Key Randomization Countermeasures to Power Analysis Attacks on Elliptic Curve Cryptosystems

by

Nevine Maurice Ebeid

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2007

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

It is essential to secure the implementation of cryptosystems in embedded devices agains side-channel attacks. Namely, in order to resist differential (DPA) attacks, randomization techniques should be employed to decorrelate the data processed by the device from secret key parts resulting in the value of this data. Among the countermeasures that appeared in the literature were those that resulted in a random representation of the key known as the binary signed digit representation (BSD). We have discovered some interesting properties related to the number of possible BSD representations for an integer and we have proposed a different randomization algorithm. We have also carried our study to the τ-adic representation of integers which is employed in elliptic curve cryptosystems (ECCs) using Koblitz curves. We have then dealt with another randomization countermeasure which is based on randomly splitting the key. We have investigated the secure employment of this countermeasure in the context of ECCs.

Acknowledgements

All praise and glory is due to God for supporting and guiding me every day of my life and throughout all my work. I seize this opportunity to express my sincere gratitude to my supervisor, Professor Anwar Hasan, for his ceaseless encouragement and understanding, his wise guidance and his genuine knowledge. I thank all the professors who have taught me and enriched my mind with their precious knowledge, especially Professors Mark Aagaard, Guang Gong, Doug Stinson and Alfred Menezes. I also thank my colleagues for their suggestions and comments, especially Jaewook Chung, Arash Reyhani-Masoleh and James Muir. I am indebted to Herb Little for the great opportunity he provided me to pursue the final parts of this work. I thank my parents for their love and support, and my sister and her husband for their care and invaluable advises. I thank also our church priest, Fr. Athanasius Iskander, for his continual support and prayers. Finally, I owe this work to my beloved husband and children for bearing with me every difficulty I faced and for helping me overcome it.

To Adel, my beloved husband

Contents

List of Tables

List of Figures

List of Acronyms

BSD Binary Signed Digit

DPA Differential Power Analysis

EC Elliptic Curve

ECC Elliptic Curve Cryptosystem

ECADD Elliptic Curve (Point) Addition

ECDBL Elliptic Curve (Point) Doubling

ECSM Elliptic Curve Scalar Multiplication

NAF Non-Adjacent Form

NFA Nondeterministic Finite Automaton

RTNAF Reduced τ-adic Non-Adjacent Form

SCA Side-Channel Analysis

SPA Simple Power Analysis

τ**NAF** τ-adic Non-Adjacent Form

List of Symbols (Selected)

$\lambda(k,n)$	Number of BSD representations of length n sbits of the integer $k \in [0, 2^n - 1]$.
$\sigma(n)$	$\sum_{k=0}^{2^n-1} \lambda(k,n)$.
$\delta(k,n)$	Number of BSD representations of length $n+1$ sbits of the integer $k \in [0, 2^n - 1]$.
$\varsigma(n)$	$\sum_{k=0}^{2^n-1} \delta(k,n)$.
$\vartheta(k,n)$	Number of τ-adic representations (of length up to $l+2$ sbits) of a τNAF k of length l sbits.
τ	$\frac{1}{2}\ (\mu + \sqrt{-7})$, where $\mu \in \{-1, 1\}$.

Chapter 1

Introduction

There appears to be an increasing trend towards adapting elliptic curve cryptography [Mil86; Kob87] for different security purposes. A good illustration of this trend is the announcement of *suite B* by the NSA [NSA] which is a set of cryptographic algorithms that serves as an interoperable cryptographic base for both unclassified information and most classified information. In suite B, elliptic curves recommended in [NIST] over 256-bit prime modulus, which match the security of AES with 128-bit keys are sufficient for protecting classified information up to the *secret* level. The 384-bit prime modulus elliptic curves, matching AES with 192-bit keys are sufficient for the protection of *top secret* information, though for the sake of interoperability, the 521-bit prime modulus elliptic curves, matching 256-bit keys for AES, are used.

Elliptic curve cryptosystems (ECCs) are suitable for implementation on devices with limited memory and computational capability such as smart cards and also with limited power such as wireless handheld devices. This is due to the fact that elliptic curves over large finite fields provide the same security level as other cryptosystems such as RSA for much smaller key sizes.

However nowadays, the mathematical strength of the cryptosystem implemented on such devices as smart cards and personal wireless devices is not enough to ensure the security of the keys stored in them. It is becoming inevitable to check the resistance of these devices against side-channel attacks. Side channels include execution time [Koc96], power consumption [KJJ99; MDS99a; MDS99b], electromagnetic emanations [QS01; AARR] and computational errors due to faults in hardware [BDL01]. For recent surveys on these attacks and their countermeasures, we refer the reader to [Ava05; Osw05; Joy05; BCF06; Lan06].

Considering power analysis attacks, there are two main types that were presented by Kocher *et al.* These are *simple* and *differential* power analysis attacks (referred to as SPA and DPA respectively). Both of them are based on monitoring the power consumption of a cryptographic token while executing an algorithm that manipulates the secret key. The traces of the measured power are then analyzed to obtain significant information about the key. In some cases the key can be totally compromised and in others the search space of the key can be reduced to a computationally affordable size. In SPA, a single power trace can reveal large features of the algorithm being executed such as the iterations of the loop. Moreover, cryptosystem-specific operations such as point doubling and adding in ECCs can be identified [Cor99]. In order to resist this SPA attack, the steps of the algorithm need to be uniform across different executions.

On the other hand, even if the SPA attack does not apply, DPA attacks rely on first collecting several power traces for executions of the algorithm using the *same* key. Then, these traces are processed using various statistical tools to identify features of the processed data (operands or memory address) at a specific instant of the cryptographic algorithm, *e.g.*, its Hamming weight or the number of toggled bits compared with another data. Information about the processed data can directly lead to revealing information

about the key parts that lead to processing this specific data at this specific instant. Hence, DPA attacks are, in general, more powerful than the SPA attack. Randomization of the data processed at some instant is essential in resisting this type of attacks.

Electromagnetic emanations present another powerful side channel since the information is leaked from the device via more than one channel and is a function of space as well as of time. In [AARR], Agrawal *et al.* presented both simple (SEMA) and differential (DEMA) electromagnetic analysis attacks on smart cards and on a Palm pilot in [RAA+03]. In [AARR], they conclude that software countermeasures rely on *signal information reduction*, which is achieved by "randomization and/or frequent key refreshing within the computation", which agrees with the concept of resisting DPA attacks. Therefore, in this thesis we include the attacks on both side channels under the acronyms SPA and DPA, for simple and differential attacks, respectively. Also, in the thesis we assume that, whenever random data is needed, this data is available in a secure fashion. In other words, we assume that the random—or pseudorandom— number generator is not vulnerable to the described attacks.

Motivation

Various randomization techniques were proposed to protect the ellipitic curve scalar multiplication (ECSM), which is the core operation of ECCs, against DPA attacks. Among them were those proposed by Oswald and Aigner in [OA01] and by Ha and Moon in [HM02a]. Their approach is based on randomizing the number and the sequence of execution of operations in the scalar multiplication algorithm. This is achieved by inserting a random decision in the process of building the representation of the scalar k. The algorithms to which this randomization is applied were originally proposed to speed up the

EC scalar multiplication by producing the *non-adjacent form* (NAF) of k, which contains fewer nonzero signed bits than the binary representation. Hence, an important outcome of these algorithms is a random *binary signed digit* (BSD) representation of k.

Although randomizing the BSD representation of the key alone is not considered a countermeasure against DPA attacks [FMPV04], it was of at least mathematical interest to us to analyse these randomization algorithms and to identify their similarities and differences and the true randomness of the resulting BSD representation from each of them. Then we embarked on studying and discovering some statistical properties of this representation and we proposed a different randomization approach. We then carried the randomization concept to a different representation of integers known as the τ-adic representation which is employed in ECCs using Koblitz curves [Sol00]. Such curves were recommended in [NIST] and their use can significantly increase the performance of ECCs.

key splitting is another key randomization technique that was proposed as a DPA countermeasure [CJ01; CJ03; CQS03]. It is based on randomly splitting the key into two parts such that each part is different in every ECSM execution. This approach is a good candidate for a DPA countermeasure since the actual *value* of the key parts is randomized, and, hence, resists different forms of DPA attacks, such as the address reuse and operand reuse attacks. We studied the different forms of key splitting the key and the possible evaluation techniques for each form.

Thesis organization

In Chapter 2, we present an overview of ECCs and the different algorithms employed to carry the scalar multiplication operation. We also present a background on the SPA and DPA attacks on this operation and the countermeasures that form the base of the work

subsequently presented.

In Chapter 3, we provide a detailed analysis of the Oswald-Aigner (OA) and the Ha-Moon (HM) randomization algorithms. We investigate the randomness of the recoded keys of each of them *i.e.*, whether or not each can produce all possible BSD representations of an integer. We also present the complexity (average case) analysis of each algorithm using both Markov chain model and grammatical specification method.

We subsequently present in Chapter 4 our answers to relevant questions concerning the BSD representation of integers such as: What is the average number of BSD representations of n bit integers? Can we find the exact number of representations for a certain integer? Which n-bit integer has the maximum number of representations? We prove that the maximum number of representations is a Fibonacci number, and provide an alternate expression for that number. We also present a randomization algorithm that scans the bits of the input integer starting from the most significant end unlike the OA and HM algorithms.

We carry the HM randomization technique to the τ-adic representation of integers in Chapter 5 where the input to the algorithm is a τ-adic NAF (τNAF). We discover the τNAF that yields the maximum number of representation and prove that this number is also a Fibonacci number. Thereafter, we study the average and exact number of τ-adic representations of a τNAF. We also analyze the average-case complexity of the randomization algorithm.

In Chapter 6, we analyze the DPA resistance of specific implementations of each key splitting approach. We discuss the candidate SPA-resistant algorithms and compare the resulting performance when combined with each form of key splitting. At the end of the chapter, we discuss briefly countermeasures to DPA attacks on the ECDSA and the ECMQV algorithms.

In Chapter 7, we summarize our contribution and provide some interesting directions for future work.

In Appendix A, we provide an extensive survey on SPA and DPA countermeasures along with the computational cost of each one. We also reconcile the different types of attacks and emphasize the conditions favoring their application, which is summarized in the conclusion of that appendix.

The remaining appendices provide a background on the various analysis tools and techniques we employed in our work.

Summary of Major Contributions

- Analysis of the randomness of the BSD representations resulting from the Oswald-Aigner and Ha-Moon randomization algorithms as well as their average case complexity.
- New properties of the number of BSD and τ-adic representations, such as the average and maximum number of representations and the patterns and values of the integers or τNAFs having the maximum number of representations.
- New algorithms for generating a random representation and for counting the number of possible representations.
- Secure deployment of key splitting schemes and the ECDSA and ECMQV algorithms.

Chapter 2

Elliptic Curve Cryptosystems and Side-Channel Attacks

In this chapter, we present an overview of elliptic curve cryptosystems (ECCs). The core operation of ECCs is the elliptic curve scalar multiplication (ECSM). We briefly summarize the different algorithms employed to carry this operation that would be useful for the remainder of the thesis.

Thereafter, we give a background on how the original ECSM algorithms are susceptible to side-channel analysis (SCA) attacks. The countermeasures to these attacks that initiated the work in this thesis are briefly introduced. For a more detailed overview on SCA attacks and their countermeasures, we refer the reader to Appendix A. The survey presented by Avanzi [Ava05] was an inspiration to that appendix.

2.1 Elliptic Curve Cryptosystems

Let K be a finite field and E be an elliptic curve (EC) over K defined by the following Weierstrass equation

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

where $a_i \in K$ and $\Delta \neq 0$, where Δ is the *discriminant* of E and is defined in [HMV04, Section 3.1].

Let L be an extension field of K. Then $E(L)$ denotes the set of L-rational points (x, y) on E, where $(x, y) \in L \times L$ and satisfy (2.1), together with the *point at infinity* $\mathcal{O}$. The addition of two points on the curve is performed using a *chord-and-tangent* rule. $E(L)$ and this addition operation form an abelian group where $\mathcal{O}$ is the identity.

The point addition operation consists of finite field operations carried in the underlying field K. In the remainder of this thesis, we denote the field inversion by I, the multiplication by M, the squaring by S. The point addition is denoted by A. When the two operands of the addition are the same point, the operation is referred to as point doubling and is denoted by D.

2.1.1 Elliptic curves over prime fields

If $K = \mathbb{F}_p$, where $p > 3$ is a prime, (2.1) can be simplified to[1]

$$E : y^2 = x^3 + ax + b, \tag{2.2}$$

where a and $b \in \mathbb{F}_p$. The discriminant of this curve is $\Delta = -16(4a^3 + 27b^2)$. The *negative* of a point $P = (x, y)$ is $-P = (x, -y)$ such that $P + (-P) = \mathcal{O}$.

[1] This simplification is generally applicable when the characteristic of K is not 2 or 3.

The *affine* coordinate ($\mathcal{A}$) representation of a point $P = (x, y)$ can be replaced by *projective* coordinates representations in order to render the point addition and doubling operations less costly in terms of field operations. The following representations are the best known

- *standard* (*homogeneous*) projective coordinates ($\mathcal{P}$); the projective point $(X : Y : Z)$, $Z \neq 0$, corresponds to the affine point $(X/Z, Y/Z)$, $\mathcal{O}$ corresponds to $(0 : 1 : 0)$ and the negative of $(X : Y : Z)$ is $(X : -Y : Z)$.

- *Jacobian* projective coordinates ($\mathcal{J}$); the projective point $(X : Y : Z)$, $Z \neq 0$, corresponds to the affine point $(X/Z^2, Y/Z^3)$, $\mathcal{O}$ corresponds to $(0 : 1 : 0)$ and the negative of $(X : Y : Z)$ is $(X : -Y : Z)$.

- *Chudnovsky* coordinates ($\mathcal{C}$); the Jacobian point $(X : Y : Z)$ is represented as $(X : Y : Z : Z^2 : Z^3)$.

The number of field multiplications (M), squarings (S) and inversions (I) needed to perform the point addition and doubling are summarized in Table 2.1. In that table, it is assumed that $a = -3$ in (2.2). Theorem 3.15 of [HMV04] confirms that this assumption is without much loss of generality since about half of all isomorphism classes of elliptic curves over $\mathbb{F}_p$ have a representative with $a = -3$. Therefore, multiplication by 3 or multiplication and division by any power of 2 are not taken into account. We refer the reader to Section 3.2 of [HMV04] for the point addition and doubling formulas in the aforementioned coordinate systems.

It is estimated that the computational cost of the inversion and the squaring operations compared to the multiplication is $1I = 80M$ and $1S = 0.8M$ [HMV04]. Therefore, the use of Jacobian coordinates for point doubling and mixed coordinate for point addition is recommended [CMO98], as will be further explained later. Note that the use

Table 2.1: Field operations count for point addition and point doubling on E over $\mathbb{F}_p$

Doubling		General addition		Mixed coordinates add.	
$2\mathcal{A} \to \mathcal{A}$	$1I, 2M, 2S$	$\mathcal{A} + \mathcal{A} \to \mathcal{A}$	$1I, 2M, 1S$	$\mathcal{J} + \mathcal{A} \to \mathcal{J}$	$8M, 3S$
$2\mathcal{P} \to \mathcal{P}$	$7M, 3S$	$\mathcal{P} + \mathcal{P} \to \mathcal{P}$	$12M, 2S$	$\mathcal{J} + \mathcal{C} \to \mathcal{J}$	$11M, 3S$
$2\mathcal{J} \to \mathcal{J}$	$4M, 4S$	$\mathcal{J} + \mathcal{J} \to \mathcal{J}$	$12M, 4S$	$\mathcal{C} + \mathcal{A} \to \mathcal{C}$	$8M, 3S$
$2\mathcal{C} \to \mathcal{C}$	$5M, 4S$	$\mathcal{C} + \mathcal{C} \to \mathcal{C}$	$11M, 3S$		

of projective coordinates, especially the Chudnovsky coordinates, is at the expense of storing more coordinates. The use of mixed coordinates will be further illustrated in Section 2.1.3.

2.1.2 Elliptic curves over binary fields

If $K = \mathbb{F}_{2^m}$, (2.1) can be simplified to

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

where a and $b \in \mathbb{F}_{2^m}$. The discriminant of this curve is $\Delta = b$ and the negative of a point $P = (x, y)$ is $-P = (x, x + y)$. Such a curve is known as *non-supersingular*.

Standard and Jacobian projective coordinates are used to represent points on this type of curves in the same way as on the prime curves with the difference that the negative of $(X : Y : Z)$ is $(X : X + Y : Z)$. Another possible representation of points is using the *López-Dahab* ($\mathcal{LD}$) projective coordinates where the projective point $(X : Y : Z)$, $Z \neq 0$, corresponds to the affine point $(X/Z, Y/Z^2)$, $\mathcal{O}$ corresponds to $(1 : 0 : 0)$ and the negative of $(X : Y : Z)$ is $(X : X + Y : Z)$.

The field operations count is summarized in Table 2.2. In that table, it is assumed that $a \in \{0, 1\}$. Theorem 3.18(ii) of [HMV04] confirms that this restriction is without much loss of generality since all isomorphism classes of elliptic curves over $\mathbb{F}_{2^m}$ have a

representative with $a \in \{0, \nu\}$, where $\nu \in \mathbb{F}_{2^m}$ and $Tr(\nu) = 1$ [2]. The cost of inversion is assumed to be between $I = 5M$ and $I = 8M$. Also, for most practical implementations, the cost of squaring in $\mathbb{F}_{2^m}$ is negligible compared to multiplication and, hence, is not taken into account.

Table 2.2: Field operations count for point addition and point doubling on E over $\mathbb{F}_{2^m}$

Doubling		General addition		Mixed coordinates add.	
$2\mathcal{A} \to \mathcal{A}$	$1I, 2M$	$\mathcal{A} + \mathcal{A} \to \mathcal{A}$	$1I, 2M$		
$2\mathcal{P} \to \mathcal{P}$	$7M$	$\mathcal{P} + \mathcal{P} \to \mathcal{P}$	$13M$	$\mathcal{P} + \mathcal{A} \to \mathcal{P}$	$12M$
$2\mathcal{J} \to \mathcal{J}$	$5M$	$\mathcal{J} + \mathcal{J} \to \mathcal{J}$	$14M$	$\mathcal{J} + \mathcal{A} \to \mathcal{J}$	$10M$
$2\mathcal{LD} \to \mathcal{LD}$	$4M$	$\mathcal{LD} + \mathcal{LD} \to \mathcal{LD}$	$14M$	$\mathcal{LD} + \mathcal{A} \to \mathcal{LD}$	$8M$

2.1.3 Elliptic Curve Scalar Multiplication (ECSM)

Scalar multiplication in the group of points of an elliptic curve is analogous to exponentiation in the multiplicative group of integers modulo a fixed integer. Thus, it is the fundamental operation in EC-based cryptographic systems. The scalar multiplication, denoted kP, is the result of adding the point P to itself k times, where k is a positive integer, that is

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

and $-kP = k(-P)$. u is said to be the *order* of P if u is the smallest integer such that $uP = \mathcal{O}$.

Let E be an elliptic curve defined over $\mathbb{F}_q$. The *order* of a curve E over $\mathbb{F}_q$ is the number of points in $E(Fq)$ and is denoted by $\#E(\mathbb{F}_q)$. From Hasse's theorem [HMV04, Theorem 3.7], $\#E(\mathbb{F}_q) \approx q$. In this thesis, we focus on elliptic curves defined in the

[2] $Tr : \mathbb{F}_{2^m} \to \mathbb{F}_2$ is the *trace* function on $\mathbb{F}_{2^m}$ defined by $Tr(c) = c + c^2 + c^{2^2} + \cdots + c^{2^{m-1}}$.

standards such as those defined in [NIST]. The order of these curves is characterized by the form $\#E(\mathbb{F}_q) = h \cdot u$, where u is prime and h, the *cofactor*, is small. That is, $h = 1$ for the curves defined over prime fields and $h = 2$ or 4 for curves defined over binary fields including Koblitz curves (cf. Section 2.1.4). Hence, u is the order of the points in the *main subgroup*.

In many applications, the scalar k is a short-term (*ephemeral*) or long-term (*private*) secret (*key*). From now on, we will always assume that k is a n-bit integer, where n is the bit length of u, the order of the group of points of interest in an ECC. Also the point P may be fixed (*e.g.*, the base point of the ECC) or unknown a priori.

Let $(k_{n-1}, k_{n-2}, \ldots, k_1, k_0)_2$ be the binary representation of k, *i.e.*, $k_i \in \{0, 1\}$ for $0 \leq i < n - 1$. Thus,

$$\begin{aligned} kP &= \left(\sum_{i=0}^{n-1} k_i 2^i \right) P \\ &= 2(2(\cdots 2(2(k_{n-1}P) + k_{n-2}P) + \cdots) + k_1 P) + k_0 P \qquad (2.4) \\ &= (k_{n-1} 2^{n-1} P) + \cdots (k_1 2P) + (k_0 P) \qquad (2.5) \end{aligned}$$

Hence, kP can be computed using the straightforward *double-and-add* approach in n iterations. In fact, there are two algorithms that can be used. Algorithm 2.1, which corresponds to the expansion in (2.4), scans the bits of the scalar k from left to right, *i.e.*, from the most significant bit to the least significant one. Algorithm 2.2, corresponding to (2.5), scans the bits of k from right to left. These algorithms are analogous to the *square-and-multiply* algorithms employed in exponentiation-based cryptosystems. These algorithms are also known as the *binary algorithms* [Knu73, Section 4.6.3].

Algorithm 2.1. Left-to-Right Double-and-Add Algorithm

INPUT: $k = (k_{n-1}, \ldots, k_0)_2$ and $P \in E(\mathbb{F}_q)$.

OUTPUT: kP.

1. $Q \leftarrow \mathcal{O}$.
2. for i from $n-1$ down to 0 do
   - 2.1 $Q \leftarrow 2Q$.
   - 2.2 if $(k_i = 1)$ then $Q \leftarrow Q + P$.
3. Return(Q).

Algorithm 2.2. Right-to-Left Double-and-Add Algorithm

INPUT: $k = (k_{n-1}, \ldots, k_0)_2$ and $P \in E(\mathbb{F}_q)$.

OUTPUT: kP.

1. $Q \leftarrow \mathcal{O}$; $R \leftarrow P$.
2. for i from 0 to $n-1$ do
   - 2.1 if $(k_i = 1)$ then $Q \leftarrow Q + R$.
   - 2.2 $R \leftarrow 2R$.
3. Return(Q).

The expected number of point addition (A) and point doubling (D) operations performed in the binary algorithm (left-to-right or right-to-left) is

$$(n-1)\ D + \frac{n}{2}\ A.$$

If affine coordinates are used, the field operation count is

$$2.5n\ S + 3n\ M + 1.5n\ I.$$

Algorithm 2.1 is usually preferred since one of the addition operands is the base point P which is constant through the algorithm. This has the advantage of saving a register if P is a fixed point known a priori. Moreover, it allows the use of mixed coordinates addition. That is, when one of the operands to the addition operation is fixed, the Z-coordinate of that operand is set to and remains 1, this reduces the number of field multiplications needed to perform the point addition as illustrated in Table 2.1 and Table 2.2.

Hence if Q is stored in Jacobian coordinates and P in affine coordinates, Jacobian coordinates can be used for doubling in Algorithm 2.1 and mixed Jacobian-affine for the

addition. The field operation count is then

$$8n\ M + 5.5n\ S + (1\ I + 3\ M + 1\ S),$$

where the last three terms are needed to convert the resulting point back to affine coordinates. The binary algorithms are suitable for unknown point P.

To speed up this algorithm, different variants have been proposed. We only included the following ones that would serve as a background for the remainder of the thesis. For the reader's interest, we have included other algorithms in Appendix A. For each of the following ECSM algorithms, we included the storage cost and the computation cost. The latter consists of the point and/or field operations count in the expected running time and the precomputation phase, if any. We also mention any specific coordinate selection.

Non-Adjacent form (NAF)

The key k can be represented in *Non-adjacent Form* (NAF) $k' = (k'_n, \ldots, k'_1, k'_0)_2$, where $k'_i \in \{0, \pm 1\}$ and no two consecutive digits are non zero; that is, $k'_{i+1}k'_i = 0$ for $i \geq 0$ [Rei60; Sol00]. The NAF of an integer is unique and is at most one digit longer than its binary representation. The average density of nonzero digits among all NAFs of length n is $n/3$.

This key representation requires the slight modification of the binary algorithm that is to *subtract*, rather than add, P when $k'_i = -1$, *i.e.*, to add $-P$. This is advantageous for ECCs since the negative of a point can be obtained with a minor cost as we mentioned in Section 2.1, *e.g.*, a modular negation for curves over prime fields and a bit-wise XOR operation for curves over binary fields.

Hereafter, we will refer to the digits of a NAF representation as *signed bits* or, in short, *sbits*. Different forms of algorithms have been proposed in the literature to produce the

NAF of an integer. In [Rei60], it is modeled as a look-up table. In [MO90], it is modeled as an automaton that does not directly output the sbits but rather performs the point addition and doubling operations along with the transitions. In [Sol00], it is based on the same idea as producing the binary representation of an integer using division by 2. In Chapter 3, we will reconcile all three algorithms.

The cost of the binary algorithm using the NAF representation of k is

Storage: key k recoded ($2n$ bits, since each digit would be represented by 2 bits).

Expected running time: $(n-1)\ D + \frac{n}{3}\ A$.

Coordinate selection: same as for Algorithm 2.1.

Window methods

This method is sometimes referred to as *m−ary method*. There are different versions of window methods [MOC97; Sol00]. What is common among them is that, if the window width is w, some multiples of the point P up to $(2^w - 1)P$ are precomputed and stored and k is processed w bits at a time. k is recoded to the radix 2^w. k can be recoded in a way so that the average density of the nonzero digits in the recoding is $1/(w + \xi)$, where $0 \leq \xi \leq 2$ depends on the algorithm.

Let the number of precomputed points be t, in the precomputation stage, each point requires either a doubling or an addition to be computed also depending on the algorithm. In the main loop, the accumulator point will be doubled at least w times, then a precomputed point corresponding to the current digit of the key will be added to the accumulator.

When a window technique is referred to as *sliding window*, this means that the 0s in k are skipped so that the windows formed all contain odd integers; this cuts the number of precomputed points to about one half and decreases the number of additions in the main

loop. Another cut by half to the precomputed points is possible if the recoded windows are formed of both negative and positive digits.

This ECSM method is suitable for unknown or fixed point P. The cost is

Storage: t points, where $2^{w-2} \leq t \leq 2^w - 1$, depending on the algorithm.

Precomputation: t point operations (A or D).

Expected running time: $(n-1)\ D + \frac{n}{w+\xi}\ A$, where $0 \leq \xi \leq 2$ depending on the algorithm. Note that the number of doubling is between $n-w$ and $n-1$.

Coordinate selection: The accumulator point Q is represented in Jacobian coordinates. The precomputed points are represented in either affine or Chudnovsky coordinates (to save the inversions in the latter case), then the addition in the loop will be performed in mixed Jacobian-affine or Jacobian-Chudnovsky coordinates respectively, and the doubling will be in Jacobian coordinates as before.

Simultaneous multiple point multiplication

This method is used to compute $kP + lS$ where P may be a known point. This algorithm was referred to as *Shamir's trick* in [ElG85][3]. If k and l are n-bit integers, then their binary representations are written in a $2 \times n$ matrix called the *exponent array*. Given width w, the values $iP + jS$ are calculated for $0 \leq i, j < 2^w$. Now the algorithm performs $d = \lceil n/w \rceil$ iterations. In every iteration, the accumulator point is doubled w times and the current $2 \times w$ window over the exponent array determines the precomputed point that is to be added to the accumulator.

Algorithm 2.3. Simultaneous multiple point multiplication (Shamir-Strauss method)

INPUT: Window width w, $d = \lceil n/w \rceil$, $k = (K_{d-1}, \ldots, K_1, K_0)_{2^w}$, $l = (L_{d-1}, \ldots, L_1, L_0)_{2^w}$, and $P, S \in E(\mathbb{F}_q)$.

[3] Also according to [Ber01], it is originally due to Straus [Str64].

OUTPUT: $kP + lS$.

1. *Precomputation.* Compute $iP + jS$ for all $i, j \in [0, 2^w - 1]$.
2. $Q \leftarrow K_{d-1}P + L_{d-1}S$.
3. for i from $d-2$ down to 0 do
   3.1 $Q \leftarrow 2^w Q$.
   3.2 $Q \leftarrow Q + (K_i P + L_i S)$.
4. Return(Q).

Storage: $2^{2w} - 1$ points. For $w = 1$, 3 points. For $w = 2$, 15 points.

Precomputation: $(2^{2(w-1)} - 2^{w-1})\ D + (3 \cdot 2^{2(w-1)} - 2^{w-1} - 1)\ A$.

For $w = 1$, 1 A.

For $w = 2$, 1 $D + 11\ A$.

Expected running time: $(d-1)w\ D + \left(\frac{2^{2w}-1}{2^{2w}}d - 1\right)\ A$.

For $w = 1$, $(n-1)\ D + \left(\frac{3}{4}n - 1\right)\ A$.

For $w = 2$, $(n-1)\ D + \left(\frac{15}{32}n - 1\right)\ A$.

Using *sliding* windows can save about $\frac{1}{4}$ of the precomputed points and decrease the number of additions to $\frac{n}{w+(1/3)}$, which is about 9% saving for $w \in \{2, 3\}$.

Note that if $S = 2^{\frac{n}{2}}P$, then this method is the comb method with $w = 2$ (cf. Algorithm A.3).

Interleaving method

This method is also a multiple point multiplication method, that is we want to compute $\sum k^j P_j$ for points P_j and integers k^j. In the comb and simultaneous multiplication methods, each of the precomputed values is a sum of the multiples of the input points. In the interleaving method, each precomputed value is simply a multiple of one of the

input points. Hence, the required storage and the number of point additions at the precomputation phase is decreased at the expense of the number of point additions in the main loop.

This method is flexible in that each k^j can have a different representation, *e.g.*, different window size, as if a separate execution of a window method is performed for each $k^j P_j$ with the doubling step performed jointly on a common accumulator, as shown in [HMV04]. As an illustration, which is also useful for the remainder of this thesis, we provide the following algorithm that computes $kP + lS$ where both k and l are represented to the same base 2^w.

Algorithm 2.4. Interleaving method

INPUT: Window width w, $d = \lceil n/w \rceil$, $k = (K_{d-1}, \ldots, K_1, K_0)_{2^w}$, $l = (L_{d-1}, \ldots, L_1, L_0)_{2^w}$, and $P, S \in E(\mathbb{F}_q)$.

OUTPUT: $kP + lS$.

1. *Precomputation.* Compute iP and iS for all $i \in [0, 2^w - 1]$.
2. $Q \leftarrow K_{d-1}P$.
3. $Q \leftarrow Q + L_{d-1}S$.
4. for i from $d - 2$ down to 0 do
   - 4.1 $Q \leftarrow 2^w Q$.
   - 4.2 $Q \leftarrow Q + K_i P$.
   - 4.3 $Q \leftarrow Q + L_i S$.
5. Return(Q).

Storage: $2^{w+1} - 2$ points.

Precomputation: $2(w-1)\ D + 2(2^w - w - 1)\ A$.

Expected running time: $w(d-1)\ D + (2d-1)\frac{2^w-1}{2^w}\ A$.

In general, if different basis and/or representations are used for k and l, we have

Storage: $2t$ points, where $2^{w-2} \leq t \leq 2^w - 1$ depending on the particular window algorithm used as discussed in Section 2.1.3.

Precomputation: $2t$ point operations (A or D).

Expected running time: $(n-1)\ D + 2\frac{n}{w+i}\ A$, where $1 \leq i \leq 2$, depending on the algorithm.

2.1.4 Koblitz curves

Koblitz curves [Kob92]—originally named *anomalous binary curves*—are the curves E_a, $a \in \{0, 1\}$, defined over $\mathbb{F}_2$

$$E_a : y^2 + xy = x^3 + ax^2 + 1, \tag{2.6}$$

which is a special case of (2.3) where $b = 1$.

$E_a(\mathbb{F}_{2^m})$ is the group of $\mathbb{F}_{2^m}$-rational points on E_a. Let $\mu = (-1)^{1-a}$, that is $\mu \in \{-1, 1\}$. The order of the group is computed as

$$\#E_a(\mathbb{F}_{2^m}) = 2^m + 1 - V_m, \tag{2.7}$$

where $\{V_h\}$ is the Lucas sequence defined by

$$V_0 = 2, \quad V_1 = \mu \quad \text{and} \quad V_{h+1} = \mu V_h - 2V_{h-1} \quad \text{for } h \geq 1.$$

The value of m is chosen to be a prime number so that $\#E_a(\mathbb{F}_{2^m}) = h \cdot u$ is very nearly prime, that is $u > 2$ is prime and $h = 3 - \mu$.

The main advantage of Koblitz curves when used in public-key cryptography is that scalar multiplication of the points in the main subgroup, the group of order u, can be performed without the use of point doubling operations. This is due to the following

property. Since these curves are defined over $\mathbb{F}_{2^m}$, then if $P = (x, y)$ is a point on E_a, then the point (x^2, y^2) is on the curve, as well. That is the Frobenius (squaring, in this case) endomorphism $\tau : E_a(\mathbb{F}_{2^m}) \rightarrow E_a(\mathbb{F}_{2^m})$ defined by

$$(x, y) \mapsto (x^2, y^2), \quad \mathcal{O} \mapsto \mathcal{O}$$

is well defined. It can also be verified by point addition on E_a that

$$(x^4, y^4) + 2(x, y) = \mu \cdot (x^2, y^2).$$

Hence, the squaring map can be considered as a multiplication by the complex number τ satisfying

$$\tau^2 + 2 = \mu\tau, \tag{2.8}$$

that is

$$\tau = \frac{1}{2}\ (\mu + \sqrt{-7}).$$

The norm of τ is 2. Thus, it is beneficial to represent the key k as an element of the ring $\mathbb{Z}[\tau]$, *i.e.*,

$$k = \sum_{i=0}^{l-1} \kappa_i \tau^i \tag{2.9}$$

for some l. We can therefore carry the scalar multiplication kP of a point P on E_a more efficiently by replacing the doubling operation in the double-an-add algorithm by the squaring map.

In [Sol00], Solinas has shown how to represent k as in (2.9) in its *τ-adic non adjacent form* (τNAF) where $\kappa_i \in \{-1, 0, 1\}$ and $\kappa_{i+1}\kappa_i = 0$ for $i \geq 0$—abusing the notation, we will refer to κ_i as a sbit. However, this results in $l \approx 2m$. Therefore, he proposed a *reduced τ-adic non adjacent form* (RTNAF) for k where k is reduced modulo $\delta = (\tau^m - 1)/(\tau - 1)$, hence $l = m + a$. He has proved that in a τNAF representation the number of 0s is $\frac{2}{3}$ on average. He also mentioned that 1 and -1 are equally likely on average.

2.2 Power and Electromagnetic Analysis Attacks on ECCs

As mentioned before, the elliptic curve scalar multiplication $Q = kP$, where both P and Q are points on the curve and k is an integer, is the fundamental computation performed in ECCs. Usually both P and Q are public information and k is the secret key stored securely in the cryptosystem. The security of the system lies in the difficulty of extracting k from P and Q, which is the hard problem known as *EC discrete logarithm problem* (ECDLP).

However, the mathematically proved security of a cryptosystem does not imply its implementation security against side-channel attacks. Among those attacks are those that monitor the power consumption and/or the electromagnetic emanations of a device, *e.g.*, a smart card or a handheld device, and can infer important information about the instructions being executed or the operands being manipulated at a specific instant of interest.

These attacks are broadly divided into two categories; *simple* and *differential* analysis attacks. We will refer to the former category as SPA attacks and the latter as DPA attacks. Though SPA and DPA are the acronyms for simple power analysis and differential power analysis, respectively, are used in this thesis to include simple and differential electromagnetic analysis as well due to their extensive usage in the literature. Also, in subsequent discussions, we may only focus on power analysis attacks, since the countermeasures that we are interested in were proposed to prevent information leakage on both side-channels, the power consumption and the electromagnetic emanations.

Power analysis attacks use the fact that the instantaneous power consumption of a hardware device is related to the instantaneous computed instructions and the manipulated data. The attacker could measure the power consumption during the execution of a

cryptographic algorithm, store the waveform using a digital oscilloscope and process the information to learn the secret key. Kocher *et al.*, in [KJJ99], first introduced this type of attack on smart cards performing the DES operation. Then Messerges *et al.* [MDS02] augmented Kocher's work by providing further analysis and detailed examples of actual attacks they mounted on smart cards.

In general, SPA attacks are those based on retrieving valuable information about the secret key from a *single* leaked information—power consumption or electromagnetic emanation—trace. On the other hand, DPA attacks generally include all attacks that require more than one such trace along with some statistical analysis tools to extract the implicit information from those traces.

2.2.1 SPA Attack on ECCs and its Countermeasures

Coron [Cor99] has transferred the power analysis attacks to ECCs and has shown that an unaware implementation of EC operations can easily be exploited to mount an SPA attack. Monitoring of the power consumption enables us to visually identify large features of an ECC implementation such as the main loop in Algorithms 2.2 and 2.1. Moreover, it may also enable to recognize the exact instruction that has been executed. For example, if the difference in power consumption between point doubling (D) and point addition (A) is obvious in their respective power traces, then, by investigating one power trace of a complete execution of a double-and-add algorithm, the bits of the scalar k are revealed. That is, whenever a D is followed by A, the corresponding bit is $k_i = 1$, otherwise if D is followed by another D, then $k_i = 0$. This sequence of point operations is referred to as the DA sequence.

As a countermeasure, Coron proposed making the DA sequence uniform across the executions of the ECSM algorithm, independent of the value of k_i. Thus, he proposed

performing the point addition in every iteration and collecting its result only if $k_i = 1$, otherwise, the point addition is considered a *dummy* operation. The resulting algorithms are referred to as the *double-and-add-always* algorithms (cf. Algorithms A.4 and A.5). Another better approach is to use the Montgomery ladder [Mon87; Gou03; IT02] (see Algorithms A.6 and A.7) which has the advantage that the point addition is never dummy. Both the double-and-add algorithms and the Montgomery ladder process the key on a bit level.

