\right)^2 \times m$ as illustrated below.



At this point there are several ways of determining whether the multiplication by m is performed or not.

First, one may show that the series of consumption values in the set of ${l_b}^2$ trace segments is consistent with the series of operand values m_j presumably involved in each of these segments. To this purpose the attacker simply computes the correlation factor

between the series of Hamming weights $\mathrm{HW}(m_j)$ and the series of trace segments $T_{i,j}^{s'+2}$ — i.e. taking $D = m_j$ and $R = 0$ in the leakage model formula (2.1). In other words the trace segments are used as they would be in a vertical analysis if they were independent aligned traces. A correlation peak reveals that m is actually handled in this long-integer multiplication, and consequently that $d_{l-s-1} = 1$.

Alternatively, the correlation factor can be computed between the traces segments and the intermediate results of each t-bit multiplication $x_i \times y_j$, cf. Algorithm 2.5, with $x = R_0^{(s)}$ and $y = m$, or in other words take $D = R_{0,i}^{(s)} \times m_j$, where $R_{0,i}^{(s)}$ denotes the i-th word of $R_0^{(s)}$. This method may also be appropriate since the words of the result are written in registers at the end of the operation. In this case ${l_b}^2$ different values are available for correlating the trace segments instead of l_b previously. This diversity of data may be necessary for the success of the attack when l_b is small. Note that other intermediate values may also lead to better results depending on the hardware leakages.

Another method consists in using trace segments $T_{i,j}^{s'+3}$ of the next long-integer multiplication and estimating their correlation with the Hamming weight of the words x_i of the product $x = \left(R_0^{(s)}\right)^2 \times m$. Indeed, if the $(s'+2)$-th operation is a multiplication by m, then $R_0^{(s+1)} = x$ and the $(s'+3)$-th operation is a squaring $\left(R_0^{(s+1)}\right)^2$, manipulating the words of x.

As pointed out by Walter with the Big Mac attack [Wal01], the longer the integer manipulated and the smaller the size t of the multiplier, the larger the number ${l_b}^2$ of trace segments. Thus longer keys are more at risk with respect to horizontal analysis. For instance in an RSA 2048 bit encryption, if the long-integer multiplication is implemented using a 32-bit multiplier we obtain $(2048/32)^2 = 4096$ segments $T_{i,j}^k$ per trace T^k. Table 2.8 shows examples of values for l_b and ${l_b}^2$ for different key lengths l and multiplier sizes t. Considering that 500 trace samples may be enough to perform a correlation analysis with an average SNR — see the next section for practical results —, many implementations may be subject to this attack.

Table 2.8: Examples of values l, t (in bits), and l_b together with the number of available segments ${l_b}^2$ for horizontal correlation analysis

Key length l	Mult. size t	l_b	${l_b}^2$
2048	64	32	1024
2048	32	64	4096
1024	64	16	256
1024	32	32	1024
1024	16	64	4096
512	32	16	256
512	16	32	1024

2.6.2.2 Comparing Horizontal and Big Mac Attacks

Let us now compare the horizontal correlation analysis on exponentiation with the Big Mac attack which is another generic horizontal analysis circumventing exponent randomization.

Since the Big Mac attack deals with long-integer multiplication templates, it aims at differentiating multiplications from squarings, and fails on regular algorithms such as square-and-multiply-always and ladders methods. On the other hand, the horizontal correlation analysis targets the manipulation of intermediate results which is a more generic approach and applies to all kinds of algorithms.

Besides, the limitation of the Big Mac analysis — its ignorance of the intermediate results — is precisely the cause of its noticeable property to be applicable also when the base of the exponentiation is not known to the attacker. The Big Mac attack thus applies when the message is randomized and/or in the case of a CRT implementation of RSA. While the horizontal correlation technique does not intrinsically deal with message randomization, we explain in the next section how to break those protected implementations when the random bit-length is not sufficiently large.

2.6.2.3 Horizontal Analysis on a Blinded Exponentiation

To protect public-key implementations from differential analysis, developers usually include blinding countermeasures in their cryptographic codes, cf. Section 2.3.2. The most popular ones on RSA exponentiation are:

- additive randomization of the message/ciphertext: $m' = m + rn \bmod kn$ with $k \in [2^{\alpha-1}, 2^{\alpha} - 1]$ and r being an α-bit random value,
- multiplicative randomization of the message/ciphertext: $m' = r^e m \bmod n$ with r being an α-bit random value and e the public exponent,
- additive randomization of the exponent: $d' = d + r\varphi(n)$ with r being a random value and $\varphi(n) = (p-1)(q-1)$.

All these countermeasures prevent classical vertical side-channel analysis but efficiency is penalized as the exponent and modulus are extended by the randomization processes.

Guessing the Blinded Message Let us now consider an implementation using both an SSCA resistant algorithm, the (additive or multiplicative) message randomization method, and the secret exponent blinding. It is generally assumed in the literature that such an implementation applies state-of-the-art countermeasures. We analyze its security with regard to the horizontal correlation analysis.

As previously discussed, exponent blinding has no effect since we analyze a single trace and recovering d' is considered to be successful for an adversary. Then, assuming that the entropy of the message blinding random value r is α bits, there are 2^{α} possible values for m' knowing m and n. Attacking this implementation first requires to recover the value of r. This is achieved by performing a horizontal correlation analysis for each

possible r on the very first multiplication which computes m'. Since this multiplication is necessarily computed, the value of r is retrieved as the one showing a correlation peak — or the highest correlation peak. Once r is known by the adversary, the randomized message m' is revealed and a classical horizontal analysis can be conducted to recover the blinded exponent d' similarly to the non blinded case, except that m' should be used instead of m. Consequently, the entropy of r must be large enough (e.g. $\alpha \geq 32$) to make the number of guess unaffordable or to make it difficult in practice to distinguish the correct one.

2.6.3 Practical Results

This section presents the successful experiments we conducted to demonstrate the efficiency of the horizontal correlation analysis technique. We use here a 16-bit RISC microprocessor on which we implemented a software schoolbook long-integer multiplication based on a 16×16 bits multiplier to simulate the behavior of a coprocessor. We aim at correlating a single long-integer multiplication with one or both operands manipulated — i.e. y_j or $x_i \times y_j$.

The measurement bench is composed of a Lecroy Wavepro oscilloscope, and homemade software and electronic cards are used to acquire the power traces and process the attacks.

We first performed a classical vertical correlation analysis to characterize our implementation and measurement bench, and to validate the correlation model; then we proceeded with the horizontal correlation analysis previously described. Figure 2.14 shows a power trace segment corresponding to the beginning of a long-integer multiplication, where we identify the single-precisions multiplications using vertical lines.

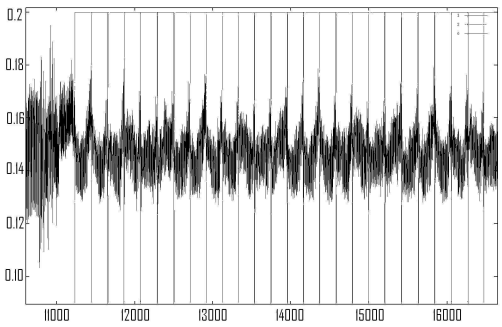


Figure 2.14: Beginning of a long-integer multiplication power trace, vertical lines delineate each $T^k_{i,j}$

2.6.3.1 Vertical Correlation Analysis

Considering an operation $x \times y$, this analysis succeeds in two cases. We obtain correlation peaks by computing the correlation between a trace segment $T_{i,j}$ and operands values x_i and y_j, or value of the product $x_i \times y_j$. Figures 2.15 and 2.16 show the correlation traces obtained in both cases with 500 power consumption traces.

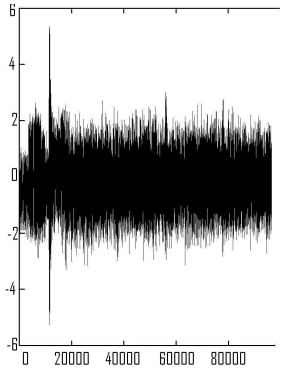


Figure 2.15: Vertical CPA on value y_j

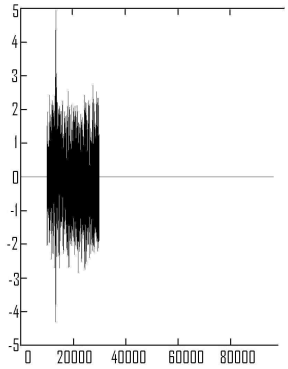


Figure 2.16: Vertical CPA on value $x_i \times y_j$

These results suggest that a horizontal correlation analysis can be performed, as explained previously, either using operand values y_i or using product values $x_i \times y_j$ for estimating the correlation with segment traces of the long-integer multiplication.

2.6.3.2 Horizontal Correlation Analysis

We present results of our attack on a 512-bit long-integer multiplication. This yields 1024 trace segments $T^k_{i,j}$ of 16-bit multiplications to mount the analysis, which should be enough for the success of our attack regarding the vertical analysis results. From the single acquired power trace, we processed the signal in order to detect the set of cycles corresponding to each 16-bit multiplication $x_i \times y_j$ and divided the single power trace in 1024 segments $T^k_{i,j}$ as depicted in Figure 2.14.

Then we performed a horizontal correlation analysis as explained in Section 2.6.2 for the two options $D = R^{(s)}_{0,i} \times m_j$ and $D = m_j$. In both cases, the operation executed is successfully identified as shown in figures 2.17 and 2.18. In each figure, the gray trace shows a greater correlation than the black one and corresponds to the correct guess on the operands.

Our results show that the horizontal correlation analysis can be used to recover a secret exponent using a single exponentiation trace when the input message is known. Although the attack is tested on a software implementation, results obtained by Amiel et al. [AFV07, Fig. 8] prove that correlation techniques are efficient on hardware co-processors (with multiplier size larger than or equal to 16 bits), and enable the attacker to locate each single-precision multiplication involved in a long-integer multiplication. We thus believe that our attack also threatens exponentiation implementations using hardware coprocessors.

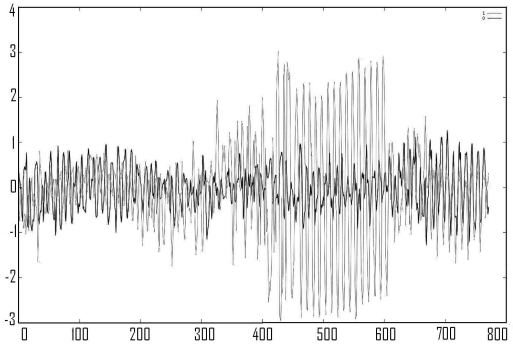


Figure 2.17: Horizontal correlation analysis on value $a_i \times m_j$.

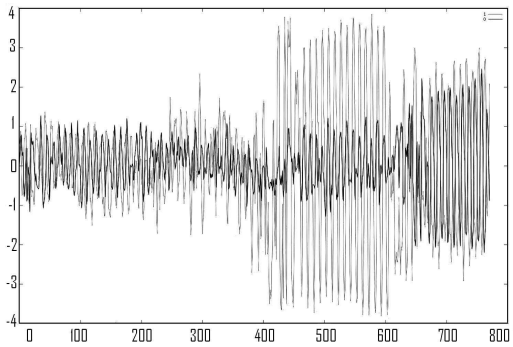


Figure 2.18: Horizontal correlation analysis on value m_j.

It should also be noted that all SSCA resistant algorithms that can be used to implement the exponentiation — either those protected with atomicity principle or regular ones as square-and-multiply-always and Montgomery ladder — may be threatened by the horizontal analysis. Therefore, we recommend to implement a resistant and efficient blinding method on the data manipulated, for instance by using additive message randomization with random values larger than or equal to 32 bits.

2.6.4 Concerns for Common Cryptosystems

We presented our analysis on straightforward implementations of the RSA signature and decryption algorithms which essentially consists of an exponentiation with the secret exponent. In the case of an RSA exponentiation using the CRT method our technique cannot be applied since the operations are performed modulo p and q which are unknown to an adversary. On the other hand, DSA and Diffie-Hellman exponentiations were until now considered immune to DSCA because the exponents are chosen at random before each execution. Although this naturally protects these cryptosystems from vertical analysis, horizontal correlation analysis can recover a secret exponent using a single side-channel trace. Thus, DSA and Diffie-Hellman exponentiations are prone to this attack and additional countermeasures must be used in embedded implementations.

It is worthwhile noticing that ECC cryptosystems are theoretically also concerned by horizontal correlation analysis. However, since key lengths are considerably shorter very few traces per scalar multiplication will be available for the attack. Besides, scalar multiplication involves point doublings and point additions instead of field multiplications and squarings. Each point operation requires about 10 modular multiplications, thus correlation computation could take advantage of all the corresponding trace segments. Nevertheless, a factor of about 10 should not balance the key length reduction which has a quadratic influence on the number of available trace segments.

Our contribution enforces the necessity of using sufficiently large random numbers for blinding in secure implementations and highlights the fact that increasing the key lengths in the next years could improve the efficiency of some side-channel attacks. Our attack threatens implementations which have been considered secure up to now.

Therefore, this new potential risk should then be taken into account when developing embedded products.

Further work could target the use of other values and distinguishers in the horizontal correlation analysis to improve its efficiency. Possible ideas include: using more intermediate values, guessing simultaneously many bits of the secret exponent to increase the number of available traces for the analysis, and using different models like the bivariate one for correlation factor computation.

2.7 Long-Integer Multiplication Blinding and Shuffling

In this section, we address the issue of implementing side-channel analysis countermeasures within the multi-precision multiplication algorithm. It is well known that two different directions can be chosen when preventing differential side-channel analysis: blinding and shuffling sensitive operations [RPD09]. Surprisingly, shuffling arithmetic operations in public-key algorithms has been little studied up to now.

Therefore, we propose new countermeasures for modular multiplication implementation using either both blinding and shuffling, or shuffling only. Although the following techniques were designed to circumvent the horizontal analysis presented in Section 2.6, we believe that they should prevent various kinds of side-channel analysis, such as collision analysis. The content of this section has been partially published [CFG+10] and led to patent registration [CFG+11c; CFG+11a].

Our countermeasures are first presented on the classical schoolbook multiplication algorithm in Section 2.7.1 and then on the modular Montgomery multiplication in Section 2.7.2.

2.7.1 Schoolbook Multiplication

In the following, we first consider using blinding on operands words inside the main loop of the schoolbook multiplication. Then we propose trading partially or completely operands blinding for shuffling, in order to speed-up the multiplication.

2.7.1.1 Operands Blinding

A full blinding countermeasure on the words x_i and y_j consists in replacing in step 5 of Algorithm 2.5 the operation

$$(w_{i+j} + x_i \times y_j) + c$$

by a blinded computation, for instance:

$$(w_{i+j} + (x_i - r_1) \times (y_j - r_2)) + r_1 \times y_j + r_2 \times x_i - r_1 \times r_2 + c$$

with r_1 and r_2 two t-bit random values. For efficiency purposes, the values $r_2 \times x_i$, $r_1 \times y_j$, $r_1 \times r_2$ should be computed once and stored[2].

Such a blinded schoolbook multiplication requires ${l_b}^2 + 2l_b + 1$ single-precision multiplications, i.e. $2l_b + 1$ extra multiplications compared to the non-blinded version. Note also that $2(t + 2l)$ additional bits are required to store the aforementioned precomputed values.

In the following we improve this countermeasure by mixing data blinding and internal loops shuffling.

[2]These precomputations must also be protected from side-channel analysis. For instance, it is possible to shuffle their processing, which allows $(2l_b + 1)!$ different sequences.

2.7.1.2 Shuffling Rows and Blinding Columns

This method consists in randomizing the way the words x_i are processed by the algorithm. In other words, it shuffles the sequence of *rows* in the schoolbook multiplication — i.e. the processing of words x_i in Algorithm 2.5. On the other hand it remains necessary to blind the words of y since *columns* are still processed in the regular sequence. Algorithm 2.22 details this countermeasure.

Algorithm 2.22 Schoolbook long-integer multiplication with rows shuffling and columns blinding

Input: $x = (x_{l_b-1}x_{l_b-2}\dots x_0)_b$, $y = (y_{l_b-1}y_{l_b-2}\dots y_0)_b$
Output: $x \times y$
Uses: $w = (w_{2l_b-1}w_{2l_b-2}\dots w_0)$ and $4t$ extra bits to store r and z

1: Pick at random a permutation vector $\alpha = (\alpha_{l_b-1}\dots\alpha_0)$ in $[0, l_b-1]$
2: $w \leftarrow (00\dots0)$
3: **for** $h = 0$ **to** $l_b - 1$ **do**
4: $\quad i \leftarrow \alpha_h$
5: $\quad$ Pick at random r in $[1, b-1]$
6: $\quad z \leftarrow r \times x_i$
7: $\quad c \leftarrow 0$
8: $\quad$ **for** $j = 0$ **to** $l_b - 1$ **do** ▷ $w \leftarrow w + b^{\alpha_h} x_{\alpha_h} \times y$
9: $\quad\quad (uv)_b \leftarrow w_{i+j} + x_i \times (y_j - r) + c + z$
10: $\quad\quad w_{i+j} \leftarrow v$
11: $\quad\quad c \leftarrow u$
12: $\quad$ **for** $j = l_b$ **to** $2l_b - i - 1$ **do** ▷ carry propagation
13: $\quad\quad (uv)_b \leftarrow w_{i+j} + c$
14: $\quad\quad w_{i+j} \leftarrow v$
15: $\quad\quad c \leftarrow u$
16: **return** w

Considering the horizontal correlation analysis, an attacker has to know precisely the way operands words are manipulated in the algorithm to compute the correlation estimate. Shuffling the processing of words x_i thus protects from an adversary performing a correlation between the series x_i and $T^k_{i,j}$ if the size of the random permutation is big enough to deter an exhaustive search.

The random permutation provides $l_b!$ different sequences for the execution of the loop on multiplication rows. For example, using a 32-bit multiplier, the number of possible execution sequences for a 1024-bit long-integer multiplication amounts to $32! > 2^{117}$ possibilities, which we believe to be more than enough.

Compared to the previous countermeasure, Algorithm 2.22 requires only $l^2 + l$ t-bit multiplications — i.e. l extra multiplications compared to Algorithm 2.5 and $4t$ bits of additional storage. Note also that an additional carry propagation loop is required due to the randomization effect. The cost of this extra processing is not straightforward to evaluate and depends on the implementation. For instance, considering a hardware device, it may induce a negligible timing overhead, but may increase the chip area.

Remark 17. As written above, Algorithm 2.22 is subject to a timing analysis on the carry propagation loop, since the number of iterations depends on i. Observing this number of iterations allows an adversary to recover the random permutation and then apply another analysis. Although we leave the algorithm as is for readability purposes, an implementation should always perform $2l_b$ iterations in the carry propagation loop to withstand timing analysis. This observation holds for the following algorithms as well.

Remark 18. One may argue that in the case of very small l_b values such a countermeasure would not be efficient due to the small number of different random sequences. Remember here that if l_b is very small, the horizontal correlation analysis is not efficient either because of the small number of trace segments.

2.7.1.3 Shuffling Rows and Columns

We propose a variant of the previous countermeasure in which the execution sequences of both internal loops of the algorithm are randomized. This means randomizing both rows and columns of the schoolbook multiplication. The main advantage of this method is that operand words do not have to be blinded anymore to prevent horizontal analysis. It is detailed in Algorithm 2.23.

Algorithm 2.23 Schoolbook long-integer multiplication with rows and columns shuffling

Input: $x = (x_{l_b-1}x_{l_b-2}\dots x_0)_b$, $y = (y_{l_b-1}y_{l_b-2}\dots y_0)_b$
Output: $x \times y$
Uses: $w = (w_{2l_b-1}w_{2l_b-2}\dots w_0)$ and l extra bits to store $c = (c_{l_b-1}c_{l_b-2}\dots c_0)$

1: Pick at random two permutation vectors α, β in $[0, l_b - 1]$
2: $w \leftarrow (00\dots0)$
3: **for** $h = 0$ **to** $l_b - 1$ **do**
4: $\quad i \leftarrow \alpha_h$
5: $\quad c \leftarrow (00\dots0)$
6: $\quad$ **for** $k = 0$ **to** $l_b - 1$ **do** ▷ $w \leftarrow w + b^{\alpha_h} x_{\alpha_h} \times y$
7: $\quad\quad j \leftarrow \beta_k$
8: $\quad\quad (uv)_b \leftarrow w_{i+j} + x_i \times y_j$
9: $\quad\quad w_{i+j} \leftarrow v$
10: $\quad\quad c_j \leftarrow u$
11: $\quad u \leftarrow 0$
12: $\quad$ **for** $j = 1$ **to** l_b **do** ▷ carries addition
13: $\quad\quad (uv)_b \leftarrow w_{i+j} + c_{j-1} + u$
14: $\quad\quad w_{i+j} \leftarrow v$
15: $\quad$ **for** $j = l_b + 1$ **to** $2l_b - i - 1$ **do** ▷ carry propagation
16: $\quad\quad (uv)_b \leftarrow w_{i+j} + u$
17: $\quad\quad w_{i+j} \leftarrow v$
18: **return** w

Using this method, the number of random sequences of ${l_b}^2$ single-precision multiplications is increased to $({l_b}!)^2$. For instance, a 512-bit integer multiplication using a 32-bit multiplier can be processed in $(16!)^2 > 2^{88}$ different ways.

Unlike the two previous countermeasures, Algorithm 2.23 requires no extra single-precision multiplication compared to Algorithm 2.5. On the other hand, it requires an extra vector of l bits for carry propagation, which is more than Algorithm 2.22.

2.7.2 Montgomery Modular Multiplication

Considering that algorithms using interleaved modular multiplication and reduction are often used in practice for memory savings, we present hereafter different implementations of the multiplication rows/columns shuffling on the interleaved Montgomery modular multiplication.

Unfortunately, Montgomery reductions cannot be interleaved in a straightforward manner between rows computations in Algorithms 2.22 and 2.23. Indeed, a reduction step requires the carry propagation to complete before taking place. Thus, all rows have to be computed before starting to reduce the result, which is precisely a non interleaved method.

We propose the following compromise: the long-integer multiplication is divided into σ stages such that $\sigma \mid l_b$, each stage i comprising the computation of rows $x_{i\delta} \times y$ to $x_{i\delta+\delta-1} \times y$ where $0 \leq i < \sigma$, and $\delta = l_b/\sigma$. Subsequently, an interleaved reduction takes place at the end of each stage and the intermediate results are limited to $\delta t + l$ bits. This method is presented in Algorithm 2.24 with the rows shuffling and the columns blinding, and in Algorithm 2.25 with the rows and columns shuffling. The reduction substages are protected as well using blinding, resp. shuffling, in Algorithm 2.24, resp. Algorithm 2.25.

Note that in algorithms 2.24 and 2.25, we do not perform the conditional final subtraction since it should be avoided in practice to prevent timing analysis [Wal02].

These algorithms provide a low-level countermeasure against various side-channel analysis, including horizontal correlation analysis and some SSCA such as collision analysis. However, it is still necessary to apply some other countermeasures such as exponent/scalar blinding when an implementation is exposed to DSCA.

Algorithm 2.24 Interleaved Montgomery modular multiplication with rows shuffling and columns blinding

Input: $x = (x_{l_b-1} x_{l_b-2} \dots x_0)_b$, $y = (y_{l_b-1} y_{l_b-2} \dots y_0)_b$, $n = (n_{l_b-1} n_{l_b-2} \dots n_0)_b$
Output: $xy2^{-l} \bmod n$
Uses: $w = (w_{l_b+\delta} w_{l_b+\delta-1} \dots w_0)_b$ and $4t$ extra bits to store r and z

1: $n'_0 \leftarrow -{n_0}^{-1} \bmod b$
2: $w \leftarrow (00 \dots 0)$
3: **for** $s = 0$ **to** $l_b - 1$ **by** δ **do**
`Multiplication substage`
4: Pick at random a permutation vector $\alpha = (\alpha_{\delta-1} \dots \alpha_0)$ in $[0, \delta - 1]$
5: **for** $h = 0$ **to** $\delta - 1$ **do**
6: $i \leftarrow \alpha_h$
7: Pick at random r in $[1, b - 1]$
8: $z \leftarrow r \times x_{s+i}$
9: $u \leftarrow 0$
10: **for** $j = 0$ **to** $l_b - 1$ **do** ▷ $w \leftarrow w + b^{\alpha_h} x_{\alpha_h} \times y$
11: $(uv)_b \leftarrow w_{i+j} + x_{s+i} \times (y_j - r) + u + z$
12: $w_{i+j} \leftarrow v$
13: **for** $j = l_b$ **to** $l_b + \delta - i - 1$ **do** ▷ carry propagation
14: $(uv)_b \leftarrow w_{i+j} + u$
15: $w_{i+j} \leftarrow v$
`Reduction substage`
16: **for** $i = 0$ **to** $\delta - 1$ **do**
17: $q \leftarrow w_0 \times n'_0 \bmod b$ ▷ $q \leftarrow -w_0/n_0$
18: Pick at random r in $[1, b - 1]$
19: $z \leftarrow (r \ll t) - r$ ▷ $z \leftarrow (b-1)r$
20: $u \leftarrow r$
21: **for** $j = 0$ **to** $l_b - 1$ **do** ▷ $w \leftarrow w + q \times n$
22: $(uv)_b \leftarrow (w_j + z) + q \times n_j + u$
23: $w_j \leftarrow v$
24: **for** $j = l_b$ **to** $l_b + \delta - i - 1$ **do** ▷ carry propagation
25: $(uv)_b \leftarrow (w_j + z) + u$
26: $w_j \leftarrow v$
27: $w_{l_b+\delta-i} \leftarrow u - r$
28: $(w_{l_b+\delta} w_{l_b+\delta-1} \dots w_0)_b \leftarrow (0\, w_{l_b+\delta} w_{l_b+\delta-1} \dots w_1)_b$ ▷ $(w)_b \leftarrow (w)_b \gg 1$
29: **return** w

Algorithm 2.25 Interleaved Montgomery modular multiplication with rows and columns shuffling

Input: $x = (x_{l_b-1}x_{l_b-2}\dots x_0)_b$, $y = (y_{l_b-1}y_{l_b-2}\dots y_0)_b$, $n = (n_{l_b-1}n_{l_b-2}\dots n_0)_b$
Output: $xy2^{-l} \bmod n$
Uses: $w = (w_{l_b+\delta}w_{l_b+\delta-1}\dots w_0)_b$ and l extra bits to store $c = (c_{l_b-1}c_{l_b-2}\dots c_0)$