Window methods process the key on a digit (window) level. The basic version of this method, that is where $\xi = 0$ in Section 2.1.3, is inherently uniform since in most iterations, w D operations are followed by 1 A, except for possibly when the digit is 0. Therefore, *fixed-sequence window methods* were proposed [Möl01; OT03; Thé06] in order to recode the digits of the key such that the digit set does not include 0. We will discuss the recoding that appeared in [Thé06] in detail in Section 6.4.1.

2.2.2 DPA Attack on ECCs and its Countermeasures

When the relation between the instructions executed by a cryptographic algorithm and the key bits is not directly observable from the power signal, an attacker can apply *differential power analysis* (DPA). DPA attacks are in general more threatening and more powerful than SPA attacks because the attacker does not need to know as many details about how the algorithm was implemented. The technique also gains strength by using statistical analysis and digital signal processing techniques on a large number of power consumption signals to reduce noise and to amplify the *differential* signal. The latter is indicated by a peak, if any, in the plot of the processed data. This peak appears only if the attacker's guess of a bit or a digit of the secret key is correct. The attacker's goal is to retrieve partial or full information about a long-term key that is employed in

several ECSM executions.

As for the SPA attack, Kocher *et al.* were the first to introduce the DPA attack on a smart card implementation of DES [KJJ99]. Techniques to strengthen the attack and a theoretical basis for it were presented by Messerges *et al.* in [MDS99a; MDS02]. Coron applied the DPA attack to ECCs [Cor99]. We provide more details about first-order DPA attack methodology on ECCs, other variants and the different countermeasures that were proposed in Section A.3 and Section A.4.

As we concluded in Section A.5, in order to resist DPA attacks, it is important to randomize the *value* of the long-term key involved in the ECSM across the different executions. Some of the countermeasures that were based on randomizing the key representation [OA01; HM02a] were proven to be inadequate since the intermediate point computed in the accumulator Q at a certain iteration remained one of two possible values [FMPV04]. The constancy of the value of this intermediate point is an integral part in the success of first-order DPA attacks. However, these randomization algorithms were of mathematical interest to us since their outcome was a random *binary signed digit* (BSD) representation of the key. We were interested in studying the properties of this representation (Chapter 4) as well as the application of the randomization algorithms to the τNAF of an integer in order to produce a random τ-adic representation (Chapter 5).

A potential DPA countermeasure is known as *key splitting* [Joy05]. It is based on randomly splitting the key into two parts such that each part is different in every ECSM execution. An additive splitting using subtraction is attributed to Clavier and Joye [CJ01][4]. It is based on computing

$$kP = (k - r)P + rP, \tag{I}$$

[4]The authors mention that the idea of splitting the data was abstracted in [CJRR99].

where r is a n-bit random integer, that is, of the same bit length as k.

Alternatively, Ciet and Joye [CJ03] suggest the following additive splitting using division, that is, k is written as

$$k = \lfloor k/r \rfloor r + (k \bmod r).$$

Hence, if we let $k_1 = (k \bmod r)$, $k_2 = \lfloor k/r \rfloor$ and $S = rP$, we can compute

$$kP = k_1 P + k_2 S, \tag{II}$$

where the bit length of r is $n/2$. They also suggest that (II) should be evaluated with Shamir-Strauss method as in Algorithm 2.3. However, they did not mention whether the same algorithm should be used to evaluate (I).

The following multiplicative splitting was proposed by Trichina and Bellezza [CQS03] where r is a random integer invertible modulo u, the order of P. The scalar multiplication kP is then evaluated as

$$kP = [kr^{-1} \pmod u](rP). \tag{III}$$

To evaluate (III), two scalar multiplications are needed; first $R = rP$ is computed, then $kr^{-1}R$ is computed.

In Chapter 6, we will further discuss the different splitting methods and their SPA- and DPA-resistant evaluation.

2.3 Conclusion

In this chapter, we have presented a background on elliptic curve cryptosystems (ECCs) along with the different methods used to compute the scalar multiplication (ECSM), which is the core operation of ECCs, and the various costs associated with them.

We have also provided a brief background on simple (SPA) and differential (DPA) power and electromagnetic analysis attacks on the classical ECSM algorithms. We have included the countermeasures to both types of attacks that were of interest to us and that serve as the basis for the work presented in this thesis.

Chapter 3

Analysis of the BSD Randomization Algorithms

As a countermeasure to DPA attacks, Oswald and Aigner [OA01] and Ha and Moon [HM02a] have, each, proposed a randomization technique to the binary algorithm (Algorithm 2.1). Though not explicitly mentioned by the former authors, both techniques yield a *binary signed digit* (BSD) recoded version of the key, which is the scalar k in Algorithm 2.1. Studying the similarities and the differences between the two techniques has been of mathematical interest to us.

In this chapter, we investigate the randomness of the recoded keys resulting from the Oswald-Aigner (OA) and the Ha-Moon (HM) algorithms, *i.e.*, whether or not each can produce all possible BSD representations of an integer. We prove that this is true for the HM technique while not possible with the OA technique.

We also present the complexity (average case) analysis of each algorithm using both Markov chain model and grammatical specification method. We find that the OA random-

ization method incurs slightly more point additions on average than the binary algorithm. Our analytical result is different from the experimental one provided by the authors. As for the HM algorithm, we use the grammatical specification method to confirm the results obtained by the authors using the Markov chain model.

3.1 The Proposed Algorithms

In this section, we present a brief overview of the OA and the HM algorithms.

3.1.1 Oswald-Aigner Randomized Automaton

As a countermeasure to DPA attacks on EC scalar multiplication, Oswald and Aigner [OA01] proposed a randomization technique to Morain and Olivos' automata [MO90]. Automaton 2 was proposed by the latter authors in order to speed up the elliptic curve scalar multiplication by implicitly generating the non-adjacent form (NAF) of the scalar. Figure 3.2 shows the OA randomized version of Automaton 2 (Figure 3.1). In the following we discuss the randomized automaton.

The inputs to the automaton are the key k and a point P on the elliptic curve. The output is $Q = kP$. The initial condition is $Q = \mathcal{O}$. The bits of $k = (k_{n-1}, \ldots, k_0)_2$ are scanned from the least significant to the most significant (from right to left). States are represented by circles and transitions by arrows along with the corresponding scanned bit. Each bit scanned, which we will refer to as k_{i+1}, triggers a transition from a state to another. In Automaton 2, each state is labeled according to the sequence of bits that led to that state starting from state 0. This labeling scheme was preserved in the randomized automaton. Also in both automata, the original and the randomized, the most significant bit of k, k_{n-1}, is assumed to be always 1, whence the exit transitions.

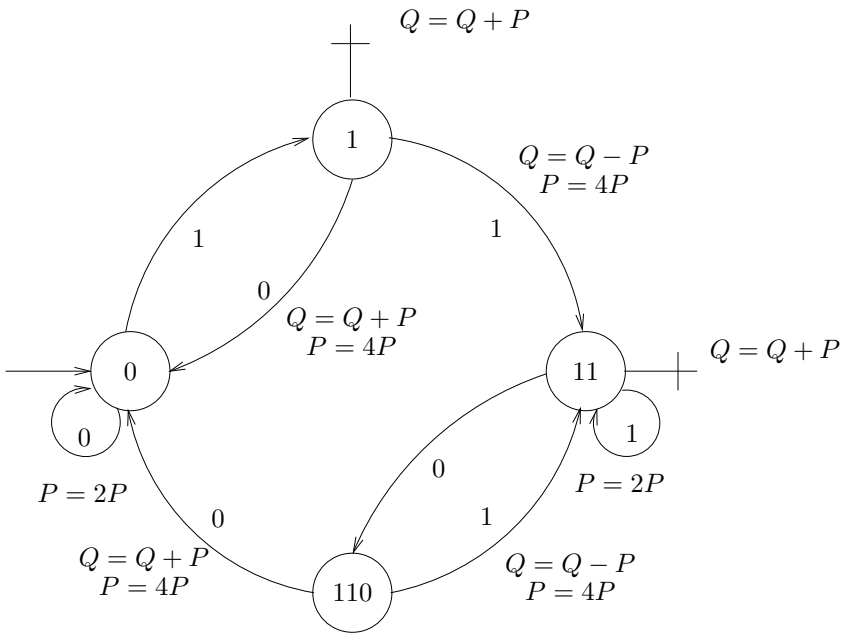

Figure 3.1: Automaton 2.

For every state other than state 0, when the bit scanned $k_{i+1} = 1$, a random variable e is drawn. If $e = 0$, the algorithm follows the original path of Morain-Olivos' Automaton 2 (dashed arrows) and performs a point subtraction and a point doubling–or just a point doubling. Otherwise, it proceeds as in the binary algorithm (dotted-dashed arrows) to perform a point addition and point doubling.[1] Transitions that are not affected by the randomization are depicted as solid arrows. Due to the random variable e, a different sequence of operations is performed each time the algorithm is executed for the same key k. As a side effect, the algorithm can be interpreted as recoding k into one of its *binary*

[1]We have interchanged the values of e from the original paper for the transitions departing from state 11 when the bit scanned is 1. This is in order to make them consistent with the other transitions that depend on the value of e.

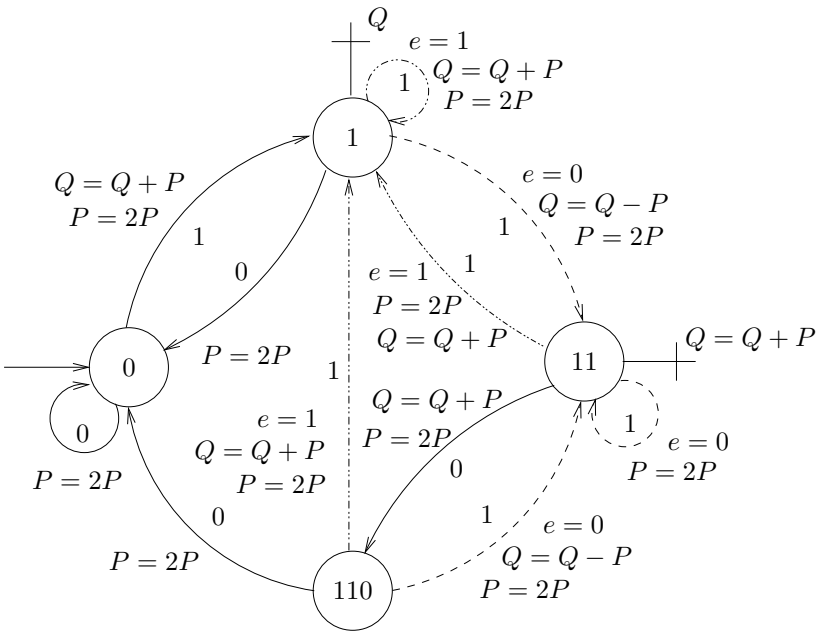

Figure 3.2: Randomized version of Automaton 2.

signed digit (BSD) representations. A BSD representation k' of an integer $k \in [0, 2^n - 1]$ is a base-2 representation denoted by $(k'_n, k'_{n-1}, \ldots, k'_0)_{\text{BSD}}$ where $k'_i \in \{-1, 0, 1\}$. Note that Automaton 2, and hence, its randomized version, may produce a BSD representation that is one sbit longer than the binary representation. We will call the k'_is *signed bits*, or *sbits* for short, and -1 will be written as $\overline{1}$. The sbits of k' are not mentioned explicitly by the authors but can be deduced from the operations performed:

- A transition where $P = 2P$, *i.e.*, a point doubling, is performed indicates that a 0 was prepended to k'.
- A transition where a point addition (resp. subtraction) and a point doubling are performed indicates that a 1 (resp. $\overline{1}$) was prepended to k'.

- Only in the transition from state 11 to state 1 when $k_{i+1} = 1$ and $e = 1$, a point doubling is performed before the point addition. This is equivalent to prepending a 0 and then a 1 to k'. This specific transition is a flaw in the algorithm that was exploited by the cryptanalysis of the randomized version of Automaton 1 presented by Okeya and Sakurai in [OS02a] and that of Automaton 2 presented by Walter in [Wal04a].

3.1.2 Ha-Moon Randomized Algorithm

The DPA countermeasure provided by Ha and Moon [HM02a] was based on inserting a random decision in Reitwiesner's canonical recoding algorithm [EK94; Rei60]. The input to both algorithms is the binary representation of k and the output is k', the NAF of k in the case of Reitwiesner's algorithm, and a random BSD representation in the case of the HM algorithm. Both algorithms scan the bits of k from right to left. There is an auxiliary carry variable c_i which is initially set to 0. Based on the current bit k_i, the current auxiliary carry c_i and the next scanned bit k_{i+1}, the output sbit k'_i and the next auxiliary carry c_{i+1} are determined.

Ha and Moon noticed that the outcomes of the algorithm, the auxiliary carry c_{i+1} and the NAF sbit k'_i, have the value $c_{i+1}2^{i+1} + k'_i 2^i = 2^i(2c_{i+1} + k'_i)$. Therefore, whenever $c_{i+1}k'_i = 01$ they can be equivalently represented as $c_{i+1}k'_i = 1\overline{1}$ and vice-versa. To insert randomness in the algorithm, they generated a random n-bit integer $r = (r_{n-1}, r_{n-2}, \ldots, r_0)_2$. The random bits of r determine the decision to be taken, if applicable, for the algorithm outcome with each scanned bit.

Both algorithms were presented by their authors as look-up tables; we represent them in this chapter as automata to show the point operations performed with each transition, instead of storing k' and then executing the binary algorithm. Also this representation

makes the HM algorithm easily comparable with the analogous OA algorithm discussed previously. The exit states of the automata are as shown assuming that a 0 is prepended to k, that is, $k_n = 0$. The arrows are labeled $k_{i+1}/c_{i+1}, k'_i$. The dashed arrows in Figure 3.3 represent the transitions that were randomized by the HM algorithm as in Figure 3.4.

We notice that the behavior of Reitwiesner's algorithm as presented in Figure 3.3 is identical to that of Automaton 2 in Figure 3.1. In fact, the point doubling operations in the latter can be rearranged to be identical to the former without loss of generality and also without affecting its performance. They also perform the same operations on exit if $k_{n-1} = 1$ as assumed in Automaton 2. We have labeled the states in Figure 3.3 in a way to reflect this analogy.

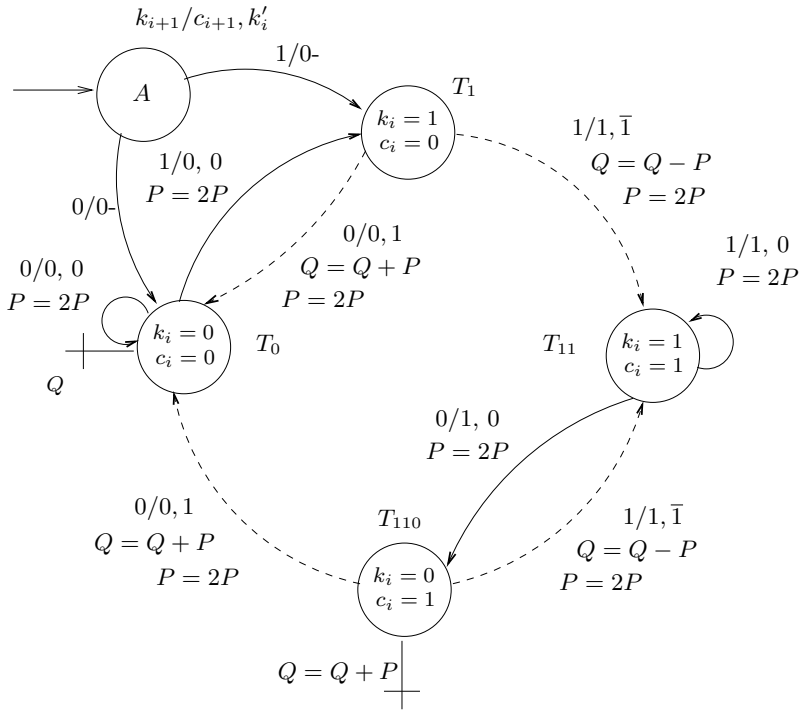

Figure 3.3: Automaton of Reitwiesner's canonical recoding algorithm.

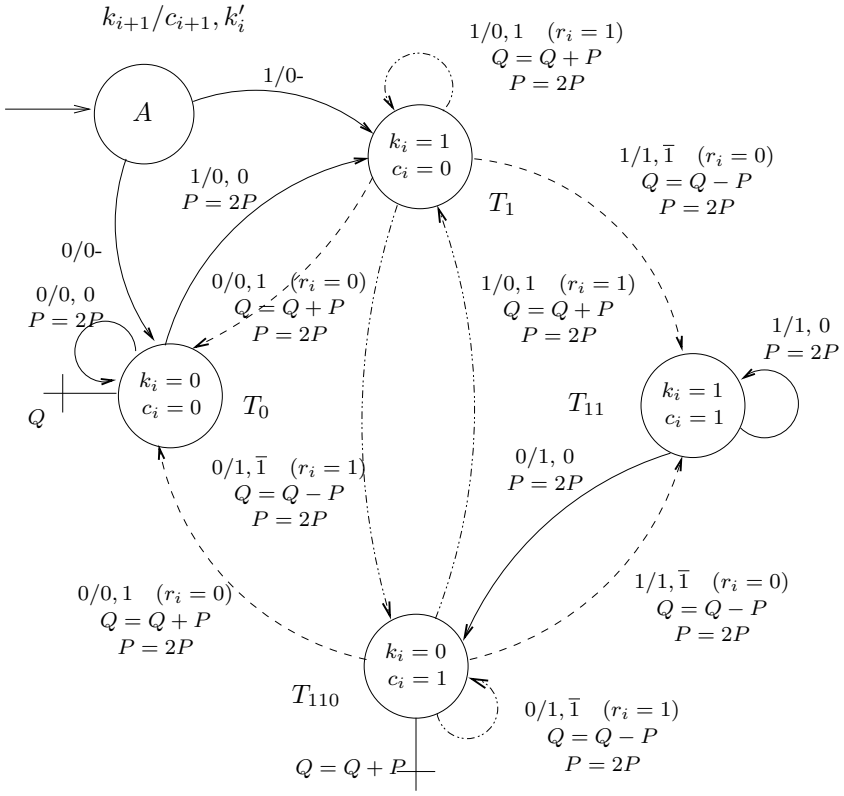

Figure 3.4: Automaton of Ha-Moon's randomization algorithm.

3.2 Randomness of the Recoded Keys

In this section, we discuss the randomness of the recoded keys resulting from the OA and the HM randomization algorithms. That is, we investigate whether or not they can produce any of all the possible BSD representations of an integer.

3.2.1 Randomness of the Keys of the OA Automaton

We will illustrate the following observations using an example of a small integer. Let $k = 15 = (1111)_2$, three successive random decisions will be drawn, which we will denote as e_1, e_2 and e_3. In Table 3.1, we give the different representations k' generated for all possible random decisions taken and the corresponding operations performed. We denote the elliptic curve (EC) point addition, subtraction and doubling as A, S and D respectively.

Table 3.1: The possible outcomes of the randomized algorithm for $k = 15$.

Case	$e_1e_2e_3$	k'	Operations Performed
1	000	$100\bar{1}1$	AD SD D D A
2	001	$100\bar{1}1$	AD SD D DA
3	010	$100\bar{1}1$	AD SD DA SD A
4	011	$100\bar{1}1$	AD SD DA AD
5	100	$10\bar{1}11$	AD AD SD D A
6	101	$10\bar{1}11$	AD AD SD DA
7	110	$1\bar{1}111$	AD AD AD SD A
8	111	01111	AD AD AD AD

We note the following in Table 3.1.

- When the sequence DA is followed by SD, as in case 3, the subtraction cancels the effect of the addition on the result and the addition and subtraction operations are then redundant.
- When the sequence DA is followed by AD, as in case 4, this sequence of operations DAAD is equivalent to DDA, that is, it yields the same result. Thus, there is one extra addition operation.

Therefore, in the first four cases shown in Table 3.1, though the random decisions drawn were different, the resulting generated representations k' are the same. This means that,

for a scalar k where the binary representation contains a block of four consecutive 1s, half of the possible resulting k' after the processing of this block are the same. In the following lemmas, we generalize this argument. Let $\mathcal{S}$ be a string. The notation $\langle \mathcal{S} \rangle^d$ denotes $\mathcal{S}$ repeated d times. For example, $(\langle 0\ 1 \rangle^3)_2$ is $(0\ 1\ 0\ 1\ 0\ 1)_2$. b^d for $b \in \{\overline{1}, 0, 1\}$ denotes a sequence of where b appears d times.

Lemma 3.1 *Starting from state 0 of the OA Randomized Automaton 2, the processing of a block of consecutive 1s in the binary representation of k yields the same BSD representation with probability $\frac{1}{2}$.*

Proof. Let $t+1$ be the length of the block of consecutive 1s in k that the algorithm starts scanning when at state 0. We have a t-tuple of random decisions $\underline{e} = (e_1\ \ldots\ e_t)$. Assume that the first random decision drawn is 0, *i.e.*, $e_1 = 0$. The algorithm will then move to state 11 after generating a 1 and a $\overline{1}$. At this point, no matter what is the value of the remaining terms of $\underline{e}$, the operations performed will be equivalent to successive $t-1$ doubling operations and an addition at the end, which translates into generating $t-1$ 0s and a 1 from right to left.

- This is obvious in the case where $\underline{e} = (0\ 0\ \ldots\ 0)$.
- In the case where $\underline{e} = (0\ \langle 1\ 0 \rangle^j\ \ldots)$ the sequence of operations starting from state 11 is $\langle \mathrm{DASD} \rangle^j$ which is equivalent to $\langle \mathrm{DD} \rangle^j$. When exiting the state 11, *i.e.*, when a 0 is scanned or when the most significant bit is reached, the algorithm performs a point addition.
- In the case where $\underline{e} = (0\ 1^j\ \ldots)$ the sequence of operations starting from state 11 is $\mathrm{DA}\langle \mathrm{AD} \rangle^{j-1}$. This sequence is equivalent to $\mathrm{DDA}\langle \mathrm{AD} \rangle^{j-2} = \mathrm{DDDA}\langle \mathrm{AD} \rangle^{j-3} = \cdots = \langle \mathrm{D} \rangle^j \mathrm{A}$.

It is obvious that the mix between the above cases covers all the possibilities for $\underline{e} = (0\ e_2\ \ldots\ e_t)$ and that it also yields a sequence of doubling operations ending with an addition. Hence, for half of the possible values for $\underline{e}$, the BSD representation generated for the $t+1$-long block of 1s is $(10^{t-1}\overline{1}1)_{\mathrm{BSD}}$. □

We can also extend the previous argument to the case where $\underline{e} = (1\ 0\ e_3\ \ldots\ e_t)$. In this case the resulting BSD representation will be $(10^{t-2}\overline{1}11)_{\mathrm{BSD}}$ and so forth. This leads to the following more general lemma.

Lemma 3.2 *Starting from state 0 of the OA Randomized Automaton 2, the processing of a block of $t+1$ consecutive 1s in the binary representation of k yields $t+1$ possible BSD representations ${}_1k'$, ${}_2k', \ldots,$ ${}_tk'$ and ${}_{t+1}k'$ with the following probability of appearance $Pr({}_1k') = \frac{1}{2}, Pr({}_2k') = \frac{1}{4}, \ldots Pr({}_tk') = \frac{1}{2^t}$ and $Pr({}_{t+1}k') = \frac{1}{2^t}$*

We also note from Table 3.1 that the NAF of $k = 15$ which is $(1000\overline{1})_{\mathrm{BSD}}$ was not among the different BSD representations generated. This suggests the following lemma.

Lemma 3.3 *The NAF of k cannot be generated by the OA Randomized Automaton 2, unless the binary representation of k is a NAF.*

Proof. Compared to the original automaton by Morain and Olivos [MO90], Oswald and Aigner have redistributed the operations performed with the original transitions. For the first automaton, when in state 0 and $k_{i+1} = 1$, no operations were performed until another bit is scanned and if it is another 1, a subtraction is performed, otherwise, an addition. Whereas in the randomized automaton, there is always an addition performed with the first 1 scanned starting from state 0, hence the least significant sbit of k' will be 1 and can never be $\overline{1}$. Since, in essence, the NAF of an integer is produced by using the

following transformation [MO90]

$$1^r 01^t \mapsto 10^r \overline{1} 0^{t-1} \overline{1}, \tag{3.1}$$

this suggests that, unless the binary representation of k was sparse (*e.g.*, $k = (10)_{10}$), the OA algorithm will not generate the NAF of k as one of its random outcomes. □

3.2.2 Randomness of the Keys of the HM Algorithm

First, we will establish the analogy between Reitwiesner's canonical recoding algorithm and Solinas' NAF-recoding algorithm [Sol00]. This will help us verify whether or not the HM algorithm can produce any of all the possible BSD representations of an integer.

Solinas' algorithm is based on the following idea. To derive the binary expansion of an integer, we divide it by 2, store the remainder (0 or 1), and repeat the process with the quotient. To derive the NAF of an integer, using the method proposed by Solinas, we divide it repeatedly by 2 as well, allowing remainders to be 0, 1 or -1. If the remainder should be ± 1, we choose whichever makes the quotient even so that the next division yields a remainder of 0.

Algorithm 3.1. Solinas' NAF-computing algorithm

INPUT: k, a positive integer

OUTPUT: $k' = \text{NAF}(k)$

1. $i \leftarrow 0$
2. while $k > 0$ do
   - 2.1 if k is odd then
     - 2.1.1 $k'_i \leftarrow 2 - (k \bmod 4)$
     - 2.1.2 $k \leftarrow k - k'_i$

2.2 else

$k'_i \leftarrow 0$

2.3 $k \leftarrow k/2$

2.4 $i \leftarrow i + 1$

In the following, we will compare Reitwiesner's (Figure 3.3) and Solinas' algorithms and the decisions taken by them for different input bits. For simplicity, we will refer to these two algorithms as RNAF and SNAF respectively. There are two cases for k.

Case 1: k is odd

Case 1.1: $k \equiv 1 \pmod 4$

In SNAF, k'_i is set to 1, a 1 is subtracted from k and then k is divided by 2, which makes this subtraction unnecessary in this case if k is just shifted to the right by one bit.

This is equivalent in RNAF to the case where $k_i + c_i = 1$ and $k_{i+1} = 0$. k'_i is set to 1 and $c_{i+1} = 0$; hence, there is no change in the remaining bits of k.

Case 1.2: $k \equiv 3 \pmod 4$

In SNAF, k'_i is set to $\overline{1}$, a 1 is added to k and then k is divided by 2. In this case, alternatively, the addition can be performed after the division by 2.

This is equivalent in RNAF to the case where $k_i + c_i = 1$ and $k_{i+1} = 1$. k'_i is set to $\overline{1}$ and $c_{i+1} = 1$.

Case 2: k is even

In SNAF, k'_i is set to 0 and k is divided by 2.

In RNAF, this is the case where $k_i + c_i \equiv 0 \pmod 2$. For any value of k_{i+1}, k'_i is set to 0 and c_{i+1} is set to be equal to c_i to propagate the carry, if any.

Nevertheless, Reitwiesner's canonical recoding algorithm has better performance than Solinas' algorithm. The reason for that is that the former sets c_{i+1} appropriately to save the carry, if any, instead of performing the subtraction or addition as in step 2.1.2 in Algorithm 3.1 and causing the carry to propagate up to the most significant bit with every iteration.

Lemma 3.4 *Ha and Moon's randomization algorithm can generate any random BSD representation of length $n+1$ sbits for an n-bit integer k.*

Proof. As was mentioned before, in HM algorithm a random decision r_i is taken whenever $k_i + c_i = 1$, that is whenever $k \equiv 1$ or $3 \pmod 4$. We can insert the same random decision in Algorithm 3.1 by Solinas by modifying step 2.1.1 as follows

$$k'_i \leftarrow (-1)^{r_i} \ [2 - (k \bmod 4)].$$

Using this modification, we can see that when k is odd the remainder can be either 1 or $\overline{1}$. Since there are no other possible values for the remainder, the proof is established.□

3.3 Average-Case Analysis of the Randomized Algorithms

In this section, we present the average case (complexity) analysis of the OA and HM algorithms. That is, we calculate how many point doubling and point addition (subtraction) operations are performed on average. The basic assumptions are that the n-bit integer k is uniformly distributed in the range $[0, 2^n - 1]$. Thus, each bit of k is equally likely to be 0 or 1, *i.e.*, $Pr(k_i = 0) = Pr(k_i = 1) = \frac{1}{2}$ for $0 \leq i < n$. The same assumption applies to the random decisions taken.

3.3.1 Average-Case Analysis of the OA Automaton

We provide this analysis using both the Markov chains method and the grammatical specification method.

Analysis Using Markov Chains

We use this method to find the limiting probability of occurrence of point additions or subtractions in the proposed algorithm (please refer to Appendix B for theoretical background on Markov chains). We follow the same procedure as in [HM02a; EK94]. We define the states of the Markov chain to be the triplet (A_i, k_{i+1}, e_i) where A_i denotes one of the states of the algorithm, namely 0, 1, 110 or 11. We will refer to these states as a, b, c and d respectively. The possible state transitions are shown in Table 3.2. Note that in the table, o_i denotes the operation performed, i.e., it takes the value 1 or $\overline{1}$ when a point addition or subtraction is performed respectively and 0 otherwise. We cannot consider o_i as the sbits of the resulting random BSD representation since the current resulting o_i may not always be prepended to the previous ones, but may be added or subtracted to the same sbit position of the previous one as was explained in Section 3.2.1.

From Table 3.2, we deduce the probability matrix $\mathbf{P}$. We then obtain the limiting probability vector $\boldsymbol{\pi} = (\pi_0 \ \ldots \ \pi_w)$ as the unique solution of

$$\begin{aligned} \boldsymbol{\pi}\mathbf{P} &= \boldsymbol{\pi}, \\ \sum_{j=0}^{w} \pi_j &= 1 \end{aligned} \tag{3.2}$$

where, in this case, $w = 15$.

Table 3.2: State transition table for the randomized automaton of Figure 3.2.

State		Output	Next state			
			(k_{i+2}, e_{i+1})			
s_i	(A_i, k_{i+1}, e_i)	(A_{i+1}, o_i)	$(0,0)$	$(0,1)$	$(1,0)$	$(1,1)$
s_0	$(a,0,0)$	$(a,0)$	s_0	s_1	s_2	s_3
s_1	$(a,0,1)$	$(a,0)$	s_0	s_1	s_2	s_3
s_2	$(a,1,0)$	$(b,1)$	s_4	s_5	s_6	s_7
s_3	$(a,1,1)$	$(b,1)$	s_4	s_5	s_6	s_7
s_4	$(b,0,0)$	$(a,0)$	s_0	s_1	s_2	s_3
s_5	$(b,0,1)$	$(a,0)$	s_0	s_1	s_2	s_3
s_6	$(b,1,0)$	$(d,1)$	s_{12}	s_{13}	s_{14}	s_{15}
s_7	$(b,1,1)$	$(b,1)$	s_4	s_5	s_6	s_7
s_8	$(c,0,0)$	$(a,0)$	s_0	s_1	s_2	s_3
s_9	$(c,0,1)$	$(a,0)$	s_0	s_1	s_2	s_3
s_{10}	$(c,1,0)$	$(d,1)$	s_{12}	s_{13}	s_{14}	s_{15}
s_{11}	$(c,1,1)$	$(b,1)$	s_4	s_5	s_6	s_7
s_{12}	$(d,0,0)$	$(c,1)$	s_8	s_9	s_{10}	s_{11}
s_{13}	$(d,0,1)$	$(c,1)$	s_8	s_9	s_{10}	s_{11}
s_{14}	$(d,1,0)$	$(d,0)$	s_{12}	s_{13}	s_{14}	s_{15}
s_{15}	$(d,1,1)$	$(b,1)$	s_4	s_5	s_6	s_7

$$\mathbf{P} = \begin{pmatrix}
\frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \\
0 & 0 & 0 & 0 & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \\
0 & 0 & 0 & 0 & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \\
0 & 0 & 0 & 0 & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix}$$

Solving (3.2) using MAPLE, we obtain

$$\begin{aligned}\boldsymbol{\pi} = (&0.1071428572, 0.1071428572, 0.1071428572, 0.1071428572,\\ &0.08928571431, 0.08928571431, 0.08928571432, 0.08928571431,\\ &0.01785714286, 0.01785714286, 0.01785714286, 0, 01785714286,\\ &0.03571428573, 0.03571428573, 0.0357142857, 0.03571428568).\end{aligned}$$

from which we can find the probability of additions (or subtractions)

$$Pr(o_i \neq 0) = \pi_2 + \pi_3 + \pi_6 + \pi_7 + \pi_{10} + \pi_{11} + \pi_{12} + \pi_{13} + \pi_{15} = 0.5357142859.$$

Thus, we conclude from this analysis that the average number of point addition operations for Oswald and Aigner's algorithm is about 3.6% more than that of the binary algorithm; whereas the authors, using experimental results, have stated that it was 9%. We have not studied the details of the authors' experiments that had lead to this result.

Analysis Using Grammatical Specification

Now we analyze Oswald and Aigner's randomized automaton (Figure 3.2) using the grammatical specification method [Gre83; HMU01; SF96] that was used by Morain and Olivos [MO90] to analyze their automata (for background on this method, please see Appendix C). This is an alternative method to the Markov chain analysis.

The language recognized by the randomized automaton is $L = \{(0,0),(0,1),(1,0),(1,1)\}^*\{(1,0),(1,1)\}$. The symbols of this language represent the pair (k_{i+1}, e_i) given that the most significant bit of k is 1. The grammar that generates L consists of

- the terminal alphabet $T = \{(0,0),(0,1),(1,0),(1,1)\}$,

- the nonterminal alphabet $N = \{T_0, T_1, T_{11}, T_{110}\}$ corresponding to the states 0, 1, 11 and 110 of the automaton respectively,
- the start symbol $S = A$ (not shown in the graph since it is a dummy state that leads directly to T_0 without scanning any input symbol).
- the productions P as follows

$$\begin{aligned}
A &\rightarrow T_0,\\
T_0 &\rightarrow (0,0)\ T_0 \mid (0,1)\ T_0 \mid (1,0)\ T_1 \mid (1,1)\ T_1,\\
T_1 &\rightarrow (0,0)\ T_0 \mid (0,1)\ T_0 \mid (1,0)\ T_{11} \mid (1,1)\ T_1 \mid \epsilon,\\
T_{11} &\rightarrow (0,0)\ T_{110} \mid (0,1)\ T_{110} \mid (1,0)\ T_{11} \mid (1,1)\ T_1 \mid \epsilon,\\
T_{110} &\rightarrow (0,0)\ T_0 \mid (0,1)\ T_0 \mid (1,0)\ T_{11} \mid (1,1)\ T_1.
\end{aligned}$$

The system of equations corresponding to the productions is

$$\begin{aligned}
A &= T_0\\
T_0 &= 2zuT_0 + 2zu^2T_1,\\
T_1 &= 2zuT_0 + zu^2T_1 + zu^2T_{11} + 1,\\
T_{11} &= 2zu^2T_{110} + zuT_{11} + zu^2T_1 + u,\\
T_{110} &= 2zuT_0 + zu^2T_1 + zu^2T_{11}.
\end{aligned} \tag{3.3}$$

We have solved the system of equations (3.3) using MAPLE to obtain the bivariate *generating function* $A(z, u)$. Using *partial fraction decomposition* and *Taylor expansion*, we can write $A(z, u)$ in the following form

$$A(z,u) = \sum_{n,m} a_{n,m} z^n u^m$$

where $a_{n,m}$, denoted as $[z^n u^m]A(z,u)$, is the number of strings with n symbols and with cost m.

Since the terminal alphabet consists of four equally probable symbols, we obtain the *probability generating function*

$$A_p(z,u) = A(\frac{z}{4}, u),$$

from which we deduce the generating function for the *expectation*

$$a_p(z) = \left.\frac{\partial A_p}{\partial u}\right|_{u=1}.$$

The coefficient of z^n in the series $a_p(z)$, *i.e.*, $[z^n]a_p(z)$ is the expected cost of the processing of a string of length n. The most significant bit of k is always assumed to be 1 [MO90]; hence, only $n-1$ bits of k are involved in the processing. Therefore, the expected cost is

$$\begin{aligned}\frac{1}{2}\frac{1}{4^{n-1}}[z^n]a(z) = [z^n]2a_p(z) &= [z^n]2a(z/4)\\ &= \frac{43}{28}\,n + \frac{117}{196} + O(2^{-\frac{3}{2}n}),\\ &= 1.535714286\,n + 0.5969387755 + O(2^{-\frac{3}{2}n}).\end{aligned}$$

This result shows that the total number of doubling and addition/subtraction operations per bit is 1.5357 on average, which agrees with the result obtained using Markov chains, since one doubling operation is performed per bit.

3.3.2 Average-Case Analysis of the HM Algorithm

The authors have analyzed their algorithm using the Markov chains method. We present here the analysis of the algorithm using the grammatical specification method.

For the sake of completeness, we first analyze the automaton of Reitweiser's canonical recoding algorithm (see Figure 3.3). The language recognized by this automaton is $L = \{0,1\}^*0$. The symbols of this language represent k_{i+1} given that a 0 is prepended to k. The grammar that generates L consists of

- $T = \{0,1\}$,
- $N = \{T_0, T_1, T_{11}, T_{110}\}$ as the states were denoted in Figure 3.3,
- $S = A$,
- the productions P as follows

$$\begin{aligned} A &\rightarrow 0\ T_0 \mid 1\ T_1, \\ T_0 &\rightarrow 0\ T_0 \mid 1\ T_1, \mid \epsilon, \\ T_1 &\rightarrow 0\ T_0 \mid 1\ T_{11}, \\ T_{11} &\rightarrow 0\ T_{110} \mid 1\ T_{11}, \\ T_{110} &\rightarrow 0\ T_0 \mid 1\ T_{11} \mid \epsilon. \end{aligned}$$

The system of equations corresponding to the productions is

$$\begin{aligned} A &= 2zT_0 + 2zT_1, \\ T_0 &= zuT_0 + zuT_1 + 1, \\ T_1 &= zu^2T_0 + zu^2T_{11}, \\ T_{11} &= zuT_{110} + zuT_{11}, \\ T_{110} &= zu^2T_0 + zu^2T_{11} + u. \end{aligned} \tag{3.4}$$

The expected cost was found to be

$$\begin{aligned}[z^{n+1}]2a(z/2) &= \frac{4}{3}\,(n+1) - \frac{8}{9} - (1 - \frac{(-1)^{(n+1)+1}}{9})2^{-(n+1)} \\ &= \frac{4}{3}\,n + \frac{4}{9} - (1 - \frac{(-1)^{n+2}}{9})2^{-(n+1)},\end{aligned}$$

which is identical up to the second term to that of Automaton 2 [MO90]. This result also agrees with Solinas' [Sol00], that is, the average number of nonzero sbits in the NAF of an n-bit integer, which is also the average number of additions/subtractions in the scalar multiplication algorithm, is $\frac{1}{3}\,n$.

The automaton shown in Figure 3.4 representing the HM algorithm operates on the language $L = \{(0,0),(0,1),(1,0),(1,1)\}^*\{(0,0),\ (0,1)\}$ with the productions being as follows

$$\begin{aligned}A &\to (0,0)\ T_0 \mid (0,1)\ T_0 \mid (1,0)\ T_1 \mid (1,1)\ T_1, \\ T_0 &\to (0,0)\ T_0 \mid (0,1)\ T_0 \mid (1,0)\ T_1 \mid (1,1)\ T_1 \mid \epsilon, \\ T_1 &\to (0,0)\ T_0 \mid (0,1)\ T_{110} \mid (1,0)\ T_{11} \mid (1,1)\ T_1, \\ T_{11} &\to (0,0)\ T_{110} \mid (0,1)\ T_{110} \mid (1,0)\ T_{11} \mid (1,1)\ T_{11}, \\ T_{110} &\to (0,0)\ T_0 \mid (0,1)\ T_{110} \mid (1,0)\ T_{11} \mid (1,1)\ T_1 \mid \epsilon.\end{aligned}$$

The corresponding system of equations is

$$\begin{aligned}A &= 2zT_0 + 2zT_1, \\ T_0 &= 2zuT_0 + 2zuT_1 + 1, \\ T_1 &= zu^2T_0 + zu^2T_{110} + zu^2T_{11} + zu^2T_1, \\ T_{11} &= 2zuT_{110} + 2zuT_{11}, \\ T_{110} &= zu^2T_0 + zu^2T_{110} + zu^2T_{11} + zu^2T_1 + u.\end{aligned} \tag{3.5}$$

The expected cost for the automaton is

$$[z^{n+1}]2a(z/4) = 1.5\,(n+1) - 1 + ((n+1)+1)2^{-(n+1)+2}$$
$$= 1.5\,n + 0.5 + (n+2)2^{-n+1}$$

agreeing with the result obtained by the authors using Markov chains method.

3.4 Conclusion

We have considered in this chapter the randomized algorithms proposed by Oswald and Aigner and by Ha and Moon. We have compared both algorithms from the viewpoint of randomness of the encoded keys and demonstrated that the HM algorithm can in fact generate any possible BSD representation of the key while this is not the case for the OA algorithm . Although both randomized automata proposed by Oswald and Aigner were shown to be breakable by [Wal04a; OS02a; KW03], neither of these papers reported their key randomness shortage. We have also shown how the random decision proposed by Ha and Moon can be simply inserted in Solinas NAF generating algorithm.

We have also compared the average case (complexity) analysis of the concerned algorithms using both Markov chains and grammatical specifications method. Hence, our work confirms that Ha-Moon's has the same average computational complexity as the binary algorithm while Oswald-Aigner's algorithm executes 3.5% more point additions/subtractions on average. This is a new result since the experimental one reported by the authors has shown that the additional number of point additions is 9% on average.

Chapter 4

Binary Signed Digit Representations of Integers

Applications of signed digit representations of an integer include computer arithmetic, cryptography, and digital signal processing. Recall that a *binary signed digit* (BSD) representation of an integer $k \in [0, 2^n - 1]$ is a base-2 representation denoted by $(k'_n, k'_{n-1}, \ldots, k'_0)_{\text{BSD}}$ where $k'_i \in \{-1, 0, 1\}$. We refer to the k'_is as *signed bits*, or *sbits* for short, and -1 is written as $\overline{1}$.

An integer can have several BSD representations. For example, $k = (9)_{10}$ can be written as $(01001)_{\text{BSD}} = (0101\overline{1})_{\text{BSD}} = (1\overline{1}001)_{\text{BSD}}$ among other possibilities. Among the possible BSD representations of an integer there are two unique representations: one is the conventional binary representation where there are no $\overline{1}$s, and the other is the *non-adjacent form* (NAF). The NAF of k is characterized by having no two adjacent non-zero sbits, and hence, having the number of 0s about $\frac{2}{3}n$. This representation was of particular importance in speeding up the scalar multiplication operation in elliptic-curve

cryptosystems [MO90; Sol00]. Especially that in those cryptosystems, the use of negative bits does not incur any noticeable extra computations.

In this chapter, we first consider a number of relevant questions concerning random BSD representations of an integer. These questions, which are not necessarily restricted to cryptographic applications of BSD representations and can be fundamentally important from the mathematical point of view, are: For an integer $k \in [0, 2^n - 1]$, what is the average number of BSD representations that are of length n or $n+1$ sbits and what is the exact number of representations? For integers in this range, which one has the maximum number of representations? We first present the answers to these questions and prove that the maximum number of representation of an n-bit integer is a Fibonacci number. Thereafter, we present an algorithm that calculates the exact number of representations of k in $O(n)$.

We then present an algorithm that generates a random BSD representation for an integer k by scanning its bits starting from the most significant end in $O(n)$. Also, we provide modifications to this algorithm to obtain a BSD generation algorithm that can produce all BSD representations of an integer k in $O(\varphi^n)$ in the worst case, where $\varphi \approx 1.618$ is the golden ratio [Kos01]. The latter algorithm helps us formulate the maximum number of BSD representations of an integer among all integers of length n, and, hence, provide an alternate expression for a Fibonacci number. We also demonstrate the effect of prepending 0s to k on the number of its BSD representations. The main contents of this chapter were published as [EH07]. Notations used throughout this chapter are as defined on page 35 prior to Lemma 3.1.

4.1 Number of Binary Signed Digit Representations

Considering the binary representation of k as one of its BSD representations, different BSD representations for k can be obtained by replacing 01 with $1\bar{1}$ and vice versa and by replacing $0\bar{1}$ with $\bar{1}1$ and vice versa [Rei60, Equations 8.4.4- and 8.4.5-]. For example if $k = (11)_{10}$ is represented in $n = 5$ bits, *i.e.*, $k = (01011)_2$, the different BSD representations for k are: 01011, $011\bar{1}1$, $0110\bar{1}$, $1\bar{1}011$, $1\bar{1}1\bar{1}1$, $1\bar{1}10\bar{1}$, $10\bar{1}\bar{1}1$, $10\bar{1}0\bar{1}$. The second representation can be obtained from the first one by replacing the second occurrence of 01 with $1\bar{1}$. The third representation can then be obtained from the second one by replacing the $\bar{1}1$ with $0\bar{1}$, and so forth. Those replacements are done exhaustively until all possible BSD representations for k are obtained.

The binary representation of k must include at least one 0 that precedes a 1, so that from it we can obtain all other BSD representations.

4.1.1 Useful Lemmas

In the following we present some lemmas related to the number of BSD representations of an integer k. These lemmas will be used to derive the main results of this chapter.

Let $\lambda(k, n)$ be the number of BSD representations of $k \in [0, 2^n - 1]$ that are n sbits long. Then the following lemmas hold.

Lemma 4.1

(i) $\lambda(0, n) = 1$,

(ii) $\lambda(1, n) = n$,

(iii) $\lambda(2^i, n) = n - i$.

Proof.

(i) This is obvious since in two's complement representation, the value 0 is represented with n consecutive zeros. Therefore, there exists no choices for alternative representations.

Let us assume that there is some other BSD representation for the integer $k = 0 = \sum_{i=0}^{n-1} k'_i 2^i$ where $k' \in \{\overline{1}, 0, 1\}$. Then this representation must contain one or more sbits of the value 1 or $\overline{1}$. For example, let us assume that there is a representation where $k'_j = 1$ for some $0 \leq j \leq n-1$, then the summation of the remaining sbits with their appropriate weights should be -2^j. The largest absolute value that the sbits $(k'_{j-1}, \ldots, k'_0)_2$ can take is $\sum_{i=0}^{j-1} 2^i = 2^j - 1$. The smallest absolute value that is greater than 0 that the sbits $(k'_{n-1}, \ldots, k'_{j+1})_2$ can take is 2^{j+1}. The difference between these two values is $2^{j+1} - 2^j + 1 = 2^j + 1$. That is there is no possible assignment for the remaining sbits resulting in a value of -2^j.

(ii) The possible BSD representations for 1 in n sbits are $0^{n-1}1$, $0^{n-2}1\overline{1}$, $0^{n-3}1\overline{1}\overline{1}$, $\ldots$, $1\overline{1}^{n-1}$. Their total number is n.

This is true since, for any $t \in [0, n]$

$$2^t - \sum_{i=0}^{t-1} 2^i = 2^t - \frac{2^t - 1}{2 - 1} = 1.$$

(iii) The possible BSD representations for 2 in n sbits are $0^{n-2}10$, $0^{n-3}1\overline{1}0$, $0^{n-4}1\overline{1}\overline{1}0$, $\ldots$, $1\overline{1}^{n-2}0$. Note that these are the same representations for 1 when its binary representation is $(n-1)$ bits long with a 0 appended as the least significant sbit. Their total number is $n - 1$.

That is,

$$\lambda(2, n) = \lambda(1, n-1) = n - 1.$$

Hence,

$$\lambda(4, n) = \lambda(2, n-1) = \lambda(1, n-2) = n-2.$$

By induction,

$$\lambda(2^i, n) = n - i. \qquad \square$$

Lemma 4.2 *For* $2^{n-1} \leq k \leq 2^n - 1$,

$$\lambda(k, n) = \lambda(k - 2^{n-1}, n-1).$$

Proof. An integer k in this range would have the binary form $(1, k_{n-2}, \ldots, k_0)_2$ and its value is $2^{n-1} + d$, where $d = (k_{n-2}, \ldots, k_0)_2$, and is $(n-1)$ bits long. The BSD representations for d in $n-1$ sbits are of the form $(k'_{n-2}, \ldots, k'_0)_{\text{BSD}}$ with $k'_{n-2} \neq \overline{1}$, otherwise, d would be negative.

The BSD representations of k are then of the form $(1, k'_{n-2}, \ldots, k'_0)_{\text{BSD}}$. The two most significant sbits are either 10 or 11. In both cases, no new BSD representations can be generated. Thus, k will have the same BSD representations as for d with an added 1 as the most significant sbit. $\square$

Lemma 4.3 *For* k *even,*

$$\lambda(k, n) = \lambda(\frac{k}{2}, n-1).$$

Proof. In this case, the integer k is of the form $(k_{n-1}, \ldots, k_1, 0)_2 = 2d$ where $d = (k_{n-1}, \ldots, k_1)_2$, and is $(n-1)$ bits long. The BSD representations for d in $n-1$ sbits are of the form $(k'_{n-1}, \ldots, k'_1)_{\text{BSD}}$. When d is multiplied by 2 to obtain k, it is shifted left by one and a 0 is added to the least significant position. The same is done to each of its BSD representations. In all of them, the least two significant sbits will be either $\overline{1}0$, 00 or 10. In all three cases, no new BSD representations can be generated. Thus, k will have the same BSD representations as for d with an added 0 as the least significant sbit. $\square$

Lemma 4.4 *For k odd,*

$$\lambda(k,n) = \lambda(k-1,n) + \lambda(k+1,n), \tag{a}$$

or

$$\lambda(k,n) = \lambda\left(\frac{k-1}{2}, n-1\right) + \lambda\left(\frac{k+1}{2}, n-1\right). \tag{b}$$

Proof. There are two cases to consider.

Case 1: $k \equiv 1 \pmod 4$, that is k ends with 01, $k-1$ ends with 00 and $k+1$ ends with 10.

From Lemma 4.3, the BSD representations for $k+1$ in n sbits are the same as those for $\frac{k+1}{2}$ with a 0 added as the least significant sbit. If this 0 is replaced by a $\overline{1}$, those representations will be possible representations for k. If we think we should start replacing the rightmost $1\overline{1}$ with 01 to generate new representations, we will find out that all representations that end with a 1 will be accounted for by considering those of $k-1$.

Also from Lemma 4.3, the BSD representations for $k-1$ in n sbits are the same as those for $\frac{k-1}{4}$ in $n-2$ sbits with 00 added as the least significant sbits. If the rightmost 0 is replaced with a 1, those representations will be possible representations for k. If we think we should start replacing the rightmost 01 with $1\overline{1}$ to generate new representations, we will find out that all representations that end with a $\overline{1}$ have been accounted for by considering those of $k+1$ as mentioned before.

Case 2: $k \equiv 3 \pmod 4$, that is k ends with 11, $k-1$ ends with 10 and $k+1$ ends with 00.

The same argument holds as in case 1. The BSD representations of $k-1$ can be possible representations for k by replacing the least significant 0 with 1. Also the BSD representations for $k+1$ can be possible representations for k by replacing the least significant 0 with $\overline{1}$.

There are no other possible representations for k. This is obvious from the fact that any BSD representation for k should have the rightmost sbit either $\overline{1}$ or 1, it can not be 0. Otherwise, $k \equiv 0 \pmod 2$ which is not the case since k is odd. So, if the rightmost 1 or $\overline{1}$ in any representation of k is replaced with 0 then this will be one of the representations of the even number preceding or following k respectively as shown in (a). Using Lemma 4.3, (b) is obtained. □

4.1.2 Number of BSD Representations of Length n

Here we will investigate the total number of BSD representations for all integers in the range $[0, 2^n - 1]$ that are n sbits long. We will denote that total number by $\sigma(n)$. Table 4.1 gives an example of $\lambda(k, n)$ and $\sigma(n)$ for small n.

We can intuitively find an expression for $\sigma(n)$ as follows. In the BSD system, the integers k, represented in n sbits, would be in the range $-2^n < k < 2^n$. There are 3^n different combinations for k. We consider in this case non-negative integers k, *i.e.*, $0 \leq k < 2^n$. In fact, $-k$ has the same BSD representations as k with the 1s replaced with $\overline{1}$s and vice versa. Thus, the total number of non-negative combinations is $\frac{3^n+1}{2}$. We will now use the previous lemmas to prove that

$$\sum_{k=0}^{2^n-1} \lambda(k, n) = \sigma(n) = \frac{3^n + 1}{2}.$$

Let

$$\varepsilon(n) = \sum_{k=1}^{2^{n-1}-1} \lambda(k, n). \tag{4.1}$$

From Lemma 4.1(i) and Lemma 4.2, we have

$$\sigma(n) = \sigma(n-1) + \varepsilon(n) + 1. \tag{4.2}$$

Table 4.1: $\lambda(k,n)$ and $\sigma(n)$ for small n.

$n=1$		$n=2$		$n=3$		$n=4$		$n=5$		$n=5$ *cont'd*	
k	$\lambda(k,n)$	k	$\lambda(k,n)$	k	$\lambda(k,n)$	k	$\lambda(k,n)$	k	$\lambda(k,n)$	k	$\lambda(k,n)$
0	1	0	1	0	1	0	1	0	1	16	1
1	1	1	2	1	3	1	4	1	5	17	4
		2	1	2	2	2	3	2	4	18	3
		3	1	3	3	3	5	3	7	19	5
				4	1	4	2	4	3	20	2
				5	2	5	5	5	8	21	5
				6	1	6	3	6	5	22	3
				7	1	7	4	7	7	23	4
						8	1	8	2	24	1
						9	3	9	7	25	3
						10	2	10	5	26	2
						11	3	11	8	27	3
						12	1	12	3	28	1
						13	2	13	7	29	2
						14	1	14	4	30	1
						15	1	15	5	31	1
$\sigma(n)=2$		$\sigma(n)=5$		$\sigma(n)=14$		$\sigma(n)=41$				$\sigma(n)=122$	

If we substitute for $\sigma(n)$ recursively in this equation we obtain

$$\sigma(n) = \sigma(1) + \varepsilon(2) + \cdots + \varepsilon(n) + n - 1. \tag{4.3}$$

From Lemma 4.3 we have

$$\begin{aligned}\sum_{k=2,\ k \text{ even}}^{2^{n-1}-1} \lambda(k,n) &= \sum_{k=2,\ k \text{ even}}^{2^{n-1}-2} \lambda\left(\frac{k}{2}, n-1\right) = \sum_{i=1}^{2^{n-2}-1} \lambda(i, n-1) \\ &= \varepsilon(n-1). \end{aligned} \tag{4.4}$$

From Lemma 4.4 we have

$$\begin{aligned}\sum_{k=1,\ k \text{ odd}}^{2^{n-1}-1} \lambda(k,n) &= \sum_{k=1,\ k \text{ odd}}^{2^{n-1}-1} \left(\lambda\left(\frac{k-1}{2}, n-1\right) + \lambda\left(\frac{k+1}{2}, n-1\right)\right) \\ &= \sum_{i=0}^{2^{n-2}-1} \lambda(i, n-1) + \sum_{i=1}^{2^{n-2}} \lambda(i, n-1).\end{aligned}$$

From Lemma 4.1(iii), $\lambda(2^{n-1}, n) = 1$, and also from Lemma 4.1(i) we have

$$\sum_{k=1,\ k \text{ odd}}^{2^{n-1}-1} \lambda(k,n) = 2 + 2\,\varepsilon(n-1) \tag{4.5}$$

where the last equality follows from (4.4). Substituting from (4.4) and (4.5) into (4.1), we have

$$\begin{aligned}\varepsilon(n) &= \varepsilon(n-1) + 2 + 2\,\varepsilon(n-1) \\ &= 3\,\varepsilon(n-1) + 2. \end{aligned} \tag{4.6}$$

With recursive substitution for $\varepsilon(n)$ and the fact that $\varepsilon(1) = 0$, we obtain

$$\begin{aligned}\varepsilon(n) &= 3^{n-1}\varepsilon(1) + 2 + 2 \cdot 3 + 2 \cdot 3^2 + \cdots + 2 \cdot 3^{n-2} \\ &= 2 \cdot \frac{3^{n-1} - 1}{3 - 1} \\ &= 3^{n-1} - 1. \end{aligned} \tag{4.7}$$

Finally we use (4.7) and the fact that $\sigma(1) = 2$ (from Table 4.1) to evaluate (4.3)

$$\begin{aligned}\sigma(n) &= 2 + (3-1) + \cdots + (3^{n-1}-1) + n - 1 \\ &= 2 + (n-1) - (n-1) + 3 \cdot \frac{3^{n-1}-1}{3-1} \\ &= \frac{3^n + 1}{2}. \end{aligned} \tag{4.8}$$

4.1.3 Number of BSD Representations of Length $n+1$

The NAF of an integer may be one sbit longer than its binary representation [Rei60; Sol00]. As mentioned before, the algorithms that generate a random BSD representation of an integer are each based on a NAF-generating algorithm [HM02a; EH03a] (cf. Chapter 3). Therefore, we are interested in knowing the number of BSD representations of an n-bit integer k that are $n+1$ sbits long. Moreover, for those integers that are in the range $[2^{n-1}, 2^n - 1]$, we will see the effect of having a 0 as the most significant bit in their binary representation on the number of their BSD representations and on the distribution of the number of representations among all n-bit integers as opposed to the previous section. The effect of prepending more 0s to the binary representation of integers on the number of their BSD representation is studied in Section 4.3.3.

Lemma 4.5 *Let $\delta(k, n)$ be the number of BSD representations of $k \in [0, 2^n - 1]$ in $n+1$ sbits. Then, we have*

$$\delta(k, n) = \lambda(k, n) + \lambda(2^n - k, n).$$

Proof. A part of the BSD representations of k that are $n+1$-sbit long are those that have a 0 as the most significant sbit and their number is $\lambda(k, n)$, as was defined in the previous section. If we change the most significant 0 in these representations to a 1, *i.e.*, add 2^n to the value of k, we should add $-(2^n - k)$ in the remaining n sbits, that is the

negative of the two's complement of k. $2^n - k$ has $\lambda(2^n - k, n)$ BSD representations of length n. The negative of these representations is obtained by replacing the 1s with $\bar{1}$s and vice versa. □

The same argument applies to the two's complement of k

$$\begin{aligned}\delta(2^n - k, n) &= \lambda(2^n - k, n) + \lambda(k, n) \\ &= \delta(k, n).\end{aligned} \tag{4.9}$$

For $k = 0$, $\delta(0, n) = \lambda(0, n) = 1$. From (4.9), we conclude that, for $k \in [0, 2^n - 1]$, the distribution of $\delta(k, n)$ is symmetric around $k = 2^{n-1}$.

Let $\varsigma(n)$ be the total number of BSD representations of length $n + 1$ for all integers $k \in [0, 2^n - 1]$. Then, we have

$$\begin{aligned}\varsigma(n) &= \sum_{k=0}^{2^n-1} \delta(k, n) = 1 + \sum_{k=1}^{2^n-1} \delta(k, n) \\ &= 1 + \sum_{k=1}^{2^n-1} \big(\lambda(k, n) + \lambda(2^n - k, n)\big) = 1 + 2 \sum_{k=1}^{2^n-1} \lambda(k, n) \\ &= 1 + 2 \left(\frac{3^n + 1}{2} - 1 \right) \\ &= 3^n.\end{aligned} \tag{4.10}$$

Remark 4.1 *We conclude that, for any n-bit integer, the average number of its—$(n + 1)$ sbits long—BSD representations is roughly* $\left(\frac{3}{2}\right)^n$

Table 4.2 gives an example of $\delta(k, n)$ and $\varsigma(n)$ for small n. It is clear from Table 4.1 and Table 4.2 and from the definitions of $\lambda(k, n)$ and $\delta(k, n)$ that, for $0 \leq k < 2^{n-1}$,

$$\lambda(k, n) = \delta(k, n - 1) \tag{4.11}$$

since in this range, the binary representation of k has a 0 as the leftmost bit. In other words, the algorithm that computes $\lambda(k, n)$ that we present in Section 4.2 can be used to

Table 4.2: $\delta(k,n)$ and $\varsigma(n)$ for small n.

$n=1$		$n=2$		$n=3$		$n=4$		$n=5$		$n=5$ *cont'd*	
k	$\delta(k,n)$	k	$\delta(k,n)$	k	$\delta(k,n)$	k	$\delta(k,n)$	k	$\delta(k,n)$	k	$\delta(k,n)$
0	1	0	1	0	1	0	1	0	1	16	2
1	2	1	3	1	4	1	5	1	6	17	9
		2	2	2	3	2	4	2	5	18	7
		3	3	3	5	3	7	3	9	19	12
				4	2	4	3	4	4	20	5
				5	5	5	8	5	11	21	13
				6	3	6	5	6	7	22	8
				7	4	7	7	7	10	23	11
						8	2	8	3	24	3
						9	7	9	11	25	10
						10	5	10	8	26	7
						11	8	11	13	27	11
						12	3	12	5	28	4
						13	7	13	12	29	9
						14	4	14	7	30	5
						15	5	15	9	31	6
$\varsigma(n)=3$		$\varsigma(n)=9$		$\varsigma(n)=27$		$\varsigma(n)=81$				$\varsigma(n)=243$	

compute $\delta(k,n)$ as

$$\delta(k,n) = \lambda(k, n+1), \tag{4.12}$$

for $0 \leq k < 2^{n-1}$.

4.1.4 Integer with Maximum Number of BSD Representations

It is desirable to know which integer $k \in [0, 2^n - 1]$ has the maximum number of BSD representations of length n or $n+1$ sbits. We note from (4.12) that the integer with the largest $\delta(k,n)$ is the same one with the largest $\lambda(k, n+1)$. Also from the symmetry of

$\delta(k,n)$ around 2^{n-1} (Eq. (4.9)), we note that there are two values of k for which $\delta(k,n)$ is the maximum value. We will denote them as $k_{max_1,n}$ and $k_{max_2,n}$. From (4.9) we have $k_{max_2,n} = 2^n - k_{max_1,n}$. In the following theorem, we will consider only $k_{max_1,n}$ and drop the subscript 1.

Theorem 4.1 *For $n \geq 3$,*

$$\delta(k_{max,n}, n) = \delta(k_{max,n-1}, n-1) + \delta(k_{max,n-2}, n-2).$$

Proof. We will prove this theorem by induction. From Table 4.2, we see that the theorem is true for $n = 3$. Now we assume that it is true up to an arbitrary $n = i - 1$. That is, using (4.12) we can write

$$\lambda(k_{max,i-1}, i) = \lambda(k_{max,i-2}, i-1) + \lambda(k_{max,i-3}, i-2) \tag{4.13}$$

Also, from Lemma 4.4, we know that

$$\lambda(k_{max,i-1}, i) = \lambda\left(\frac{k_{max,i-1}-1}{2}, i-1\right) + \lambda\left(\frac{k_{max,i-1}+1}{2}, i-1\right). \tag{4.14}$$

From Lemma 4.3 and Lemma 4.4, $k_{max,i}$ must be an odd integer. Let k be an i-bit odd integer, then $k = \left(\frac{k-1}{2}\right) + \left(\frac{k+1}{2}\right)$. Obviously, one of these two terms is an odd integer and the other is the preceding or following even integer. We will denote those two integers as k_o and k_e, respectively. From Lemma 4.4, we have

$$\lambda(k, i+1) = \lambda(k_o, i) + \lambda(k_e, i).$$

For k to be equal $k_{max,i}$, one or both of the following conditions must be true:

- $k_o = k_{max,i-1}$, and $k_e \equiv 2 \pmod 4$ (Lemma 4.3 and 4.4).
- k_e has the maximum number of representations among all even integers. From Lemma 4.3, this is equivalent to saying that $k_e = 2 \cdot k_{max,i-2}$.

We have four cases for the value of an odd k modulo 8. In each case, if the first condition is verified, we will prove that the second condition is also verified, which will prove the theorem.

Case 1: $k \equiv 1 \pmod 8$

$\Rightarrow k_e = \frac{k-1}{2} \equiv 0 \pmod 4$.

This violates the first condition.

Case 2: $k \equiv 3 \pmod 8$

$\Rightarrow k_o = \frac{k-1}{2} \equiv 1 \pmod 4 \Rightarrow k_e = \frac{k+1}{2} \equiv 2 \pmod 4$.

Assume $k_o = k_{max,i-1}$

$\Rightarrow \frac{k_{max,i-1}-1}{2} \equiv 0 \pmod 2$ (even).

Hence, from (4.13) and (4.14), we have

$$k_{max,i-2} = \frac{k_{max,i-1}+1}{2},$$
$$2 \cdot k_{max,i-2} = k_{max,i-1} + 1 = k_e. \tag{4.15}$$

That is, the second condition is verified.

Case 3: $k \equiv 5 \pmod 8$

$\Rightarrow k_e = \frac{k-1}{2} \equiv 2 \pmod 4 \Rightarrow k_o = \frac{k+1}{2} \equiv 3 \pmod 4$.

Assume $k_o = k_{max,i-1}$

$\Rightarrow \frac{k_{max,i-1}-1}{2} \equiv 1 \pmod 2$ (odd).

Hence, from (4.13) and (4.14), we have

$$k_{max,i-2} = \frac{k_{max,i-1}-1}{2},$$
$$2 \cdot k_{max,i-2} = k_{max,i-1} - 1 = k_e. \tag{4.16}$$

That is, the second condition is verified.

Case 4: $k \equiv 7 \pmod 8$

$\Rightarrow \frac{k-1}{2} \equiv 3 \pmod 4$ (odd) $\Rightarrow k_e = \frac{k+1}{2} \equiv 0 \pmod 4$.

This violates the first condition. □

From the proof of Theorem 4.1, we can deduce the following corollary.

Corollary 4.1

$$k_{max,n} = k_{max,n-1} + 2 \cdot k_{max,n-2}.$$

Values and patterns of $k_{max_1,n}$ and $k_{max_2,n}$

Now, we will derive a formula for $k_{max_1,n}$ and $k_{max_2,n}$. From the proof of Theorem 4.1, we see that when $k_{max_1,n} \equiv 3 \pmod 4$ (Case 2), $k_{max_1,n-1} \equiv 1 \pmod 4$ and

$$k_{max_1,n} = 2 \cdot k_{max_1,n-1} + 1, \tag{4.17}$$

and that when $k_{max_1,n} \equiv 1 \pmod 4$ (Case 3), $k_{max_1,n-1} \equiv 3 \pmod 4$ and

$$k_{max_1,n} = 2 \cdot k_{max_1,n-1} - 1. \tag{4.18}$$

Thus, we see that with every increment of n ($n > 2$), cases 2 and 3 alternate. For $n = 3$, from Table 4.2 we have $k_{max_1,3} = 3 \equiv 3 \pmod 4$. Hence, Case 2 occurs when n is odd and Case 3 occurs when n is even.

For n even, we can substitute from (4.17) into (4.18) to obtain

$$\begin{aligned} k_{max_1,n} &= 2\ k_{max_1,n-1} - 1 \\ &= 2\ (2\ k_{max_1,n-2} + 1) - 1 \\ &= 4\ k_{max_1,n-2} + 1. \end{aligned} \tag{4.19}$$

Using recursive substitution,

$$\begin{aligned} k_{max_1,n} &= 4 \cdot 4 \cdot 4 \cdots k_{max_1,2} + 1 + 4 + 16 + \cdots \\ &= (2^2)^{\frac{n-2}{2}} \cdot 1 + \frac{(2^2)^{\frac{n-2}{2}} - 1}{2^2 - 1} \\ &= \frac{1}{3}(2^n - 1). \end{aligned} \tag{4.20}$$

For n odd, we can follow the same derivation procedure to obtain

$$k_{max_1,n} = \frac{1}{3}(2^n + 1). \tag{4.21}$$

As for $k_{max_2,n}$, for n even we have

$$\begin{aligned} k_{max_2,n} &= 2^n - \frac{1}{3}(2^n - 1) \\ &= \frac{1}{3}(2^{n+1} + 1) \\ &= k_{max_1,n+1} \end{aligned} \tag{4.22}$$

where the last equality follows from (4.21).

Similarly, for n odd, we have

$$\begin{aligned} k_{max_2,n} &= \frac{1}{3}(2^{n+1} - 1) \\ &= k_{max_1,n+1}. \end{aligned} \tag{4.23}$$

The sbit pattern of $k_{max_1,n}$ for even n and for odd n can be deduced from the previous discussion. From (4.19) we can deduce that for n even, $k_{max_1,n}$ is of the form $(\langle 0\ 1\rangle^{\frac{n}{2}})_2$ and hence from (4.18) for n odd, $k_{max_1,n}$ is of the form $(\langle 0\ 1\rangle^{\frac{n-1}{2}} 1)_2$.

Thus, after specifying the binary structure of the integer with maximum number of BSD representations, we will use it as the worst-case input to our left-to-right generation algorithm in Section 4.3.2. We will hence derive an expression for the number of BSD representations of that integer.

$\delta(k_{max,n}, n)$ as a Fibonacci number

The Fibonacci numbers form a sequence defined by the following recurrence relation [Kos01]

$$F(0) = 0, \quad F(1) = 1, \quad F(n) = F(n-1) + F(n-2), \ n > 1. \tag{4.24}$$

The closed-form expression of Fibonacci numbers, which is known as Binet's formula, is

$$F(n) = \frac{\varphi^n - (1-\varphi)^n}{\sqrt{5}}, \tag{4.25}$$

where

$$\varphi = \frac{1+\sqrt{5}}{2}, \tag{4.26}$$

φ is known as *the golden ratio*.

From Theorem 4.1, we can see that the values of $\delta(k_{max,n}, n)$ for $n \geq 1$ form a Fibonacci sequence, where from Table 4.2, we have

$$\delta(k_{max,1}, 1) = 2 = F(3),$$
$$\delta(k_{max,1}, 2) = 3 = F(4),$$

hence,

$$\delta(k_{max,n}, n) = F(n+2) = \frac{\varphi^{n+2} - (1-\varphi)^{n+2}}{\sqrt{5}}. \tag{4.27}$$

In Section 4.3.2, we will derive another expression for $\delta(k_{max,n}, n) = \lambda(k_{max,n}, n+1)$.

4.2 Algorithm to Compute the Number of BSD Representations for an Integer

In this section, we will present an algorithm that computes $\lambda(k, n)$ for any integer $k \in [0, 2^n - 1]$. This algorithm is based on the lemmas presented in Section 4.1.1.

Algorithm 4.1. Number of BSD representations of an integer k in n sbits

INPUT: $k \in [0, 2^n - 1]$ and n.

OUTPUT: $C = \lambda(k, n)$.

external $\lambda 2(k_e, k_o, w_e, w_o, n)$. /*computed by Algorithm 4.2 that follows.*/

1. if $(k = 0)$ then

   $C \leftarrow 1$.

2. else if $(k = 1)$ then

   $C \leftarrow n$.

3. else if $(k \geq 2^{n-1})$ then

   $C \leftarrow \lambda(k - 2^{n-1}, n - 1)$.

4. else if (k is even) then

   $C \leftarrow \lambda(\frac{k}{2}, n - 1)$.

5. else

   5.1 if $(k \equiv 1 \pmod 4)$ then

   $C \leftarrow \lambda 2(\frac{k-1}{2}, \frac{k+1}{2}, 1, 1, n - 1)$.

   5.2 else

   $C \leftarrow \lambda 2(\frac{k+1}{2}, \frac{k-1}{2}, 1, 1, n - 1)$.

6. Return(C).

Algorithm 4.1 uses Lemma 4.1(i) and 4.1(ii) to return the value of $\lambda(k, n)$ directly if the value of k is either 0 or 1. Otherwise, it uses Lemmas 4.2 and 4.3 to trim k recursively from any leading 1s or trailing 0s since they do not add to the number of BSD representations of k as mentioned in the proofs of these lemmas. Then, this algorithm calls Algorithm 4.2 to find the actual number of BSD representations of k which is then

an odd integer in the range $[0, 2^{n'-1}-1]$, for some $n' \leq n$. Hence, Lemma 4.4 is applicable to this k.

Algorithm 4.2. Auxiliary algorithm used by Algorithm 4.1 to compute $\lambda(k, n)$

INPUT: k_e, k_o, w_e, w_o and n.

OUTPUT: $c = \lambda 2(k_e, k_o, w_e, w_o, n)$.

1. if ($k_o = 1$ AND $k_e = 2$) then

   $c \leftarrow n * w_o + (n-1) * w_e$.

2. else

   2.1 if ($k_e \equiv 0 \pmod 4$) then

      2.1.1 if ($k_o \equiv 1 \pmod 4$) then

         $c \leftarrow \lambda 2(\frac{k_e}{2}, \frac{k_e}{2} + 1, w_o + w_e, w_o, n-1)$.

      2.1.2 else

         $c \leftarrow \lambda 2(\frac{k_e}{2}, \frac{k_e}{2} - 1, w_o + w_e, w_o, n-1)$.

   2.2 else

      2.2.1 if ($k_o \equiv 1 \pmod 4$) then

         $c \leftarrow \lambda 2(\frac{k_e}{2} - 1, \frac{k_e}{2}, w_o, w_o + w_e, n-1)$.

      2.2.2 else

         $c \leftarrow \lambda 2(\frac{k_e}{2} + 1, \frac{k_e}{2}, w_o, w_o + w_e, n-1)$.

3. Return(c).

Using Lemma 4.4, $\lambda(k, n)$ for k odd consists of two other evaluations of the same function λ; one is for an even integer, k_e which is the closest even integer to $k/2$, and the other is for the preceding or the following odd integer, k_o. If we start using Lemmas 4.3 and 4.4 recursively to evaluate λ for k_e and k_o respectively, at each iteration there will be always two terms for the λ function multiplied by a certain weight each, w_e and w_o.

In general, at the i^{th} iteration

$$\lambda(k,n) = w_{e,n-i}\ \lambda(k_{e,n-i}, n-i) + w_{o,n-i}\ \lambda(k_{o,n-i}, n-i).$$

At the beginning, $w_{e,n} = w_{o,n} = 1$. From Lemma 4.3 we have

$$\lambda(k_{e,n-i}, n-i) = \lambda\left(\frac{k_{e,n-i}}{2}, n-i-1\right).$$

From Lemma 4.4 we have

$$\lambda(k_{o,n-i}, n-i) = \lambda\left(\frac{k_{o,n-i}-1}{2}, n-i-1\right) + \lambda\left(\frac{k_{o,n-i}+1}{2}, n-i-1\right).$$

There are two possible cases for k_e in each iteration and two corresponding subcases for k_o.