1: $n'_0 \leftarrow -{n_0}^{-1} \bmod b$
2: $w \leftarrow (00\dots0)$
3: **for** $s = 0$ **to** $l_b - 1$ **by** δ **do**
`Multiplication substage`
4: Pick at random a permutation vector $\alpha = (\alpha_{\delta-1}\dots\alpha_0)$ in $[0, \delta-1]$
5: **for** $h = 0$ **to** $\delta - 1$ **do**
6: $i \leftarrow \alpha_h$
7: Pick at random a permutation vector $\beta = (\beta_{l_b-1}\dots\beta_0)$ in $[0, l_b-1]$
8: $c \leftarrow (00\dots0)$
9: **for** $k = 0$ **to** $l_b - 1$ **do** ▷ $w \leftarrow w + b^{\alpha_h}x_{\alpha_h} \times y$
10: $j \leftarrow \beta_k$
11: $(uv)_b \leftarrow w_{i+j} + x_{s+i} \times y_j$
12: $w_{i+j} \leftarrow v$; $c_j \leftarrow u$
13: $u \leftarrow 0$
14: **for** $j = 1$ **to** l_b **do** ▷ carries addition
15: $(uv)_b \leftarrow w_{i+j} + c_{j-1} + u$
16: $w_{i+j} \leftarrow v$
17: **for** $j = l_b + 1$ **to** $l_b + \delta - i - 1$ **do** ▷ carry propagation
18: $(uv)_b \leftarrow w_{i+j} + u$
19: $w_{i+j} \leftarrow v$
`Reduction substage`
20: **for** $i = 0$ **to** $\delta - 1$ **do**
21: $q \leftarrow w_0 \times n'_0 \bmod b$ ▷ $q \leftarrow -w_0/n_0$
22: Pick at random a permutation vector $\beta = (\beta_{l_b-1}\dots\beta_0)$ in $[0, l_b-1]$
23: $c \leftarrow (00\dots0)$
24: **for** $k = 0$ **to** $l_b - 1$ **do** ▷ $w \leftarrow w + q \times n$
25: $j \leftarrow \beta_k$
26: $(uv)_b \leftarrow w_j + q \times n_j$
27: $w_j \leftarrow v$; $c_j \leftarrow u$
28: $u \leftarrow 0$
29: **for** $j = 1$ **to** l_b **do** ▷ carries addition
30: $(uv)_b \leftarrow w_j + c_{j-1} + u$
31: $w_j \leftarrow v$
32: **for** $j = l_b + 1$ **to** $l_b + \delta - i - 1$ **do** ▷ carry propagation
33: $(uv)_b \leftarrow w_j + u$
34: $w_j \leftarrow v$
35: $w_{l_b+\delta-i} \leftarrow u$
36: $(w_{l_b+\delta}w_{l_b+\delta-1}\dots w_0)_b \leftarrow (0\, w_{l_b+\delta}w_{l_b+\delta-1}\dots w_1)_b$ ▷ $(w)_b \leftarrow (w)_b \gg 1$
37: **return** w

2.8 CoCo (Side-)Channel Analysis on AES

We already mentioned in Section 2.2.2.1 a specific approach for side-channel analysis that consists in using information leakages to detect collisions between data manipulated in algorithms. While the rest of this thesis deals with public-key cryptography, this section focuses on side-channel collision attacks on block ciphers. In the following, we refine some known collision techniques and illustrate our analysis on the widespread AES algorithm. This work has been published [CFG+11b].

A side-channel collision attack against a block cipher is presented by Schramm, Wollinger, and Paar in 2003 [SWP03]. This attack uses differential analysis to exploit collisions in adjacent S-Boxes of the DES algorithm. Schramm, Leander, Felke, and Paar [SLFP04] then propose an attack against AES to detect collisions in the output of the first round `MixColumns`. Later, Bogdanov [Bog07] improved this attack by looking for equal S-Box inputs in several AES executions. He then studied statistical techniques to detect collisions between power traces [Bog08]. Finally, Moradi, Mischke, and Eisenbarth [MME10] proposed a correlation based collision attack to defeat an AES implementation using masked S-Boxes.

In this section, we present two Collision-Correlation (CoCo) attacks on software AES implementations protected against first-order side-channel analysis using masked S-Boxes and practical results on both simulated and real power traces. Our attacks are much more efficient and more generic compared to the one presented by Moradi et al. [MME10]. Moreover we believe our techniques to be applicable to other embedded implementations of symmetric block ciphers.

The remainder of this study is organized as follows: Section 2.8.1 presents the two AES first-order protected implementations targeted by our study. Then in Section 2.8.2 we present our attacks and practical results on simulated power traces and on a physical integrated circuit. Finally, Section 2.8.3 deals with possible countermeasures.

2.8.1 Targeted Implementations

AES is the well-known block cipher algorithm selected in 2001 by the NIST [NIST01], as the candidate designed by Daemen and Rijmen, in order to replace the DES. It is probably the currently most used symmetric encryption algorithm.

For the sake of simplicity, we focus on AES-128 which includes 10 rounds, each one comprising four functions: `AddRoundKey`, `SubBytes`, `ShiftRows`, and `MixColumns`. It encrypts a 128-bit message $M = (m_0, \ldots, m_{15})$ using a 128-bit secret key $K = (k_0, \ldots, k_{15})$ and produces a 128-bit ciphertext $C = (c_0, \ldots, c_{15})$. Note however that the techniques presented hereafter are easily applicable to AES-192 and AES-256.

The only non-linear function in the AES is `SubBytes` (also referred to as the S-Box S in the following) which is a substitution function defined by the pseudo-inversion I in $\mathbb{F}_{2^8}$ and an affine transformation. In this work, we consider the two following solutions that have been proposed to protect this function against first-order attacks.

2.8.1.1 Blinded Lookup Table

The first targeted implementation uses a *masked* substitution table as proposed by Kocher et al. [KJJ98], and Akkar and Giraud [AG01]. This masked table S' is defined by $S'(x_i \oplus u_i) = S(x_i) \oplus v_i$, with u_i (resp. v_i) the mask of the i-th input byte x_i (resp. output byte) of function `SubBytes`, $x_i, y_i, u_i, v_i \in \mathbb{F}_{2^8}$, $0 \leq i \leq 15$. This table is usually computed before the AES execution and stored in volatile memory.

We further consider that the same masks u and v are applied on all S-Boxes during one execution (or a round at least) of the algorithm, i.e. $u_i = u$ and $v_i = v$ for $0 \leq i \leq 15$. We believe that this hypothesis is realistic for embedded security products considering that an expensive recomputation of the 256-byte substitution table S' is necessary for each new pair (u, v) and that the storage of many masked tables is not conceivable in memory constrained devices.

2.8.1.2 Blinded Inversion Calculation

An alternative solution has been proposed by Oswald, Mangard, Pramstaller, and Rijmen [OMPR05] and improved by Canright and Batina [CB08]. It consists in computing the inversion in $\mathbb{F}_{2^8}$ using a multiplicative mask. To do this efficiently it is proposed to decompose the computation using inversions in the subfield $\mathbb{F}_{2^4}$ (and possibly in $\mathbb{F}_{2^2}$). Such a masking method is well suited for hardware implementations.

We recall some properties of the masked inversion. Let I' denote the masked pseudo-inversion such that $I'(x_i \oplus u_i) = I(x_i) \oplus u_i$. The element $x_i \oplus u_i$ in $\mathbb{F}_{2^8}$ is mapped to a pair $(x_{i,h} \oplus u_{i,h}, x_{i,l} \oplus u_{i,l})$ in $\mathbb{F}_{2^4} \times \mathbb{F}_{2^4}$ such that $x_i \oplus u_i = (x_{i,h} \oplus u_{i,h})X + (x_{i,l} \oplus u_{i,l})$, for some X in $\mathbb{F}_{2^8} \setminus \mathbb{F}_{2^4}$. As detailed by Oswald et al. [OMPR05] many calculations occur on these subfield elements to compute the masked inversion of $x_i \oplus u_i$. In these formulas, neither $x_{i,h}$ nor $x_{i,l}$ is directly inverted in $\mathbb{F}_{2^4}$ but the following value:

$$d_i \oplus u_{i,h} = x_{i,h}^2 \times p_0 \oplus (x_{i,h} \times x_{i,l}) \oplus x_{i,l}^2 \oplus u_{i,h} \ ,$$

where p_0 depends on the field polynomial used to define the quadratic extension of $\mathbb{F}_{2^4}$. Then the masked inversion in $\mathbb{F}_{2^4}$ of $d_i \oplus u_{i,h}$ gives $d_i^{-1} \oplus u_{i,h}$ and is used to compute $I'(x_i \oplus u_i)$.

The 16 input bytes of `SubBytes` are blinded using different masks u_i, but one can notice that input and output masks of the inversion stage are identical. Therefore another threat to take into consideration is the zero value power analysis. This technique has been introduced by Golić and Tymen [GT03], and Mangard, Oswald, and Popp [MOP07], and implemented against masked inversion by Moradi et al. [MME10]. It also applies to the improved version of Canright and Batina [CB08] when input and output of the inversion are masked with the same value.

2.8.1.3 Measurements and Validation of Implementations

Trace Acquisition We have developed software implementations on a contact smart card using a 16-bit RISC CPU with low power consumption. Two different methods were used to validate our attacks.

First, we used *simulated traces*: a proprietary tool was used to simulate power traces based on the chip architecture and the code executed. This tool generates ideal power consumption traces without any noise which allows to validate in practice the resistance of an implementation to a set of side-channel attacks leaving aside the acquisition and signal processing problems.

Second, we used *real traces*: we made physical measurements on the chip itself using a MicroPross MP100 reader and a Lecroy WavePro numerical oscilloscope.

First-Order Resistance Validation Since our aim is to present techniques able to defeat first-order protected devices, we performed classical first-order differential and correlation analysis on the two implementations presented above before testing our collision attacks.

We applied DPA and CPA to the `AddRoundKey`, `SubBytes`, and `MixColumns` functions at the first and the last rounds of our implementations. We also performed detailed SPA for each input byte value using many average traces to detect any noticeable (biased) power traces that would reveal a potential leakage: no leakage has been observed. We also verified that both implementations were immune to zero value power analysis and to the attack presented by Moradi et al [MME10].

We have thus verified that, to the best of our knowledge, both considered AES implementations are resistant to known first-order attacks. Nevertheless we present in the next section two new collision techniques which jeopardize these implementations.

2.8.2 Description of our CoCo Attacks

In this section, we present the general principle of CoCo attacks and then detail how it can be applied to the two considered AES implementations.

Collision based analysis is also known as *cross-correlation* attacks [WWM11] and *multiple-differential collision* attacks [Bog08]. We prefer the term *collision-correlation* attacks since cross-correlation may be ambiguous depending on the context, and we deem multiple-differential collision too generic for our method.

2.8.2.1 The CoCo Recipe

The principle of the attacks presented in this section is to detect internal collisions between data processed in blinded S-Boxes on the first round of an AES execution. We demonstrate in the following that if (i) we are able to detect that the same data is processed at instants t_0 and t_1, and (ii) the S-Boxes are blinded such that either the same mask is applied to all message bytes or the mask is identical at the input and the output of each S-Box, then it is possible to infer information on the secret key with very few traces.

In the following, we denote by $(T^n)_{1\leq n\leq N}$ a set of N power traces captured from a device processing N encryptions of the same message M. Then we consider two

instructions[3] whose processing starts at times t_0 and t_1 and denote by ω the number of samples acquired per instruction processing. As depicted in Figure 2.19 we finally consider $\Theta_0 = (T^n_{t_0})_n$ and $\Theta_1 = (T^n_{t_1})_n$ the two series of power consumptions segments at instants t_0 and t_1.

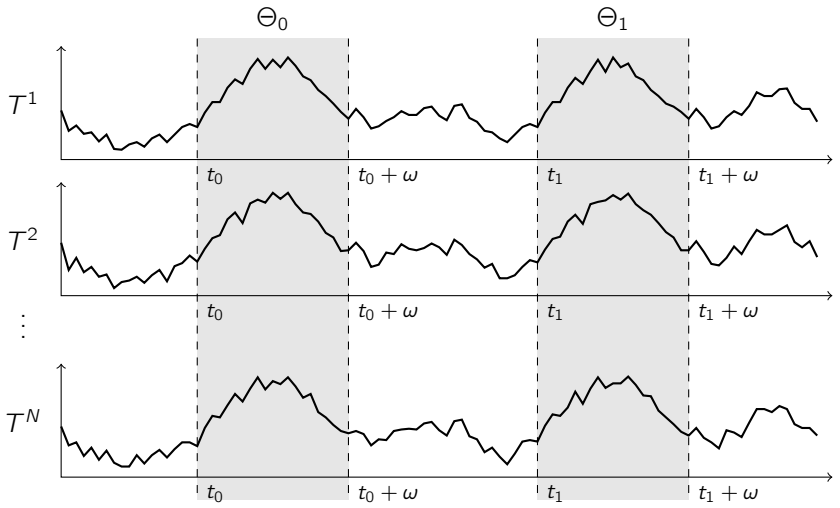


Figure 2.19: Overview of the collision-correlation attack

Note that in practice the N power traces should start at the same instant of the encryption and be perfectly aligned. Such conditions generally require signal processing to be performed first. Note also that as the sampling rate is usually such that $\omega > 1$ points are acquired per instruction, we can generalize the definition of Θ_0 and Θ_1 as being series of ω-sample trace segments instead of series of single power consumption samples.

The final stage of the attack consists in applying a statistical treatment to (Θ_0, Θ_1) in order to identify if the same data was involved in $T^n_{t_0}$ and $T^n_{t_1}$ for $1 \leq n \leq N$. Let $\texttt{Collision}(\Theta_0, \Theta_1)$ denote a decision function returning `true` or `false` depending on whether this property is presumed to be fulfilled or not. Such a decision function would usually compare the value of a synthetic criterion with an experimentally determined threshold. Possible examples of such a criterion include the mean squared difference — the mean being taken over the N traces as well as over the ω samples —, the least squared difference with binary or ternary voting [Bog08], and the Pearson correlation factor.

We used this latter criterion in our study. However, contrary to the classical CPA presented in Section 2.3.1.3, we do not compare a consumption value computed using a theoretical model to an experimental measurement sample. Instead, we estimate the correlation factor between the two series of trace segments Θ_0 and Θ_1 at time offset t

[3] In our attacks we consider only the correlation between two identical instructions, but it may even be possible to detect that two different instructions manipulate identical data, e.g. by spotting a data bus using eletromagnetic analysis.