Case 1: $k_{e,n-i} \equiv 0 \pmod 4 \quad \Rightarrow \quad \frac{k_{e,n-i}}{2}$ is even

Case 1.1: $k_{o,n-i} \equiv 1 \pmod 4$, *i.e.*, $k_{o,n-i} = k_{e,n-i} + 1 \Rightarrow \frac{k_{o,n-i}-1}{2}$ is even

Hence,

$k_{e,n-i-1} = \frac{k_{o,n-i}-1}{2} = \frac{k_{e,n-i}}{2}$

$k_{o,n-i-1} = \frac{k_{o,n-i}+1}{2} = \frac{k_{e,n-i}}{2} + 1$

Case 1.2: $k_{o,n-i} \equiv 3 \pmod 4$, *i.e.*, $k_{o,n-i} = k_{e,n-i} - 1 \Rightarrow \frac{k_{o,n-i}-1}{2}$ is odd

Hence,

$k_{e,n-i-1} = \frac{k_{o,n-i}+1}{2} = \frac{k_{e,n-i}}{2}$

$k_{o,n-i-1} = \frac{k_{o,n-i}-1}{2} = \frac{k_{e,n-i}}{2} - 1$

In case 1, the weights are updated as follows

$w_{e,n-i-1} = w_{o,n-i} + w_{e,n-i}$

$w_{o,n-i-1} = w_{o,n-i}$

Case 2: $k_{e,n-i} \equiv 2 \pmod 4 \quad \Rightarrow \quad \frac{k_{e,n-i}}{2}$ is odd

Case 2.1: $k_{o,n-i} \equiv 1 \pmod 4$, *i.e.*, $k_{o,n-i} = k_{e,n-i} - 1$

Hence,

$k_{e,n-i-1} = \frac{k_{o,n-i}-1}{2} = \frac{k_{e,n-i}}{2} - 1$

$k_{o,n-i-1} = \frac{k_{o,n-i}+1}{2} = \frac{k_{e,n-i}}{2}$

Case 2.2: $k_{o,n-i} \equiv 3 \pmod 4$, *i.e.*, $k_{o,n-i} = k_{e,n-i} + 1$

Hence,

$k_{e,n-i-1} = \frac{k_{o,n-i}+1}{2} = \frac{k_{e,n-i}}{2} + 1$

$k_{o,n-i-1} = \frac{k_{o,n-i}-1}{2} = \frac{k_{e,n-i}}{2}$

In case 2, the weights are updated as follows

$w_{e,n-i-1} = w_{o,n-i}$

$w_{o,n-i-1} = w_{o,n-i} + w_{e,n-i}$

We depict in Figure 4.1 some examples that illustrate the iterations of Algorithm 4.2 for some chosen values of k. In these examples we have used the notation (k) that means $\lambda(k, n)$. We have also used a tree-like representation where, for a parent node k, the number of BSD representations is equal to the sum of the number of BSD representations of its child nodes. The weight update procedure is depicted by the number of links

between the parent and child nodes. The examples are chosen such that k_e and k_o that are passed as arguments to Algorithm 4.2, *e.g.*, 12 and 13 in Figure 4.1 (a), correspond to the four different cases discussed.

For the time and space complexity of Algorithm 4.1—including its usage of Algorithm 4.2—it is clear that it runs in $O(n)$ time and occupies $O(n)$ bits in memory. This time complexity results from the fact that both algorithms deal with the integer k one bit at a time. As for the space complexity, the new values generated for k and n by Algorithm 4.1 and for k_e, k_o, w_e, w_o and n by Algorithm 4.2 can replace the old values in memory. That is, though the algorithms are illustrated in a recursive form, they can be transformed into an iterative form.

4.3 Left-to-Right BSD Randomization and Generation Algorithms

The algorithms that generate a random BSD representation of an integer [OA01; HM02a; EH03a] (cf. Chapter 3) scan its bits from the least significant to the most significant bit, *i.e.*, from right to left. This is because they are based on the NAF-generating algorithms [Rei60; MO90; Sol00] that scan the bits of the integer in the same direction.

In this section, we present an algorithm that generates a random BSD representation of k while scanning it from left to right. Then we modify it in order to generate *all* of the possible BSD representations of k. The modified algorithm helps us demonstrate the exponential growth with n of the number of BSD representations of $k_{max_1,n}$, as well as the effect of prepending 0s to any integer k.

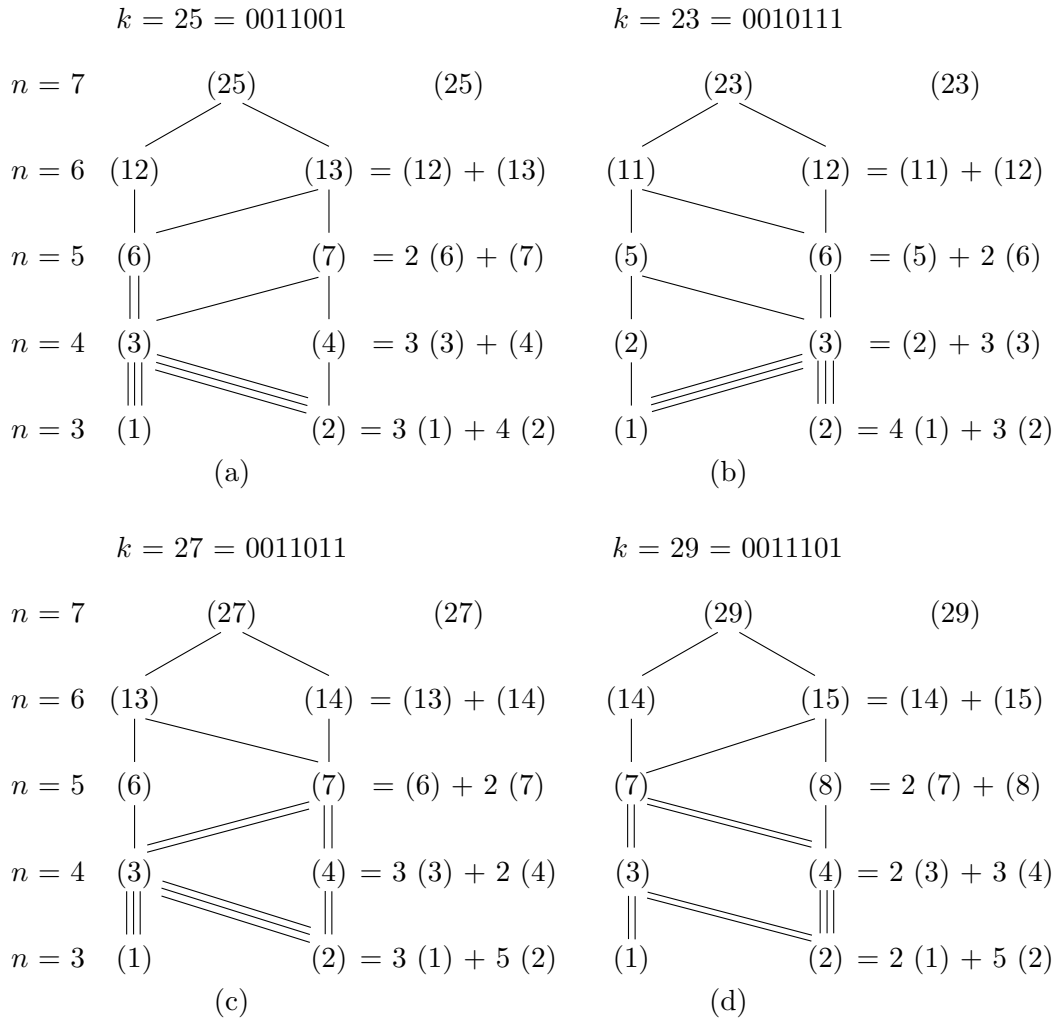

Figure 4.1: (a) $\lambda(25, 7)$, corresponds to case 1.1. (b) $\lambda(23, 7)$, corresponds to case 1.2. (c) $\lambda(27, 7)$, corresponds to case 2.1. (d) $\lambda(29, 7)$, corresponds to case 2.2.

4.3.1 Left-to-Right Randomization Algorithm

We start with an example. Let $k = 10001001 = 2^7 + 2^4 + 2^0$. That is k is the result of adding the following three integers:

$$\begin{aligned} k &= 10000000 \\ &+ 00001000 \\ &+ 00000001 \end{aligned}$$

We will consider the different BSD representations for each integer separately and then add those different representations together to get those of k.

The first integer has no other representation according to Lemma 4.1(iii). From the same lemma, the second integer has the following additional representations $0001\bar{1}000$, $001\bar{1}\bar{1}000$, $01\bar{1}\bar{1}\bar{1}000$ and $1\bar{1}\bar{1}\bar{1}\bar{1}000$. If we add this last representation to the first integer, then k would need more than 8 sbits in this example to represent it, so we will not take it into consideration. Finally the third integer has this set of additional representations $0000001\bar{1}$, $00000\bar{1}\bar{1}$, $0000\bar{1}\bar{1}\bar{1}$, ... and $01\bar{1}\bar{1}\bar{1}\bar{1}\bar{1}\bar{1}$. We will notice that we only need to take into account the first three representations. This is because all the representations starting from the third one when added to the first two integers will yield a 0 in sbit position 3, $\bar{1}$s in the lower positions and a sbit pattern in the upper positions that has been accounted for in the different representations of the second integer.

Also, we can consider that this representation of k

$$\begin{aligned} k &= 10000000 \\ &+ 001\bar{1}\bar{1}000 \\ &+ 0000\bar{1}\bar{1}\bar{1} \\ \hline &= 101\bar{1}0\bar{1}\bar{1}\bar{1} \end{aligned}$$

could be obtained from this one

$$\begin{aligned} k &= 10000000 \\ &+ 001\overline{1}\overline{1}000 \\ &+ 00000\overline{1}\overline{1}\overline{1} \\ \hline &= 101\overline{1}\overline{1}\overline{1}\overline{1} \end{aligned}$$

after changing the $\overline{1}1$ at sbit positions 3 and 2 to $0\overline{1}$.

Thus, the underlying idea of the algorithm is that the binary representation of k is subdivided into groups of bits—of different lengths—such that each group is formed by a number of consecutive 0s ending with a single 1. For each of these groups a random BSD representation is chosen as in the proof of Lemma 4.1(ii). Whenever the choice yields a $\overline{1}$ at the end of a group—which happens when any representation for that group other than the binary one is chosen—and a 1 at the beginning of the next group—which happens when the representation chosen for the next group is the one that has no 0s—another random decision is taken so as whether to leave those two sbits (*i.e.*, $\overline{1}1$) as they are or to change them to $0\overline{1}$.

The choice of a random representation for a certain group is done by counting the number of 0s in it, say z, and choosing a random integer $t \in [0, z]$ which will be the number of 0s to be written to the output. If t is equal to z, the last sbit to be written to the output for this group is 1, and it is actually written before considering the next group. Otherwise, t 0s, a 1 and only $z - t - 1$ $\overline{1}$s are written to the output, that is the last $\overline{1}$ is not written, but saved as the value of a variable labeled as *last*. We then do the same for the next group. If for the next group $t = 0$, we take a random decision whether to write $\overline{1}1$ to the output or $0\overline{1}$ at the boundary of the two groups. This leads to the following algorithm. Note that a 0 is prepended to k so that the BSD representation k'

generated is of length $n+1$ sbits.

Algorithm 4.3. Left-to-Right Randomization of an integer's BSD representation

---

INPUT: $k = (k_{n-1}, \ldots, k_0)_2$.

OUTPUT: $k' = (k'_n, \ldots, k'_0)_{\mathrm{BSD}}$, a random BSD representation of k.

1. Set $k_n \leftarrow 0;\ i \leftarrow n+1;\ last \leftarrow 1$.

2. for j from n down to 0 do

   if $(k_j = 1)$ then

   2.1 $t \leftarrow_R [0, i-j-1]$.

   2.2 if $(t = 0$ AND $last = \overline{1})$ then

   2.2.1 $c \leftarrow_R \{0, 1\}$.

   2.2.2 if $(c = 0)$ then

   $k'_i \leftarrow \overline{1}$.

   $i \leftarrow i-1;\ k'_i \leftarrow 1$.

   2.2.3 else

   $k'_i \leftarrow 0$.

   2.3 else

   2.3.1 if $(last = \overline{1})$ then

   $k'_i \leftarrow \overline{1}$.

   2.3.2 while $(t > 0)$ do

   $i \leftarrow i-1;\ k'_i \leftarrow 0;\ t \leftarrow t-1$.

   2.3.3 $i \leftarrow i-1;\ k'_i \leftarrow 1$.

   2.4 if $(i = j)$ then

   $last \leftarrow 1$.

   2.5 else

   2.5.1 while $(i > j+1)$ do

   $i \leftarrow i-1;\ k'_i \leftarrow \overline{1}$.

2.5.2 $i \leftarrow i - 1;\ last \leftarrow \overline{1}$.

3. if $(last = \overline{1})$ then
   $k'_i \leftarrow \overline{1}$.

4. while $(i > 0)$ do
   $i \leftarrow i - 1;\ k'_i \leftarrow 0$.

We note the following about the algorithm:

- The algorithm runs in $O(n)$ time.

- The sbits of k' are written to the output one at a time in their correct order from left to right. This means that there is no need to store k' in an application where the sbits are processed from left to right as they are generated. For example, it is advantageous to perform the elliptic curve scalar multiplication from left to right, especially when a mixed (projective and affine) coordinate system is used for saving computational cost in scalar multiplication [LD99; CMO98]. If the randomization of the BSD representation of the key is needed during the scalar multiplication, then Algorithm 4.3 can be readily interleaved with point doubling and point addition operations. Thus, it is more beneficial than Ha-Moon's algorithm [HM02a] (cf. Chapter 3), where the generated representation is first stored in order to use it in the scalar multiplication. Note that a BSD representation would probably need twice the storage required for the binary representation, if each sbit is internally represented by two bits.

- The minimum and/or the maximum number of 0s allowed in each group of the resulting BSD representation can be set by changing the range from which t is randomly chosen in step 2.1, *e.g.*, the minimum value can be some fraction of

$z = i - j - 1$. This is interesting if it is desired to keep the Hamming weight of the representation low. Moreover, in step 2.2.1, a bias could be given to choosing 1 more than 0 in order to make it more likely to choose $0\bar{1}$ at the group boundaries than $\bar{1}1$.

- When there is a group with a long run of 0s in k, considering the possible random representation for that group, we can see that the most significant bit of that group will be 0 with high probability among the resulting representations. This is also the case for a long run of 1s. Though each of those 1s forms a group, the randomization at the boundary of the group ensures that the most significant bit of that run is also more likely to be 0. That is $01111 = 1\bar{1}111 = 10\bar{1}11 = 100\bar{1}1 = 1000\bar{1}$. This agrees with the observation of Fouque *et al.* [FMPV04] on their right-to-left attack that, after a long run of 0s or 1s, the probability of the sbit being 0 becomes close to 1.

- The number of representations of a group is the length of that group. Assume that an n-bit integer k has all groups of the same length $l \geq 2$ (since for $l = 1$, k can be considered as having one group as in the previous note). If we do not consider the randomization at the group boundaries, then the lower bound on the number of representations of k, $\lambda(k, n)$, is $(l)^{\frac{n}{l}}$. If we consider the boundary randomization as one more representation of the group—except for the last group, then the upper bound on $\lambda(k, n)$ is $(l+1)^{\frac{n}{l}-1}\, l \approx (l+1)^{\frac{n}{l}}$. The actual number is closer to the upper boundary than the lower one. The upper boundary strictly decreases with l. We conclude that an integer with more groups has a larger number of BSD representations than another integer of the same length with fewer groups. As mentioned in the previous note, a run of 1s can be considered as one group. Thus,

the former conclusion is equivalent to saying that an integer with a better random distribution of its bits has more representations than another integer with longer runs of 0s or 1s. In Section 4.3.2, we will derive an expression for the number of BSD representations when $l = 2$, which will be of the order of the upper boundary.

- It could be further investigated whether the output BSD representations of this algorithm as well as those of the HM algorithm (cf. Section 3.1.2) are equally probable.

4.3.2 Left-to-Right Generation Algorithm

The algorithm presented here is a modified version of Algorithm 4.3 that recursively and exhaustively substitutes every group of 0s ending with a 1 with one of its possible forms. It also takes into consideration the alternative substitutions at the group boundary when the representation of a group ends with a $\overline{1}$ and that of the next group starts with 1. This algorithm can be used as a straight-forward check that Algorithm 4.3 is capable of generating any possible—$(n+1)$ sbits long—BSD representation of an integer k in the range $[1, 2^n - 1]$, *i.e.*, there is no BSD representation of k that cannot be generated by that algorithm. It was tested on small values of n.

Algorithm 4.4. Left-to-Right Generation of all BSD representations of k

---

INPUT: $k = (k_{n-1}, \ldots, k_0)_2$.

OUTPUT: all possible strings $k' = (k'_n, \ldots, k'_0)_{\text{BSD}}$.

1. Subdivide k from left to right into groups of consecutive 0s each ending with a single 1. Store the length of each group in a look-up table G. Let g be the index of the table.

2. Set $g \leftarrow 0;\ i \leftarrow n+1;$
   $last \leftarrow 1;\ j \leftarrow i - G[g];\ k' \leftarrow \langle\rangle.$

3. for $t = 0$ to $i - j - 1$ do

   `ChooseForm`$(k, g, t, i, j, last, k')$.

Algorithm 4.5. `ChooseForm`$(k, g, t, i, j, last, k')$, a recursive procedure employed by Algorithm 4.4.

INPUT: k, g, t, i, j, $last$ and k'.

OUTPUT: returns k', a string of sbits, as a possible BSD representation of k.

1. if $(t > 0)$ OR $(last = 1)$ then //this step is equivalent to step 2.3 in Algorithm 4.3.

   1.1 if $(last = \overline{1})$ then

   $k' \leftarrow k'|\overline{1}$. //concatenate k' with $\overline{1}$.

   1.2 while $(t > 0)$ do

   $i \leftarrow i - 1;\ k' \leftarrow k'|0;\ t \leftarrow t - 1$.

   1.3 $i \leftarrow i - 1;\ k' \leftarrow k'|1$.

2. if $(i = j)$ then

   $last \leftarrow 1$.

3. else

   3.1 while $(i > j + 1)$ do

   $i \leftarrow i - 1;\ k \leftarrow k'|\overline{1}$.

   3.2 $i \leftarrow i - 1;\ last \leftarrow \overline{1}$.

4. if $(j = 0)$ then

   4.1 if $(last = \overline{1})$ then

   $k' \leftarrow k'|\overline{1}$.

   4.2 Return(k').

5. $g \leftarrow g + 1;\ j \leftarrow j - G[g]$.

6. if $(j = 0)$ AND (k is even) then

6.1 if $(i > 0)$ then

6.1.1 if $(last = \overline{1})$ then

$k' \leftarrow k'|\overline{1}$.

6.1.2 while $(i > 0)$ do

$i \leftarrow i - 1;\ k' \leftarrow k'|0$.

6.2 Return(k').

7. $t \leftarrow 0$.

8. if $(last = \overline{1})$ then

`ChooseForm`$(k, g, t, i - 1, j, last, k'|\overline{1}1)$.

`ChooseForm`$(k, g, t, i, j, last, k'|0)$.

9. else

`ChooseForm`$(k, g, t, i, j, last, k')$.

10. for t from 1 to $i - j - 1$ do

`ChooseForm`$(k, g, t, i, j, last, k')$.

To better explain the different behaviors of the algorithm, we present in Figure 4.2 the tree explored by the algorithm for $k = 21$ and $n = 5$. This tree is explored by the algorithm in a *depth-first* fashion. That is, the recursive function `ChooseForm` is first called from Algorithm 4.4 at node a in the figure. Then this function calls itself at node b and then at node c where it returns with the first BSD representation for $k = 21$ which is $1\overline{1}1\overline{1}1\overline{1}$. With the flow control at node b the function calls itself at node d where the second BSD representation is generated and so forth.

The algorithm proceeds as follows. The integer k is first subdivided into groups of bits where each group consists of consecutive 0s and a single 1 at the end as for Algorithm 4.3. Starting from the leftmost group, a particular BSD representation for that group is chosen

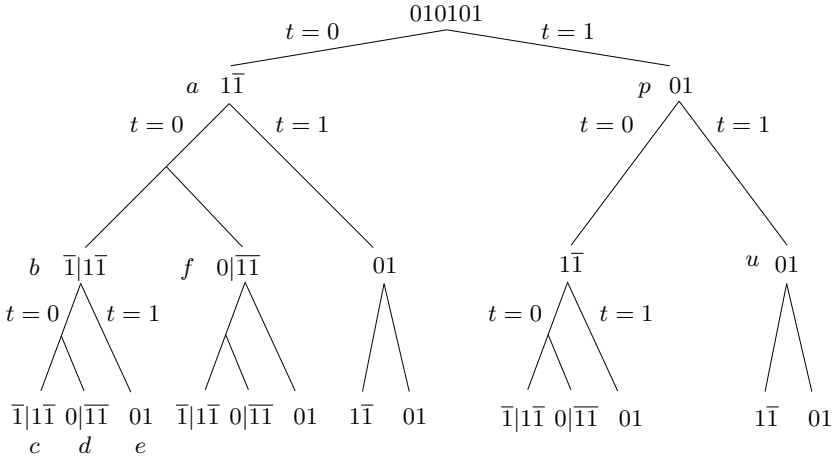

Figure 4.2: The tree explored by Algorithm 4.4 for $k = 21$ and $n = 5$.

starting with the one that contains no 0s (*i.e.*, $t = 0$, where t is the number of 0s to be written to the output for that group as before). The representation is formed inside the function `ChooseForm` which takes t as one of its arguments. In turn, this function goes through the possible values of t for the following group and, for each one, calls itself to form the corresponding BSD representation of that group. When t is equal to 0, the two possible alternatives at the group boundary are considered as was explained in Section 4.3.1. For example, in Figure 4.2, the last sbit in the group at node a may remain $\overline{1}$ or change to 0 depending on the random decision taken at the group boundary when $t = 0$ for the next group. This is why this last sbit is written at nodes b and f before the symbol '|' which designates the boundary between the groups.

The *worst-case* complexity analysis of Algorithm 4.4 is presented in the following. The worst case occurs for the integer with the maximum number of BSD representations in the range $[1, 2^n - 1]$. There are actually two integers with this property for any given n, which we referred to as $k_{max_1,n}$ and $k_{max_2,n}$ in Section 4.1.4. We mentioned that,

for n even, $k_{max_1,n}$ is of the form $(\langle 0\ 1\rangle^{\frac{n}{2}})_2$. For example, $k_{max_1,6} = 21 = (010101)_2$ (see Figure 4.3). We also mentioned that, for any n, $k_{max_2,n} = k_{max_1,n+1}$. For example, $k_{max_1,5} = 11 = (01011)_2$ and $k_{max_2,5} = 21 = (10101)_2$ (see Figure 4.3). Therefore, our analysis is conducted on those integers k of the binary form $(\langle 0\ 1\rangle^{\frac{n}{2}})_2$ for n even and $(1\langle 0\ 1\rangle^{\frac{n-1}{2}})_2$ for n odd. In the following discussion, we will drop the subscript n from $k_{max_1,n}$ and $k_{max_2,n}$ for simplicity, since it will be obvious from the context.

For n even, we have the following.

$n = 2$: $k_{max_1} = k_{max_2} = 1 = (01)_2$,

$\delta(1, 2) = \lambda(1, 3) = 3$.

The tree explored for this integer is of the same structure as the one having as root the node b in Figure 4.2. The difference is that, in this case, t can take the values 0, 1 and 2 and the corresponding representations are $1\overline{11}$, $01\overline{1}$and 001, respectively, with only one representation for $t = 0$.

$n = 4$: $k_{max_1} = 5 = (0101)_2$,

$\delta(5, 4) = \lambda(5, 5) = 8 = 3 \cdot 3 - 1$.

The tree explored for this integer is of the same structure as the one having as root a in Figure 4.2.

$n = 6$: $k_{max_1} = 21 = (010101)_2$,

$\delta(21, 6) = \lambda(21, 7) = 21 = 3 \cdot 3 \cdot 3 - (3.1 + 3)$.

The tree explored for this integer is illustrated in Figure 4.3.

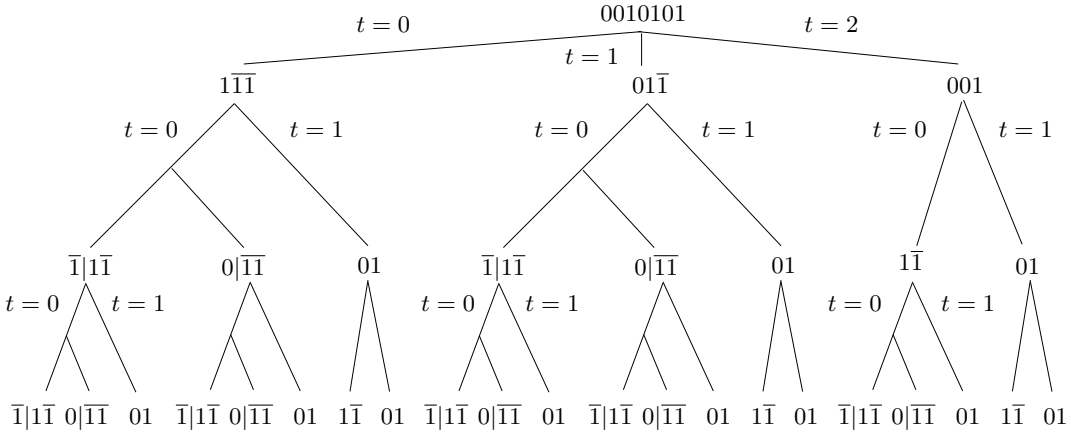

Figure 4.3: The tree explored by Algoritm 4.4 for $k = 21$ and $n = 6$.

Let $m = \frac{n}{2}$. By induction, we can deduce the following

$$\begin{aligned}\lambda(k_{max_1}, n+1) = 3^m - (m-1)3^{m-2} + \left(\sum_{i_1=1}^{m-3} i_1\right) 3^{m-4} \\ - \left(\sum_{i_1=1}^{m-5}\sum_{i_2=1}^{i_1} i_2\right) 3^{m-6} + \left(\sum_{i_1=1}^{m-7}\sum_{i_2=1}^{i_1}\sum_{i_3=1}^{i_2} i_3\right) 3^{m-8} - \cdots .\end{aligned} \tag{4.28}$$

For n odd, we have the following.

$n = 1$: $k_{max_1} = k_{max_2} = 1 = (1)_2$,

$\delta(1,1) = \lambda(1,2) = 2$.

The tree explored for this integer is simply the same as the one having as root the node u in Figure 4.2.

$n = 3$: $k_{max_2} = 5 = (101)_2$,

$\delta(5,3) = \lambda(5,4) = 5 = 2 \cdot 3 - 1$.

The tree explored for this integer is the same as the one having as root the node p in Figure 4.2.

$n = 5$: $k_{max_2} = 21 = (10101)_2$,

$\delta(21, 5) = \lambda(21, 6) = 13 = 2 \cdot 3 \cdot 3 - (3 + 2 \cdot 1)$.

The tree for this integer is the one illustrated in Figure 4.2.

Let $m = \frac{n-1}{2}$. By induction we can deduce the following

$$\begin{aligned}\lambda(k_{max_2}, n+1) = 2 \cdot 3^m &- \left[3^{m-1} + 2(m-1)3^{m-2}\right] \\ &+ \left[(m-2)3^{m-3} + 2\left(\sum_{i_1=1}^{m-3} i_1\right)3^{m-4}\right] \\ &- \left[\left(\sum_{i_2=1}^{m-4} i_2\right)3^{m-5} + 2\left(\sum_{i_1=1}^{m-5}\sum_{i_2=1}^{i_1} i_2\right)3^{m-6}\right] \\ &+ \left[\left(\sum_{i_2=1}^{m-6}\sum_{i_3=1}^{i_2} i_3\right)3^{m-7} + 2\left(\sum_{i_1=1}^{m-7}\sum_{i_2=1}^{i_1}\sum_{i_3=1}^{i_2} i_3\right)3^{m-8}\right] \\ &- \cdots .\end{aligned} \tag{4.29}$$

From this discussion, for any n the maximum number of BSD representations generated by Algorithm 4.4 appears to be $O(3^{\lfloor \frac{n}{2} \rfloor})$. However, a tighter bound on the maximum number of BSD representations is obtained from (4.27) and is given by $O(\varphi^n)$, where the golden ratio $\varphi \approx 1.618 < 3^{\frac{1}{2}}$.

On a related note, Equations (4.28) and (4.29) can be considered another solution for the Fibonacci number $F(n-2)$ (cf. (4.27)). The first few terms of these formulas can then be used as an approximation to a Fibonacci number where floating point arithmetic is not available.

4.3.3 Effect of Prepending 0s to k on the Number of its BSD Representations

In this section, we show how the number of BSD representations of k increases if we lengthen its binary representation by adding 0s at the most significant end.

If we compare Figure 4.2 with Figure 4.3, we see that for the same integer $k = 21$, increasing n from 5 to 6 had the effect of increasing the number of branches emerging from the root by one. The added branch has the same tree structure as the leftmost branch a in Figure 4.2. This is because the number of BSD representations of the first group—recall how the integer is subdivided into groups—has increased by one. Since all representations of a group, except for the original binary representation, end with a $\overline{1}$, the added representation would generate two alternatives when, for the next group, $t = 0$. If we increase n to 7, another subtree like the one having as root a will be added to the tree. The same subtree is repeated with every 0 prepended to the binary representation of k. It is easy to verify that this is true for any integer k.

As was mentioned before, the subtree having as root the node a is the tree explored for $k = 5$ and $n = 4$. In general, the subtree that is repeated is the one formed for the integer with the binary representation having the same groups as k except for the leftmost group, *i.e.*, with the most significant 1 removed. This integer can be expressed as $k - 2^{\lfloor \log_2 k \rfloor}$ for any k that is not a power of 2. A 0 should replace the most significant 1 that was removed in order to ensure a correct number of branches at the first level, that is to compensate for the other branch corresponding to the case when $t = 0$ and $last = \overline{1}$. Hence, the bit length of $k - 2^{\lfloor \log_2 k \rfloor}$ should be $\lfloor \log_2 k \rfloor + 1$. That is

$$\begin{aligned} \Delta(k) &= \lambda(k, n+1) - \lambda(k, n), \\ &= \lambda(k, n+i+1) - \lambda(k, n+i) \quad \text{for any } i \geq 0, \\ &= \lambda(k - 2^{\lfloor \log_2 k \rfloor}, \lfloor \log_2 k \rfloor + 1). \end{aligned} \tag{4.30}$$

where $\Delta(k)$ is the number of leaves in the repeated subtree. Based on (4.30), we have the following theorem.

Theorem 4.2 *If Δn is the number of 0s prepended to the binary representation of an integer k, then the number of its BSD representations increases by $\Delta n \cdot \Delta(k)$.*

4.4 Experimental Results

In this section, we present experimental results related to the speed and usage of Algorithm 4.2. We consider an application where we need to choose long integers, *e.g.*, $n = 160$, having a large number of BSD representations, *e.g.*, more than 2^{40}.

First, we have shown in Section 4.2 that Algorithm 4.1 runs in $O(n)$. As a simple illustration, for $n = 160$, the integer that has the maximum number of BSD representations is $k_{max_1,160} = (\langle 0\ 1 \rangle^{80})_2$ and the number of these representations is

$$\delta(k_{max_1,160}, 160) = (9E449CF5F9D5F28B6248B9097ED8)_{hex}.$$

This result was computed in approximately $0.38\mu s$ on a 1.5 GHz Pentium M processor (Centrino technology), using the BN (big numbers) library[1]. This is an empirical indication of how much time this algorithm can take when executed on a concurrent laptop platform. As we mentioned previously, the algorithm could be modified to run iteratively rather than recursively. Each iteration would then consist of simple operations such as copy/move, shift right, increment/decrement and addition operations.

In Section 4.3.2, we have shown that $\delta(k_{max_1,n}, n)$ is $O(\varphi^n)$. As we have mentioned in the example above, for $n = 160$, this number is a 111-bit integer. Using this information, one can proceed as follows in order to choose an integer with a large number of BSD representations:

[1]Provided by Eric Young as part of his implementation of SSL, known as *openssl*. Available from `http://www.openssl.org/source/openssl-0.9.7d.tar.gz`

Step 1: Specify the minimum number of BSD representations that an integer should have, that is a selection threshold T.

Step 2: Choose at random an integer k of length n.

Step 3: Use that integer as an input to Algorithm 4.1 to compute the number of its BSD representations, $\delta(k, n)$.

Step 4: If $\delta(k, n) \geq T$, then accept the integer k. Otherwise, reject it and go to Step 2.

From *the theory of runs* [Kal85, Sec. 2.7], if k is an n-bit integer with $\frac{n}{2}$ 0s and $\frac{n}{2}$ 1s, then the most probable number of runs in k is between $\frac{n}{2}$ and $\frac{n}{2} + 3$, that is the probability that k would have several long runs of 0s and 1s—and hence fewer runs—is small. Moreover, from the same theory, we can calculate the probability of having h runs for such an integer for any h. For example, for $n = 160$, an integer with 80 0s and 80 1s has the following probabilities of having h runs.

$$Pr(h = 20) = 9 \times 10^{-25},$$
$$Pr(h = 40) = 1.7 \times 10^{-11}.$$

Based on this theory and the last note in Section 4.3.1, it is expected that the percentage of integers that would be rejected by the above selection procedure is negligible for large n. We can illustrate this fact experimentally as follows.

Let $\alpha = \log(\delta(k, n))/\log(\delta(k_{max_1,n}, n))$. For $n = 160$, let the selection threshold be $T = 2^{40}$, then we would like to know the percentage of integers in the range $[0, 2^n - 1]$ having $\alpha < 40/111 = 0.36036$. For large n, it is computationally infeasible to calculate this percentage in a deterministic way. However, for small n ($n < 32$), our experiments show that this percentage is negligible and is strictly decreasing as n increases. For

example, this percentage is 0.00132%, 0.00093% and 0.00065% for $n = 29, 30$ and 31, respectively. If we set $T = 2^{80}$, then the percentage of rejected integers, *i.e.*, those having $\alpha < 80/111 = 0.72072$ is 10.09%, 9.62% and 9.17% for $n = 29, 30$ and 31, respectively. The actual value of T will depend on the application and, as mentioned in Step 1 on page 86, it can be set based on the minimum number of BSD representations that an integer should have.

4.5 Conclusion

In this chapter, we have presented some interesting issues related to the number of binary-signed digit (BSD) representations of an integer $k \in [0, 2^n - 1]$, such as the average number of representations among integers of the same length and the bit patterns of $k_{max_1,n}$ and $k_{max_2,n}$, *i.e.*, the integers of length n bits that have the maximum number of BSD representations. We have presented the recurrence that governs the number of representations of such integers and have, hence, proved that it is a Fibonacci number and is $O(\varphi^n)$, where $\varphi \approx 1.618$ is the golden ratio [Kos01]. To the best of our knowledge, there has been no precedence to our results in regard to these issues.

We have introduced an algorithm that calculates in $O(n)$ the exact number of BSD representations of k that are of length n sbits, and have illustrated the algorithm's efficiency for $n = 160$, which is of interest for elliptic curve cryptographic applications. We have also introduced an algorithm that generates in $O(n)$ a random BSD representation of k by scanning its bits starting from the most significant end, and outputs the sbits in their correct order one at a time. In addition, we have presented an algorithm that can generate all BSD representations of an integer, which has helped us provide another expression for the number of representations of $k_{max_1,n}$. We have also proved that

prepending 0s to the binary representation of an integer results in only a linear increase in the number of its BSD representations and presented a way to compute this increase.

We have also presented some experimental results that show that the percentage of integers of a certain length having a relatively small number of representations is negligible.

Moreover, the formula we have provided for $k_{max_1,n}$ is considered an alternate formula for Fibonacci numbers that can be computed without using floating point arithmetic, or the first few terms of which used as an approximation. It is interesting to study how many terms are necessary for a *good* approximation.

Chapter 5

τ-adic Representations of Integers

Koblitz curves [Kob92], E_a where $a \in \{0, 1\}$, are elliptic curves defined over $\mathbb{F}_2$ (cf. Section 2.1.4). Their advantageous characteristic is the Frobenius mapping which can be exploited to replace the point doubling operation with a simple squaring of the underlying field elements, *i.e.*, the point coordinates [Sol00]. Hence, the scalar multiplication algorithm of a point $P \in E(\mathbb{F}_{2^m})$ can be executed in a much shorter time. This technique is generally not as efficient when using an arbitrary endomorphism. In order to use this mapping efficiently, Solinas [Sol00] has shown how to represent the scalar k in a number system of base τ, that is

$$k = \sum_{i=0}^{l-1} \kappa_i \tau^i, \tag{5.1}$$

where $\kappa_i \in \{-1, 0, 1\}$ and τ is a complex number representing the squaring map and satisfying

$$\tau^2 + 2 = \mu\tau, \qquad \mu = (-1)^{1-a}. \tag{5.2}$$

His representation is also characterized by being a *non-adjacent form* (NAF) where no two adjacent symbols are non-zeros, *i.e.*, $\kappa_i \kappa_{i+1} = 0$ for $i \geq 0$ in order to minimize the

number of point additions—abusing the notation, we will refer to κ_i as a signed bit or *sbit*.

In this chapter, we first present our experimental results on an open problem proposed by Solinas. This problem questions the uniform distribution of points resulting from multiplying a randomly chosen τ-adic NAF (τNAF) by an input point.

We then present an efficient algorithm that takes as input the τ-adic NAF representation and produces a random τ-adic representation for the same scalar value. The symbols of the randomized τ-adic representation are output one at a time from right to left which allows the execution of the right-to-left scalar multiplication along with the randomization algorithm without the need to store the new representation. The model of our algorithm has enabled us to derive a number of interesting results with regard to τ-adic representations that we present subsequently. Our results include the characteristics of τNAFs that have the maximum number of representations and formulas describing that number, the average Hamming density of the representations. We then present deterministic methods for determining both the average and the exact number of representations of τNAFs of a certain length.

5.1 τNAFs of length $m + a$ and their Distribution

In [Sol00], Solinas has first shown how to represent k as in (5.1) as a τNAF. However, the direct recoding method results in $l \approx 2m$. Therefore, he proposed a *reduced τ-adic non adjacent form* (RTNAF) for k where k is first reduced modulo $\delta = (\tau^m - 1)/(\tau - 1)$ resulting in $l = m + a$. He has proved that in a τNAF representation the number of 0s is $\frac{2}{3}$ on average. He also mentioned that 1 and -1 are equally likely on average.

To obtain a key k represented in a reduced τNAF, we can choose $k \in [1, u - 1]$ where

u is the prime order of the main subgroup, and apply Solinas' method to produce its RTNAF. Alternatively, as Solinas suggests [Sol00], we can directly choose a τNAF of length $m + a$ as follows: the first sbit is generated according to the following probability distribution

$$\kappa_i = \begin{cases} 0 & Pr(0) = 1/2 \\ 1 & Pr(1) = 1/4 \\ \overline{1} & Pr(\overline{1}) = 1/4. \end{cases} \tag{5.3}$$

We follow each 1 or $\overline{1}$ with a 0, and after each 0 the subsequent sbit is generated according to the distribution in (5.3).

This method can be verified as follows. We can consider the sequence of sbits in a random τNAF as a Markov chain of three states, namely 0, 1 and $\overline{1}$. We have the limiting probabilities as follows [Sol00]

$$\pi_0 = 2/3 \quad \text{and} \quad \pi_1 = \pi_{\overline{1}} = 1/6. \tag{5.4}$$

Also, from the properties of the NAF representation, we know that a 1 or a $\overline{1}$ must be followed by a 0. Hence we have the following transition probabilities

$$P_{10} = P_{\overline{1}0} = 1 \quad \text{and} \quad P_{11} = P_{1\overline{1}} = P_{\overline{1}1} = P_{\overline{1}\overline{1}} = 0. \tag{5.5}$$

It remains to determine P_{00}, P_{01} and $P_{0\overline{1}}$, which we can calculate by solving the equation

$$\boldsymbol{\pi}\mathbf{P} = \boldsymbol{\pi}, \tag{5.6}$$

where $\boldsymbol{\pi} = (\pi_0\ \pi_1\ \pi_{\overline{1}})$ and $\mathbf{P}$ is the transition matrix

$$\mathbf{P} = \begin{pmatrix} P_{00} & P_{01} & P_{0\overline{1}} \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix} \tag{5.7}$$

We obtain a unique solution to (5.6) which is

$$P_{00} = 1/2 \quad \text{and} \quad P_{01} = P_{0\bar{1}} = 1/4. \tag{5.8}$$

The sequence obtained by this method is selected from the set of all τNAFs of length $m + a$. As stated by Solinas [Sol00], their number is the integer closest to $2^{m+a+2}/3$, whereas the order of the main subgroup is $u \approx 2^{m-2+a}$. That is the average number of sequences that, when multiplied by a given point P, would lead to the same point in the main subgroup is 16/3. The deviation from this average is an open problem. We have calculated this deviation experimentally for E_1 over small fields as follows.

We have generated all τNAFs of length $m + a$ for small m. We have then reduced each of them modulo δ, and stored how many times each of the u lattice point $\lambda_0 + \lambda_1\tau$ ($\lambda_i \in \mathbb{Z}$) in $\mathcal{V}$, which is the region spanned by the elements of $\mathbb{Z}[\tau]/\delta\mathbb{Z}[\tau]$, is mapped. The mean and standard deviation of the distribution of the number of mappings for $E_1(\mathbb{F}_{2^m})$ for small m are shown in Table 5.1.

Table 5.1: The mean and standard deviation of the number of times the lattice points of the region $\mathcal{V}$ are mapped by all τNAFs of length $m + 1$.

m	7	11	17	19	23
u	71	991	65587	262543	4196903
mean	4.803	5.511	5.329	5.325	5.330
standard deviation	0.721	0.734	0.523	0.502	0.482

As we can see from Table 5.1, the mean approaches $\frac{16}{3}$ as m increases. Moreover, the deviation is small and is decreasing starting from $m = 11$. Also, in our experiments the number of times a lattice point was mapped was at most 8. Hence, if these results can be generalized to larger fields, then we can provide an affirmative answer to the question

whether it is cryptographically secure to choose a point multiplier (key) by choosing a random τNAF.

5.2 Randomizing the τ-adic Representation of an Integer

Now, having the key k represented as a τNAF, we will present a randomization algorithm to obtain a different τ-adic representation of the key. The technique used in this algorithm is similar to the one used by Ha and Moon [HM02a] (cf. Chapter 3) to randomize the binary representation of the key. The difference is in the state representation which is similar to the one we used in Figure 3.4. We presented the underlying idea of the HM algorithm as follows. A carry bit is initialized to 0 and the input binary representation is scanned, one bit at a time, starting from the least significant one. Whenever the sum of the current scanned bit and the carry is $0 \pmod 2$, the output sbit is 0, otherwise, if the sum is 1, a random decision is drawn as to whether send a 1 or a -1 to the output. In all cases the carry bit is updated properly. For example, if the current sum is 1 and the output is chosen to be -1, then 2 should be added to the remaining input bits, this is ensured by setting the carry bit to 1.

Similarly, the underlying idea of our τ-adic randomization algorithm is as follows. The sbits of the input τNAF are scanned starting from the least significant end. Whenever the scanned bit value, added to the current carry sbit, is 1, a random decision is drawn based on which the current output sbit is determined. If the latter was chosen to be 1, no change occurs in the carry sbits and the following sbit of the input as well as that of the carry are scanned. On the other hand, if the output sbit was chosen to be $\overline{1}$, this is equivalent to subtracting 2 from the current τNAF and should be compensated by adding 2 back to it. For the curve E_0, $2 = -\tau^2 - \tau = (\overline{1}\overline{1}0)_\tau$ and for the curve E_1, $2 = -\tau^2 + \tau = (\overline{1}10)_\tau$.

Hence, the addition of 2 to the remaining sbits of the τNAF is handled by adding the τ-adic representation of 2 to the carry sbits. Since whenever the candidate output bit is ± 1, a random decision is taken, then all possible τ-adic representations of length up to $l+2$, as will be explained, for the input τNAFcan be output by this algorithm. This idea is captured in the following pseudocode where the subscript τ is omitted since it is implied. The length of the output representation, the prepending of three 0s to the input as well as the number of carry sbits needed will be explained in the subsequent discussion of the algorithm implementation.

Algorithm 5.1. Randomization of the τ-adic representation

INPUT: $k = (\kappa_{l-1}, \ldots, \kappa_1, \kappa_0)$ where k is a τNAF.

OUTPUT: $k' = (d_{l+1}, \ldots, d_0)$, a random τ-adic representation of k.

1. Prepend $(\kappa_{l+2}, \kappa_{l+1}, \kappa_l) = (0, 0, 0)$ to k.

2. $(c_{2_i}, c_{1_i}, c_{0_i}) \leftarrow (0, 0, 0)$. // carry sbits.

3. for i from 0 to $l+2$ do

   3.1 $b_i \leftarrow \kappa_i + c_{0_i}$; $r_i \leftarrow_R \{0, 1\}$. // r_i is random bit

   3.2 if $(b_i = 0)$ then

   $d_i \leftarrow 0$; $(c_{2_{i+1}}, c_{1_{i+1}}, c_{0_{i+1}}) \leftarrow (0, c_{2_i}, c_{1_i})$.

   3.3 else if $(b_i = \pm 2)$ then

   $d_i \leftarrow 0$; $(c_{2_{i+1}}, c_{1_{i+1}}, c_{0_{i+1}}) \leftarrow (0, c_{2_i}, c_{1_i}) \pm 2/\tau$.

   // for E_0, $2/\tau = (\overline{11})$ and for E_1, $2/\tau = (\bar{1}1)$.

   3.4 else // $b_i = \pm 1$

   3.4.1 if $(r_i = 0)$ then

   $d_i \leftarrow b_i$; $(c_{2_{i+1}}, c_{1_{i+1}}, c_{0_{i+1}}) \leftarrow (0, c_{2_i}, c_{1_i})$.

3.4.2 else

$$d_i \leftarrow -b_i;\ (c_{2_{i+1}}, c_{1_{i+1}}, c_{0_{i+1}}) \leftarrow (0, c_{2_i}, c_{1_i}) + b_i * 2/\tau.$$

As an illustration of the algorithm outcome, let $k = (10\overline{1})_\tau$ be the input τNAF, then the algorithm would output one of the following representations for the curve E_0: $(10\overline{1})_\tau, (\overline{11}10\overline{1})_\tau$, $(\overline{11}011)_\tau, (\overline{1}1\overline{1}1)_\tau$. Note that the τNAF is a possible output if the input sbits are sent to the output unchanged and the carry remains 0. Moreover, since the τNAF of an element of the ring $\mathbb{Z}[\tau]$ is unique [Sol00, Theorem 1], the other representations have adjacent non-zero sbits, *i.e.*, are not τNAFs.

The algorithm can be implemented as a look-up table as in Table 5.2 for the curve E_1. It is also best described as a *nondeterministic finite automaton* (NFA) as in Figure 5.1, which is discussed in Section 5.5.2. As mentioned above, the sbit sequence of the key is scanned from the least significant end to the most significant end. The current state s_i is the combination of the current sbit κ_i and the carry sbits $(c_{2_i}\ c_{1_i}\ c_{0_i})_\tau$. Based on the next sbit κ_{i+1} and the random decision bit r_i, the output sbit d_i and the next state s_{i+1} are determined. Depending on whether κ_0 is $\overline{1}$, 0 or 1 the first state S_0 will be s_4, s_{12} or s_{20} respectively where the carry sbits are initialized to 0. Note that only the states in Table 5.2 are reachable, that is, not all combinations of the carry sbits occur in the algorithm. Moreover, by verifying the different states of the algorithm, we can observe that only three carry sbits are needed.

We will illustrate the calculation of the carry sbits and the state transitions using the following example. Let $k = (10010\overline{1})_\tau$. Then, $\kappa_0 = \overline{1}$ and $c_{2_0} = c_{1_0} = c_{0_0} = 0$ ($S_0 = s_4$). If $r_0 = 0$, $d_0 = \kappa_0 = \overline{1}$, the carry sbits do not change and the next state $S_1 = s_{12}$. Otherwise, $d_0 = 1$; to change the value of κ_0 from $\overline{1}$ to 1, we should add (-2) to the remaining sbits of k. For the curve E_1, $-2 = \tau^2 - \tau = (1\overline{1}0)_\tau$. This results in the carry

sbits being $c_{2_1} = 0, c_{1_1} = 1, c_{0_1} = \overline{1}$, and the next state $S_1 = s_{14}$.

The output sbit d_i is determined by $\kappa_i + c_{0_i}$. If the latter is 0 or ± 2, then $d_i = 0$, and the carry sbits are adjusted accordingly, *e.g.*, as in the states s_2 and s_3 in Table 5.2. Otherwise, if $\kappa_i + c_{0_i} = \pm 1$, then if $r_i = 0$, then $d_i = \kappa_i + c_{0_i}$, else $d_i = -(\kappa_i + c_{0_i})$ and a $\pm(\overline{1}1)_\tau$ is added to $(c_{2_i}, c_{1_i})_\tau$. Note that the output d_i is determined along with the next state S_{i+1}. In other words, when the algorithm is in state S_i, the last sbit that was sent to the output is d_{i-1}.

In Figure 5.1, the arrows are labeled with κ_{i+1}/d_i. Solid arrows correspond to transitions where r_i is $\times$, *i.e.*, only one transition per value of κ_{i+1} is possible. Dashed arrows correspond to $r_i = 0$ and dotted arrows correspond to $r_i = 1$.

Table 5.2: State transition table for the randomized τ-audic representation for the curve E_1.

State					Input		Output				Next state
S_i	κ_i	c_{2_i}	c_{1_i}	c_{0_i}	κ_{i+1}	r_i	d_i	$c_{2_{i+1}}$	$c_{1_{i+1}}$	$c_{0_{i+1}}$	S_{i+1}
s_1	$\overline{1}$	0	$\overline{1}$	0	0	0	$\overline{1}$	0	0	$\overline{1}$	s_{11}
					0	1	1	1	0	0	s_{16}
s_2	$\overline{1}$	0	$\overline{1}$	1	0	$\times$	0	0	0	$\overline{1}$	s_{11}
s_3	$\overline{1}$	0	0	$\overline{1}$	0	$\times$	0	0	1	$\overline{1}$	s_{14}
s_4	$\overline{1}$	0	0	0	0	0	$\overline{1}$	0	0	0	s_{12}
					0	1	1	0	1	$\overline{1}$	s_{14}
s_5	$\overline{1}$	0	0	1	0	$\times$	0	0	0	0	s_{12}
s_6	$\overline{1}$	0	1	$\overline{1}$	0	$\times$	0	0	1	0	s_{15}
s_7	$\overline{1}$	0	1	0	0	0	$\overline{1}$	0	0	1	s_{13}
					0	1	1	0	1	0	s_{15}
s_8	0	$\overline{1}$	0	0	$\overline{1}$	$\times$	0	0	$\overline{1}$	0	s_1
					0	$\times$	0	0	$\overline{1}$	0	s_9
					1	$\times$	0	0	$\overline{1}$	0	s_{17}
s_9	0	0	$\overline{1}$	0	$\overline{1}$	$\times$	0	0	0	$\overline{1}$	s_3
					0	$\times$	0	0	0	$\overline{1}$	s_{11}

S_i	κ_i	c_{2_i}	c_{1_i}	c_{0_i}	κ_{i+1}	r_i	d_i	$c_{2_{i+1}}$	$c_{1_{i+1}}$	$c_{0_{i+1}}$	S_{i+1}
					1	$\times$	0	0	0	$\overline{1}$	s_{19}
s_{10}	0	0	$\overline{1}$	1	$\overline{1}$	0	1	0	0	$\overline{1}$	s_3
					$\overline{1}$	1	$\overline{1}$	0	$\overline{1}$	0	s_1
					0	0	1	0	0	$\overline{1}$	s_{11}
					0	1	$\overline{1}$	0	$\overline{1}$	0	s_9
					1	0	1	0	0	$\overline{1}$	s_{19}
					1	1	$\overline{1}$	0	$\overline{1}$	0	s_{17}
s_{11}	0	0	0	$\overline{1}$	$\overline{1}$	0	$\overline{1}$	0	0	0	s_4
					$\overline{1}$	1	1	0	1	$\overline{1}$	s_6
					0	0	$\overline{1}$	0	0	0	s_{12}
					0	1	1	0	1	$\overline{1}$	s_{14}
					1	0	$\overline{1}$	0	0	0	s_{20}
					1	1	1	0	1	$\overline{1}$	s_{22}
s_{12}	0	0	0	0	$\overline{1}$	$\times$	0	0	0	0	s_4
					0	$\times$	0	0	0	0	s_{12}
					1	$\times$	0	0	0	0	s_{20}
s_{13}	0	0	0	1	$\overline{1}$	0	1	0	0	0	s_4
					$\overline{1}$	1	$\overline{1}$	0	$\overline{1}$	1	s_2
					0	0	1	0	0	0	s_{12}
					0	1	$\overline{1}$	0	$\overline{1}$	1	s_{10}
					1	0	1	0	0	0	s_{20}
					1	1	$\overline{1}$	0	$\overline{1}$	1	s_{18}
s_{14}	0	0	1	$\overline{1}$	$\overline{1}$	0	$\overline{1}$	0	0	1	s_5
					$\overline{1}$	1	1	0	1	0	s_7
					0	0	$\overline{1}$	0	0	1	s_{13}
					0	1	1	0	1	0	s_{15}
					1	0	$\overline{1}$	0	0	1	s_{21}
					1	1	1	0	1	0	s_{23}
s_{15}	0	0	1	0	$\overline{1}$	$\times$	0	0	0	1	s_5
					0	$\times$	0	0	0	1	s_{13}
					1	$\times$	0	0	0	1	s_{21}
s_{16}	0	1	0	0	$\overline{1}$	$\times$	0	0	1	0	s_7
					0	$\times$	0	0	1	0	s_{15}
					1	$\times$	0	0	1	0	s_{23}
s_{17}	1	0	$\overline{1}$	0	0	0	1	0	0	$\overline{1}$	s_{11}
					0	1	$\overline{1}$	0	$\overline{1}$	0	s_9
s_{18}	1	0	$\overline{1}$	1	0	$\times$	0	0	$\overline{1}$	0	s_9
s_{19}	1	0	0	$\overline{1}$	0	$\times$	0	0	0	0	s_{12}
s_{20}	1	0	0	0	0	0	1	0	0	0	s_{12}

S_i	κ_i	c_{2_i}	c_{1_i}	c_{0_i}	κ_{i+1}	r_i	d_i	$c_{2_{i+1}}$	$c_{1_{i+1}}$	$c_{0_{i+1}}$	S_{i+1}
					0	1	$\overline{1}$	0	$\overline{1}$	1	s_{10}
s_{21}	1	0	0	1	0	$\times$	0	0	$\overline{1}$	1	s_{10}
s_{22}	1	0	1	$\overline{1}$	0	$\times$	0	0	0	1	s_{13}
s_{23}	1	0	1	0	0	0	1	0	0	1	s_{13}
					0	1	$\overline{1}$	$\overline{1}$	0	0	s_8

The algorithm keeps scanning the l sbits of the input τ-adic NAF, starting from the least significant end, moving from a state to another according to the look-up table. When the most significant sbit κ_{l-1} is reached, the algorithm is in state S_{l-1}, with the last output bit d_{l-2}.

To exit the algorithm from the state S_{l-1}, the value of the current input sbit κ_{l-1} should be added to the carry $(c_{2_{l-1}}\ c_{1_{l-1}}\ c_{0_{l-1}})_\tau$ and sent to the output. We can see from Table 5.2 that, for all states, the result of this addition cannot exceed three sbits. Hence, the output τ-adic representation can be of length at most $l+2$. This exit step is equivalent to prepending at most three 0s to the τNAF and continuing the algorithm as before with all subsequent random decisions $r_i = 0$. The algorithm then stops when the state s_{12} is reached, since in this state $\kappa_i = c_{2_i} = c_{1_i} = c_{0_i} = 0$. As with adding the carry to the current sbit, it can be verified from Table 5.2 that the paths from all states to s_{12} are at most three transitions long. We will refer to those paths as exit paths. However, from some states, there exist two exit paths that satisfy this length restriction. For example, if $S_{l-1} = s_4$, then $S_l = s_{12}$ and $d_{l-1} = \overline{1}$. Alternatively, $S_l = s_{14}$, $S_{l+1} = s_{13}$, and $S_{l+2} = s_{12}$, with the respective output $d_{l-1} = 1$, $d_l = \overline{1}$, $d_{l+1} = 1$. Other states that have two possible exit paths are $s_7, s_{10}, s_{11}, s_{13}, s_{14}, s_{17}$ and s_{20}.

The same randomization technique can be applied to the τ-adic representation of integers when the points are on the curve E_0. In this case, $2 = -\tau^2 - \tau = (\overline{1}10)_\tau$, which

will produce different carry sbits than for the curve E_1, and hence different states. Those states and the transitions between them are listed in Table 5.3. For this curve, the states that have two possible exit paths are $s_2, s_4, s_9, s_{11}, s_{13}, s_{15}, s_{20}$ and s_{22}. We have included the representations of the τNAFs of length $1 \leq l \leq 6$ on the curve E_0 in Appendix D.1. As can be seen in this appendix, the number of representations is not uniform among the τNAFs, which is expected to be true for any length l. Hence, it is not favorable to choose a key by choosing a random τ-adic expansion.

Table 5.3: State transition table for the randomized τ-adic representation for the curve E_0.

State					Input		Output				Next state
S_i	κ_i	c_{2_i}	c_{1_i}	c_{0_i}	κ_{i+1}	r_i	d_i	$c_{2_{i+1}}$	$c_{1_{i+1}}$	$c_{0_{i+1}}$	S_{i+1}
s_1	$\overline{1}$	0	$\overline{1}$	$\overline{1}$	0	$\times$	0	0	1	0	s_{14}
s_2	$\overline{1}$	0	$\overline{1}$	0	0	0	$\overline{1}$	0	0	$\overline{1}$	s_{11}
					0	1	1	0	1	0	s_{14}
s_3	$\overline{1}$	0	0	$\overline{1}$	0	$\times$	0	0	1	1	s_{15}
s_4	$\overline{1}$	0	0	0	0	0	$\overline{1}$	0	0	0	s_{12}
					0	1	1	0	1	1	s_{15}
s_5	$\overline{1}$	0	0	1	0	$\times$	0	0	0	0	s_{12}
s_6	$\overline{1}$	0	1	0	0	0	$\overline{1}$	0	0	1	s_{13}
					0	1	1	$\overline{1}$	0	0	s_8
s_7	$\overline{1}$	0	1	1	0	$\times$	0	0	0	1	s_{13}
s_8	0	$\overline{1}$	0	0	$\overline{1}$	$\times$	0	0	$\overline{1}$	0	s_2
					0	$\times$	0	0	$\overline{1}$	0	s_{10}
					1	$\times$	0	0	$\overline{1}$	0	s_{18}
s_9	0	0	$\overline{1}$	$\overline{1}$	$\overline{1}$	0	$\overline{1}$	0	0	$\overline{1}$	s_3
					$\overline{1}$	1	1	0	1	0	s_6
					0	0	$\overline{1}$	0	0	$\overline{1}$	s_{11}
					0	1	1	0	1	0	s_{14}
					1	0	$\overline{1}$	0	0	$\overline{1}$	s_{19}
					1	1	1	0	1	0	s_{22}
s_{10}	0	0	$\overline{1}$	0	$\overline{1}$	$\times$	0	0	0	$\overline{1}$	s_3

S_i	κ_i	c_{2_i}	c_{1_i}	c_{0_i}	κ_{i+1}	r_i	d_i	$c_{2_{i+1}}$	$c_{1_{i+1}}$	$c_{0_{i+1}}$	S_{i+1}
					0	$\times$	0	0	0	$\overline{1}$	s_{11}
					1	$\times$	0	0	0	$\overline{1}$	s_{19}
s_{11}	0	0	0	$\overline{1}$	$\overline{1}$	0	$\overline{1}$	0	0	0	s_4
					$\overline{1}$	1	1	0	1	1	s_7
					0	0	$\overline{1}$	0	0	0	s_{12}
					0	1	1	0	1	1	s_{15}
					1	0	$\overline{1}$	0	0	0	s_{20}
					1	1	1	0	1	1	s_{23}
s_{12}	0	0	0	0	$\overline{1}$	$\times$	0	0	0	0	s_4
					0	$\times$	0	0	0	0	s_{12}
					1	$\times$	0	0	0	0	s_{20}
s_{13}	0	0	0	1	$\overline{1}$	0	1	0	0	0	s_4
					$\overline{1}$	1	$\overline{1}$	0	$\overline{1}$	$\overline{1}$	s_1
					0	0	1	0	0	0	s_{12}
					0	1	$\overline{1}$	0	$\overline{1}$	$\overline{1}$	s_9
					1	0	1	0	0	0	s_{20}
					1	1	$\overline{1}$	0	$\overline{1}$	$\overline{1}$	s_{17}
s_{14}	0	0	1	0	$\overline{1}$	$\times$	0	0	0	1	s_5
					0	$\times$	0	0	0	1	s_{13}
					1	$\times$	0	0	0	1	s_{21}
s_{15}	0	0	1	1	$\overline{1}$	0	1	0	0	1	s_5
					$\overline{1}$	1	$\overline{1}$	0	$\overline{1}$	0	s_2
					0	0	1	0	0	1	s_{13}
					0	1	$\overline{1}$	0	$\overline{1}$	0	s_{10}
					1	0	1	0	0	1	s_{21}
					1	1	$\overline{1}$	0	$\overline{1}$	0	s_{18}
s_{16}	0	1	0	0	$\overline{1}$	$\times$	0	0	1	0	s_6
					0	$\times$	0	0	1	0	s_{14}
					1	$\times$	0	0	1	0	s_{22}
s_{17}	1	0	$\overline{1}$	$\overline{1}$	0	$\times$	0	0	$\overline{1}$	$\overline{1}$	s_{11}
s_{18}	1	0	$\overline{1}$	0	0	0	1	0	0	$\overline{1}$	s_{11}
					0	1	$\overline{1}$	1	0	0	s_{16}
s_{19}	1	0	0	$\overline{1}$	0	$\times$	0	0	0	0	s_{12}
s_{20}	1	0	0	0	0	0	1	0	0	0	s_{12}
					0	1	$\overline{1}$	0	$\overline{1}$	$\overline{1}$	s_9
s_{21}	1	0	0	1	0	$\times$	0	0	$\overline{1}$	$\overline{1}$	s_9
s_{22}	1	0	1	0	0	0	1	0	0	1	s_{13}
					0	1	$\overline{1}$	0	$\overline{1}$	0	s_{10}

S_i	κ_i	c_{2_i}	c_{1_i}	c_{0_i}	κ_{i+1}	r_i	d_i	$c_{2_{i+1}}$	$c_{1_{i+1}}$	$c_{0_{i+1}}$	S_{i+1}
s_{23}	1	0	1	1	0	$\times$	0	0	$\bar{1}$	0	s_{10}

The Ha-Moon randomization algorithm [HM02a] was proposed as a countermeasure to differential power analysis (DPA) attacks on ECCs. The output of the algorithm was a random binary signed-digit (BSD) representation of the input binary representation of the key. The resulting BSD representation is then used by the binary algorithm (cf. Section 2.1.3) allowing negative digits. Later on, it was shown in [FMPV04; SPL04], that this randomization method does not serve its purpose since the number of intermediate points possibly computed at any iteration of the binary algorithm is only two. Guessing the value of an intermediate point is at the core of a conventional DPA attack. The reason behind this limited number of intermediate points is that the following relation always hold for some key $k = (k_{n-1}, \ldots, k_0)_2$ with BSD representation $k' = (k'_n, \ldots, k'_0)_2$, where $k'_i \in \{-1, 0, 1\}$

$$\sum_{i=0}^{j-1} k_i 2^i = \sum_{i=0}^{j-1} k'_i 2^i + c_j 2^j, \tag{5.9}$$

for any $0 < j \leq n$, where $c_j \in \{0, 1\}$ is the carry bit in the Ha-Moon algorithm. The carry takes only one of two values, and so does the intermediate point computed by the binary algorithm using the BSD representation of the key.

Similar arguments apply to the τ-adic representation. However, from Tables 5.2 and 5.3, we can see that the carry sbits can take one of 9 possible values. Hence, the adequacy of this randomization method as a DPA countermeasure depends on the application and the life length of the key. It is interesting to study the probability of occurrence of a certain intermediate value of the representation in relation to the sbits values of the original τ-NAF similar to the study presented by [FMPV04] on the BSD representation. It is also interesting to investigate the number of carry patterns that would result if the

input is the shortest τ-adic representation of an element of $\mathbb{Z}[\tau]$ [Sta93]. Note that in this case, the number of carry sbits may be more than 3 and it is not guaranteed that the number of states is finite.

Algorithm 5.1 is a right-to-left randomization algorithm. It could be investigated whether the left-to-right randomization approach we presented in Section 4.3.1, or a similar one, is applicable to the τ-adic representations.

5.3 τNAF with the Maximum Number of Representations

Let k be a τNAF of length l sbits, possibly having 0(s) as the leading sbit(s), and let $\vartheta(k, l)$ be the number of τ-adic representations of k. Note that those representations are of length at most $l + 2$ as in Section 5.2. In the following, we will focus our discussion on "positive" τNAFs, *i.e.*, those having $\kappa_{l-1} = \kappa_{l-2} = \ldots = \kappa_i = 0$ and $\kappa_{i-1} = 1$ for some $0 < i \leq l$. Since $-k$ is obtained from k by interchanging the $\overline{1}$s with the 1s, in the same way, the representations of $-k$ can be obtained from those of k, hence, $\vartheta(k, l) = \vartheta(-k, l)$. Let $k_{max,l}$ be the τNAF of length l that has the maximum number of representations among other τNAFs of the same length (cf. Table D.7 in Appendix D.2). Also, let $\alpha(k, l)$ be the number of representations of k that are of length at most l sbits. Then, we can prove the following theorem.

Theorem 5.1 *Let $l \geq 1$ and $w = \lfloor \frac{l-1}{2} \rfloor$. For l odd, $k_{max,l} = \tau^{2w} + \sum_{i=0}^{w-1} (-1)^{w-1-i} \tau^{2i}$. For l even, $k_{max,l} = \sum_{i=0}^{w} (-1)^{w-i} \tau^{2i}$. And for any τNAF k of length up to $l + 3$, $\alpha(k, l + 2) \leq \vartheta(k_{max,l}, l)$. Moreover, for $l \geq 3$,*

$$\vartheta(k_{max,l}, l) = \vartheta(k_{max,l-1}, l - 1) + \vartheta(k_{max,l-2}, l - 2).$$

In order to prove the theorem, we will need the following lemmas.

Lemma 5.1 *If k is divisible by τ^e then $\vartheta(k,l) = \vartheta(\frac{k}{\tau^e}, l-e)$.*

Proof. Looking at Table 5.2 and Table 5.3, we find that random decisions are made at the states where $\kappa_i + c_{0_i} = \pm 1$. In this case, there are two possible transitions emerging from these states, that is there are two possible paths that can be followed, each yielding a family of representations where the sbit d_i is either 1 or $\overline{1}$.

When the least significant sbit(s) is (are) 0, the algorithm enters state s_{12} and does not exit this state until the first 1 or $\overline{1}$ is encountered. Until then, there are no new representations that are formed, and the least significant 0s are sent to the output as they are. Any other representation formed thereafter will have the same number of least significant 0s as k.

In other words, if k is divisible by τ^e, so are its representations. That is, they will all have e least significant 0s. Therefore, the possible representations for k when represented in l sbits will be the same representations for $\frac{k}{\tau^e}$ when represented in $l-e$ sbits with e 0s appended to each of the latters. □

Lemma 5.2 *If k is a τNAF of length l and $k \equiv (-1)^b \pmod{\tau}$ where $b \in \{0,1\}$, then the τNAF of $k + (-1)^b$ is of length at most $l+3$.*

Proof. To convert a number in a τ-adic form into a τNAF, we can use the transformations given by Gordon [Gor98] for the curve E_1. The following transformations (and their negatives) are the equivalent ones for the curve E_0.

$$\tau + 1 \rightarrow -\tau^2 - 1 \quad (11 \rightarrow \overline{1}0\overline{1}), \tag{5.10}$$

$$\tau - 1 \rightarrow -\tau^3 + 1 \quad (1\overline{1} \rightarrow \overline{1}001), \tag{5.11}$$

$$2 \rightarrow \tau^3 + \tau \qquad (2 \rightarrow 1010). \tag{5.12}$$

Now, consider the following cases for the least significant sbits of $k \equiv 1 \pmod{\tau}$ when 1 is added, where the transformation (5.12) is used after the addition. Other cases are recursions of the following ones. The subscript τ was removed since it applies to all of the following representations.

$(\ldots \bar{1}001) + 1 = (\ldots 0010)$,

$(\ldots 1001) + 1 = (\ldots 2010)$,

$(\ldots 0101) + 1 = (\ldots 1110) = (\ldots 00\bar{1}0)$, using (5.10)

$(\ldots \bar{1}0\bar{1}01) + 1 = (\ldots \bar{1}1\bar{1}10) = (\ldots 010\bar{1}0)$, using -(5.11) (*i.e.*, the negative of (5.11))

$(\ldots 10\bar{1}01) + 1 = (\ldots 11\bar{1}10) = (\ldots 210\bar{1}0)$, using -(5.11)

$(\ldots 100\bar{1}01) + 1 = (\ldots 101\bar{1}10) = (\ldots 1110\bar{1}0) = (\ldots 00\bar{1}0\bar{1}0)$, using -(5.11) and (5.10)

$(\ldots \bar{1}00\bar{1}01) + 1 = \ldots = (\bar{2}0\bar{1}0\bar{1}0)$, using -(5.11) and (5.10).

When any of the transformations (5.10) to (5.12) is used, the resulting carry will either cancel an existing sbit, be added to a 0 or result in a 2 or -2. We can see from the above cases that the absolute result of adding a carry to an sbit will not exceed 2. Thus, the resulting τNAF of $k+1$ is at most 3 sbits longer than k. The same argument applies to $k \equiv -1 \pmod{\tau}$. □

Lemma 5.3 *For any τNAF $k \equiv (-1)^b \pmod{\tau}$ of length l, where $b \in \{0,1\}$, we have*

$$\vartheta(k,l) = \vartheta(\frac{k-(-1)^b}{\tau^2}, l-2) + \alpha(\frac{k+(-1)^b}{\tau}, l+1).$$

Proof. We will consider here the case of $k \equiv 1 \pmod{\tau}$ but the same arguments apply to $k \equiv -1 \pmod{\tau}$. Recall that $\vartheta(k,l)$ is the number of representations of k that are of length at most $l+2$. Since $k \mod \tau \neq 0$, this is also true for the τ-adic representations of k. That is, their least significant sbit (LSSB) will be either 1 or $\bar{1}$. For those representations that have 1 as the LSSB, if this 1 is replaced with 0, they will become representations of $k-1$. Since k is a τNAF, then $k-1$ is a τNAF divisible by τ^2. From Lemma 5.1,

we know that the number of representations of $k-1$ is $\vartheta(k-1,l) = \vartheta(\frac{k-1}{\tau^2}, l-2)$ and that those representations will have their 2 LSSBs equal to 00. Therefore, they can all be used as representations of k by replacing the least significant 0 with 1.

On the other hand, for those representations that have $\overline{1}$ as their LSSB, if this $\overline{1}$ is replaced with 0, they will become representations of $k+1$. Since $2 = (\overline{1}10)_\tau$ for the curve E_1 and $2 = (\overline{11}0)_\tau$ for the curve E_0, we can see that $k+1 \equiv 0 \pmod{\tau}$, hence all the representations of $k+1$ have 0 as their LSSB. Those representations that are of length $l+2$, with their least significant 0 replaced with $\overline{1}$, are counted among the $\vartheta(k,l)$ representations of k and their number is $\alpha(k+1, l+2) = \alpha(\frac{k+1}{\tau}, l+1)$. □

The following lemmas are carried on E_0 but there exists corresponding lemmas on E_1.

Lemma 5.4 *For l odd and $w = \frac{l-1}{2}$, if $k = \tau^{2w} + \sum_{i=0}^{w-1}(-1)^{w-1-i}\tau^{2i}$, then $\sum_{i=0}^{w-1}(-1)^{w-i}\tau^{2i+1} + (-1)^w$ is among the representations of k. In other words, $\frac{k-(-1)^w}{\tau} = \frac{k+(-1)^{w-1}}{\tau} = \sum_{i=0}^{w-1}(-1)^{w-i}\tau^{2i}$.*

Proof. Without loss of generality, let w be odd, then $k = (1\ 0\ 1\ 0\ \overline{1}\ 0\ \ldots\ 1\ 0\ \overline{1}\ 0\ 1)_\tau$. When the least significant 1 is replaced by $\overline{1}$, $2 = (\overline{11}0)_\tau$ is added to k. Hence,

$$\begin{aligned}
k &= (1\ 0\ 1\ 0\ \overline{1}\ 0\ \ldots\ 1\ 0\ \overline{2}\ \overline{1}\ \overline{1})_\tau \\
&= (1\ 0\ 1\ 0\ \overline{1}\ 0\ \ldots\ 2\ 1\ 0\ \overline{1}\ \overline{1})_\tau \\
&= \ldots \\
&= (1\ 0\ 1\ 0\ \overline{2}\ \overline{1}\ \ldots\ 0\ 1\ 0\ \overline{1}\ \overline{1})_\tau \\
&= (1\ 0\ 2\ 1\ 0\ \overline{1}\ \ldots\ 0\ 1\ 0\ \overline{1}\ \overline{1})_\tau \\
&= (0\ \overline{1}\ 0\ 1\ 0\ \overline{1}\ \ldots\ 0\ 1\ 0\ \overline{1}\ \overline{1})_\tau.
\end{aligned}$$

□

Lemma 5.5 *For l even and $w = \lfloor \frac{l-1}{2} \rfloor = \frac{l}{2} - 1$, if $k = \sum_{i=0}^{w}(-1)^{w-i}\tau^{2i}$, then $\tau^{2w+3} + \tau^{2w+1} + \sum_{i=0}^{w-1}(-1)^{w-1-i}\tau^{2i+1} + (-1)^{w-1}$ is among the representations of k. In other*

words, $\frac{k-(-1)^{w-1}}{\tau} = \frac{k+(-1)^{w}}{\tau} = \tau^{2w+2} + \tau^{2w} + \sum_{i=0}^{w-1}(-1)^{w-1-i}\tau^{2i}$

Proof. Without loss of generality, let w be odd. Then k is of the form $(0\ 1\ 0\ \overline{1}\ 0\ \ldots\ 1\ 0\ \overline{1})_\tau$. As before, the least significant $\overline{1}$ can be replaced by 1 and $-2 = (110)_\tau$ added to k. Hence, we obtain the following

$$\begin{aligned} k = &\quad (0\ 1\ 0\ \overline{1}\ 0\ \ldots\ 2\ 1\ 1)_\tau \\ = &\quad \ldots \\ = &\quad (0\ 1\ 0\ \overline{2}\ \overline{1}\ \ldots\ 0\ 1\ 1)_\tau \\ = &\quad (0\ 2\ 1\ 0\ \overline{1}\ \ldots\ 0\ 1\ 1)_\tau \\ = &\quad (\overline{1}\ \overline{1}\ 0\ 1\ 0\ \overline{1}\ \ldots\ 0\ 1\ 1)_\tau \\ = &\ (1\ 0\ 1\ 0\ 1\ 0\ \overline{1}\ \ldots\ 0\ 1\ 1)_\tau. \end{aligned}$$

□

Lemma 5.6 *Let k be τNAF of length l with $\kappa_{l-1} = 1$ $(\overline{1})$. Then, the representations of k that are of length $l+2$ will have $d_{l+1} = \overline{1}$ (1), where d_i are the sbits output from the algorithm as in Table 5.3. Moreover, if $d_{l-1} = \overline{1}$ in any of the representations of k, then the length of this representation is $l+2$.*

Considering Table 5.3, when the most significant sbit $\kappa_{l-1} = 1$ is read, the algorithm will be in one of the states s_{17} to s_{23}. Representations that are of length $l+2$ are resulting from those states that have exit paths consisting of three transitions as single exit paths (s_{21} and s_{23}) or as alternate paths (s_{20} and s_{22}). It can be easily checked from the table that the last output sbit in all such paths is $\overline{1}$. It can be also checked that $d_{l-1} = \overline{1}$ occurs only on the alternate exit paths from s_{20} and s_{22}, hence the second part of the lemma is proved. The same arguments applies for $\kappa_{l-1} = \overline{1}$. □

Now we employ the previous lemmas to prove Theorem 5.1 by induction.