$(0 \leq t \leq \omega - 1)$ to identify similarities between waveform pairs $T^n_{t_0}$ and $T^n_{t_1}$. Thus the estimation of the correlation factor is expressed as:

$$\hat{\rho}_{\Theta_0,\Theta_1}(t) = \frac{N\sum(T^n_{t_0+t}T^n_{t_1+t}) - \sum T^n_{t_0+t}\sum T^n_{t_1+t}}{\sqrt{N\sum(T^n_{t_0+t})^2 - (\sum T^n_{t_0+t})^2}\sqrt{N\sum(T^n_{t_1+t})^2 - (\sum T^n_{t_1+t})^2}}$$

where summations are taken over $1 \leq n \leq N$, and $\Theta_i(t) = (T^n_{t_i+t})_n$ for $i \in \{0,1\}$.

`Collision`(Θ_0, Θ_1) thus consists in comparing $\max_{0\leq t\leq \omega-1}(\hat{\rho}_{\Theta_0,\Theta_1}(t))$ to a given threshold. In our experiments a preliminary characterization of the targeted device enabled us to find proper values for ω and the threshold.

Note that with the CoCo technique we compute the correlation factor between a set of real power consumptions Θ_0 with another set of real power consumptions Θ_1, rather than with model dependent estimates. As Bogdanov already described about binary and ternary voting techniques [Bog08], an interesting property of this method is that, unlike Hamming weight based CPA, our criterion does not rely on a particular leakage model. The consequences of this are that (i) the attack is more generic and requires less knowledge of the targeted device, and (ii) secret S-Boxes may be attacked as well as known ones.

As said above, correlating two moments (time segments) on different traces has already been applied by Moradi et al. [MME10] on a particular AES implementation. However they collect many traces obtained by encrypting random messages and average them according to the value of an S-Box input byte. This results in 2^8 averaged traces for each byte position, from which they try to detect collisions between two bytes. They successfully carried out this attack on their implementation of the Canright and Batina [CB08] first-order protected implementation. However as indicated by the authors their implementation presented a remaining first-order leakage based on zero-value attacks. We applied Moradi et al. attack to the first-order protected implementations considered in this study without success. We thus consider that this attack is not applicable to most first-order protected implementations. Indeed averaging different traces implies the use of new random mask values which should spoil the influence of the unmasked data and make the collision of intermediate values undetectable. The technique we develop in this section improves on Moradi et al. attack in order to detect data collisions by comparing two moments on the same trace and repeating it on many executions without the destructive averaging process. In the following we detail two applications of our attack on two different implementations.

2.8.2.2 Attack on the Blinded Lookup-Table Implementation

First, we present an application of the CoCo principle on the implementation described in Section 2.8.1.1. This attack targets the execution of the first round `SubBytes` function. Each of the 16 masked input bytes $x'_i = x_i \oplus u$ is replaced by a masked output byte $y'_i = y_i \oplus v$ where $y'_i = S'(x'_i)$. We try to detect when two `SubBytes` inputs (or outputs) are equal within the first AES round as depicted on Figure 2.20.

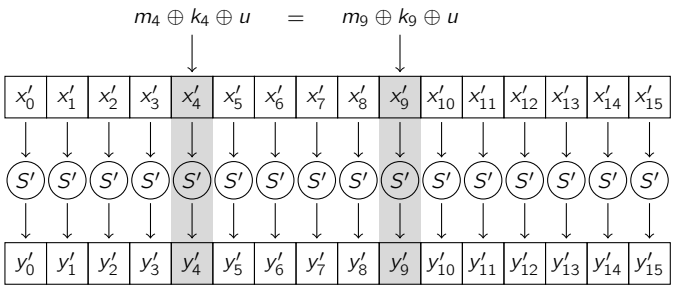


Figure 2.20: Collision between the computation of two S-Boxes on bytes 4 and 9 on the blinded lookup-table implementation

Detecting a collision in the first AES round between bytes i_1 and i_2 yields $x_{i_1} \oplus u = x_{i_2} \oplus u$ and considering that $x_i = m_i \oplus k_i \oplus u$ implies the following relation of the two involved key bytes:

$$k_{i_1} \oplus k_{i_2} = m_{i_1} \oplus m_{i_2} \ . \tag{2.6}$$

Practically, we encrypted N times the same message M and collected the N traces corresponding to the first AES round. For each of the N traces we identified the 16 instants t_i corresponding to the beginning of the computation $S'(x_i \oplus u)$. This allowed us to extract 16 segments from each trace and construct the series Θ_i used for collision-correlation as explained in Section 2.8.2.1.

Performing $\texttt{Collision}(\Theta_{i_1}, \Theta_{i_2})$ for all the 120 possible pairs (i_1, i_2) yields a set of relations $(i_1, i_2, m_{i_1} \oplus m_{i_2})$ given by Eq. (2.6). By repeating this process for several random messages M one can accumulate enough relations so that the secret key is recovered up to a guess on one key byte only.

Based on 10 000 simulations we observed that on average 59 random messages (each one being encrypted N times) provide enough relations to retrieve the key up to an unknown byte.

Practical Results on Simulated Traces The threshold of `Collision` was fixed to having at least one point equal to 1 among the ω points of the correlation trace. Under this condition our attack was successful for $N = 16$. Since a mean of 59 different messages is required, then $16 \times 59 = 944$ traces are sufficient on the average for the attack to succeed on simulated traces were S-Boxes are performed sequentially.

Figures 2.21 and 2.22 show the correlation traces obtained for two different messages. Both figures present the 120 outputs of $\hat{\rho}_{\Theta_{i_1},\Theta_{i_2}}(t)$, $i_1 < i_2$ for each message. The black trace on Figure 2.21 corresponds to a collision found for the first message between the computation of two S-Boxes, while other traces in grey reveal that no other collision occur. One can observe on Figure 2.22 that the second message yields no collision.

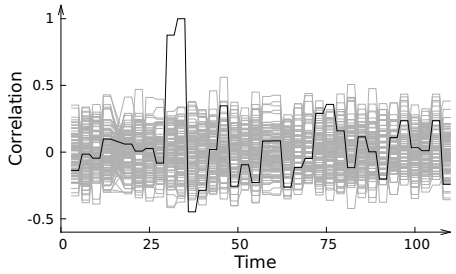


Figure 2.21: Correlation obtained using simulated traces for a message giving a collision (in black)

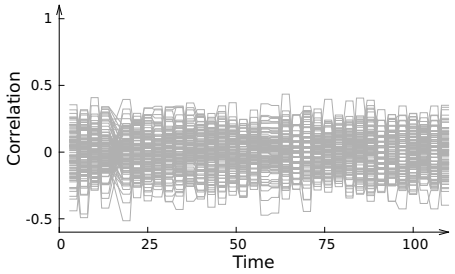


Figure 2.22: Correlation obtained using simulated traces for a message giving no collision

Practical Results on Real Traces The attack was successful using $N = 25$ so that less than 1 500 traces allow to recover the key. Notice how few traces are needed to detect a collision by correlation. This confirms that the CoCo technique is more efficient than classical model-based CPA which would not obtain high correlation levels with only 25 traces. Figure 2.23 shows an example of a correlation peak when an equality between two S-Box outputs occurs, while Figure 2.24 shows the correlation trace when all S-Box outputs are different.

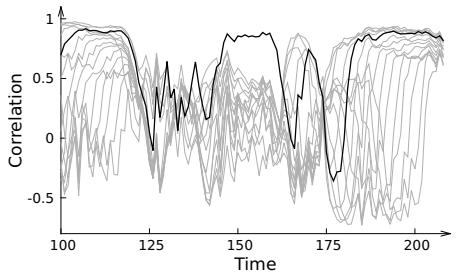


Figure 2.23: Correlation peak obtained using real traces when a collision occurs (in black)

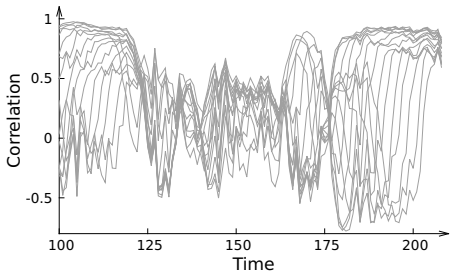


Figure 2.24: No correlation peak occurs in $[130, 160]$ using real traces when data differ

Note that in the case of real traces the threshold is slightly different. To identify a clear relation between two S-Box outputs the correlation trace must be greater than 0.8 in the interval $[130, 160]$. So only these $\omega = 30$ points must be considered when computing $\texttt{Collision}(\Theta_0, \Theta_1)$.

Attack Improvement The method for obtaining information about the key as described above basically exploits collision events where a pair (i_1, i_2) of indices gives a high correlation between Θ_{i_1} and Θ_{i_2} revealing the value of $k_{i_1} \oplus k_{i_2}$. While very informative, such collision events occur much less frequently than non-collision ones, that is when Θ_{i_1} and Θ_{i_2} show no significant correlation between each other. Non-collision events individually bring little information — namely that $k_{i_1} \oplus k_{i_2}$ is different from $m_{i_1} \oplus m_{i_2}$ — but they are so numerous that it appears worth trying to exploit them also.

This improvement halves the number of required traces. However we do not develop this point here for a sake of simplicity but refer the interested reader to our full paper [CFG+11b].

2.8.2.3 Attack on the Blinded Inversion Implementation

The previous attack cannot be applied to the blinded inversion implementation described in Section 2.8.1.2 since the different S-Box input and output bytes are masked with different values u_i. However there may exist a possible leakage leading to what we may call a *zero & one values attack*.

One can notice that values 0 and 1 produce a collision between the input and the output of the masked pseudo-inversion stage I' as depicted on Figure 2.25. This is due to the following properties of the pseudo-inversion:

$$\begin{aligned} I(0) = 0 \quad &\Rightarrow \quad I'(0 \oplus u_i) = 0 \oplus u_i \\ I(1) = 1 \quad &\Rightarrow \quad I'(1 \oplus u_i) = 1 \oplus u_i \end{aligned}$$

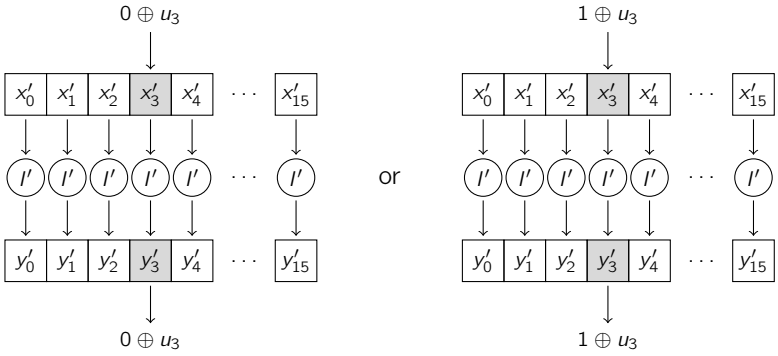


Figure 2.25: Collision between the input and the output on byte 3 of the blinded inversion I' (values 0 and 1 lead to a collision)

The two cases leading to a collision are indistinguishable from one another. Detecting a collision between the input and the output of a blinded inversion gives either $x'_i = 0 \oplus u_i$ or $x'_i = 1 \oplus u_i$ which reveals a key byte up to one single bit:

$$k_i = m_i \quad \text{or} \quad k_i = m_i \oplus 1 \ .$$

Assume we want to recover the 7 most significant bits of k_0. For every even byte value g we encrypt N times a single message M with $m_0 = g$ and collect the corresponding power consumption traces $T^{n,g}$, $1 \leq n \leq N$. Note that in this attack we need to guess the 7 most significant bits only, because the least significant one is indistinguishable. Let's denote t_0 and t_1 the instants when $x_0 \oplus u_0$ is loaded before the pseudo-inversion I, and when the result is stored respectively. For each of the N traces we extract the two segments $T^{n,g}_{[t_0,t_0+\omega-1]}$ and $T^{n,g}_{[t_1,t_1+\omega-1]}$, and construct the series $\Theta^g_0 = (T^{n,g}_{[t_0,t_0+\omega-1]})_n$ and $\Theta^g_1 = (T^{n,g}_{[t_1,t_1+\omega-1]})_n$. For this step of our attack it is

helpful to have some experience on the targeted implementation to identify precisely where these two segments are located.

Applying the decision function $\texttt{Collision}(\Theta_0^g, \Theta_1^g)$ for all the 128 possible values g will reveal two possibilities for k_0. Repeating this step for all key bytes allows the key space to be reduced to 2^{16} values only. Note that a trick which allows to considerably reduce the number of traces is to encrypt the messages $M^g = (g, g, \ldots, g)$ with all bytes equal.

Results on Simulated Traces As for the previous attack on simulated traces, a relation is established when at least one point within the ω-sample correlation trace is equal to 1. As previously, the attack is successful using $N = 16$ traces for each key guess. Figure 2.26 shows the 128 correlation traces for all possible guesses on k_0. The black trace corresponds to the correct guess for k_0.

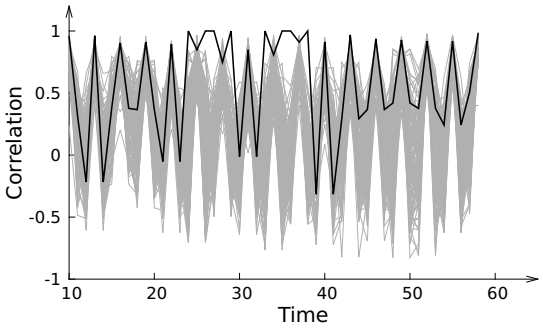


Figure 2.26: Correlation obtained using simulated traces on the pseudo-inversion of the first byte in $\mathbb{F}_{2^8}$

The attack on this second implementation has thus been validated on simulated traces. Regrettably, we did not acquired real traces for this implementation due to a lack of time. Based on what has been observed on the previous attack (successful results obtained using simulations have led to successful results on the chip in practice), we believe that the attack would be successful on the real chip too, using a value for N of the same order to what was necessary for the first attack.

2.8.3 Countermeasures and Concerns for Practical Implementations

The attacks presented in this section defeat first-order protected implementations. We highlight the fact that this kind of attack is more powerful and practical than previous second-order power analyses and increases the risk of these implementations being broken in practice. This confirms the necessity for developers to take into account how collisions of masked data may be unsafe in cryptographic implementations. A possible countermeasure could be the use of second (or higher) order resistant schemes. To the best of our knowledge, the best solution should be the countermeasure presented

by Rivain and Prouff [RP10]. It allows the implementation of proven d-th order DPA resistant AES for any $d \geq 1$.

Another countermeasure against our first attack may simply consist in executing the `SubBytes` function in a random sequence. Even if this method is not theoretically perfect, it may be enough to resist second-order attacks in practice. Considering the second implementation, its main weakness is the use of the same mask before and after each byte pseudo-inversion. If the result is masked with a different value then the CoCo attack is no longer feasible.

It should also be noted that, depending on the quality of the hardware countermeasures provided by the device, these attacks can become more complicated in practice.

Though we presented practical results on software implementations, we believe that this technique may also be a threat for hardware coprocessors. Therefore the collision threat should be taken into consideration by developers and designers during their embedded cryptographic design.

Conclusion

In the first chapter of this thesis, we provide a comprehensive overview of the options available to implement ECC scalar multiplication in embedded devices with industrial constraints. We review most of known techniques suitable for low-resources devices and emphasize on the different solutions to protect them against the simple side-channel analysis. In our survey, all field operations are taken into account, including additions and subtractions, as we show that, on most smart cards equipped with an arithmetic coprocessor, every field operation has a non-negligible minimal cost. This study highlights that some classical optimizations may not be suited to this context such as the S–M trade-off strategy.

This careful comparison of existing solutions leads us to propose our own implementation of the right-to-left scalar multiplication using a new atomic pattern. We demonstrate that this option has one of the lowest costs among all techniques suitable for general curves over $\mathbb{F}_p$ with respect to our specific constraints.

In the second chapter, we focus on physical cryptanalysis. In this aim, we recall how modular multiplication is generally implemented in embedded devices. Then, we give an overview of existing side-channel and fault analyses, together with common countermeasures, with emphasis on ECC and RSA.

We propose a new exponentiation scheme for RSA in which all multiplications are traded for squarings, thus ensuring immunity against various side-channel attacks. Our algorithms use the atomicity principle to resist simple-side channel analysis and yield faster exponentiation than classical regular algorithms on devices provided with a dedicated squaring operation. Besides, we show that our technique provides better results in the context of parallelized operations than previous algorithms. This observation is not limited to the scope of embedded devices and may be worthwhile for everyone interested into high-speed RSA implementation.

While the first side-channel attacks are mostly based on the structure of exponentiation and scalar multiplication algorithms, we point out that new attacks should also consider the underlying implementation of arithmetic operations. With respect to the classical implementation of modular multiplication, we devise the horizontal side-channel analysis on the exponentiation, which recovers a secret key using a single execution trace, while differential side-channel analysis generally requires many of them. This attack highlights the necessity to consider cryptographic implementations as a whole when designing countermeasures. Extending this study to ECC scalar multiplication requires further investigation.

We propose new modular multiplication algorithms to prevent attacks such as the horizontal analysis at a lower level than most of previous countermeasures by shuffling the internal sequence of single-precision multiplications. Further work is necessary to assess the chip area and timing overhead induced by this solution.

Finally, we study how collision-correlation analysis may threaten AES implementations protected against first-order differential analysis when the same mask is used in several S-Boxes, or when the same mask is used to blind the input and the output of an S-Box. This work demonstrates how sound countermeasures may not be as efficient as initially thought when efficiency trade-offs are made in implementations.

Bibliography

Publications

[CFG+10] C. Clavier, B. Feix, G. Gagnerot, M. Roussellet, and V. Verneuil. "Horizontal correlation analysis on exponentiation". In: *ICICS 2010*. Ed. by M. Soriano, S. Qing, and J. López. Vol. 6476. Lecture Notes in Computer Science. Springer, 2010, pp. 46–61 (cit. on pp. 3, 135, 144).

[CFG+11b] C. Clavier, B. Feix, G. Gagnerot, M. Roussellet, and V. Verneuil. "Improved collision-correlation power analysis on first order protected AES". In: *CHES 2011*. Ed. by B. Preneel and T. Takagi. Vol. 6917. Lecture Notes in Computer Science. Springer, 2011, pp. 49–62 (cit. on pp. 3, 150, 157).

[CFG+11g] C. Clavier, B. Feix, G. Gagnerot, M. Roussellet, and V. Verneuil. "Square always exponentiation". In: *INDOCRYPT 2011*. Ed. by D. J. Bernstein and S. Chatterjee. Vol. 7107. Lecture Notes in Computer Science. Springer, 2011, pp. 40–57 (cit. on pp. 3, 121).

[GV10] C. Giraud and V. Verneuil. "Atomicity improvement for elliptic curve scalar multiplication". In: *CARDIS 2010*. Ed. by D. Gollmann, J.-L. Lanet, and J. Iguchi-Cartigny. Vol. 6035. Lecture Notes in Computer Science. Springer, 2010, pp. 80–101 (cit. on pp. 2, 70).

[Ver09] V. Verneuil. "Courbes elliptiques et attaques par canaux auxiliaires". In: *Multisystem & internet security cookbook – MISC* (July 2009), pp. 54–63.

Patents

[CFG+11a] C. Clavier, B. Feix, G. Gagnerot, M. Roussellet, and V. Verneuil. "Circuit intégré protégé contre une analyse par canal auxiliaire horizontale". Pat. FR2956933. Inside Contactless. Sept. 2, 2011 (cit. on p. 144).

[CFG+11c] C. Clavier, B. Feix, G. Gagnerot, M. Roussellet, and V. Verneuil. "Integrated circuit protected against horizontal side channel analysis". Pat. US2011246789. Inside Contactless. Oct. 6, 2011 (cit. on p. 144).

[CFG+11d] C. Clavier, B. Feix, G. Gagnerot, M. Roussellet, and V. Verneuil. "Procédé de cryptographie comprenant une opération d'exponentiation". Pat. Request 100592 FR. Inside Secure. Feb. 25, 2011 (cit. on p. 121).

[CFG+11e] C. Clavier, B. Feix, G. Gagnerot, M. Roussellet, and V. Verneuil. "Procédé de test de la resistance d'un circuit intégré à une analyse par canal auxiliaire". Pat. FR2956932. Inside Contactless. Sept. 2, 2011 (cit. on p. 135).

[CFG+11f] C. Clavier, B. Feix, G. Gagnerot, M. Roussellet, and V. Verneuil. "Process for testing the resistance of an integrated circuit to a side channel analysis". Pat. US2011246119. Inside Contactless. Oct. 6, 2011 (cit. on p. 135).

[GV09] C. Giraud and V. Verneuil. "Method and apparatus for cryptographic data processing". Pat. EP2275925. Oberthur Technologies. July 6, 2009 (cit. on p. 70).

References

[ABF+03] C. Aumüller, P. Bier, W. Fischer, P. Hofreiter, and J.-P. Seifert. "Fault attacks on RSA with CRT: concrete results and practical countermeasures". In: *CHES 2002*. Ed. by B. S. K. Jr., Ç. K. Koç, and C. Paar. Vol. 2523. Lecture Notes in Computer Science. Springer, 2003, pp. 260–275 (cit. on p. 118).

[AFMV07] F. Amiel, B. Feix, L. Marcel, and K. Villegas. "Passive and active combined attacks – Combining fault attacks and side channel analysis". In: *FDTC 2007*. Ed. by L. Breveglieri, S. Gueron, I. Koren, D. Naccache, and J.-P. Seifert. IEEE Computer Society, 2007, pp. 92–99 (cit. on pp. 119, 126).

[AFT+09] F. Amiel, B. Feix, M. Tunstall, C. Whelan, and W. P. Marnane. "Distinguishing multiplications from squaring operations". In: *SAC 2008*. Ed. by R. M. Avanzi, L. Keliher, and F. Sica. Vol. 5381. Lecture Notes in Computer Science. Springer, 2009, pp. 346–360 (cit. on pp. 108, 121, 126).

[AFV07] F. Amiel, B. Feix, and K. Villegas. "Power analysis for secret recovering and reverse engineering of public key algorithms". In: *SAC 2007*. Ed. by C. M. Adams, A. Miri, and M. J. Wiener. Vol. 4876. Lecture Notes in Computer Science. Springer, 2007, pp. 110–125 (cit. on pp. 106, 135, 141).

[AG01] M.-L. Akkar and C. Giraud. "An implementation of DES and AES, secure against some attacks". In: *CHES 2001*. Ed. by Ç. K. Koç, D. Naccache, and C. Paar. Vol. 2162. Lecture Notes in Computer Science. Springer, 2001, pp. 309–318 (cit. on p. 151).

[AK96] R. Anderson and M. Kuhn. "Tamper resistance – a cautionary note". In: *Proceedings of the 2nd USENIX workshop on electronic commerce*. USENIX Association, 1996, pp. 1–11 (cit. on p. 115).

[AMW07] C. M. Adams, A. Miri, and M. J. Wiener, eds. *Selected areas in cryptography, 14th international workshop, sac 2007, ottawa, canada, august 16-17, 2007, revised selected papers*. Vol. 4876. Lecture Notes in Computer Science. Springer, 2007.

[ANSI05] American National Standards Institute. *X9.62 Public key cryptography for the financial service industry: the Elliptic Curve Digital Signature Algorithm (ECDSA)*. Nov. 2005 (cit. on pp. 9, 22, 99).

[APSQ06] C. Archambeau, E. Peeters, F.-X. Standaert, and J.-J. Quisquater. "Template attacks in principal subspaces". In: *CHES 2006*. Ed. by L. Goubin and M. Matsui. Vol. 4249. Lecture Notes in Computer Science. Springer, 2006, pp. 1–14 (cit. on p. 99).

[ARRS05] D. Agrawal, J. R. Rao, P. Rohatgi, and K. Schramm. "Templates as master keys". In: *CHES 2005*. Ed. by J. R. Rao and B. Sunar. Vol. 3659. Lecture Notes in Computer Science. Springer, 2005, pp. 15–29 (cit. on p. 99).

[AT03] T. Akishita and T. Takagi. "Zero-value point attacks on elliptic curve cryptosystem". In: *ISC 2003*. Ed. by C. Boyd and W. Mao. Vol. 2851. Lecture Notes in Computer Science. Springer, 2003, pp. 218–233 (cit. on pp. 110, 111).

[Bar87] P. Barrett. "Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor". In: *CRYPTO '86*. Ed. by A. M. Odlyzko. Vol. 263. Lecture Notes in Computer Science. Springer, 1987, pp. 311–323 (cit. on p. 134).

[BBJ+08] D. J. Bernstein, P. Birkner, M. Joye, T. Lange, and C. Peters. "Twisted Edwards curves". In: *AFRICACRYPT 2008*. Ed. by S. Vaudenay. Vol. 5023. Lecture Notes in Computer Science. Springer, 2008, pp. 389–405 (cit. on pp. 19, 78, 79).

[BCO03] É. Brier, C. Clavier, and F. Olivier. *Optimal statistical power analysis*. Cryptology ePrint Archive, Report 2003/152. 2003. URL: `http://eprint.iacr.org/` (cit. on p. 105).

[BCO04] É. Brier, C. Clavier, and F. Olivier. "Correlation power analysis with a leakage model". In: *CHES 2004*. Ed. by M. Joye and J.-J. Quisquater. Vol. 3156. Lecture Notes in Computer Science. Springer, 2004, pp. 16–29 (cit. on p. 105).

[BDL01] D. Boneh, R. DeMillo, and R. Lipton. "On the importance of eliminating errors in cryptographic computations". In: *Journal of cryptology* 14.2 (2001). An earlier version was published at EUROCRYPT'97 [BDL97], pp. 101–119 (cit. on p. 83).

[BDL96] D. Boneh, R. DeMillo, and R. Lipton. *New threat model breaks crypto codes*. Press Release. Bellcore, Sept. 1996 (cit. on p. 115).

[BDL97] D. Boneh, R. DeMillo, and R. Lipton. "On the importance of checking cryptographic protocols for faults". In: *EUROCRYPT '97*. Ed. by W. Fumy. Vol. 1233. Lecture Notes in Computer Science. Springer, 1997, pp. 37–51 (cit. on pp. 83, 115, 165).

[BECN+06] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan. "The sorcerer's apprentice guide to fault attacks". In: *IEEE, Special issue on cryptography and security* 94.2 (2006), pp. 370–382 (cit. on p. 115).

[Ber01] D. J. Bernstein. *A software implementation of NIST P-224*. Talk given at ECC 2001. Oct. 2001. URL: `http://cr.yp.to/talks/2001.10.29/slides.ps` (cit. on p. 12).

[Ber02] D. J. Bernstein. *Pippenger's exponentiation algorithm*. To be incorporated into author's High-speed cryptography book. Jan. 2002. URL: `http://cr.yp.to/papers.html\#pippenger` (cit. on p. 85).

[Ber06] D. J. Bernstein. "Curve25519: new Diffie-Hellman speed records". In: *PKC 2006*. Ed. by M. Yung, Y. Dodis, A. Kiayias, and T. Malkin. Vol. 3958. Lecture Notes in Computer Science. Springer, 2006, pp. 207–228 (cit. on p. 117).

[BGLR08] L. Batina, B. Gierlichs, and K. Lemke-Rust. "Comparative evaluation of rank correlation based DPA on an AES prototype chip". In: *ISC 2008*. Ed. by T.-C. Wu, C.-L. Lei, V. Rijmen, and D.-T. Lee. Vol. 5222. Lecture Notes in Computer Science. Springer, 2008, pp. 341–354 (cit. on p. 106).