Proof. From the algorithm using Table 5.3, we can verify the following (cf. Tables D.1 to D.6 in Appendix D.2):

- $\vartheta((1)_\tau, 1) = 2$, those two representations are $(1)_\tau, (\overline{111})_\tau$. $k_{max,1} = 1$.
- $\vartheta((1)_\tau, 2) = 3$, those representations are $(1)_\tau, (\overline{111})_\tau, (101\overline{1})_\tau$. From Lemma 5.1, we have $\vartheta((10)_\tau, 2) = \vartheta((1)_\tau, 1) = 2$. So, $k_{max,2} = 1$.
- $\vartheta((101)_\tau, 3) = 5$. $k_{max,3} = 101$. The 5 representations are $(101)_\tau$, $(\overline{111}01)\tau$, $(\overline{11})_\tau$, $(111\overline{1})_\tau$, $(\overline{1}0\overline{1}1\overline{1})_\tau$. The first 2 representations are the same representations of $(100)_\tau$ for $l = 3$, with 1 as the least significant sbit instead of 0. From Lemma 5.1, we have $\vartheta((100)_\tau, 3) = \vartheta((1)_\tau, 1) = 2$. The remaining 3 representations are the same representations of $(\overline{1})_\tau$ for $l = 2$ shifted left by τ with $\overline{1}$ added. Note that the representations of $\overline{1}$ are the negative of the representations of 1. Hence, $\vartheta((101)_\tau, 3) = \vartheta((1)_\tau, 2) + \vartheta((1)_\tau, 1)$.
- For all τNAFs k of length up to $l+3 = 6$, $\alpha(k, 5) \leq \vartheta(k_{max,3}, 3)$. It is also true that $\alpha(k, 3) \leq \vartheta(k_{max,1}, 1)$ and $\alpha(k, 4) \leq \vartheta(k_{max,2}, 2)$, not only for τNAFs of lengths up to 4 and 5, respectively but also for those up to length 6.

We see that Theorem 5.1 is true for $l = 1, 2$ and 3. Now assume that it is true up to some length $l - 1$. We first prove that

$$\vartheta(k_{max,l}, l) = \vartheta(k_{max,l-1}, l-1) + \vartheta(k_{max,l-2}, l-2)$$

From Lemma 5.1, $k_{max,l} \equiv (-1)^b \pmod{\tau}$, for $b \in \{0, 1\}$. From Lemma 5.3, we know that

$$\vartheta(k_{max,l}, l) = \vartheta(\frac{k_{max,l} - (-1)^b}{\tau^2}, l-2) + \alpha(\frac{k_{max,l} + (-1)^b}{\tau}, l+1),$$

where at least one of the following conditions is true:

- $\frac{k_{max,l}-(-1)^b}{\tau^2} = k_{max,l-2}$.
- $\alpha(\frac{k_{max,l}+(-1)^b}{\tau}, l+1) = \vartheta(k_{max,l-1}, l-1)$, since, from Lemma 5.2, $\frac{k_{max,l}+(-1)^b}{\tau}$ will be of length at most $l+2$ and we assume that for any τNAF k of length up to $l+2$, $\alpha(k, l+1) \leq \vartheta(k_{max,l-1}, l-1)$ is true.

If there exists a τNAF k of length l for which both conditions are simultaneously true, then this k is $k_{max,l}$.

Let l be odd and k of length l be equal to $\tau^{2w}+\sum_{i=0}^{w-1}(-1)^{w-1-i}\tau^{2i}$ where $w = \frac{l-1}{2}$, that is $k \equiv (-1)^{w-1} \pmod{\tau}$. Then, we have $\frac{k-(-1)^{w-1}}{\tau^2} = \tau^{2(w-1)} + \sum_{i=0}^{w-2}(-1)^{w-2-i}\tau^{2i} = k_{max,l-2}$ (from our assumption that Theorem 5.1 is true up to τNAFs of length $l-1$). Also, from Lemma 5.4, we have $\frac{k+(-1)^{w-1}}{\tau} = \sum_{i=0}^{w-1}(-1)^{w-i}\tau^{2i} = -k_{max,l-1}$. Since $\alpha(-k_{max,l-1}, l+1) = \vartheta(-k_{max,l-1}, l-1) = \vartheta(k_{max,l-1}, l-1)$, then $k = k_{max,l}$.

Now, let l be even and k of length l be equal to $\sum_{i=0}^{w}(-1)^{w-i}\tau^{2i}$ where $w = \lfloor\frac{l-1}{2}\rfloor = \frac{l}{2} - 1$, that is $k \equiv (-1)^w \pmod{\tau}$. Then, we have $\frac{k-(-1)^w}{\tau^2} = \sum_{i=0}^{w-1}(-1)^{w-1-i}\tau^{2i} = k_{max,l-2}$. Also, from Lemma 5.5, $\frac{k+(-1)^w}{\tau} = \tau^{2w+2} + \tau^{2w} + \sum_{i=0}^{w-1}(-1)^{w-1-i}\tau^{2i} = \tau^{2w+2} + k_{max,l-1}$. According to Lemma 5.6, the representations of $k_{max,l-1}$ that are of length $l+1$ have their most significant term equal to $-\tau^{2w+2}$. Therefore, all the representations of $\tau^{2w+2} + k_{max,l-1}$ will be of length at most $l+1$ and can be used as representations for k by shifting them to the left by one sbit and adding to them $(-1)^{w-1}$. Hence, $\alpha(\tau^{2w+2} + k_{max,l-1}, l+1) = \vartheta(k_{max,l-1}, l-1)$, and $k = k_{max,l}$.

Now, it remains to prove that for all τNAFs k of length up to $l+3$, $\alpha(k, l+2) \leq \vartheta(k_{max,l}, l)$. We have already assumed that for any τNAF k of length up to $l+2$, $\alpha(k, l+1) \leq \vartheta(k_{max,l-1}, l-1) < \vartheta(k_{max,l}, l)$ is true. Now, let k be a τNAF of length $l+3$. If

$k \equiv 0 \pmod{\tau}$, from Lemma 5.1 we have,

$$\begin{aligned}\alpha(k, l+2) &= \alpha(\frac{k}{\tau}, l+1) \\ &\leq \vartheta(k_{max,l-1}, l-1), \text{by assumption} \\ &< \vartheta(k_{max,l}, l).\end{aligned}$$

Otherwise, if $k \equiv (-1)^b \pmod{\tau}$, then some of the representations of k will have 1 as their LSSB and the others will have $\overline{1}$. Without loss of generality, let $b = 0$. From Lemma 5.3, the representations that end with 1 and are of length $l+2$, are those of $\frac{k-1}{\tau^2}$ that are of length l with an appended 01. Hence, their number is $\alpha(\frac{k-1}{\tau^2}, l) \leq \vartheta(k_{max,l-2}, l-2)$. On the other hand, the representations of k that end with $\overline{1}$ and are of length $l+2$ are those of $\frac{k+1}{\tau}$ that are of length $l+1$ with an appended $\overline{1}$. Their number is $\alpha(\frac{k+1}{\tau}, l+1) \leq \vartheta(k_{max,l-1}, l-1)$. Note that $\frac{k-1}{\tau^2}$ and $\frac{k+1}{\tau}$ are τNAFs of length $l+1$ and $l+2$, respectively. Hence, we have

$$\begin{aligned}\alpha(k, l+2) &= \alpha(\frac{k-1}{\tau^2}, l) + \alpha(\frac{k+1}{\tau}, l+1) \\ &\leq \vartheta(k_{max,l-2}, l-2) + \vartheta(k_{max,l-1}, l-1) \\ &\leq \vartheta(k_{max,l}, l).\end{aligned}$$

□

It is important to notice that the recurrence relation of $\vartheta(k_{max,l}, l)$ in Theorem 5.1 is identical to the recurrence we obtained for the maximum number of binary signed digit (BSD) representations of an integer in Theorem 4.1. Since the values $\vartheta(k_{max,1}, 1) = 2$ and $\vartheta(k_{max,2}, 2) = 3$ agree with the values of $\delta(k_{max,n}, n)$ for $n = 1, 2$ in the BSD system, then the expressions we derived for $\delta(k_{max,n}, n)$, namely (4.27), (4.28) and (4.29), are directly applicable to $\vartheta(k_{max,l}, l)$ by replacing n with l. Hence we can conclude that $\vartheta(k_{max,l}, l)$ is $O(\varphi^l)$, where $\varphi \approx 1.618$ is the golden ratio [Kos01].

5.4 Average Hamming Density of the Representations

We assume that the τNAF k has been randomly chosen among all τNAFs of length $m + a$ as was suggested by Solinas [Sol00]. Since the decision bit r_i is also randomly chosen, the transition from a state S_i to the next state S_{i+1} does not depend on the previous states $S_{i-1}, S_{i-2}, \ldots$. Thus, this process is an irreducible ergodic Markov chain (cf. Appendix B). Therefore, the limiting probabilities of all states can be calculated using (B.1).

We can write the transition matrix for the states of Table 5.2 as follows

$$\mathbf{T} = \begin{pmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\frac{1}{4} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{4} & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & \frac{1}{4} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{4} & 0 & 0 & 0 & 0 \\
\frac{1}{8} & 0 & \frac{1}{8} & 0 & 0 & 0 & 0 & 0 & \frac{1}{4} & 0 & \frac{1}{4} & 0 & 0 & 0 & 0 & 0 & \frac{1}{8} & 0 & \frac{1}{8} & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & \frac{1}{8} & 0 & \frac{1}{8} & 0 & 0 & 0 & 0 & 0 & \frac{1}{4} & 0 & \frac{1}{4} & 0 & 0 & 0 & 0 & 0 & \frac{1}{8} & 0 & \frac{1}{8} & 0 \\
0 & 0 & 0 & \frac{1}{4} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{4} & 0 & 0 & 0 \\
0 & \frac{1}{8} & 0 & \frac{1}{8} & 0 & 0 & 0 & 0 & 0 & \frac{1}{4} & 0 & \frac{1}{4} & 0 & 0 & 0 & 0 & 0 & \frac{1}{8} & 0 & \frac{1}{8} & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & \frac{1}{8} & 0 & \frac{1}{8} & 0 & 0 & 0 & 0 & 0 & \frac{1}{4} & 0 & \frac{1}{4} & 0 & 0 & 0 & 0 & 0 & \frac{1}{8} & 0 & \frac{1}{8} \\
0 & 0 & 0 & 0 & \frac{1}{4} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{4} & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{4} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{4} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix}.$$

Let $\boldsymbol{\eta} = (\eta_0 \ \ldots \ \eta_{22})$ be the vector of limiting probabilities of the states of Table 5.2.

We can calculate the values in that vector by solving the following equations for Markov chains

$$\begin{aligned} \boldsymbol{\eta}\mathbf{T} &= \boldsymbol{\eta}, \\ \sum_{j=0}^{22} \eta_j &= 1. \end{aligned} \tag{5.13}$$

This yields the following

$$\begin{aligned} \boldsymbol{\eta} = (&\frac{13}{1152}, \frac{1}{144}, \frac{43}{2304}, \frac{107}{1152}, \frac{43}{2304}, \frac{1}{144}, \frac{13}{1152}, \frac{13}{2304}, \frac{9}{256}, \frac{91}{1152}, \frac{1}{18}, \\ &\frac{91}{288}, \frac{1}{18}, \frac{91}{1152}, \frac{9}{256}, \frac{13}{2304}, \frac{13}{1152}, \frac{1}{144}, \frac{43}{2304}, \frac{107}{1152}, \frac{43}{2304}, \frac{1}{144}, \frac{13}{1152}). \end{aligned}$$

The average Hamming density of the randomized representation can be obtained by summing the limiting probabilities of the states that have as output $d_i = 1$ or $\overline{1}$.

$$\begin{aligned} Pr(d_i = 1 \text{ or } d_i = \overline{1}) &= \eta_0 + \eta_3 + \eta_6 + \eta_9 + \eta_{10} + \eta_{12} + \eta_{13} + \eta_{16} + \eta_{19} + \eta_{22} \\ &= 0.5 \end{aligned}$$

Similarly, the transition matrix for the states of Table 5.3, which is for curve E_0, can be formed. By solving (5.13) for the matrix obtained, the vector of limiting probabilities is found to be

$$\begin{aligned} \boldsymbol{\eta} = (&\frac{1}{144}, \frac{13}{1152}, \frac{43}{2304}, \frac{107}{1152}, \frac{43}{2304}, \frac{13}{1152}, \frac{1}{144}, \frac{13}{2304}, \frac{91}{1152}, \\ &\frac{9}{256}, \frac{1}{18}, \frac{91}{288}, \frac{1}{18}, \frac{9}{256}, \frac{91}{1152}, \frac{13}{2304}, \frac{1}{144}, \frac{13}{1152}, \frac{43}{2304}, \frac{107}{1152}, \frac{43}{2304}, \frac{13}{1152}, \frac{1}{144} \end{aligned}$$

Hence, we have

$$\begin{aligned} Pr(d_i = 1 \text{ or } d_i = \overline{1}) &= \eta_1 + \eta_3 + \eta_5 + \eta_8 + \eta_{10} + \eta_{12} + \eta_{14} + \eta_{17} + \eta_{19} + \eta_{21} \\ &= 0.5 \end{aligned}$$

We can see that for both curves the average Hamming density for the randomized representation is 0.5.

5.5 Average and Exact Number of Representations

In this section, we first show how to obtain the average number of representations for a τNAF of length l by finding the total number of representations for all τNAFs of length l and dividing it by the number of those τNAFs. Then, we show how the exact number of representations for a τNAF can also be found.

5.5.1 Number of τNAFs of Length l

We first prove that the number of τNAFs of length l is the integer closest to $2^{l+2}/3$ as was stated by Solinas [Sol00]. That is, this number is

$$\frac{2^{l+2}-1}{3} = \sum_{i=0}^{\frac{l}{2}} 2^{2i}, \quad \text{for } l \text{ even}, \tag{5.14}$$

and

$$\frac{2^{l+2}+1}{3} = \sum_{i=0}^{\frac{l+1}{2}} 2^{2i+1} + 1, \quad \text{for } l \text{ odd}. \tag{5.15}$$

The number of non adjacent sequences of length l is the number of ways of placing i non-zero symbols in $l+1-i$ possible positions, such that no two non-zero symbols are adjacent, where $0 \leq i \leq \lceil \frac{l}{2} \rceil$. Each of the i nonzero symbols can be 1 or -1, yielding 2^i choices for their values. Hence, the number of sequences can be expressed as

$$\sum_{i=0}^{\lceil l/2 \rceil} \binom{l+1-i}{i} 2^i. \tag{5.16}$$

Now we will prove by induction that (5.16) is equivalent to (5.14) and (5.15). It can be easily verified that this is the case for $l = 0$ and 1. Now assume that it is true up to some $l = t-1$ where t is even. We will use the following identity [Kal85]

$$\binom{a+1}{e} = \binom{a}{e-1} + \binom{a}{e}, \tag{5.17}$$

for any real number a and integer e, where by definition

$$\binom{a}{e} = 0 \quad \text{for } e < 0. \tag{5.18}$$

If a is an integer,

$$\binom{a}{e} = 0 \quad \text{for } e > a. \tag{5.19}$$

We have

$$\sum_{i=0}^{t/2} \binom{t+1-i}{i} 2^i = \sum_{i=0}^{t/2} \binom{t-i}{i-1} 2^i + \sum_{i=0}^{t/2} \binom{t-i}{i} 2^i. \tag{5.20}$$

The second term of (5.20) evaluates to

$$\sum_{i=0}^{\lceil \frac{t-1}{2} \rceil} \binom{(t-1)+1-i}{i} 2^i = \frac{2^{t+1}+1}{3} \tag{5.21}$$

by using (5.15).

As for the first term of (5.20), let $j = i-1$. Note that the first term of the summation is 0 from (5.18). Hence, the summation becomes

$$\begin{aligned}
\sum_{j=0}^{t/2} \binom{t-j-1}{j} 2^{j+1} &= 2 \sum_{j=0}^{\frac{t-2}{2}+1} \binom{(t-2)+1-j}{j} 2^j \\
&= 2 \left[\sum_{j=0}^{\frac{t-2}{2}} \binom{(t-2)+1-j}{j} 2^j + \binom{\frac{t}{2}-1}{\frac{t}{2}} 2^{\frac{t}{2}} \right] \\
&= 2 \left[\frac{2^t - 1}{3} + 0 \right] \\
&= \frac{2^{t+1}-2}{3},
\end{aligned} \tag{5.22}$$

using (5.14) and (5.19).

The sum of (5.22) and (5.21) yields

$$\sum_{i=0}^{t/2} \binom{t+1-i}{i} 2^i = \frac{2^{t+2}-1}{3}. \tag{5.23}$$

The proof can be similarly carried for t odd. □

5.5.2 Number of Possible Representations for all τNAFs of Length l

In the following, we will consider the representations of τNAFs on the curve E_1, though the procedure we followed applies to those on the curve E_0. The states of the algorithm in Table 5.2, together with an initial state s_0 form a nondeterministic finite automaton (NFA) Γ with alphabet $\{\overline{1}, 0, 1\}$ as illustrated in Figure 5.1. Three directed edges labeled $\overline{1}$, 0 and 1 begin at s_0 and end at s_4, s_{12} and s_{20}, respectively. The final state of Γ is $s12$. Γ accepts the language described by the regular expression $(\varepsilon|1|\overline{1})(0|01|0\overline{1})^*(000)$. This regular expression represents non-adjacent forms when scanned from the least significant end. Three zeros are prepended in order to ensure that the final state s_{12} is reached for any input NAF string as was explained in Section 5.2.

Since an NFA is a directed graph (cf. Appendix E), it can be described by an *adjacency matrix* $M = (m_{ij})$ for $0 \leq i, j \leq 23$, such that $m_{ij} = 1$ if there is a directed edge from vertex i to vertex j in Γ and 0 otherwise. The number of directed paths of length l from vertex i to vertex j is the ij-th entry of the matrix M^l.

We can also define an adjacency matrix for each input symbol. For example, M_0 has a 1 in the ij-th entry if there is a directed edge labeled 0 from vertex i to vertex j. Note that since in the automaton considered, starting at some vertex i, there is only one edge labeled with just one of the input symbols that ends at state j, for $0 \leq i, j \leq 23$, and there are no edges labeled with the empty string ε, we have

$$M = M_{\overline{1}} + M_0 + M_1.$$

Therefore, in order to find all possible paths in Γ for input NAF strings of length l with three prepended 0s, we compute

$$M^l M_0^3 \tag{5.24}$$

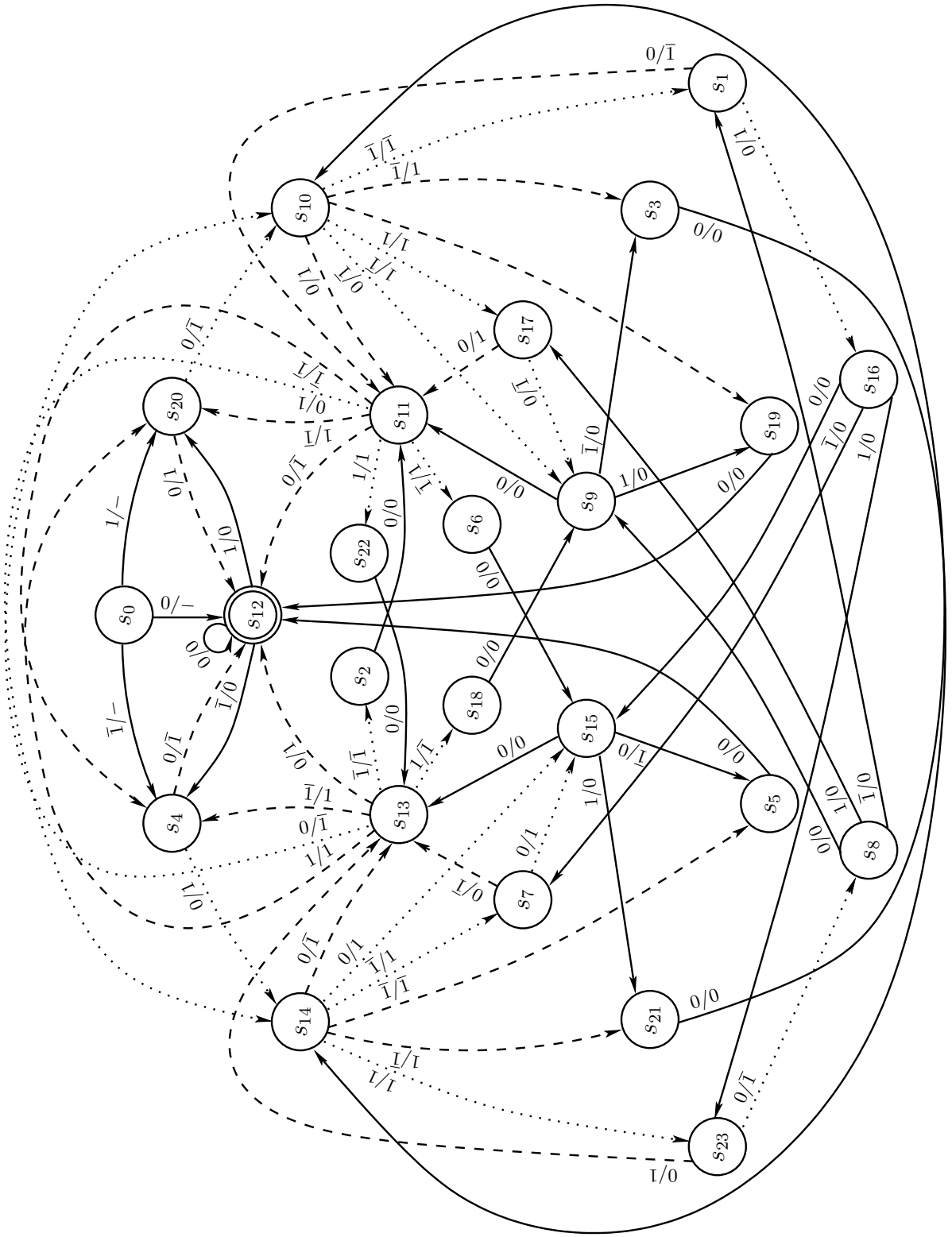

Figure 5.1: NFA corresponding to Table 5.2

and retrieve its (0,12)th entry. By computing this entry for the different values of l recommended by NIST [NIST] (163, 233, 283, 409, 571) using MAPLE, we have deduced that it is the integer closest to

$$1.304812 \cdot 3^l. \tag{5.25}$$

Hence, from (5.14), (5.15) and (5.25), the average number of representations of a τNAF of length l in the range [163, 571] is the integer closest to

$$0.9786 \left(\frac{3}{2}\right)^l. \tag{5.26}$$

The matrix multiplication in (5.24) can be performed by MAPLE in 0.41 seconds for $l = 163$ and in 0.83 seconds for $l = 571$ on a Pentium M processor.

5.5.3 Exact Number of Representations for a τNAF

The use of adjacency matrices can also be extended to find the number of paths corresponding to a specific input string. That is for a τNAF $k = (\kappa_{l-1}, \ldots, \kappa_1, \kappa_0)_\tau$, the number of possible representations is

$$M_{\kappa_0} M_{\kappa_1} \cdots M_{\kappa_{l-1}} M_0^3 \tag{5.27}$$

We have included the adjacency matrices for the automaton in Figure 5.1 in Appendix E.

5.6 Conclusion

In this chapter, we have introduced a new method of randomizing the τ-adic representation of a key in ECCs using Koblitz curves. The input to the randomization algorithm is a τNAF of length l. The output of the algorithm is a random τ-adic sequence of the

same value as the input. The sbits of the resulting sequence are output one at a time from the least significant to the most significant which allows the simultaneous execution of the scalar multiplication operations. The length of the random representation is at most $l + 2$. We have proved that the average Hamming density of all representations for all τNAFs of the same length is 0.5.

We have also presented the pattern of τNAFs with maximum number of representations and the recurrence governing that number which shows that it is a Fibonacci number and, hence, is $O(\varphi^l)$. By modeling our algorithm as a nondeterministic finite automaton and by using adjacency matrices, we have presented a deterministic method to determine the average and the exact number of representations of a τNAF, where the average number is very close to $\left(\frac{3}{2}\right)^l$ for $l \in [163, 571]$. It is interesting to note the similarity of the results obtained here to those obtained for the BSD representation of integers in Chapter 4 such as the average Hamming density and the average and maximum number of representations. To the best of our knowledge, this randomization approach to the τ-adic representations and the results we derived from applying it are the first of their kind [Has01; Joy03].

Also of interest is to investigate how this randomization method and the associated properties of the representation can be carried to any complex radix with norm 2 or any arbitrary norm. Note that this complex number should satisfy an equation such as (5.2), in order to be able to recursively replace digits with a larger absolute value than those in the digit set with the latter ones during the randomization procedure.

Chapter 6

On Key Splitting Methods

From the important conclusion we have drawn in Appendix A, we know that key randomization is essential in protecting a cryptographic algorithm from DPA attacks in their different forms. That is, the actual value of the key that is processed—either on a bit level or a digit level—to compute the elliptic curve scalar multiplication (ECSM) should be randomized; not just the key representation. The randomization is performed before the ECSM execution, for which an efficient SPA-resistant algorithm is used. A candidate key value randomization method is the key splitting (cf. Section 2.2.2).

In this chapter, we study different forms of key splitting along with their strengths and weaknesses. We also discuss the candidate SPA-resistant algorithms and compare the resulting performance when combined with each form of key splitting. At the end of the chapter, we present countermeasures to DPA attacks on the ECDSA and the ECMQV algorithms. Our contributions are summarized in Section 7.

6.1 Additive Splitting Using Subtraction (scheme I)

This approach was suggested by Clavier and Joye in [CJ01] and revisited by Ciet [Cie03] as follows. In order to compute the point kP, the n-bit key k is written as $k = k_1 + k_2$, such that $k_1 = k - r$ and $k_2 = r$, where r is a random integer of length n bits. Then kP is computed as

$$kP = k_1P + k_2P. \tag{6.1}$$

It is important to note that each of the terms of (6.1) should be evaluated separately and their results combined at the end using point addition. That is the multiple-point multiplication methods that use a common accumulator to save doubling operations such as Algorithms 2.3 and 2.4—whether at the bit level ($w = 1$) or window level ($w > 1$)—should not be used, even when a countermeasure against SPA is employed (cf. Section A.2). This observation is based on the following lemma. Let $k_{b\to a}$ denote $\lfloor (k \bmod 2^{b+1})/2^a \rfloor$ or, simply, the bits of k from bit position b down to bit position a, with $b \geq a$.

Lemma 6.1 *Let splitting scheme I in* (6.1) *be evaluated using Algorithm 2.3 with* $w = 1$ *(*$d = n$*). Then, at the end of some iteration* j*,* $0 < j \leq n-1$*, there are only two possible values for* Q*, those are* $[k_{n-1\to j}]\ P$ *or* $[k_{n-1\to j} - 1]\ P$*.*

Proof. Algorithm 2.3—and similarly Algorithm 2.4—computes the required point by scanning $k_1 = (k_{1_{n-1}}, \ldots, k_{1_0})_2$ and $k_2 = (k_{2_{n-1}}, \ldots, k_{2_0})_2$ from the most significant end down to the least significant end. Hence, at the end of iteration j, the accumulator Q contains the value

$$\begin{aligned} Q &= k_{1_{n-1\to j}}P + k_{2_{n-1\to j}}P, \\ &= [k_{1_{n-1\to j}} + k_{2_{n-1\to j}}]\ P. \end{aligned} \tag{6.2}$$

We can write k, k_1 and k_2 as

$$k = k_{n-1\to j}2^j + k_{j-1\to 0}, \tag{6.3}$$

$$k_i = k_{i_{n-1\to j}}2^j + k_{i_{j-1\to 0}}, \quad i \in \{1, 2\}. \tag{6.4}$$

Since $k = k_1 + k_2$, we have

$$k_{1_{j-1\to 0}} + k_{2_{j-1\to 0}} = k_{j-1\to 0} + b\,2^j, \quad \text{where } b \in \{0, 1\} \tag{6.5}$$

and

$$k_{1_{n-1\to j}} + k_{2_{n-1\to j}} = k_{n-1\to j} - b. \qquad \square$$

Therefore, this evaluation method would still be vulnerable to DPA attacks as described in Section A.3. The attack would proceed in the same way, whether the algorithm processes a single bit or a digit per iteration, though it would be more involved in the latter case depending on the digit size. The attacker can double the number of traces gathered and compute the necessary intermediate points as if there was no countermeasure in place.

Hence each term of (6.1) should be computed separately using a SPA-resistant algorithm such as the Montgomery ladder algorithm or fixed-sequence window method as on pages 161 and 164, respectively, in Section A.2.1. A point addition operation is performed at the end. Note that if a window method is used, the precomputed multiples of the input point are calculated once and used with both scalar multiplications k_1P and k_2P.

As we mentioned before, the splitting of the key k into k_1 and k_2 is performed before every ECSM execution. If the key splitting process, *e.g.*, the subtraction in this case, is processing the key every time, then an attacker can obtain less noisy information about the key words such as their Hamming weights by averaging the side-channel trace obtained from the key splitting process [MS00]. Moreover, if it is difficult for the attacker to locate

the instances where the key is manipulated, then by correlating different traces, he can detect where the same data is processed. Therefore, it is desirable to use a previously split version of the key to generate the new one. Hence in (6.1), k_1 and k_2 can be refreshed as

$$k_1 \pm r_t, \qquad k_2 \mp r_t,$$

before the t-th execution of the ECSM, where the addition/subtraction is modulo the group order of the points on the elliptic curve and r_t is an n-bit random integer.

6.2 Additive Splitting Using Division (scheme II)

As an alternative to the previous splitting, Ciet and Joye [CJ03] suggest that a random divisor r be chosen and the key k written as $k = g * r + h$, where $g = \lfloor k/r \rfloor$ and $h = k \bmod r$. Let $S = rP$, then kP can be computed as

$$kP = gS + hP. \tag{6.6}$$

We choose the bit length of r to be $l = \lceil n/2 \rceil$. That is, r is chosen uniformly at random from the range $[2^{l-1}, 2^l - 1]$. Hence, the bit length of g is at most $\lfloor n/2 \rfloor + 1 \leq l + 1$ and at least l and that of h is at most l [JV02].

An ECSM is first performed to compute the point S, where the scalar is of size half that of k. Then, unlike splitting scheme I, a multiple point multiplication method can be safely used. In the following, we will justify this assertion.

Let the representations of k, g and h to the base 2^w, for some $w \geq 1$, be $(K_{2z-1}, \ldots, K_1, K_0)_{2^w}$, $(G_z, \ldots, G_1, G_0)_{2^w}$ and $(H_{z-1}, \ldots, H_1, H_0)$, respectively, where $z = \lceil l/w \rceil$, that is $l \leq zw \leq l + w - 1$. If $l < zw$, then $G_z = 0$, otherwise, if $l = zw$, then $G_z \leq 1$. As before, let $K_{b \to a}$ denote $\lfloor (k \mod 2^{bw+1})/2^{aw} \rfloor$ or, simply, the w-bit digits of k from digit position b down to digit position a, with $b \geq a$. Let (6.6) be evaluated using Algorithm 2.3

or Algorithm 2.4, replacing d by $z+1$ in these algorithms and setting $H_z = 0$. Then at the end of some iteration j, $1 < j \leq z$, the accumulator Q contains the value

$$\begin{aligned} Q &= G_{z\to j}\ S + H_{z\to j}\ P, \\ &= (G_{z\to j} * r + H_{z\to j})P. \end{aligned} \tag{6.7}$$

Let $k^j = G_{z\to j} * r + H_{z\to j}$, which is of length $2z - j$ w-bit digits[1]. In general—exceptions follow—, $k^j \neq K_{2z-1\to j}$. This is true since $K_{2z-1\to j} = G_{z\to j} * r + h_j$, where h_j is the l-bit remainder of the division of $K_{2z-1\to j}$ by r. Since $K_{2z-1\to j} \neq k$, then from the division theorem, the pair $(G_{z\to j}, h_j)$ is not equal to (g, h), hence, in general, $h_j \neq H_{z\to j}$.

Major Collisions

A *major collision* is defined as the occurrence of $k^j = K_{2z-1\to j}$ at some iteration $j \in [1, z-1]$. The intermediate point computed at this value of k^j is the same value that would be computed when no countermeasure is in place. The condition of this collision is provided by the following lemma.

Lemma 6.2 *For some $j \in [1, z-1]$, $k^j = K_{2z-1\to j}$ iff $G_{j-1\to 0} = 0$.*

Proof. We have

$$\begin{aligned} k &= g * r + h, \\ &= (G_{z\to j} * 2^{jw} + G_{j-1\to 0}) * r + (H_{z\to j} * 2^{jw} + H_{j-1\to 0}), \\ &= k^j * 2^{jw} + G_{j-1\to 0} * r + H_{j-1\to 0}. \end{aligned} \tag{6.8}$$

[1]In the case where $l = zw$ and $G_z = 1$, the bit length of $G_{z\to j}$ is $(z-j)w + 1$. Though the bit length of r is l, for all $j \in [0, z]$, the bit length of $G_{z\to j} * r$ is at most $(2z-j)w$ and not $(2z-j)w+1$. Otherwise, the bit length of $G_{z\to 0} * r = g * r$ would be $2zw + 1$, which is one bit longer than that of k.

But $k = K_{2z-1\to j} * 2^{jw} + K_{j-1\to 0}$. Hence, if $G_{j-1\to 0} = 0$, we have $k^j = K_{2z-1\to j}$ and $K_{j-1\to 0} = H_{j-1\to 0}$. On the other hand, if $k^j = K_{2z-1\to j}$, then $\lfloor (G_{j-1\to 0} * r)/2^{jw} \rfloor = 0$. However, $r \geq 2^{l-1}$, that is, $r \geq 2^{(z-1)w}$. Hence, $G_{j-1\to 0} = 0$ [2]. □

The probability of the occurrence of this collision is around 2^{-jw}. That is, it increases with the iterations of a multiple-point multiplication ECSM algorithm. It is negligible in the first iterations that are critical for the attacker in a DPA attack as explained in Section A.3.

Moreover, these collisions can be avoided when evaluating (6.6) for all j as follows. After performing the division of k by r, the quotient g is inspected. If the least significant w bits are found to be 0, another r is chosen. Note that this incurs a negligible reduction in the choice space of r from 2^{l-1} to approximately $2^{l-1} - 2^{l-w-1}$.

Another way to avoid these collisions is to make the quotient g always odd. That is if g is even, it is decremented by one and h is updated by adding r to it. This may increase the bit length of h to $l + 1$.

Minor Collisions

A *minor collision* occurs when at some iteration $j \in [1, z]$, for two values of r: r_1 and r_2, such that $r_1 \neq r_2$, we have $k_1^j = k_2^j \neq K_{2z-1\to j}$.

The conditions favoring these collisions are not straightforward to analyze. Some of them occur when $h_1 = h_2$, but also many of them occur with $g_1 \neq g_2$ and $h_1 \neq h_2$. Also in some cases, collisions occur when $\gcd(r_1, r_2) \neq 1$, where gcd is the greatest common divisor.

In the following we refer to ϕ as a *t-time collision* value, if at iteration j, $k^j = \phi$ for

[2] If $l = zw$ and $G_z = 1$, then for $j = z$, a major collision occurs if $G_{z-1\to 0} = 0$ or 1, since then $r \geq 2^{l-1} = 2^{zw-1}$.

t different values of r. We have conducted some experiments to study the probability of happening of these collisions for $n = 40$ and 50 when divided by all divisors of length 20 and 25 bits, respectively, with window width $w = 4$. We found that for different values of j, after excluding the values of g with w least significant bits, about 63% of the values of k^j on average were collision-free. About 25.6% were two-times collision values. The maximum number of collisions t for some value varied with the iteration; it was higher towards the middle iterations than the first and last iterations. For example, in the middle iterations, some 40-bit integers exhibited k^j values with up to 132-times collision and up to 1735-times for 50-bit integers. The density of values that have the higher number of collisions is usually 1 or 2. On the other hand, after the first iteration, the maximum number of collisions we obtained was 12 for 40-bit integers and 23 for 50-bit integers.

Our goal was to prove experimentally the insignificance of these collisions, especially in the first iterations. Moreover, any doubt about these collisions can be eliminated by randomizing once the projective coordinates of the accumulator Q after the initialization step (*e.g.*, step 3 of Algorithm 2.4 on page 18) as explained in Section A.3.2.