[BJ02] É. Brier and M. Joye. "Weierstraß elliptic curves and side-channel attacks". In: *PKC 2002*. Ed. by D. Naccache and P. Paillier. Vol. 2274. Lecture Notes in Computer Science. Springer, 2002, pp. 335–345 (cit. on p. 42).

[BJ03] O. Billet and M. Joye. "The Jacobi model of an elliptic curve and side-channel analysis". In: *AAECC-15*. Ed. by M. P. C. Fossorier, T. Høholdt, and A. Poli. Vol. 2643. Lecture Notes in Computer Science. Springer, 2003, pp. 34–42 (cit. on p. 17).

[BL07a] D. J. Bernstein and T. Lange. *Analysis and optimization of elliptic-curve single-scalar multiplication*. Cryptology ePrint Archive, Report 2007/455. 2007. URL: `http://eprint.iacr.org/` (cit. on p. 32).

[BL07b] D. J. Bernstein and T. Lange. "Faster addition and doubling on elliptic curves". In: *ASIACRYPT 2007*. Ed. by K. Kurosawa. Vol. 4833. Lecture Notes in Computer Science. Springer, 2007, pp. 29–50 (cit. on pp. 18, 19, 79).

[BL07c] D. J. Bernstein and T. Lange. "Inverted Edwards coordinates". In: *AAECC-17*. Ed. by S. Boztas and H. feng Lu. Vol. 4851. Lecture Notes in Computer Science. Springer, 2007, pp. 20–27 (cit. on p. 19).

[BLW03] B. den Boer, K. Lemke, and G. Wicke. "A DPA attack against the modular reduction within a CRT implementation of RSA". In: *CHES 2002*. Ed. by B. S. K. Jr., Ç. K. Koç, and C. Paar. Vol. 2523. Lecture Notes in Computer Science. Springer, 2003, pp. 228–243 (cit. on pp. 103, 105).

[BMM00] I. Biehl, B. Meyer, and V. Müller. "Differential fault analysis on elliptic curve cryptosystems". In: *CRYPTO 2000*. Ed. by M. Bellare. Vol. 1880. Lecture Notes in Computer Science. Springer, 2000, pp. 131–146 (cit. on pp. 116, 117).

[BNP07] A. Boscher, R. Naciri, and E. Prouff. "CRT RSA algorithm protected against fault attacks". In: *WISTP 2007*. Ed. by D. Sauveron, C. Markantonakis, A. Bilas, and J.-J. Quisquater. Vol. 4462. Lecture Notes in Computer Science. Springer, 2007, pp. 229–243 (cit. on p. 118).

[Bog07] A. Bogdanov. "Improved side-channel collision attacks on AES". In: *SAC 2007*. Ed. by C. M. Adams, A. Miri, and M. J. Wiener. Vol. 4876. Lecture Notes in Computer Science. Springer, 2007, pp. 84–95 (cit. on p. 150).

[Bog08] A. Bogdanov. "Multiple-differential side-channel collision attacks on AES". In: *CHES 2008*. Ed. by E. Oswald and P. Rohatgi. Vol. 5154. Lecture Notes in Computer Science. Springer, 2008, pp. 30–44 (cit. on pp. 150, 152–154).

[BOS06] J. Blömer, M. Otto, and J.-P. Seifert. "Sign change fault attacks on elliptic curve cryptosystems". In: *FDTC 2006*. Ed. by L. Breveglieri, I. Koren, D. Naccache, and J.-P. Seifert. Vol. 4236. Lecture Notes in Computer Science. Springer, 2006, pp. 36–52 (cit. on p. 118).

[BS97] E. Biham and A. Shamir. "Differential fault analysis of secret key cryptosystems". In: *CRYPTO '97*. Ed. by B. S. K. Jr. Vol. 1294. Lecture Notes in Computer Science. Springer, 1997, pp. 513–525 (cit. on p. 115).

[BV07] Y.-J. Baek and I. Vasyltsov. "How to prevent DPA and fault attack in a unified way for ECC scalar multiplication – Ring extension method". In: *ISPEC 2007*. Ed. by E. Dawson and D. S. Wong. Vol. 4464. Lecture Notes in Computer Science. Springer, 2007, pp. 225–237 (cit. on p. 118).

[CB08] D. Canright and L. Batina. "A very compact "perfectly masked" S-box for AES". In: *ACNS 2008*. Ed. by S. M. Bellovin, R. Gennaro, A. D. Keromytis, and M. Yung. Vol. 5037. Lecture Notes in Computer Science. 2008, pp. 446–459 (cit. on pp. 151, 154).

[CCD00] C. Clavier, J.-S. Coron, and N. Dabbous. "Differential power analysis in the presence of hardware countermeasures". In: *CHES 2000*. Ed. by Ç. K. Koç and C. Paar. Vol. 1965. Lecture Notes in Computer Science. Springer, 2000, pp. 252–263 (cit. on p. 102).

[Cer00] Standards for Efficient Cryptography Group. *SEC 2 : Recommended elliptic curve domain parameters*. Certicom Research, 2000. URL: `http://www.secg.org/collateral/sec2\_final.pdf` (cit. on p. 9).

[CFA+05] H. Cohen, G. Frey, R. Avanzi, C. Doche, T. Lange, K. Nguyen, and F. Vercauteren. *Handbook of elliptic and hyperelliptic curve cryptography*. Chapman & Hall/CRC, 2005 (cit. on p. 9).

[CFR10] J.-C. Courrège, B. Feix, and M. Roussellet. "Simple power analysis on exponentiation revisited". In: *CARDIS 2010*. Ed. by D. Gollmann, J.-L. Lanet, and J. Iguchi-Cartigny. Vol. 6035. Lecture Notes in Computer Science. Springer, 2010, pp. 65–79 (cit. on p. 98).

[CG09] C. Clavier and K. Gaj, eds. *Cryptographic hardware and embedded systems - ches 2009, 11th international workshop, lausanne, switzerland, september 6-9, 2009, proceedings*. Vol. 5747. Lecture Notes in Computer Science. Springer, 2009.

[Cie03] M. Ciet. "Aspects of fast and secure arithmetics for elliptic curve cryptography". PhD thesis. Université Catholique de Louvain, June 2003 (cit. on p. 109).

[CJ05a] M. Ciet and M. Joye. "Elliptic curve cryptosystems in the presence of permanent and transient faults". In: *Designs, codes and cryptography* 36.1 (July 2005), pp. 33–43 (cit. on p. 117).

[CJ05b] M. Ciet and M. Joye. "Practical fault countermeasures for chinese remaindering based RSA". In: *FDTC'05*. Ed. by L. Breveglieri and I. Koren. 2005, pp. 124–132. URL: `http://joye.site88.net/papers/CJ05fdtc.pdf` (cit. on p. 118).

[CJLM06] M. Ciet, M. Joye, K. Lauter, and P. L. Montgomery. "Trading inversions for multiplications in elliptic curve cryptography". In: *Designs, codes and cryptography* 39.2 (2006), pp. 189–206 (cit. on p. 15).

[CMCJ04] B. Chevallier-Mames, M. Ciet, and M. Joye. "Low-cost solutions for preventing simple side-channel analysis: side-channel atomicity". In: *IEEE Transactions on computers* 53.6 (2004), pp. 760–768 (cit. on pp. 48, 50, 86, 123).

[CMO98] H. Cohen, A. Miyaji, and T. Ono. "Efficient elliptic curve exponentiation using mixed coordinates". In: *ASIACRYPT '98*. Ed. by K. Ohta and D. Pei. Vol. 1514. Lecture Notes in Computer Science. Springer, 1998, pp. 51–65 (cit. on pp. 16, 24).

[Com90] P. G. Comba. "Exponentiation cryptosystems on the IBM PC". In: *IBM systems journal* 29.4 (1990), pp. 526–538 (cit. on p. 87).

[Cor99] J.-S. Coron. "Resistance against differential power analysis for elliptic curve cryptosystems". In: *CHES'99*. Ed. by Ç. K. Koç and C. Paar. Vol. 1717. Lecture Notes in Computer Science. Springer, 1999, pp. 292–302 (cit. on pp. 40, 97, 109–111).

[CPR07] J.-S. Coron, E. Prouff, and M. Rivain. "Side channel cryptanalysis of a higher order masking scheme". In: *CHES 2007*. Ed. by P. Paillier and I. Verbauwhede. Vol. 4727. Lecture Notes in Computer Science. Springer, 2007, pp. 28–44 (cit. on p. 106).

[CRR03] S. Chari, J. Rao, and P. Rohatgi. "Template attacks". In: *CHES 2002*. Ed. by B. S. K. Jr., Ç. K. Koç, and C. Paar. Vol. 2523. Lecture Notes in Computer Science. Springer, 2003, pp. 13–29 (cit. on pp. 98, 99).

[CS11] S. Chatterjee and P. Sarkar. *Identity-based encryption*. Springer, 2011. URL: `http://books.google.fr/books?id=a5mkUnEPMooC` (cit. on p. 9).

[DH76] W. Diffie and M. Hellman. "New directions in cryptography". In: *IEEE Transaction on information theory* 22.6 (Nov. 1976), pp. 644–654 (cit. on p. 84).

[Dhe98] J.-F. Dhem. "Design of an efficient public-key cryptographic library for RISC-based smart cards". PhD thesis. Université Catholique de Louvain, 1998 (cit. on p. 87).

[DI06] C. Doche and L. Imbert. "Extended double-base number system with applications to elliptic curve cryptography". In: *INDOCRYPT 2006*. Ed. by R. Barua and T. Lange. Vol. 4329. Lecture Notes in Computer Science. Springer, 2006, pp. 335–348 (cit. on p. 22).

[DV11] V. Dupaquis and A. Venelli. "Redundant modular reduction algorithms". In: *CARDIS 2011*. Ed. by E. Prouff. Vol. 7079. Lecture Notes in Computer Science. Springer, 2011, pp. 102–114 (cit. on p. 113).

[Edw07] H. M. Edwards. "A normal form for elliptic curves". In: *Bulletin of the American Mathematical Society* 44 (2007), pp. 393–422. URL: `www.ams.org/bull/2007-44-03/S0273-0979-07-01153-6/home.html` (cit. on p. 18).

[EFD] D. J. Bernstein and T. Lange. *Explicit-formulas database*. URL: `http://www.hyperelliptic.org/EFD` (cit. on pp. 12, 18).

[EMV07] EMV Co. *EMV Book 2, Security and key management, version 4.1z ECC, with support for elliptic curve cryptography*. May 2007. URL: `http://www.emvco.com/specifications.aspx?id=160` (cit. on p. 5).

[EPAF] P. Longa. *ECC point arithmetic formulae*. URL: `patricklonga.bravehost.com/jacobian.html` (cit. on p. 15).

[FGKS02] W. Fischer, C. Giraud, E. W. Knudsen, and J.-P. Seifert. *Parallel scalar multiplication on general elliptic curves over $\mathbb{F}_p$ hedged against non-differential side-channel attacks*. Cryptology ePrint Archive, Report 2002/007. 2002. URL: `http://eprint.iacr.org/` (cit. on pp. 42, 43).

[FLRV08] P.-A. Fouque, R. Lercier, D. Réal, and F. Valette. "Fault attack on elliptic curve Montgomery ladder implementation". In: *FDTC 2008*. Ed. by L. Breveglieri, S. Gueron, I. Koren, D. Naccache, and J.-P. Seifert. IEEE Computer Society, 2008, pp. 92–98 (cit. on p. 117).

[FV03] P.-A. Fouque and F. Valette. "The doubling attack: why upwards is better than downwards". In: *CHES 2003*. Ed. by C. D. Walter, Ç. K. Koç, and C. Paar. Vol. 2779. Lecture Notes in Computer Science. Springer, 2003, pp. 269–280 (cit. on pp. 94, 95).

[GBTP08] B. Gierlichs, L. Batina, P. Tuyls, and B. Preneel. "Mutual information analysis". In: *CHES 2008*. Ed. by E. Oswald and P. Rohatgi. Vol. 5154. Lecture Notes in Computer Science. Springer, 2008, pp. 426–442 (cit. on p. 107).

[Gir06] C. Giraud. "An RSA implementation resistant to fault attacks and to simple power analysis". In: *IEEE Transactions on computers* 55.9 (Sept. 2006), pp. 1116–1120 (cit. on p. 118).

[Gir08] D. Giry. *Keylength – Cryptographic key length recommendation*. Approved by J.-J. Quisquater. 2008. URL: `http://www.keylength.com` (cit. on p. 8).

[GJM10] R. R. Goundar, M. Joye, and A. Miyaji. "Co-Z addition formulæ and binary ladders on elliptic curves". In: *CHES 2010*. Ed. by S. Mangard and F.-X. Standaert. Vol. 6225. Lecture Notes in Computer Science. Springer, 2010, pp. 65–79 (cit. on pp. 15, 42, 46).

[GJM+11] R. R. Goundar, M. Joye, A. Miyaji, M. Rivain, and A. Venelli. "Scalar multiplication on Weierstraß elliptic curves from co-Z arithmetic". In: *Journal of cryptographic engineering* 1.2 (2011), pp. 161–176 (cit. on pp. 46, 48).

[GLIC10] D. Gollmann, J.-L. Lanet, and J. Iguchi-Cartigny, eds. *Smart card research and advanced application, 9th ifip wg 8.8/11.2 international conference, cardis 2010, passau, germany, april 14-16, 2010. proceedings*. Vol. 6035. Lecture Notes in Computer Science. Springer, 2010.

[GMO01] K. Gandolfi, C. Mourtel, and F. Olivier. "Electromagnetic analysis: concrete results". In: *CHES 2001*. Ed. by Ç. K. Koç, D. Naccache, and C. Paar. Vol. 2162. Lecture Notes in Computer Science. Springer, 2001, pp. 251–261 (cit. on p. 102).

[Gor98] D. M. Gordon. "A survey of fast exponentiation methods". In: *Journal of algorithms* 27 (1998), pp. 129–146 (cit. on p. 28).

[Gou02] L. Goubin. "A refined power-analysis attack on elliptic curve cryptosystems". In: *PKC 2003*. Ed. by Y. Desmedt. Vol. 2567. Lecture Notes in Computer Science. Springer, 2002, pp. 199–210 (cit. on pp. 98, 110, 111).

[GT03] J. Golić and C. Tymen. "Multiplicative masking and power analysis of AES". In: *CHES 2002*. Ed. by B. S. K. Jr., Ç. K. Koç, and C. Paar. Vol. 2523. Lecture Notes in Computer Science. Springer, 2003, pp. 198–212 (cit. on p. 151).

[GT04] C. Giraud and H. Thiebeauld. "A survey on fault attacks". In: *CARDIS 2004*. Ed. by J.-J. Quisquater, P. Paradinas, Y. Deswarte, and A. A. E. Kalam. Kluwer, 2004, pp. 159–176 (cit. on p. 115).

[HLM+10] F. Herbaut, P.-Y. Liardet, N. Meloni, Y. Teglia, and P. Véron. "Random Euclidean addition chain generation and its application to point multiplication". In: *INDOCRYPT 2010*. Ed. by G. Gong and K. C. Gupta. Vol. 6498. Lecture Notes in Computer Science. Springer, 2010, pp. 238–261 (cit. on pp. 61, 62).

[HMA+08] N. Homma, A. Miyamoto, T. Aoki, A. Satoh, and A. Shamir. "Collision-based power analysis of modular exponentiation using chosen-message pairs". In: *CHES 2008*. Ed. by E. Oswald and P. Rohatgi. Vol. 5154. Lecture Notes in Computer Science. Springer, 2008, pp. 15–29 (cit. on p. 97).

[HMV03] D. Hankerson, A. Menezes, and S. Vanstone. *Guide to elliptic curve cryptography*. Springer Professional Computing Series, Jan. 2003 (cit. on pp. 8, 28, 30).

[HWCD08] H. Hisil, K. K.-H. Wong, G. Carter, and E. Dawson. "Twisted Edwards curves revisited". In: *ASIACRYPT 2008*. Ed. by J. Pieprzyk. Vol. 5350. Lecture Notes in Computer Science. Springer, 2008, pp. 326–343 (cit. on p. 20).

[ICAO06] International Civil Aviation Organization. *Doc 9303, Part 1. Machine readable travel document, Volume 2. Specifications for electronically enabled passports with biometric identification capability, sixth edition*. 2006 (cit. on p. 5).

[IEEE04] IEEE Computer Society. *IEEE 1363a-2004 Standard specifications for public-key cryptography – Amendment 1: Additional techniques*. Sept. 2004 (cit. on p. 9).

[ISO02] ISO/IEC. *15946-1 Information technology – Security techniques – Cryptographic techniques based on elliptic curves – Part 1: General*. 2002 (cit. on p. 9).

[IT02] T. Izu and T. Takagi. "A fast parallel elliptic curve multiplication resistant against side channel attacks". In: *PKC 2002*. Ed. by D. Naccache and P. Paillier. Vol. 2274. Lecture Notes in Computer Science. Springer, 2002, pp. 280–296 (cit. on p. 42).

[JKP03] B. S. K. Jr., Ç. K. Koç, and C. Paar, eds. *Cryptographic hardware and embedded systems - ches 2002, 4th international workshop, redwood shores, ca, usa, august 13-15, 2002, revised papers*. Vol. 2523. Lecture Notes in Computer Science. Springer, 2003.

[Joy04] M. Joye. "Smart-card implementation of elliptic curve cryptography and DPA-type attacks". In: *CARDIS 2004*. Ed. by J.-J. Quisquater, P. Paradinas, Y. Deswarte, and A. A. E. Kalam. Kluwer, 2004, pp. 115–126 (cit. on p. 109).

[Joy07] M. Joye. "Highly regular right-to-left algorithms for scalar multiplication". In: *CHES 2007*. Ed. by P. Paillier and I. Verbauwhede. Vol. 4727. Lecture Notes in Computer Science. Springer, 2007, pp. 135–147 (cit. on p. 43).

[Joy08] M. Joye. "Fast point multiplication on elliptic curves without precomputation". In: *WAIFI 2008*. Ed. by J. von zur Gathen, J. L. Imaña, and Ç. K. Koç. Vol. 5130. Lecture Notes in Computer Science. Springer, 2008, pp. 36–46 (cit. on pp. 24, 26, 54).

[Joy09] M. Joye. "Highly regular m-ary powering ladders". In: *SAC 2009*. Ed. by M. J. Jacobson Jr., V. Rijmen, and R. Safavi-Naini. Vol. 5867. Lecture Notes in Computer Science. Springer, 2009, pp. 350–363 (cit. on pp. 44, 45).

[JPS05] M. Joye, P. Paillier, and B. Schoenmakers. "On second-order differential power analysis". In: *CHES 2005*. Ed. by J. R. Rao and B. Sunar. Vol. 3659. Lecture Notes in Computer Science. Springer, 2005, pp. 293–308 (cit. on p. 106).

[JQ01] M. Joye and J.-J. Quisquater. "Hessian elliptic curves and side-channel attacks". In: *CHES 2001*. Ed. by Ç. K. Koç, D. Naccache, and C. Paar. Vol. 2162. Lecture Notes in Computer Science. Springer, 2001, pp. 412–420 (cit. on p. 18).

[JQ04] M. Joye and J.-J. Quisquater, eds. *Cryptographic hardware and embedded systems - ches 2004: 6th international workshop cambridge, ma, usa, august 11-13, 2004. proceedings*. Vol. 3156. Lecture Notes in Computer Science. Springer, 2004.

[JT01] M. Joye and C. Tymen. "Protections against differential analysis for elliptic curve cryptography". In: *CHES 2001*. Ed. by Ç. K. Koç, D. Naccache, and C. Paar. Vol. 2162. Lecture Notes in Computer Science. Springer, 2001, pp. 386–400 (cit. on pp. 111, 113).

[JY03] M. Joye and S.-M. Yen. "The Montgomery powering ladder". In: *CHES 2002*. Ed. by B. S. K. Jr., Ç. K. Koç, and C. Paar. Vol. 2523. Lecture Notes in Computer Science. Springer, 2003, pp. 291–302 (cit. on pp. 86, 119).

[KA96] Ç. K. Koç and T. Acar. "Analyzing and comparing Montgomery multiplication algorithms". In: *IEEE Micro* 16 (1996), pp. 26–33 (cit. on p. 89).

[Kim01] K. Kim, ed. *Information security and cryptology - icisc 2001, 4th international conference seoul, korea, december 6-7, 2001, proceedings*. Vol. 2288. Lecture Notes in Computer Science. Springer, 2001.

[KJJ98] P. Kocher, J. Jaffe, and B. Jun. *Introduction to differential power analysis and related attacks*. Tech. rep. Cryptography Research Inc., 1998. URL: `http://www.cryptography.com/resources/whitepapers/DPATechInfo.pdf` (cit. on pp. 83, 101, 102, 151).

[KJJ99] P. C. Kocher, J. Jaffe, and B. Jun. "Differential power analysis". In: *CRYPTO '99*. Ed. by M. J. Wiener. Vol. 1666. Lecture Notes in Computer Science. Springer, 1999, pp. 388–397 (cit. on pp. 83, 91, 101, 102, 106).

[KK99] O. Kommerling and M. Kuhn. "Design principles for tamper resistant smartcard processors". In: *The USENIX workshop on smartcard technology (Smartcard '99)*. 1999, pp. 9–20 (cit. on p. 84).

[KNP01] Ç. K. Koç, D. Naccache, and C. Paar, eds. *Cryptographic hardware and embedded systems - ches 2001, third international workshop, paris, france, may 14-16, 2001, proceedings*. Vol. 2162. Lecture Notes in Computer Science. Springer, 2001.

[KO62] A. A. Karatsuba and Y. P. Ofman. "Multiplication of multidigit numbers on automata". In: *Doklady Akademii Nauk SSSR* 45(2):293294 (1962) (cit. on p. 87).

[Koc96] P. C. Kocher. "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems". In: *CRYPTO '96*. Ed. by N. Koblitz. Vol. 1109. Lecture Notes in Computer Science. Springer, 1996, pp. 104–113 (cit. on p. 83).

[KP00] Ç. K. Koç and C. Paar, eds. *Cryptographic hardware and embedded systems - ches 2000, second international workshop, worcester, ma, usa, august 17-18, 2000, proceedings*. Vol. 1965. Lecture Notes in Computer Science. Springer, 2000.

[KP99] Ç. K. Koç and C. Paar, eds. *Cryptographic hardware and embedded systems, first international workshop, CHES'99, worcester, ma, usa, august 12-13, 1999, proceedings*. Vol. 1717. Lecture Notes in Computer Science. Springer, 1999.

[KQ07] C. H. Kim and J.-J. Quisquater. "Fault attacks for CRT based RSA: new attacks, new results, and new countermeasures". In: *WISTP 2007*. Ed. by D. Sauveron, C. Markantonakis, A. Bilas, and J.-J. Quisquater. Vol. 4462. Lecture Notes in Computer Science. Springer, 2007, pp. 215–228 (cit. on p. 118).

[KT93] K. Koyama and Y. Tsuruoka. "Speeding up elliptic cryptosystems by using a signed binary window method". In: *CRYPTO '92*. Ed. by E. F. Brickell. Vol. 740. Lecture Notes in Computer Science. Springer, 1993, pp. 345–357 (cit. on p. 28).

[LG09] P. Longa and C. H. Gebotys. "Fast multibase methods and other several optimizations for elliptic curve scalar multiplication". In: *PKC 2009*. Ed. by S. Jarecki and G. Tsudik. Vol. 5443. Lecture Notes in Computer Science. Springer, 2009, pp. 443–462 (cit. on p. 22).

[LM08a] P. Longa and A. Miri. "Fast and flexible elliptic curve point arithmetic over prime fields". In: *IEEE Transactions on computers* 57.3 (2008), pp. 289–302 (cit. on p. 34).

[LM08b] P. Longa and A. Miri. "New composite operations and precomputation scheme for elliptic curve cryptosystems over prime fields". In: *PKC 2008*. Ed. by R. Cramer. Vol. 4939. Lecture Notes in Computer Science. Springer, 2008, pp. 229–247 (cit. on p. 15).

[LM08c] P. Longa and A. Miri. *New multibase non-adjacent form scalar multiplication and its application to elliptic curve cryptosystems (extended version)*. Cryptology ePrint Archive, Report 2008/052. 2008. URL: `http://eprint.iacr.org/` (cit. on pp. 22, 34).

[LM10] M. Lochter and J. Merkle. *Elliptic curve cryptography Brainpool standard curves and curve generation*. RFC 5639. ECC Brainpool, Mar. 2010. URL: `http://tools.ietf.org/html/rfc5639` (cit. on pp. 7, 9).

[Lon07] P. Longa. "Accelerating the scalar multiplication on elliptic curve cryptosystems over prime fields". MA thesis. Canada: School of Information Technology and Engineering, University of Ottawa, 2007 (cit. on pp. 12, 52, 54, 55).

[LS01] P.-Y. Liardet and N. Smart. "Preventing SPA/DPA in ECC systems using the Jacobi form". In: *CHES 2001*. Ed. by Ç. K. Koç, D. Naccache, and C. Paar. Vol. 2162. Lecture Notes in Computer Science. Springer, 2001, pp. 401–411 (cit. on p. 18).

[MDS99a] T. Messerges, E. Dabbish, and R. Sloan. "Investigations of power analysis attacks on smartcards". In: *The USENIX workshop on smartcard technology (Smartcard '99)*. 1999, pp. 151–161 (cit. on p. 102).

[MDS99b] T. Messerges, E. Dabbish, and R. Sloan. "Power analysis attacks of modular exponentiation in smartcard". In: *CHES'99*. Ed. by Ç. K. Koç and C. Paar. Vol. 1717. Lecture Notes in Computer Science. Springer, 1999, pp. 144–157 (cit. on pp. 103, 135).

[Mel07] N. Meloni. "New point addition formulae for ECC applications". In: *WAIFI 2007*. Ed. by C. Carlet and B. Sunar. Vol. 4547. Lecture Notes in Computer Science. Springer, 2007, pp. 189–201 (cit. on pp. 14, 48, 58–61).

[Mes00] T. Messerges. "Using second-order power analysis to attack DPA resistant software". In: *CHES 2000*. Ed. by Ç. K. Koç and C. Paar. Vol. 1965. Lecture Notes in Computer Science. Springer, 2000, pp. 238–251 (cit. on p. 106).

[MME10] A. Moradi, O. Mischke, and T. Eisenbarth. "Correlation-enhanced power analysis collision attack". In: *CHES 2010*. Ed. by S. Mangard and F.-X. Standaert. Vol. 6225. Lecture Notes in Computer Science. Springer, 2010, pp. 125–139 (cit. on pp. 150–152, 154).

[MMM04] H. Mamiya, A. Miyaji, and H. Morimoto. "Efficient countermeasures against RPA, DPA, and SPA". In: *CHES 2004*. Ed. by M. Joye and J.-J. Quisquater. Vol. 3156. Lecture Notes in Computer Science. Springer, 2004, pp. 343–356 (cit. on p. 112).

[MO09] M. Medwed and E. Oswald. "Template attacks on ECDSA". In: *WISA 2008*. Ed. by K.-I. Chung, K. Sohn, and M. Yung. Vol. 5379. Lecture Notes in Computer Science. Springer, 2009, pp. 14–27 (cit. on p. 99).

[Mon85] P. Montgomery. "Modular multiplication without trial division". In: *Mathematics of computation* 44.170 (Apr. 1985), pp. 519–521 (cit. on p. 87).