6.2.1 Computing New Quotients and Remainders From Old Ones

As mentioned at the end of Section 6.1, it is desirable to perform the splitting without involving the key every time. That is, given g_1, r_1, h_1 and r_2, we would like to compute g_2 and h_2 without internally computing k or any part thereof. We provide the underlying idea in the following. The implementation detailed steps are depicted in Algorithm 6.2.

$$\begin{aligned} g_2 &= \left\lfloor \frac{g_1 r_1 + h_1}{r_2} \right\rfloor, \\ &= \left\lfloor \frac{g_1 r_1}{r_2} \right\rfloor + \left\lfloor \frac{h_1}{r_2} \right\rfloor + \left\lfloor \frac{(g_1 r_1) \bmod r_2 + h_1 \bmod r_2}{r_2} \right\rfloor. \end{aligned} \tag{6.9}$$

$$\left\lfloor \frac{g_1 r_1}{r_2} \right\rfloor = \left\lfloor \frac{(\lfloor g_1/r_2 \rfloor \; r_2 + g_1 \bmod r_2)(\lfloor r_1/r_2 \rfloor \; r_2 + r_1 \bmod r_2)}{r_2} \right\rfloor,$$

$$= \left\lfloor \frac{g_1}{r_2} \right\rfloor \; r_1 + \left\lfloor \frac{r_1}{r_2} \right\rfloor \; (g_1 \bmod r_2) + \left\lfloor \frac{(g_1 \bmod r_2) \; (r_1 \bmod r_2)}{r_2} \right\rfloor. \quad (6.10)$$

$$(g_1 r_1) \bmod r_2 = [(g_1 \bmod r_2)(r_1 \bmod r_2)] \bmod r_2. \quad (6.11)$$

$$h_2 = (g_1 r_1 + h_1) \bmod r_2,$$

$$= [(g_1 r_1) \bmod r_2 + h_1 \bmod r_2] \bmod r_2. \quad (6.12)$$

Since $r_i \in [2^{l-1}, 2^l - 1]$, then g_i is in the range $[2^{l-1}, 2^{l+1} - 1]$ and h_i is in the same range as r_i. Hence, we have

$$\left\lfloor \frac{g_1}{r_2} \right\rfloor \leq 3, \quad \left\lfloor \frac{h_1}{r_2} \right\rfloor \leq 1 \quad \text{and} \quad \left\lfloor \frac{(g_1 r_1) \bmod r_2 + h_1 \bmod r_2}{r_2} \right\rfloor \leq 1.$$

It is important to note that if $g_1 < r_2$ and $r_1 < r_2$, the multiplication $(g_1 \bmod r_2)(r_1 \bmod r_2)$ will result in computing a value very close to the upper half of the bits of k. Therefore, in this case, we add the following modification

$$\left\lfloor \frac{(g_1 \bmod r_2) \; (r_1 \bmod r_2)}{r_2} \right\rfloor = \left\lfloor \frac{g_1 r_1}{r_2} \right\rfloor = \left\lfloor \frac{(g_1 + r_2) r_1}{r_2} \right\rfloor - r_1, \quad (6.13)$$

that is, we add r_2 to g_1 before performing the multiplication by r_1 and the division by r_2 and we subtract r_1 from g_2. Note that this modification does not affect the computation of h_2 since

$$(g_1 + r_2) \; r_1 \equiv g_1 r_1 \pmod{r_2}.$$

6.3 Multiplicative Splitting (scheme III)

This splitting was proposed by Trichina and Bellezza [CQS03] where r is a random integer invertible modulo u, where u is the order of P. The scalar multiplication kP is then

evaluated as

$$kP = [kr^{-1} \pmod u](rP). \tag{6.14}$$

This splitting involves two scalar multiplications, first $R = rP$ is computed, then $kr^{-1}R$ is computed. In this case, the length of r needs not be n bits as k but can be, say 80 bits, so that the first scalar multiplication is not as expensive as the second one.

However, if it is desirable that the key k be not involved in the computation of $kr^{-1} \pmod u$ before every scalar multiplication execution, then the chain of multiplied r_i is stored from the first up to the t-th scalar multiplication; $\boldsymbol{r}_t = r_t \cdot r_{t-1} \cdots r_1 \pmod u$ for $1 \leq i \leq t$, and the key is stored as $\boldsymbol{k}_t = k \cdot r_1^{-1} \cdots r_{t-1}^{-1} \cdot r_t^{-1} \pmod u$. Before the following scalar multiplication kP_{t+1}, r_{t+1} is chosen, then $\boldsymbol{r}_{t+1} = r_{t+1} \cdot \boldsymbol{r}_t \pmod u$, $R_{t+1} = \boldsymbol{r}_{t+1}P_{t+1}$, $\boldsymbol{k}_{t+1} = \boldsymbol{k}_t \cdot r_{t+1}^{-1} \pmod u$ and, finally, $\boldsymbol{k}_{t+1}R_{t+1}$ are computed. In this case, the length of $\boldsymbol{r}_i$ will reach n bits after a few executions.

In the following section, we present modular inversion and multiplication algorithms that are SPA-resistant.

6.4 Implementation Details

In this section, we provide details on our implementation of the different splitting methods. The implementation was targeting a handheld device with a *not very constrained memory*. Therefore, our priority was reducing first the computational cost then the storage cost.

6.4.1 Fixed-Sequence Window Method

This is the method we have employed as an SPA countermeasure. The recoding of the key to the base 2^w was performed with a method similar to the one suggested by Thériault

in [Thé06] and by Lim in [Lim04]. The digit set used is $\{\pm 1, \pm 3, \ldots, \pm(2^w - 1)\}$ as in [Thé06; Lim04; OT03]. That is, only the odd multiples of the input point are computed. Moreover, we have made use of the available storage to also store the negatives of those multiples for the following reason. Though in the literature point addition and subtraction are used interchangeably without incurring a side-channel leakage, there is some minor cost involved with performing the point subtraction (a modular subtraction for prime fields or an XOR operation for binary fields, cf. Section 2.1). In order not to reveal the sign of the bit or digit involved, the negation of the required point in an iteration can be always performed and either the point or its negative chosen for addition. Another solution could be to modify the point addition operation to take as input the sign of the point to be added and perform the required operation if the point is negative or a dummy one if it is positive.

We have found that storing the negatives of the precomputed points was a simpler solution since we have preferred not to modify the existing efficient doubling and addition operations using Jacobian coordinates and Jacobian-affine coordinates, respectively (cf. Section 2.1).

We have adopted the idea presented by Thériault [Thé06, Section 5.2] for the fixed left-to-right recoding of the scalar into signed odd digits. We refer the reader to the author's paper for details on the recoding and its correctness. We show explicitly how the precomputed points are stored at contiguous locations and how the recoding serves as a mapping from the scalar current window to the required multiple of the input point.

In [Thé06], an n-bit integer $e = (e_{n-1}, \ldots, e_0)_2$, which is to be multiplied by an input point P is recoded to the base 2^w as follows. First, it is assumed that $e_0 = 1$. If $e_0 = 0$, the order of P, which is a prime for curves used in cryptographic applications, can be added to e. Let $d = \lceil n/w \rceil$. Padding e with $dw - n$ 0s, the recoding of e is $(\varepsilon_{d-1}, \ldots, \varepsilon_0)_{2^w}$

such that

$$\varepsilon_{d-1} = 1 + \sum_{i=1}^{w-1} e_{(d-1)w+i}\ 2^i = 1 + 2 * e_{dw-1\ \rightarrow\ (d-1)w+1}, \tag{6.15}$$

$$\varepsilon_j = 1 - 2^w + \sum_{i=1}^{w} e_{jw+i}\ 2^i = 1 - 2^w + 2 * e_{(j+1)w\ \rightarrow\ jw+1}, \quad 0 \leq j \leq d-2. \tag{6.16}$$

Note that the bits needed to form a digit ε_j are not those of the corresponding digit $E_j = e_{(j+1)w-1\ \rightarrow\ jw}$ but they include the least significant bit of the digit E_{j+1} and exclude the least significant bit of the digit E_j. Therefore, in our algorithm we first shift right (SHR) e by 1 bit, where the least significant bit is known to be always 1. Note that this also has the benefit that if $dw = n$, the addition of the group order—which is an n-bit prime—will not result in an extra digit E_d. Let $e' = \mathtt{SHR}(e)$ and $E'_j = e'_{(j+1)w-1\ \rightarrow\ jw}$, the mapping from a digit E'_j to ε_j is

$$\begin{aligned} \varepsilon_{d-1} &= 2 * E'_{d-1} + 1, \\ &= 2 * (E'_{d-1} + 2^{w-1}) - (2^w - 1), \end{aligned} \tag{6.17}$$

$$\varepsilon_j = 2 * E'_j - (2^w - 1), \quad 0 \leq j \leq d-2. \tag{6.18}$$

Hence, we define a mapping

$$\mathcal{R} : [0, 2^w - 1] \rightarrow \{-(2^w - 1), -(2^w - 3), \ldots, (2^w - 3), (2^w - 1)\}$$

such that

$$\mathcal{R}(x) = 2x - (2^w - 1).$$

This mapping is used to build the table, T, of precomputed multiples of P, such that $T[E'_j] = \mathcal{R}(E'_j)P = \varepsilon_j P$ for $0 \leq j \leq d-2$ and $T[E'_{d-1} + 2^{w-1}] = \mathcal{R}(E'_{d-1} + 2^{w-1})P = \varepsilon_{d-1}P$. For example, for $w = 4$, $T[0] = -15P$, $T[1] = -13P$, $\ldots$, $T[7] = -P$, $T[8] = P$, $\ldots$, $T[14] = 13P$, $T[15] = 15P$. The following is the ECSM algorithm using this fixed left-to-right recoding.

Algorithm 6.1. Fixed-sequence window method

INPUT: Window width w, $d = \lceil n/w \rceil$, an n-bit odd integer e and $P \in E(\mathbb{F}_p)$.

OUTPUT: eP

1. *Precomputation.*
   - 1.1 $T[2^{w-1}] \leftarrow P$.
   - 1.2 $T[2^w - 1] \leftarrow 2P$.
   - 1.3 for i from 2^{w-1} to $2^w - 2$ do
     $T[i+1] \leftarrow T[i] + T[2^w - 1]$.
   - 1.4 for i from $2^{w-1} - 1$ down to 0 do
     $T[i] \leftarrow -T[2^w - 1 - i]$.
2. $e' = \mathtt{SHR}(e) = (E'_{d-1} \dots E'_0)_{2^w}$.
3. $Q \leftarrow T[E'_{d-1} + 2^{w-1}]$.
4. for i from $d-2$ down to 0 do
   - 4.1 $Q \leftarrow 2^w Q$.
   - 4.2 $Q \leftarrow Q + T[E'_i]$.
5. Return(Q).

Storage: 2^w points.

Precomputation: $1\ D + (2^{w-1} - 1)\ A$.

Running time: $(d-1)w\ D + (d-1)\ A$.

Coordinate selection: As we mentioned in the window method in Section 2.1.3, page 15, for prime fields, it is more efficient to represent the precomputed points using affine coordinates and the accumulator Q using Jacobian coordinates. Hence, the doubling operation is performed using Jacobian coordinates and the addition operation using Jacobian-affine coordinates [HMV04]. In the precomputation step, the doubling can be performed

on affine coordinates to obtain $2P$, which is then used in all the subsequent additions. Therefore, these additions can be performed using Jacobian-affine coordinates and then all the points can be converted to affine coordinates, the cost of each conversion being $1I+3M+1S$ (cf. Section 2.1). Using simultaneous inversion technique [Mon87; HMV04], we can save $2^{w-1}-2$ inversions by replacing them by $3(2^{w-1}-2)$ multiplications. This is particularly useful for prime fields where $1I \approx 80M$ [HMV04]. The cost of this conversion is $1I + 3(2^{w-1} - 2)M + (2^{w-1} - 1)(3M + 1S) = 1I + 3(2^w - 3)M + (2^{w-1} - 1)S$. This technique is also useful for binary fields if the computational cost of an inversion exceeds that of three multiplications. Note that for binary fields, the Lopez-Dahab coordinates are more efficient than the Jacobian coordinates [HMV04].

Combining the fixed-sequence window method with splitting scheme II

Algorithm 6.1 can be used to compute rP that is used in both (6.6) and (6.14). In this case, r can be chosen to be always odd. It can also be combined with Algorithm 2.4—or Algorithm 2.3— to compute (6.6). In this case, if g and/or h is even, which is each of bit length half that of k, we should not resolve to adding the order of P which would result in loosing the efficiency provided by this splitting method. Hence, we suggest the following.

We have $k_0 = g_0 r_0 + h_0$. If $k_0 = 0$, we can chose r such that $r_0 = 1$. In this case, if $g_0 = 1$, then $h_0 = 1$ and if $g_0 = 0$, $h_0 = 0$. In the latter case, g can be decremented by one, as suggested previously to avoid major collisions, and r added to h; as a result, we obtain $g_0 = h_0 = 1$. Similarly, if $k_1 = 1$, we can force $r_0 = 0$, in which case $h_0 = 1$ irrespective of g_0 and even if g is decremented and r is added to h when $g_0 = 0$.

Another generic solution when e in Algorithm 6.1 is not necessarily odd is to keep the value of the least significant bit, e_0, that is shifted out by step 2. After the main loop in

step 4, if $e_0 = 0$, we subtract P from Q, that is we perform $Q + T[2^{w-1} - 1]$, otherwise, we can perform a dummy point addition. Hence we can write

$$e = (\varepsilon_{d-1}, \ldots, \varepsilon_0)_{2^w} - (1 - e_0).$$

For example, we would need to use this method when computing rP if r is forced to be even. If we use this method when computing (6.6), we do not need to force g and h to be odd. However, major collisions will reappear associated with slightly different patterns of g as explained in the following lemma.

We use the following notation for the recoding of g and h.

$$\begin{aligned} g' &= \mathtt{SHR}(g), & h' &= \mathtt{SHR}(h), \\ &= (G'_{z-1}, \ldots, G'_0)_{2^w}, & &= (H'_{z-1}, \ldots, H'_0)_{2^w}, \\ \gamma_j &= \mathcal{R}(G'_j), & \eta_j &= \mathcal{R}(H'_j), \end{aligned}$$

where $z = \lceil l/w \rceil$, $l = \lceil n/2 \rceil$ and $0 < j < z - 1$. When (6.6) is evaluated using Algorithm 2.4 combined with the recoding in Algorithm 6.1, then at some iteration $j \in [0, z-1]$, the accumulator point $Q = \kappa^j P$, where $\kappa^j = \gamma_{z-1 \to j} * r + \eta_{z-1 \to j}$.

Lemma 6.3 *For some $j \in [1, z-1]$, $\kappa^j = K_{2z-1 \to j} + b$, where $b \in \{0, 1\}$, iff $g_{jw \to 0} = 2^{jw}$. Moreover, $b = 1 - h_{jw}$.*

Proof. We have

$$k = \kappa^j * 2^{jw} + (\gamma_{j-1 \to 0} - (1 - g_0)) * r + (\eta_{j-1 \to 0} - (1 - h_0)). \tag{6.19}$$

Also

$$k = K_{2z-1 \to j} * 2^{jw} + K_{j-1 \to 0}. \tag{6.20}$$

From [Thé06], we know that

$$\eta_{j-1} = H_{j-1} - (1 - h_{jw})2^w + (1 - h_{(j-1)w}). \tag{6.21}$$

Hence, (6.19) becomes

$$\begin{aligned} k = {} & (\kappa^j - (1 - h_{jw})) * 2^{jw} + (\gamma_{j-1\to 0} - (1 - g_0)) * r \\ & + (H_{j-1} + (1 - h_{(j-1)w})) * 2^{(j-1)w} + (\eta_{j-2\to 0} - (1 - h_0)). \end{aligned} \tag{6.22}$$

From Lemma 6.2, $\kappa^j - (1 - h_{jw}) = K_{2z-1\to j}$ iff

$$(\gamma_{j-1\to 0} - (1 - g_0)) = 0 \tag{6.23}$$

If $g_0 = 1$, there is no collision since, due to the recoding properties, $\gamma_{j-1\to 0} \neq 0$. Hence, when $g_0 = 0$, the condition becomes

$$\gamma_{j-1\to 0} = 1 \tag{6.24}$$

It can be easily verified that this condition holds when

$$\gamma_i = \begin{cases} 1 & i = j - 1, \\ -15 & 0 \leq i < j - 1. \end{cases} \tag{6.25}$$

This is particularly true since γ_i cannot be 0. By doing the inverse mapping $\mathcal{R}^{-1}(\gamma_i)$, we obtain

$$G'_i = \begin{cases} 2^{w-1} & i = j - 1, \\ 0 & 0 \leq i < j - 1. \end{cases} \tag{6.26}$$

from which we deduce $g_{jw\to 0} = (10 \ldots 0)_2 = 2^{jw}$. □

Since the condition of major collision depends on the value of the least significant $jw + 1$ bits of g the probability of the occurrence of this collision is around 2^{-jw+1} for both values of h_{jw} which are expected to be equally likely.

As suggested in Section 6.2, these collisions can be avoided for all j by excluding the g values having w least significant 0s. Though this method will also exclude values that cause no collisions, it is easier than checking for every $j \in [1, z-2]$ whether (6.26) is satisfied.

This recoding had also positive effects on minor collision. We observed the following

- The average number of collision-free intermediate values, after excluding the g values as mentioned, had increased to 80%. About 16% were two-times collision values. The maximum number of collisions for a value had also decreased.
- We have noticed that collisions that occurred when using even divisors were uncorrelated to those that occurred when using odd divisors. That is, when all the divisors space is used with some integer k, the number of values for each type of collision (collision free, two-times, three-times, ..., maximum number of collision) was the sum of the corresponding numbers gathered for the odd divisors and the even divisors separately for the same k.
- If major collisions are avoided by making g always odd, then when the recoding is used, collisions tend to be less occurring when larger integers k are divided by smaller divisors r.

6.4.2 Successive Division

We have described in Section 6.2.1 the underlying idea of computing a new quotient and remainder from old ones knowing the old divisor and the new one. In this section, we provide the corresponding algorithm.

Algorithm 6.2. Successive Division

INPUT: r_1, $g_1 = \lfloor k/r_1 \rfloor$, $h_1 = k \bmod r_1$ and r_2.

OUTPUT: $g_2 = \lfloor k/r_2 \rfloor$, $h_2 = k \bmod r_2$.

1. $\mathfrak{g} \leftarrow g_1$; $\mathfrak{r} \leftarrow r_1$. // $(g_1 \bmod r_2)$ and $(r_1 \bmod r_2)$ will be computed in $\mathfrak{g}$ and $\mathfrak{r}$, resp.
2. $g_2 \leftarrow 0$; $h_2 \leftarrow h_1$.
3. if $(\mathfrak{g} \leq r_2)$ AND $(\mathfrak{r} < r_2)$ then
   $\mathfrak{g} \leftarrow \mathfrak{g} + r_2$; $g_2 \leftarrow g_2 - r_1$.
4. else
   4.1 while $(\mathfrak{g} > r_2)$ do // This loop is executed at most three times.
   $\mathfrak{g} \leftarrow \mathfrak{g} - r_2$; $g_2 \leftarrow g_2 + r_1$.
   4.2 if $(\mathfrak{r} > r_2)$ then
   $\mathfrak{r} \leftarrow \mathfrak{r} - r_2$; $g_2 \leftarrow g_2 + \mathfrak{g}$.
5. $b = \mathfrak{g} * \mathfrak{r}$.
6. $g_2 \leftarrow g_2 + \lfloor b/r_2 \rfloor$; $h_2 \leftarrow h_2 + b \bmod r_2$.
7. while $(h_2 > r_2)$ do // This loop is executed at most twice.
   $g_2 \leftarrow g_2 + 1$; $h_2 \leftarrow h_2 - r_2$.
8. Return(g_2, h_2).

Note that the SPA information leaked from Algorithm 6.2 is not critical. However, in our implementation we have balanced all the paths in order to prevent any information leakage. We have employed the idea of replacing comparisons with subtractions and sign verification. Also, because of the available storage, we computed once the two's complements and the doubles of some values and stored them for subsequent use. Hence, the only operations used in the algorithm were addition and sign check operations. For example, the loop in step 7 is unfolded into two iterations one of which is inserted under step 4.2. Each iteration is implemented as follows, where `ONE` $= 1$ and `negR2` $= -r_2$

```
h2Sign = add(h2, negR2);
if (h2Sign == 1)  // h2 < 0
  add(h2, r2);
else
  add(g2, ONE);
```

Few dummy operations were inserted in some branches of the algorithm to balance the operations in each branch.

Integer division

The division algorithm used in step 6 is a non-restoring division algorithm [HVZ02; JV02]. The algorithm in [JV02] assumed that the dividend and the divisor were not signed and kept track of the sign change separately. In our algorithm, the dividend and the divisor are each internally represented by an array of w-bit digits that contains at least one sign bit. After the division is performed the dividend array contains the remainder concatenated with the quotient.

Also, Joye and Villegas [JV02] had proposed their algorithm for a memory constrained environment. Therefore, in their algorithm, the two's complement of the divisor array was computed and stored in the same array when needed, otherwise, this operation was performed on a dummy array. The address of the target register was determined by the sign bit and the carry bit from the operations in the previous iteration. Hence, while trying to save space, their algorithm still required an extra array for the dummy operation as well as an extra array negation in the main loop. Instead, in our algorithm, we stored both the divisor and its two's complement, then, in the main loop of the division algorithm, the carry of the current addition operation, whether 0 or 1, determines whether the divisor or its complement, respectively will be added in the following iteration.

We have also modified the algorithm such that the initial condition is that, when the most significant w-bit digit of the dividend and that of the divisor are aligned, the most significant bit of the divisor is at a bit position at least as high as that of the most significant bit of the dividend. In the original algorithms, the condition is that it should be at least one position higher. By doing that we saved in some cases the need to prepend an extra 0 digit to the dividend array, which translates into a saving of unnecessary w addition and shift left operations.

We use the following notation in the algorithm. $\texttt{ADD}_x(a, b)$ adds x digits from the array b to x digits of the array a, when they are aligned at their most significant digit. The operation $\texttt{SHL}_x(a)$ ($\texttt{SHR}_x(a)$) shifts left (right) by one bit the most significant x digits of a, the `carry` from this operation is the bit shifted out. Note that the `SHR` operation here includes a sign extension, that is, the sign of the shifted array is preserved. $\texttt{lsb}(a)$ ($\texttt{msb}(a)$) denotes the least (most) significant bit of a.

Algorithm 6.3. SPA-resistant division

INPUT: a, the array containing the dividend of length u w-bit digits, and b, the array containing the divisor of length v w-bit digits, $u > v$.

OUTPUT: $q = \lfloor a/b \rfloor$, $r = a \bmod b$.

1. $d[0] \leftarrow b$; $d[1] \leftarrow -b$.
2. $\delta \leftarrow 1$.
3. for i from 0 to $(u - v) * w$ do
   - 3.1 $a \leftarrow \texttt{ADD}_v(a, d[\delta]); \delta \leftarrow$ `carry`.
   - 3.2 $\texttt{SHL}_u(a)$.
   - 3.3 $\texttt{lsb}(a) \leftarrow \delta$.
4. $q \leftarrow$ the least significant $u - v$ digits of a.
5. $\texttt{SHR}_v(a)$; $\texttt{msb}(q) \leftarrow$ `carry`. // only the first v digits of a are shifted right

6. if ($\delta = 0$) then // final restoration
   $a \leftarrow \texttt{ADD}_v(a, d[\delta])$.
7. $r \leftarrow$ most significant v digits of a.
8. Return(q, r).

6.4.3 Modular Division

As discussed in Section 6.3, in order to use splitting III as in (6.14), we need to perform a modular division, $kr^{-1} \pmod u$ or $\boldsymbol{k}_{t+1} = \boldsymbol{k}_t \cdot r_{t+1}^{-1} \pmod u$, and possibly a modular multiplication, $\boldsymbol{r}_{t+1} = r_{t+1} \cdot \boldsymbol{r}_t \pmod u$. We focused our attention on prime fields. We have chosen to use Montgomery arithmetic due to the reason that the Montgomery inversion algorithm [ScKK00] was less expensive to protect against SPA attacks than to protect other inversion algorithms such as the extended Euclidean and the binary algorithms [HMV04]. For binary fields, the almost inverse algorithm [HMV04] has almost identical steps to the almost Montgomery inverse algorithm [ScKK00] that we modified below and can be modified in the same way.

In the following algorithm, a and b are integers internally represented each by an array of w-bit digits. The length of each array is $d = \lceil n/w \rceil$ digits. Note that for the modular inversion, as mentioned by Savas and Koç [ScKK00], b needs not be less than the modulus u, but be in $[1, 2^m - 1]$, where $m = dw$. Also note that the values $R^2 \bmod u$, where $R = 2^m$, and u' are computed once per modulus, *i.e.*, per curve.

Algorithm 6.4. Modular division

INPUT: u: a n-bit prime, $d = \lceil n/w \rceil$, $m = dw$, $R^2 \pmod u = (2^m)^2 \pmod u$, $u' = u^{-1} \bmod 2^w$, $a \in [1, p-1]$ and $b \in [1, 2^m - 1]$.

OUTPUT: $ab^{-1} \pmod u$.

1. Compute $b^{-1}R \pmod{u}$ using Algorithm 6.6.
2. Compute $x = a(b^{-1}R)R^{-1} \pmod{u}$ using Algorithm 6.5.
3. Return(x).

The following algorithm is Algorithm 14.36 in [MvOV96]. We include it here for the sake of completeness.

Algorithm 6.5. Montgomery multiplication [MvOV96]

INPUT: u: a n-bit prime, $d = \lceil n/w \rceil$, $m = dw$, $u' = u^{-1} \bmod 2^w$, $x = (x_{d-1}, \ldots, x_0)_{2^w}$ and $y = (y_{d-1}, \ldots, y_0)_{2^w}$.

OUTPUT: $xy2^{-m} \pmod{u}$.

1. $A \leftarrow 0$. // $A = (a_d, a_{d-1}, \ldots, a_0)_{2^w}$
2. for i from 0 to $d-1$ do
   2.1 $u_i \leftarrow (a_0 + x_i y_0) \bmod 2^w$.
   2.2 $A \leftarrow (A + x_i y + u_i m)/2^w$
3. if $(A > u)$ then
   $A \leftarrow A - u$.
4. Return(A).

The following algorithm was presented by Savas and Koç in [ScKK00] as the *modified Kaliski-Montgomery Inverse*.

Algorithm 6.6. Montgomery inversion

INPUT: u: a n-bit prime, $d = \lceil n/w \rceil$, $m = dw$, $R^2 \pmod{u} = (2^m)^2 \pmod{u}$, $u' = u^{-1} \bmod 2^w$ and $b \in [1, 2^m - 1]$.

OUTPUT: $b^{-1}R \pmod{u}$.

1. Compute f and $x = b^{-1}2^f \pmod{u}$ using Algorithm 6.7, where $n \leq f \leq m + n$.

2. if $(f \leq m)$ then

   2.1 $x \leftarrow xR^2R^{-1} \pmod{u}$ using Algorithm 6.5. // $x = b^{-1}2^{m+f} \pmod{u}$

   2.2 $f \leftarrow f + m$. // $f > m,\ x = b^{-1}2^{f} \pmod{u}$

3. $x \leftarrow x2^{2m-f}R^{-1} \pmod{u}$ using Algorithm 6.5.
   // $x = b^{-1}2^{f}2^{2m-f}2^{-m} = b^{-1}2^{m} \pmod{u}$

4. Return(x).

We modified the *almost Montgomery inverse algorithm* of [ScKK00] to be resistant to SPA attacks as in the following algorithm. `SwapAddress`(c, d) denotes interchanging the memory addresses of the integers c and d. This is an inexpensive operation, hence its usage as a dummy operation to balance the branches of the main loop. We implemented the "if" statement in steps 3.4 and 3.5 such that the number of conditions checked per loop iteration is always three. In assembly language, this can be easily ensured. Written in Java, step 3.4 is implemented as

```
if( ( xLSb == 0 ) && ( xLSb == 0 ) && ( xLSb == 0 ) ).
```

If the condition is false, due to short-circuit evaluation, the flow control will move to the following "if" after the first check, otherwise, it will perform the check three times. The following "if"—step 3.5—is similar but with the condition checked only two times

```
if( ( yLSb == 0 ) && ( yLSb == 0 ) ).
```

Algorithm 6.7. Almost Montgomery inverse

INPUT: u: a n-bit prime, $d = \lceil n/w \rceil$, $m = dw$ and $b \in [1, 2^m - 1]$.

OUTPUT: f and $b^{-1}2^{f} \pmod{u}$, where $n \leq f \leq m + n$.

1. $x \leftarrow u;\ y \leftarrow b;\ r \leftarrow 0;\ s \leftarrow 1$.

2. $f \leftarrow 0$.

3. while $(v > 0)$ do

   3.1 $\mathcal{U} \leftarrow x - y$; $\mathcal{V} \leftarrow -\mathcal{U}$.

   3.2 $\mathcal{T} \leftarrow r + s$.

   3.3 $f \leftarrow f + 1$.

   3.4 if $(((\texttt{lsb}(x) = 0)))$ then // This "if" is special

   $\texttt{SwapAddress}(x,\mathcal{U})$; $\texttt{SwapAddress}(x,\mathcal{U})$ // dummy

   $\texttt{SHR}(x)$; $\texttt{SHL}(s)$.

   3.5 else if $((\texttt{lsb}(y) = 0))$ then // This "if" is special

   $\texttt{SwapAddress}(y,\mathcal{V})$; $\texttt{SwapAddress}(y,\mathcal{V})$ // dummy

   $\texttt{SHR}(y)$; $\texttt{SHL}(r)$.

   3.6 else if $(\mathcal{V} >= 0)$ then

   $\texttt{SwapAddress}(y,\mathcal{V})$; $\texttt{SwapAddress}(s,\mathcal{T})$

   $\texttt{SHR}(y)$; $\texttt{SHL}(r)$.

   3.7 else

   $\texttt{SwapAddress}(x,\mathcal{U})$; $\texttt{SwapAddress}(r,\mathcal{T})$

   $\texttt{SHR}(x)$; $\texttt{SHL}(s)$.

4. $\mathcal{T} \leftarrow u - r$; $\mathcal{V} \leftarrow u + \mathcal{T}$.

5. if $(\mathcal{T} > 0)$ then

   Return$(f, \mathcal{T})$

   else

   Return$(f, \mathcal{V})$.

The drawback of this algorithm is that an SPA of the number of iterations of the main loop directly leaks the value of f. If f is uniformly distributed, the search space of

b is reduced from 2^m to $2^{m-\log_2 m}$, which is not a significant reduction. It is interesting to study how f is actually distributed.

6.5 Performance Comparison

In this section, we compare the performance of the three different methods of key splitting when combined with a SPA-resistant method. Among the described SPA countermeasures in Section A.2, we have chosen the Montgomery ladder and the fixed-sequence window methods to compare. The former is the most efficient method that processes the key on the bit level provided that the point addition and doubling operation are performed using the x-coordinate only as explained in Section A.2.1 on page 161. When storage is available for precomputation, the latter performs better and makes use of existent efficient point doubling and mixed coordinate addition operations. Its performance also exceeds that of the modified comb method (page 165) for $w > 2$ and it is simpler to implement. Both the Montgomery ladder and the fixed-sequence window methods do not contain dummy field or point operations—except possibly for the last bit in the latter method as mentioned in Section 6.4.1.

The comparison is shown in Table 6.5. We note the following:

- For the Montgomery ladder,
  - $(D + A)_x$ denotes the encapsulated point addition and doubling using the x-coordinate only evaluation.
  - For both splitting schemes I and II, the two terms are evaluated separately, each yielding a point in projective coordinates (based on the idea in [IT02]), which are finally added using projective coordinates point addition denoted by

$A_{\mathcal{P}}$ [3].

- For the fixed-sequence window method,
  - the first term in square brackets is the operation count of the precomputation step and the second term is that of the running time. $D_{\mathcal{A}}$ and $D_{\mathcal{J}}$ denote point doubling using affine and Jacobian coordinates, respectively. $A_{\mathcal{JA}}$ and $A_{\mathcal{J}}$ denote point addition using Jacobian-affine and Jacobian coordinates, respectively. $\mathcal{X} = 1I + 3(2^w - 3)M + (2^{w-1} - 1)S$ as explained in Section 6.4.1. For binary fields, the usage of the Jacobian coordinates can be replaced by that of the Lopez-Dahab coordinates.
  - For splitting scheme I, the precomputation step is done once and used for the evaluation of each of the terms of the splitting. A Jacobian coordinate point addition is performed at the end.
  - For splitting scheme II, both terms are evaluated simultaneously using interleaving (cf. Algorithm 2.4 on page 18), hence the precomputation used to compute rP is reused in computing (6.6). Note that both tables of precomputed multiples of P and rP need to coexist in memory during the evaluation of (6.6). This is not the case for scheme III since there is no simultaneous evaluation.
- For splitting scheme II, recall that $d = \lceil n/w \rceil$, $l = \lceil n/2 \rceil$ and $z = \lceil l/w \rceil$. Hence, it is easy to show that $z = \lceil d/2 \rceil$.

[3] Montgomery ladder can be combined with simultaneous multiplication to evaluate scheme II in (6.6) based on the idea in [IT02, Theorem 5], by replacing $2^{\lfloor n/2 \rfloor}P$ with rP, but it would be less efficient than evaluating each term separately, since each iteration consists of three point additions and one point doubling.

- For splitting scheme III, v is the bit length of r and $\mu = \lceil v/w \rceil$. v can be a relatively small integer, *e.g.*, 80, in case the actual key k is used to compute kr^{-1} before every ECSM execution. However, if a divisor chain $\boldsymbol{r}_t$ is stored as mentioned in Section 6.3, then $v = n$.

- It is assumed that the resulting point in all cases is represented in standard or Jacobian projective coordinates, hence, an inversion and some field multiplications and or squaring operations are needed to convert the point to affine coordinates.

Table 6.1: Performance comparison of the key splitting methods combined with Montgomery ladder or fixed-sequence window method

Scheme	Montgomery ladder	fixed-sequence window
I	$2(n-1)\ (D+A)_x + A_{\mathcal{P}}$	$[1\ D_{\mathcal{A}} + (2^{w-1}-1)\ A_{\mathcal{JA}} + \mathcal{X}]+$ $[2w(d-1)\ D_{\mathcal{J}} + 2(d-1)\ A_{\mathcal{JA}} + A_{\mathcal{J}}]$
II	$3(l-1)\ (D+A)_x + A_{\mathcal{P}}$	$2[1\ D_{\mathcal{A}} + (2^{w-1}-1)\ A_{\mathcal{JA}} + \mathcal{X}]+$ $[2w(z-1)\ D_{\mathcal{J}} + 3(z-1)\ A_{\mathcal{JA}}]$
III	$((v-1)+(n-1))(D+A)_x$	$2[1\ D_{\mathcal{A}} + (2^{w-1}-1)\ A_{\mathcal{JA}} + \mathcal{X}]+$ $[w((\mu-1)+(d-1))\ D_{\mathcal{J}} + ((\mu-1)+(d-1))\ A_{\mathcal{JA}}]$

Depending on the value v, splitting scheme III may be more efficient than the other two *e.g.*, if $v < 66$ for $n = 521$ and $w = 4$, which may not be adequate for the required security of the curve P-521 [NIST]. However, if $v = n$, then scheme I would perform better and is simpler than scheme III; and scheme II would be the most efficient. Note that the latter requires an extra precomputation phase and $z-1$ extra additions compared with the ECSM using Algorithm 6.1.

We should also note that, scheme III requires finite field operations, *e.g.*, inversion and modular multiplication, which are different for computation over prime fields than

for binary fields. Also some curve specific constants need to be precomputed and stored. This is not the case for the first two splitting schemes where the computation involves integer arithmetic operations.

6.6 Countermeasures to the DPA attack on ECDSA and ECMQV

In Section A.4.2, we mentioned Messerges' attack on the ECDSA signing algorithm [Mes00], where the step that has been attacked is

$$s = k^{-1}(m + dr) \bmod g. \tag{6.27}$$

More precisely, the attack targets the result of the integer multiplication dr where d is the signer's private key and r is part of the signature which is known to the attacker.

We first want to draw the attention that this attack is also applicable to the ECMQV key agreement algorithm [HMV04, Algorithm 4.51]. In this algorithm, A and B aim to establish a shared key. Hence, A—and similarly B—computes

$$s_A = (k_A + \overline{R}_A d_A) \bmod u, \tag{6.28}$$

where k_A is a secret random value used once (a *nonce*), $\overline{R}_A = f(R_A)$ where R_A is a point computed by A and sent to B and $f(.)$ is a known function, d_A is A's private key and u is the group order. It is clear that $\overline{R}_A$ is known to the attacker; hence, Messerges' attack is applicable to the step where the multiplication $\overline{R}_A d_A$ is performed.

As a countermeasure to the attack on ECDSA, Messerges suggested multiplying both m and d by a random value ω and, after computing (6.27), multiplying s by ω^{-1}. This method requires $1I+3M$ additional operations. Note that $\xi = d\omega \bmod g$ is computed first

then $\xi r \bmod g$. That is, due to the modulo operation, no intermediate value combining r with d is computed. Hence, a second-order DPA attack as described in [JPS05] (cf. Section A.4.3) is not applicable.

As an alternative countermeasure, we suggest using the same idea as in splitting scheme III by computing $\xi = d\omega \bmod g$ and $\psi = \omega^{-1} r \bmod g$ and then $\xi\psi \bmod g$. Then this would require $1I + 2M$ additional operations. Again, no intermediate values can be computed for the partitioning required by Messerges' attack or a second-order DPA attack, since the value $\xi\psi$ is uncorrelated to dr due to the modulo operation.

A more efficient alternative is, since k in (6.27) is a nonce, to compute $\xi = k^{-1} d \bmod g$ then s is computed as

$$s = (mk^{-1} + \xi r) \bmod g,$$

requiring only $1M$ additional operation.

Similarly, to protect the ECMQV key agreement algorithm, (6.28) can be modified as follows

$$s_A = (k_A d_A^{-1} + \overline{R}_A) d_A \bmod u, \tag{6.29}$$

where d_A^{-1} can be precomputed, this requires only an additional $1M$. If it is not desirable to store both the key d_A and its inverse, then multiplicative splitting can be used here as well by choosing a random integer ω and computing $\xi = R_A \omega \bmod u$ and $\psi = \omega^{-1} d_A \bmod u$ and then $\xi\psi \bmod u$, requiring $1I + 2M$ additional operations.

6.7 Conclusions and Future Work

In this chapter, we have dealt in detail with key splitting schemes as a countermeasure to DPA attacks. We have identified the weaknesses and strengths of each scheme. For

the first scheme, we have proved by a lemma that it should not be evaluated using simultaneous point multiplication methods.