[Mon87] P. Montgomery. "Speeding the Pollard and elliptic curve methods of factorization". In: *Mathematics of computation* 48 (1987), pp. 243–264 (cit. on pp. 42, 86).

[MOP07] S. Mangard, E. Oswald, and T Popp. *Power analysis attacks: revealing the secrets of smart cards*. Springer, 2007 (cit. on p. 151).

[Mor09] F. Morain. *Edwards curves and CM curves*. eprint arXiv:0904.2243. Apr. 2009. URL: `http://arxiv.org/abs/0904.2243` (cit. on p. 78).

[MOV97] A. Menezes, P. van Oorschot, and S. Vanstone. *Handbook of applied cryptography*. CRC Press, 1997. URL: `http://www.cacr.math.uwaterloo.ca/hac/` (cit. on pp. 9, 85).

[MS00] R. Mayer Sommer. "Smartly analyzing the simplicity and the power of simple power analysis on smartcards". In: *CHES 2000*. Ed. by Ç. K. Koç and C. Paar. Vol. 1965. Lecture Notes in Computer Science. Springer, 2000, pp. 78–92 (cit. on p. 94).

[MS10] S. Mangard and F.-X. Standaert, eds. *Cryptographic hardware and embedded systems, ches 2010, 12th international workshop, santa barbara, ca, usa, august 17-20, 2010. proceedings*. Vol. 6225. Lecture Notes in Computer Science. Springer, 2010.

[Mui04] J. A. Muir. "Efficient integer representation for cryptographic operations". PhD thesis. University of Waterloo, 2004 (cit. on p. 26).

[Möl03] B. Möller. "Improved techniques for fast exponentiation". In: *ICISC 2002*. Ed. by P. J. Lee and C. H. Lim. Vol. 2587. Lecture Notes in Computer Science. Springer, 2003, pp. 298–312 (cit. on pp. 28, 31).

[NIST01] National Institute of Standards and Technology. *FIPS 197 Advanced encryption standard*. Nov. 2001 (cit. on p. 150).

[NIST06] National Institute of Standards and Technology. *FIPS 186-3 Digital signature standard*. Draft. Mar. 2006 (cit. on pp. 7, 9, 75, 84, 109).

[Nohl] K. Nohl. Webpage at the Computer Science Department of the University of Virginia. URL: `http://www.cs.virginia.edu/~kn5f/` (cit. on p. 84).

[NP02] D. Naccache and P. Paillier, eds. *Public key cryptography, 5th international workshop on practice and theory in public key cryptosystems, pkc 2002, paris, france, february 12-14, 2002, proceedings*. Vol. 2274. Lecture Notes in Computer Science. Springer, 2002.

[NS03] P. Q. Nguyen and I. Shparlinski. "The insecurity of the elliptic curve digital signature algorithm with partially known nonces". In: *Designs, codes and cryptography* 30.2 (2003), pp. 201–217 (cit. on p. 100).

[NSA05] National Security Agency. *NSA Suite B cryptography*. 2005. URL: `http://www.nsa.gov/ia/programs/suiteb_cryptography/` (cit. on p. 5).

[OMPR05] E. Oswald, S. Mangard, N. Pramstaller, and V. Rijmen. "A side-channel analysis resistant description of the AES S-box". In: *FSE 2005*. Ed. by H. Gilbert and H. Handschuh. Vol. 3557. Lecture Notes in Computer Science. Springer, 2005, pp. 413–423 (cit. on p. 151).

[OR08] E. Oswald and P. Rohatgi, eds. *Cryptographic hardware and embedded systems - ches 2008, 10th international workshop, washington, d.c., usa, august 10-13, 2008. proceedings*. Vol. 5154. Lecture Notes in Computer Science. Springer, 2008.

[PH78] S. Pohlig and M. Hellman. "An improved algorithm for computing logarithms over $GF(p)$ and its cryptographic significance". In: *IEEE Transactions on information theory* 24 (Jan. 1978), pp. 106–110 (cit. on p. 9).

[PR09] E. Prouff and M. Rivain. "Theoretical and practical aspects of mutual information based side channel analysis". In: *ACNS 2009*. Ed. by M. Abdalla, D. Pointcheval, P.-A. Fouque, and D. Vergnaud. Vol. 5536. Lecture Notes in Computer Science. 2009, pp. 499–518 (cit. on p. 107).

[PRB09] E. Prouff, M. Rivain, and R. Bevan. "Statistical analysis of second order differential power analysis". In: *IEEE Transactions on computers* 58.6 (2009), pp. 799–811 (cit. on pp. 106, 107).

[PV07] P. Paillier and I. Verbauwhede, eds. *Cryptographic hardware and embedded systems - ches 2007, 9th international workshop, vienna, austria, september 10-13, 2007, proceedings*. Vol. 4727. Lecture Notes in Computer Science. Springer, 2007.

[Riv09] M. Rivain. "Securing RSA against fault analysis by double addition chain exponentiation". In: *CT-RSA 2009*. Ed. by M. Fischlin. Vol. 5473. Lecture Notes in Computer Science. Springer, 2009, pp. 459–480 (cit. on p. 118).

[Riv11] M. Rivain. *Fast and regular algorithms for scalar multiplication over elliptic curves*. Cryptology ePrint Archive, Report 2011/338. 2011. URL: `http://eprint.iacr.org/` (cit. on p. 46).

[RM07] B. Robisson and P. Manet. "Differential behavioral analysis". In: *CHES 2007*. Ed. by P. Paillier and I. Verbauwhede. Vol. 4727. Lecture Notes in Computer Science. Springer, 2007, pp. 413–426 (cit. on p. 119).

[RP10] M. Rivain and E. Prouff. "Provably secure higher-order masking of AES". In: *CHES 2010*. Ed. by S. Mangard and F.-X. Standaert. Vol. 6225. Lecture Notes in Computer Science. Springer, 2010, pp. 413–427 (cit. on p. 159).

[RPD09] M. Rivain, E. Prouff, and J. Doget. "Higher-order masking and shuffling for software implementations of block ciphers". In: *CHES 2009*. Ed. by C. Clavier and K. Gaj. Vol. 5747. Lecture Notes in Computer Science. Springer, 2009, pp. 171–188 (cit. on p. 144).

[RS05] J. R. Rao and B. Sunar, eds. *Cryptographic hardware and embedded systems - ches 2005, 7th international workshop, edinburgh, uk, august 29 - september 1, 2005, proceedings*. Vol. 3659. Lecture Notes in Computer Science. Springer, 2005.

[Sch00] W. Schindler. "A timing attack against RSA with the chinese remainder theorem". In: *CHES 2000*. Ed. by Ç. K. Koç and C. Paar. Vol. 1965. Lecture Notes in Computer Science. Springer, 2000, pp. 109–124 (cit. on p. 89).

[Sha98] A. Shamir. "Improved method and apparatus for protecting public key schemes from timing and fault attacks". Pat. WO9852319. Yeda Research, Development Co. Ltd., and L. Fleit. Also presented at the Eurocrypt'97 rump session. Nov. 19, 1998 (cit. on p. 118).

[Sko05] S. P. Skorobogatov. *Semi-invasive attacks – A new approach to hardware security analysis*. Tech. rep. 630. University of Cambridge – Computer Laboratory, Apr. 2005. URL: `http://www.cl.cam.ac.uk/TechReports/` (cit. on p. 115).

[SLFP04] K. Schramm, G. Leander, P. Felke, and C. Paar. "A collision-attack on AES: combining side channel- and differential-attack". In: *CHES 2004*. Ed. by M. Joye and J.-J. Quisquater. Vol. 3156. Lecture Notes in Computer Science. Springer, 2004, pp. 163–175 (cit. on p. 150).

[SMBQ07] D. Sauveron, C. Markantonakis, A. Bilas, and J.-J. Quisquater, eds. *Information security theory and practices. smart cards, mobile and ubiquitous computing systems, first ifip tc6 / wg 8.8 / wg 11.2 international workshop, wistp 2007, heraklion, crete, greece, may 9-11, 2007, proceedings*. Vol. 4462. Lecture Notes in Computer Science. Springer, 2007.

[SPQ05] F.-X. Standaert, E. Peeters, and J.-J. Quisquater. "On the masking countermeasure and higher order power analysis attacks". In: *ITCC (1)*. IEEE Computer Society, 2005, pp. 562–567 (cit. on p. 106).

[STA+10] J.-M. Schmidt, M. Tunstall, R. M. Avanzi, I. Kizhvatov, T. Kasper, and D. Oswald. "Combined implementation attack resistant exponentiation". In: *LATINCRYPT 2010*. Ed. by M. Abdalla and P. S. L. M. Barreto. Vol. 6212. Lecture Notes in Computer Science. Springer, 2010, pp. 305–322 (cit. on pp. 120, 127).

[SWP03] K. Schramm, T. Wollinger, and C. Paar. "A new class of collision attacks and its application to DES". In: *FSE 2003*. Ed. by T. Johansson. Vol. 2887. Lecture Notes in Computer Science. Springer, 2003, pp. 206–222 (cit. on p. 150).

[Tar10] C. Tarnovsky. *Deconstructing a 'secure' processor*. Presentation at Black Hat DC. 2010. URL: `http://www.securitytube.net/video/945` (cit. on p. 84).

[VCS09] N. Veyrat-Charvillon and F.-X. Standaert. "Mutual information analysis: how, when and why?" In: *CHES 2009*. Ed. by C. Clavier and K. Gaj. Vol. 5747. Lecture Notes in Computer Science. Springer, 2009, pp. 429–443 (cit. on p. 107).

[VD10] A. Venelli and F. Dassance. "Faster side-channel resistant elliptic curve scalar multiplication". In: *Contemporary mathematics* 521 (2010), pp. 29–40 (cit. on pp. 46, 48).

[Ven10] A. Venelli. "Efficient entropy estimation for mutual information analysis using B-splines". In: *WISTP 2010*. Ed. by P. Samarati, M. Tunstall, J. Posegga, K. Markantonakis, and D. Sauveron. Vol. 6033. Lecture Notes in Computer Science. Springer, 2010, pp. 17–30 (cit. on p. 107).

[Ven11] A. Venelli. "Analysis of nonparametric estimation methods for mutual information analysis". In: *ICISC 2010*. Ed. by K. H. Rhee and D. Nyang. Vol. 6829. Lecture Notes in Computer Science. Springer, 2011, pp. 1–15 (cit. on p. 107).

[Vig08] D. Vigilant. "RSA with CRT: a new cost-effective solution to thwart fault attacks". In: *CHES 2008*. Ed. by E. Oswald and P. Rohatgi. Vol. 5154. Lecture Notes in Computer Science. Springer, 2008, pp. 130–145 (cit. on p. 118).

[Wal01] C. D. Walter. "Sliding windows succumbs to Big Mac attack". In: *CHES 2001*. Ed. by Ç. K. Koç, D. Naccache, and C. Paar. Vol. 2162. Lecture Notes in Computer Science. Springer, 2001, pp. 286–299 (cit. on pp. 104, 138).

[Wal02] C. D. Walter. "Precise bounds for Montgomery modular multiplication and some potentially insecure RSA moduli". In: *CT-RSA 2002*. Ed. by B. Preneel. Vol. 2271. Lecture Notes in Computer Science. Springer, 2002, pp. 30–39 (cit. on pp. 89, 147).

[Wal99] C. Walter. "Montgomery exponentiation needs no final subtractions". In: *Electronics letters* 35.21 (Oct. 1999), pp. 1831–1832 (cit. on p. 89).

[WT01] C. D. Walter and S. Thompson. "Distinguishing exponent digits by observing modular subtractions". In: *CT-RSA 2001*. Ed. by D. Naccache. Vol. 2020. Lecture Notes in Computer Science. Springer, 2001, pp. 192–207 (cit. on p. 89).

[WW04] J. Waddle and D. Wagner. "Toward efficient second-order power analysis". In: *CHES 2004*. Ed. by M. Joye and J.-J. Quisquater. Vol. 3156. Lecture Notes in Computer Science. Springer, 2004, pp. 1–15 (cit. on p. 106).

[WWM11] M. F. Witteman, J. G. J. van Woudenberg, and F. Menarini. "Defeating RSA multiply-always and message blinding countermeasures". In: *CT-RSA 2011*. Ed. by A. Kiayias. Vol. 6558. Lecture Notes in Computer Science. Springer, 2011, pp. 77–88 (cit. on p. 152).

[Yao76] A. C.-C. Yao. "On the evaluation of powers". In: *SIAM Journal on computing* 5.1 (1976), pp. 100–103 (cit. on p. 28).

[YJ00] S.-M. Yen and M. Joye. "Checking before output may not be enough against fault-based cryptanalysis". In: *IEEE Transactions on computers* 49.9 (2000), pp. 967–970 (cit. on p. 119).

[YKLM01a] S.-M. Yen, S.-J. Kim, S.-G. Lim, and S.-J. Moon. "A countermeasure against one physical cryptanalysis may benefit another attack". In: *ICISC 2001*. Ed. by K. Kim. Vol. 2288. Lecture Notes in Computer Science. Springer, 2001, pp. 414–427 (cit. on p. 119).

[YKLM01b] S.-M. Yen, S.-J. Kim, S.-G. Lim, and S.-J. Moon. "RSA speedup with residue number system immune against hardware fault cryptanalysis". In: *ICISC 2001*. Ed. by K. Kim. Vol. 2288. Lecture Notes in Computer Science. Springer, 2001, pp. 397–413 (cit. on pp. 118, 120).

[YLMH05] S.-M. Yen, W.-C. Lien, S.-J. Moon, and J. Ha. "Power analysis by exploiting chosen message and internal collisions – Vulnerability of checking mechanism for RSA-decryption". In: *Mycrypt 2005*. Ed. by E. Dawson and S. Vaudenay. Vol. 3715. Lecture Notes in Computer Science. Springer, 2005, pp. 183–195 (cit. on pp. 97, 113).