For the second scheme, we have identified the occurrences of collisions on intermediate points computed during a simultaneous point multiplication. We have classified these collisions into major and minor ones. The former type of collision is directly related to the key bits while the latter is not. We have proved by lemmas the conditions associated with the occurrence of major collisions and have shown how to effectively avoid them. We have also distinguished minor collisions and studied them experimentally on small numbers in an attempt to quantify their probability. We have suggested ways to overcome them in spite of their small likelihood. We have also recognized the effect of the fixed-sequence recoding on both the major and minor collisions. We have introduced the notion of splitting a dividend with a new divisor, given the old quotient, remainder and divisor, without computing internally any part of the dividend; we have presented and SPA resistant algorithm to carry this operation. We have as well presented an SPA-resistant integer division algorithm that is more space and time efficient than what has previously appeared in the literature.

For the third scheme, we have discussed the required finite field algorithms and presented our modifications to the almost Montgomery inverse algorithm in order to make it SPA-resistant.

We have compared the performance of the three schemes when combined each with the Montgomery ladder algorithm or the interleaving algorithm.

Finally, we have demonstrated the applicability of Messerges' ECDSA attack to the ECMQV algorithm and have presented efficient countermeasures to first-order and second-order DPA attacks on both algorithms.

It is interesting to pursue the study on minor collisions and to provide an analytical

result for their probability of occurrence as well as the effect of the fixed-sequence recoding on them. It is also desirable to know the distribution of f resulting from the almost Montgomery inverse algorithm over the integers in the range $[1, 2^m - 1]$ and its relation to the modulus.

One of the examiners suggested combining splitting schemes I and II to avoid the collisions in both schemes. An effort to such combination should carefully assess its effect on the performance of the ECSM algorithm.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

In this thesis, we have dealt with key randomization countermeasures to DPA attacks. We have discovered interesting statistical properties of the number of binary signed digit representations (BSD) and τ-adic representations of integers. We have also studied in detail the employment of key splitting techniques.

In Chapter 3, we have dealt with the randomized algorithms proposed by Oswald and Aigner (OA) and by Ha and Moon (HM) along with the NAF generating algorithms on which they were based. We have demonstrated that the HM algorithm can generate any possible BSD representation of an integer while this is not the case for the OA algorithm. We have also shown how the random decision proposed by Ha and Moon can be simply inserted in Solinas NAF generating algorithm. We have also provided a new analytical result for the average case complexity of the OA algorithm, that is it requires 3.5% more point additions/subtractions on average than the binary algorithm. We have confirmed this result using both the Markov chains and the grammatical specification methods.

In Chapter 4, we have answered a number of questions related to the number of binary-signed digit (BSD) representations of an integer $k \in [0, 2^n - 1]$, such as the average number of representations among integers of the same length and the bit patterns of $k_{max_1,n}$ and $k_{max_2,n}$, *i.e.*, the integers of length n bits that have the maximum number of BSD representations. We have presented the recurrence that governs the number of representations of such integers and have, hence, proved that it is a Fibonacci number and is $O(\varphi^n)$, where $\varphi \approx 1.618$ is the golden ratio [Kos01]. We have presented an algorithm that calculates in $O(n)$ the exact number of BSD representations of k that are of length n sbits. We have also presented an algorithm that generates in $O(n)$ a random BSD representation of k by scanning its bits starting from the most significant end, unlike the HM algorithm, and outputs the sbits one by one in their correct order. In addition, we have presented an algorithm that can generate all BSD representations of an integer, which has helped us provide an alternate expression for the Fibonacci number that can be computed without using floating point arithmetic, or the first few terms of which used as an approximation. We have also proved that prepending 0s to the binary representation of an integer results in only a linear increase in the number of its BSD representations.

In Chapter 5, we have introduced a new method of randomizing the τ-adic representation of a key in ECCs using Koblitz curves. The input to the randomization algorithm is a τNAF of length l. The symbols of the resulting sequence are output one at a time from the least significant to the most significant which allows the simultaneous execution of the scalar multiplication operations. The length of the random representation is at most $l + 2$. We have proved that the average Hamming density of all representations for all τNAFs of the same length is 0.5. We have also presented the pattern of τNAFs with maximum number of representations and the recurrence governing that number which was identical to the recurrence found in Chapter 4. By modeling our algorithm as a

nondeterministic finite automaton and by using adjacency matrices, we have presented a deterministic method to determine the average and the exact number of representations of a τNAF, where the average number is very close to $\left(\frac{3}{2}\right)^l$. There is an obvious similarity between the results of this chapter and the corresponding ones in Chapter 4.

In Chapter 6, we have dealt in detail with the key splitting schemes as a countermeasure to DPA attacks and identified the weaknesses and strengths of each scheme. We have proved that scheme I should not be evaluated using simultaneous point multiplication methods.

When those methods are used with scheme II, we identified the occurrence of some collisions on intermediate points. We have classified these collisions into major and minor ones. The former type of collision is directly related to the key bits while the latter is not. We have proved by lemmas the conditions associated with the occurrence of major collisions and have shown how to effectively avoid them. We have also distinguished minor collisions and studied them experimentally on small numbers in an attempt to quantify their probability. We have suggested ways to overcome them in spite of their small likelihood. We have also recognized the effect of the fixed-sequence recoding as an SPA countermeasure on both the major and minor collisions. We have introduced a new algorithm that performs a new splitting from an old one without computing internally any part of the private key. And we have presented an SPA-resistant integer division algorithm.

As for scheme III, we have discussed the required finite field algorithms and presented our modifications to the almost Montgomery inverse algorithm in order to make it SPA-resistant. We have compared the performance of the three schemes when combined each with the Montgomery ladder algorithm or the interleaving algorithm. Finally, we have demonstrated the applicability of Messerges' ECDSA attack to the ECMQV algorithm

and have presented efficient countermeasures to DPA attacks on both algorithms.

7.2 Future Work

In Section 4.3.2, we have provided an alternate formula for Fibonacci numbers that can be truncated and used as an approximation when floating point arithmetic is not available. It is interesting to study how many terms are necessary for a *good* approximation. Also, to investigate the conversion of this formula to the closed form expression for Fibonacci numbers which is known as Binet's formula [Kos01].

Also of interest is to investigate how the randomization method we applied to the τNAFs in Chapter 5 and the associated properties of the representation can be carried to any complex radix with norm 2 or any arbitrary norm. This complex number should satisfy an equation such as (5.2), in order to be able to recursively replace digits with a larger absolute value than those in the digit set with the latter ones during the randomization procedure.

It is interesting to pursue the study we started in Chapter 6 on minor collisions in the second splitting scheme and to provide an analytical result for their probability of occurrence as well as the effect of the fixed-sequence recoding on them.

It is desirable also to know the distribution of f resulting from the almost Montgomery inverse algorithm over the integers in the range $[1, 2^m - 1]$ and its relation to the modulus.

Appendix A

Overview of Side Channel Attacks on ECCs and Known Countermeasures

In this appendix, we first provide a brief background on other ECSM algorithms that were not included in Section 2.1.3, but that are the base for some subsequently described countermeasures. We then discuss the SPA and DPA attacks and their variants along with many of the countermeasures that appeared in the literature. Finally, we draw a comprehensive conclusion on the applicability of each type of attack and the principles of resisting it.

Our discussion is mostly focused on elliptic curves defined over fields with prime modulus since those are the ones adopted in suite B [NSA]. However, many of the countermeasures are generic or have corresponding application on curves defined over binary fields.

A.1 Elliptic Curve Scalar Multiplication Algorithms

In this section, we provide other ECSM algorithms that are essential in understanding some of the SPA and DPA countermeasures in this appendix.

Fixed-base windowing method

This is also known as Yao's method [Ber01; Yao76]. It was presented in a different form with precomputation by Brickell *et al.* [BGMW93]. In this method, k is recoded to the radix 2^w as $(K_{d-1}, \ldots, K_1, K_0)_{2^w}$ where $d = \lceil \frac{n}{w} \rceil$. Let β be the largest absolute digit in the digit set. The digit set can be formed of only positive digits ($\beta = 2^w - 1$) or can contain both positive and negative digits ($\beta = 2^{w-1}$). Let $Q_j = \sum_{i:K_i=j} 2^{wi}P - \sum_{i:K_i=-j} 2^{wi}P$ for $1 \leq j \leq \beta$. Then

$$kP = \sum_{i=0}^{d-1} K_i(2^{wi}P) = \sum_{j=1}^{\beta} j \left(\sum_{i:K_i=j} 2^{wi}P - \sum_{i:K_i=-j} 2^{wi}P \right) = \sum_{j=1}^{\beta} jQ_j.$$

$$= Q_\beta + (Q_\beta + Q_{\beta-1}) + \cdots + (Q_\beta + Q_{\beta-1} + \cdots + Q_1).$$

The right-to-left version of this method is illustrated in Algorithm A.1 (cf. [Möl02]).

Algorithm A.1. Yao's method, right-to-left version

INPUT: Window width w, $d = \lceil n/w \rceil$, $k = (K_{d-1}, \ldots, K_0)_{2^w}$ and $P \in E(\mathbb{F}_p)$.

OUTPUT: kP.

1. $Q_1, \ldots, Q_\beta \leftarrow \mathcal{O}$; $R \leftarrow P$.
2. for i from 0 to $d-1$ do

   2.1 if $(K_i > 0)$ then

   $$Q_{K_i} \leftarrow Q_{K_i} + R.$$

   else

   $$Q_{|K_i|} \leftarrow Q_{|K_i|} - R.$$

2.2 $R \leftarrow 2^w R$.

3. for $i = \beta - 1$ down to 1 do

   $Q_i \leftarrow Q_i + Q_{i+1}$.

4. for $i = 2$ to β do

   $Q_1 \leftarrow Q_1 + Q_i$.

5. Return(Q_1).

Note that the number of additions in Algorithm A.1 is $d + 2\beta - 2$, but because the accumulators are initialized to $\mathcal{O}$, there are β additions saved (other additions are also saved for the digits, if any, that did not appear in the representation of k.)

Storage: $\beta + 1$ accumulators (1 for R).

Precomputation: none

Expected running time: $w(d-1)D + (d + \beta - 2)A$.

Coordinate selection: We suggest to use Chudnovsky coordinates for the accumulators Q_i and Jacobian coordinates for R. If it is preferred to limit the space taken by the accumulators, then they can be represented in Jacobian coordinates.

The Brickell *et al.* version of the method is illustrated in Algorithm A.2.

Algorithm A.2. Brickell *et al.* version of Yao's method

INPUT: Window width w, $d = \lceil n/w \rceil$, $k = (K_{d-1}, \ldots, K_0)_{2^w}$ and $P \in E(\mathbb{F}_p)$.

OUTPUT: kP.

1. *Precomputation.* Compute $P_i = 2^{wi} P,\ 0 \leq i \leq d-1$.

2. $A \leftarrow \mathcal{O};\ B \leftarrow \mathcal{O}$.

3. for j from β down to 1 do

3.1 For each i for which $|K_i| = j$ do

if $K_i > 0$ then $B \leftarrow B + P_i$. //Add Q_j to B.

else $B \leftarrow B - P_i$.

3.2 $A \leftarrow A + B$.

4. Return(A).

Storage: d points.

Precomputation: $w(d-1)$ D.

Expected running time: $(d + \beta - 2)$ A.

Coordinate selection: If the base point P is fixed, the precomputed points are stored in affine coordinates, B in Chudnovsky coordinates and A in Jacobian coordinates. But if P is unknown point, we suggest using Jacobian coordinates for the precomputed points and A, and Chudnovsky coordinates for B.

In both versions of the algorithm the total number of addition and doubling operations is the same. Moreover, if we take a closer look, we will find that this method and the window method perform the same total number of group operations for the same key and the same 2^w-ary recoding of that key [Ber01; Ava].

Comb method

This method is attributed to Lim and Lee [LL94][1]. In this method the key k is written as a matrix of w rows and $d = \lceil \frac{n}{w} \rceil$ columns. The bits of the key are processed one column at a time. The precomputed points are

$$[a_{w-1}, \ldots, a_2, a_1, a_0]P = a_{w-1}2^{(w-1)d}P + \cdots + a_2^{2d}P + a_1 2^d P + a_o P$$

[1]But according to Bernestein [Ber01], Lim-Lee's method is a special case (single-level recursion) of Pippenger's algorithm [Pip76].

for all possible bit strings $(a_{w-1}, \ldots, a_1, a_0)$. The algorithm is as follows.

Algorithm A.3. Comb method

INPUT: Number of rows w, $d = \lceil n/w \rceil$, $k = (k_{n-1}, \ldots, k_0)_2$ and $P \in E(\mathbb{F}_p)$.

OUTPUT: kP.

1. *Precomputation.* Compute $[a_{w-1}, \ldots, a_2, a_1, a_0]P$ for all bit strings $(a_{w-1}, \ldots, a_1, a_0)$ of length w.
2. By padding k on the left with 0s if necessary, write $k = K^{w-1}|| \cdots ||K^1||K^0$, where each K^j is a bit string of length d, *i.e.*, a row of the matrix. Let K^j_i denote the ith bit of K^j.
3. $Q \leftarrow [K^{w-1}_{d-1}, \ldots, K^1_{d-1}, K^0_{d-1}]P$.
4. for i from $d-2$ down to 0 do
   - 4.1 $Q \leftarrow 2Q$.
   - 4.2 $Q \leftarrow Q + [K^{w-1}_i, \ldots, K^1_i, K^0_i]P$.
5. Return(Q).

Storage: $2^w - 1$ points.

Precomputation: $(w-1)d\ D + (2^w - 1 - w)\ A$.

Expected running time: $(d-1)\ D + \left(\frac{2^w-1}{2^w}d - 1\right)\ A$.

Coordinate selection: as in the window method.

The savings of this method are more evident for fixed point P. Yet, if P is unknown, the number of doubling operations is $n-1$ including precomputation and the savings in the addition operations are comparable to window methods.

A.2 Countermeasures Against SPA and Timing attacks

As explained by Avanzi in his survey [Ava05], SPA attacks are based on observing a single trace of power consumption or leaked emissions. By doing so, one can infer the sequence of group operations since they typically take many clock cycles. If the trace of the point doubling is distinguishable from that of the point addition and the latter operation is performed depending on the secret key bit value, the bits of the secret key can be revealed. Some of the timing attacks also exploit the irregularity of the ECSM execution. In this section, we summarize the different countermeasures that were proposed to defeat these attacks.

A.2.1 Fixed Sequence of Point Operations

The same sequence of point operations, point adding and point doubling, is executed independent of the key bits.

Double-and-Add-Always

This technique was suggested by Coron [Cor99] (Algorithm A.5 appeared in [IT02]). In order to make the sequence of doubling and adding independent of the key bit, a point addition is performed whether the key bit is 0 or 1. The point addition is *dummy* if the bit is 0.

Algorithm A.4. Left-to-Right Double-and-Add-Always

INPUT: k and $P \in E(\mathbb{F}_p)$.

OUTPUT: kP.

1. $Q[0] \leftarrow \mathcal{O}$.
2. for i from $n-1$ down to 0 do
   - 2.1 $Q[0] \leftarrow 2Q[0]$.
   - 2.2 $Q[1] \leftarrow Q[0] + P$.
   - 2.3 $Q[0] \leftarrow Q[k_i]$.
3. Return($Q[0]$).

Algorithm A.5. Right-to-Left Double-and-Add-Always

INPUT: k and $P \in E(\mathbb{F}_p)$.

OUTPUT: kP.

1. $Q[0] \leftarrow \mathcal{O}$; $Q[2] \leftarrow P$.
2. for i from 0 to $n-1$ do
   - 2.1 $Q[1] \leftarrow Q[0] + Q[2]$.
   - 2.2 $Q[2] \leftarrow 2Q[2]$.
   - 2.3 $Q[0] \leftarrow Q[k_i]$.
3. Return($Q[0]$).

Drawbacks:

- This countermeasure is vulnerable to safe-error attack [YJ00]. The safe-error is an adaptive fault-analysis attack (hardware attack). It induces a fault during an operation to know whether it is dummy or not. If it is a dummy operation, the fault does not change the output. If not, the output is incorrect. Thus, the attacker can distinguish whether it is dummy or not and reveal the secret key. The algorithm needs to be repeated several times (at least n times) with a single fault induced each time until all of the key bits are revealed. Hence, the attack applies if the key is not randomized.

- Algorithm A.4 is prone to Fouque-Valette doubling attack [FV03]. This is a variant of SPA attack, in which an attacker inputs the points P and $2P$ to the algorithm and collects the corresponding traces S_P and S_{2P}, respectively. Let $S_{P_D}(i)$ be the

portion of the trace S_P corresponding to the doubling in iteration i. If $S_{P_D}(i)$ is highly correlated with $S_{2P_D}(i+1)$, then $k_{i+1} = 0$, otherwise, $k_{i+1} = 1$. Table A.1 gives an example of the attack. The attack applies as long as neither the key nor the intermediate points are randomized.

Table A.1: Example of Fouque-Valette doubling attack

i	k_i	comput. of kP	comput. of $k(2P)$
6	1	$2 \times \mathcal{O}$	$2 \times \mathcal{O}$
		$\mathcal{O} + P$	$\mathcal{O} + 2P$
5	0	$2 \times P$	$2 \times 2P$
		$2P + P$	$4P + 2P$
4	0	$2 \times 2P$	$2 \times 4P$
		$4P + P$	$8P + 2P$
3	1	$2 \times 4P$	$2 \times 8P$
		$8P + P$	$16P + 2P$
2	1	$2 \times 9P$	$2 \times 18P$
		$18P + P$	$36P + 2P$
1	1	$2 \times 19P$	$2 \times 38P$
		$38P + P$	$76P + 2P$
0	0	$2 \times 39P$	$2 \times 78P$
		$78P + P$	$156P + 2P$
		return $78P$	return $156P$

- Another similar attack was proposed by Yen *et al.* [YLMH05]. This is an SPA attack that uses as input a point of order 2. It was presented on RSA with the message $n - 1$ as input, where n here is the modulus. This attack is not applicable to the curves in the standards that have prime order, but we mention it for the sake of completeness.

  Let P_0 be a point of order 2. If P_0 is an input to Algorithm A.4, then the interme-

diate point computed at the end of each iteration i, is $\mathcal{O}$ if $k_i = 0$ or P_0 if $k_i = 1$. Hence, collisions between the iteration intervals in a single trace can reveal the key bits. If the attack is applied to window-based or comb methods, as will follow, it can reveal whether the digit at iteration i is odd or even, that is $\lceil n/w \rceil$ bits can be revealed.

To resist this attack, we can check whether the input point P belongs to the main subgroup before starting the ECSM execution. This is done by verifying whether hP is equal to $\mathcal{O}$ where h is the cofactor. Also this attack is not possible if the base point is blinded as we suggest in Section A.3.2 or as in Section A.3.2.

- Performance penalty: the algorithm performs $nD + nA$. Using Jacobian and Jacobian-affine coordinates, the operation count is

$$12nM + 7nS + (1I + 3M + 1S).$$

Montgomery ladder

The Montgomery ladder has been first used for the Montgomery form of an elliptic curve defined over a prime field [Mon87]. The following is the algorithm as it appears in [Gou03].

Algorithm A.6. The Montgomery Ladder

INPUT: k and $P \in E(\mathbb{F}_p)$.

OUTPUT: kP.

1. $Q[0] \leftarrow P$; $Q[1] \leftarrow 2P$.
2. for i from $n-2$ down to 0 do
   - 2.1 $Q[1-k_i] \leftarrow Q[0] + Q[1]$.
   - 2.2 $Q[k_i] \leftarrow 2Q[k_i]$.

3. Return($Q[0]$).

Another version of the algorithm appeared in [IT02], making use of a third register to remove the register dependence between the addition and the doubling so that the two operations can be executed in parallel.

Algorithm A.7. The Montgomery Ladder (parallelizable)

INPUT: k and $P \in E(\mathbb{F}_p)$.

OUTPUT: kP.

1. $Q[0] \leftarrow P$; $Q[1] \leftarrow 2P$.
2. for i from $n-2$ down to 0 do
   - 2.1 $Q[2] \leftarrow 2Q[k_i]$.
   - 2.2 $Q[1] \leftarrow Q[0] + Q[1]$.
   - 2.3 $Q[0] \leftarrow Q[2 - k_i]$.
   - 2.4 $Q[1] \leftarrow Q[1 + k_i]$.
3. Return($Q[0]$).

Note that in steps 2.3 and 2.4, the addresses of the registers rather than their contents can be swapped [IIT02].

In the Montgomery ladder, the doubling (ECDBL) and addition (ECADD) are performed in every iteration and there are no dummy operations. The algorithm takes

$$(n-1)\ D + (n-1)\ A.$$

The remarkable characteristic of this algorithm is that, regardless of the value of k_i, $Q[1] - Q[0] = P = (x, y)$. This enabled the execution of the algorithm by computing only the x-coordinate of $Q[0]$ and $Q[1]$ and recovering the y-coordinate at the end. The

doubling and addition formulas needed for this purpose were introduced for a Weierstrass curve over $\mathbb{F}_p$ by Brier and Joye in [BJ02] and by Izu and Takagi in [IT02]. They proposed the use of standard projective coordinates which yields the following operation count, assuming that the base point P has the Z-coordinate $= 1$[2].

Addition: $8M + 2S$. 3 of the multiplications are by constants a, b and x.

Doubling: $6M + 3S$. 2 of the multiplications are by constants a and b.

To obtain $x(kP)$, *i.e.*, to convert back to affine coordinates: $1I$.

To recover $y(kP)$: $1I + 4M + 1S$.

We can assume that $a = -3$ as in Section 2.1.1. Also, the intermediate result of the multiplication by b from the doubling can be used in the addition. Hence, we can deduct $1M$ from the doubling and $2M$ from the addition. The total operation count is therefore

$$11nM + 5nS + (2I + 4M + 1S).$$

Using the same idea—but not the same formula—as in [IT02], the projective coordinate $Y(kP)$ can be first computed, then the point converted into affine coordinate. This has the benefit of obtaining the point in projective coordinates first if it is to be used in another point addition as in the case of the additive key splitting countermeasure (cf. Section 6.5). The cost of these two computations in this order is less than the previous order. We found the cost that the total cost becomes

$$11nM + 5nS + (1I + 10M + 2S).$$

[2]This is our operation count based on [BJ02], whereas the authors did not distinguish between M and S. It agrees with the operation count for ECADD$_m$ (using multiplicative formula to compute the x-coordinate) and ECDBL in [IT02], but the formulas for the y-recovering are different. Also, the encapsulation of ECADD and ECDBL in [IT02] yields $12M + 4S$ per iteration, if we assume $a = -3$ and we do not count multiplication by a.

Fixed-Sequence Window Method

This SPA countermeasure is based on recoding the key k to the base 2^w, where w is the window width [Möl01; OT03; Thé06] and performing the scalar multiplication using left-to-right window method as in Section 2.1.3. The difference is that the digit 0 is not in the digit set. Hence, the sequence of additions and doubling is fixed which is w doubling operations followed by one addition. The recoding of k can be carried from right to left and stored to be used in the scalar multiplication [Möl01; OT03] or from left to right and interleaved with the scalar multiplication [Thé06].

Storage: 2^{w-1} points, when using the digit set $\{\pm 1, \pm 3, \ldots, \pm(2^w - 1)\}$, this is the smallest proposed set [OT03; Thé06].

Precomputation: $1\ D + (2^{w-1} - 1)\ A$.

Running time: $n\ D + \frac{n}{w}\ A$.

NAF-based Scalar Multiplication with Dummy Point Operations

In this countermeasure, the NAF of the key k is used with the left-to-right double-and-add algorithm as in Section 2.1.3. The difference is that 2 digits are processed at a time and depending on their value, dummy point addition and/or doubling are inserted to unify the sequence $DDAD$ [CJ03].

Expected running time: $\frac{3n}{2}\ D + \frac{n}{2}\ A$.

Another similar countermeasure proposed in [HM02b] unifies the sequence DDA, but there is a recoding step prior to the scalar multiplication that recodes the w-NAF of k into another "SPA-resistant" sequence. This recoding step was removed in [CJ03].

Expected running time: $\frac{10n}{9}\ D + \frac{5n}{9}\ A$.

Drawbacks:

- Since dummy operations are inserted, this countermeasure is vulnerable to safe-error attack [YJ00] as in Section A.2.1. This attack applies if the key is not randomized.

Replacing Digit 0 in the Comb Method

This countermeasure was proposed in [HPB04]. It deals with the occurrence of 0 digits in the comb method. Recall from Section A.1 that in this method, the bits of the key k are arranged in a $w \times d$ matrix and are processed one column at a time from left to right. Therefore, recoding methods that avoid the 0 digit as in Section A.2.1 are not applicable.

The following is our interpretation of the proposed idea: if at iteration $i = a$ of Algorithm A.3, the current column $C_a = [K_a^{w-1}, \ldots, K_a^1, K_a^0]$ is all 0, the value of the next non-zero column $C_b = [K_b^{w-1}, \ldots, K_b^1, K_b^0]$, where $b < a$, replaces the value of C_a. Then, in subsequent iterations where $a > i \geq b$, the point C_bP is subtracted. This yields the correct result since after iteration $i = b$

$$\begin{aligned} Q &= \sum_{i=a+1}^{d-1} 2^{i-b} C_i P + 2^{a-b} C_b P - \sum_{i=b}^{a-1} 2^{i-b} C_b P, \\ &= \sum_{i=a+1}^{d-1} 2^{i-b} C_i P + C_b P, \\ &= \sum_{i=b}^{d-1} 2^{i-b} C_i P. \end{aligned}$$

We can see that this column processing can take place while the algorithm is executing. Yet, in the proposed countermeasure, the columns of the key are first processed from right to left and a sign is assigned to every column. The zero columns are replaced and the signs of the less significant columns adjusted. After that, the scalar multiplication is performed with the new column-sign representation.

A.2.2 Unified Addition and Doubling Formulas

Brier and Joye [BJ02] rewrote the addition of points on the general Weierstrass form of elliptic curves so that it can be used for adding two different points as well as doubling a point. For curves over $\mathbb{F}_p$, the field operations count for affine and projective coordinates is

coordinates	operations count
$\mathcal{A}$	$1I, 3M, 2S$
$\mathcal{P}$	$13M, 4S$

We did not include in the count one multiplication by a for each of the coordinate systems, assuming $a = -3$ as before. Also we separated the squaring from the multiplication count as opposed to the authors' count.

Drawbacks:

- Let P_1 and P_2 be two point on E, such that $P_1 \neq \pm P_2$. To obtain $P_3 = P_1 + P_2$ in affine coordinates, the unified formula requires the inversion of $y_1 + y_2$. If $y_1 + y_2 = 0$ (note that $x_1 \neq x_2$), an error occurs if those two points are processed [AT03]. This can happen if at iteration $i = a$ of the left-to-right scalar multiplication algorithm, there is an attempt to calculate $mP + P$, where P is the input point and $m = \sum_{i=a}^{n-1} d_i 2^{i-a}$ and P is chosen such that $y(P) + y(mP) = 0$, P is called in this case an *m-th self-collision point*. Such points could be found for small m and thus this attack can reveal few bits of the key. This attack applies if the key is not randomized.

  If projective coordinates are used, the formula contains $M = Y_1 Z_2 + Y_2 Z_1$, which will yield 0 if the attack condition is met. This in turn will result in $Z_3 = 0$, that is the point $\mathcal{O}$ which is not the correct result for P_3. We suggest to change the

addition formula so that M is not computed explicitly at the expense of two extra field multiplications. Similar changes to other addition formulas were suggested by Akishita and Takagi in [AT03].

- Another attack was presented by Walter [Wal04b] when the unified formula is used with Algorithm 2.1. The attack exploits the use of a modular multiplier that contains a conditional subtraction at the end such as the Montgomery modular multiplier (MMM). It is assumed that this subtraction can be detected with a side-channel leakage. The unified formula contains the following intermediate multiplications

$$U_1 = X_1Z_2, U_2 = X_2Z_1, \quad S_1 = Y_1Z_2, S_2 = Y_2Z_1.$$

When the unified formula is used for point doubling, the operands for the first two multiplications are the same as well as for the second two multiplications. This is not the case for point addition. Hence, if the attacker observes a subtraction in the first multiplication and not in the second, he is certain that this operation is a point addition.

The attack uses one trace of side-channel leakage, but in order to obtain a good trace with sufficient point additions identifiable, the attacker needs to set some sample space. For example to attack a key in the field P-192, a sample size of 512 can reduce the key search space to $2^{17.6}$. The author asserts that blinding the key, blinding the input point and/or randomizing the input point will not defeat the attack, but may actually help it. Yet, the attack is not feasible if Algorithm 2.2 or a window method are used.

A.2.3 Inserting Dummy Field Operations

Inserting Dummy Field Multiplications

Gebotys and Gebotys [GG02] showed how to insert dummy field operations in the addition and doubling formulas using Jacobian coordinates for Weierstrass curves over $\mathbb{F}_p$ (assuming $a = -3$). The point addition was split into two modules, and the point doubling was kept as a single module. The field arithmetic operations were arranged so that their sequence in each module is exactly the same. The dummy operations are:
In doubling: 1 M (also 2 multiplications were used to perform squaring).
In addition: 1 M + 3 modular additions/subtractions (A/S) + 10 shift left operations (ShL).
(The authors' implementation of Jacobian coordinates addition formula used $1M$ more than in [HMV04].)
Operations count:
Addition: $14M + 4S + 12A/S + 10ShL$.
Doubling: $7M + 2S + 6A/S + 5ShL$.

Drawbacks:

- Since dummy field operations are inserted, this countermeasure can be attacked by safe-error attack [YJ00] as in Section A.2.1. Though the attack will need more precision, since the fault should be inserted in a precise field multiplication rather than anywhere in a point doubling/addition operation. The attack applies if the key is not randomized.

- Performance penalty compared with using mixed coordinates for point addition.

Common Side-Channel Atomicity

This approach introduced by Chevallier-Mames *et al.* [CMCJ04] can be considered a refinement of the previous countermeasure. In this case, both the addition and doubling operations are split into elementary basic blocks called *side-channel atomic blocks*. Hence, even if the addition is performed depending on the current key bit, the sequence of doubling and addition would appear as a repetitive sequence of field operations that does not reveal the boundaries between the point operations.

They also use Jacobian coordinates for Weierstrass curves over $\mathbb{F}_p$ assuming a is random. In this case the atomic block is formed of $1M + 2A + 1N$, where A is modular addition and N is negation. All the squarings are performed as multiplications and multiplication by 2 is performed as modular addition (no shift operations are used). Dummy A and N operations are inserted in different blocks, but there are no dummy multiplications.

The main loop of the algorithm consists of those 4 operations, where the addresses of the input and output registers for each operation is stored in a look-up table.

Operations count:

Addition: 16 blocks $= 16M + 32A + 16N$.

Doubling: 10 blocks $= 10M + 20A + 10N$.

Drawbacks:

- The safe-error attack is still applicable to this countermeasure but the precision needed is more than with the previous one since there are no dummy multiplications. Also allocating a dummy operation will not directly lead to the knowledge of the key bit, since the boundaries between addition and doubling are not apparent. A

minimum number of blocks need to be identified in order to reveal whether the current point operation is an addition or a doubling. The attack applies if the key is not randomized.

- Performance penalty compared with using mixed coordinates for point addition.

A.3 Countermeasures Against DPA attacks

Even when SPA countermeasures are employed, if the sequence of key bits is the same in every scalar multiplication execution, the internal state of the device is correlated with the bit value. The typical scenario where DPA attacks are applicable is when a long-term key is processed by the ECSM of an arbitrary point on the elliptic curve. Differential attacks can be explained as follows [Cie03; Cor99].

The attacker has access to a device that performs ECSM using a left-to-right binary algorithm with some SPA countermeasure (with no precomputation) *e.g.*, Algorithm A.4. The attacker's goal is to reveal the key k that is kept secret in the device. The attacker can input different points $P_1, P_2, \ldots, P_e$ to the device and make it perform a scalar multiplication of each point by k. During each execution, the attacker collects the side-channel information associated to the computation of kP_i *e.g.*, power consumption trace. Let this information be $C_i(t)$, which is a function of time t. Note that this is a known-plaintext attack, since the attacker needs only to know the input points and not necessarily choose them. Once the attacker collects all the side-channel traces, he does not need the device to perform his statistical analysis. Let $k_{b\to a}$ denote $\lfloor (k \mod 2^{b+1})/2^a \rfloor$ or, simply, the bits of k from bit position b down to bit position a, with $b \geq a$.

Now the attacker will guess the key bits starting from the most significant bit. Assume the key is n bits long and $k_{n-1} = 1$. If $k_{n-2} = 0$ then $4P$ is computed in iteration $i = n-3$,

otherwise, $4P$ is never computed. The following table shows the value of $Q[0]$ in step 2.2 in the first 3 iterations of Algorithm A.4.

	i		
k_{n-2}	$n-1$	$n-2$	$n-3$
0	$\mathcal{O}$	$2P$	$4P$
1	$\mathcal{O}$	$2P$	$6P$

The attacker computes $4P_i$ for $i = 1, \ldots, e$, and specify a decision boolean function s_i *e.g.*, a specific bit value of $4P_i$ [3]. Then, the following function $g(t)$ is computed

$$g(t) = \langle C_i(t) \rangle_{i=1,2,\ldots,e|s_i=0} - \langle C_i(t) \rangle_{i=1,2,\ldots,e|s_i=1}, \tag{A.1}$$

where $\langle \cdot \rangle$ denotes the average operator. If $k_{n-2} = 0$, then the points $4P_i$ were computed by the device and a peak will be observed in $g(t)$ at $t = t_1$, where t_1 is the time when the point $4P$ is processed. Otherwise, no peak will be observed. This peak is sometimes called a *differential signal*.

Suppose that no peak was observed, then $k_{n-2} = 1$[4]. Now to find k_{n-3}, the attacker repeats the previous procedure by computing $12P_i$ for $1 \leq i \leq e$ [Cie03], repartitioning the traces and computing $g(t)$ where s_i is now a decision function of $12P$. If $k_{n-3} = 0$, then $12P$ is computed by the device, otherwise, $12P$ is never computed. We can conclude that, if the bits recovered by the attacker are $k_{n-1 \to t} = (k_{n-1}, k_{n-2}, \ldots, k_t)$, then the attacker recovers the bit k_{t-1} by computing the intermediate point $4k_{n-1 \to t}P$ and repeating the partitioning procedure. This point would be computed during the execution of the algorithm only if $k_{t-1} = 0$.

Now we consider the case where the attacked algorithm uses precomputed points, that is the key is represented in w-bit digits $k = (K_{d-1}, \ldots, K_1, K_0)_{2^w}$, where $d = \lceil n/w \rceil$. let

[3] Gebotys *et al.* [GHT05] found that the most significant bit of the x-coordinate was the best choice.

[4] To be certain the attacker can repeat the procedure with $6P_i$.

$K_{b\to a}$ denote $\lfloor (k \mod 2^{bw+1})/2^{aw} \rfloor$ or, simply, the w-bit digits of k from digit position b down to digit position a, with $b \geq a$. Then, the attack can be modified as follows. Assume that the attacker knows the key digits $K_{d-1\to j+1}$. For every possible guess for the key digit K_j, he computes the corresponding result in the accumulator and partitions the traces into two sets as before. The correct guess will result in a peak in $g(t)$. This is similar to the attack by Kocher *et al.* on DES [KJJ99], to reveal the 8-bit subkey of the last round. The attack gets more involved as w increases.

The decision boolean function here was a bit in the intermediate point. This function is also called *partitioning function* [MDS99a]. Other partitioning functions were proposed by Messerges *et al.* [MDS99a] when attacking DES in order to increase the peak that appears in $g(t)$. Examples of those functions are:

- d bits in the intermediate result being either all 0 or all 1. They called the DPA attack in this case *all-or-nothing d-bit DPA*.
- The Hamming weight of n bits of the intermediate result being less than $n - d$ or greater than d. Also the number of bits that were toggled, *the transition count*, can be used instead of the Hamming weight. This is referred to as *generalized d-bit DPA* attack.
- The number of address bus transitions being maximum or minimum. In DES, the address used for the S-box lookup is resulting from the XORing of the subkey with part of the ciphertext. The latter is known to the attacker. Hence, this partitioning is helpful if the target address depends on the key bits. The authors refer to the DPA attack that uses this partitioning as *address-bit DPA*. Other forms of address-bit attacks that do not use partitioning are in Section A.4.5.

In our discussion of the various DPA countermeasures, we will mention the attacks,

if any, that were reported on each one. If it is a good candidate, we will assess the performance of the proposed method, the storage it requires and the amount of protection it provides. To measure the amount of protection provided by the method, most of the authors provide the number of random choices that can be taken in the method [CQS03; IIT02]. They sometimes call the inverse of this number *attenuation ratio* (AR) [IIT02], since it means the amount by which the differential signal will be attenuated. We will mention the AR if it was calculated by the authors. If not, we will try to assess it as the inverse of the number of random choices.

A.3.1 Randomizing the Key

Randomizing the Key Representation

The following countermeasures aim at changing the key representation so that the sequence of doubling and adding operations is different for every ECSM execution. The main objective, though, is to randomize the intermediate points computed in a certain iteration among several ECSM executions that use the same secret key. This objective is not always met as we will explain.

Randomized binary signed digit representation The idea is to insert random decisions in Algorithm 2.2 when $k_i = 1$ whether to perform the point addition or not and adjust the value of the remainder of the key $\lfloor k/2^i \rfloor$ accordingly. This approach is equivalent to generating a random *binary signed digit* (BSD) representation for k with every ECSM execution of the algorithm. The BSD representation of an integer is a redundant representation with radix 2 where the digits used are $\{0, \pm 1\}$.

Two countermeasures were proposed in this direction, one by Oswald and Aigner [OA01] and the other by Ha and Moon [HM02a]. The second one can generate one of

all possible BSD representations of the key while this is not the case for the first one [EH03a]. Also, the sequence of additions and doubling (the AD sequence) generated by the first one made its cryptanalysis easier.

Drawbacks:

- Assuming that no SPA countermeasures are in place, and that the addition and doubling operations could be distinguished from their side-channel trace, the Oswald-Aigner algorithm cryptanalysis was reported in [OS02a; KW03; Wal04a].

- With the same assumptions, the Ha-Moon algorithm cryptanalysis was reported in [OH03]. The idea was that the AD sequence can reveal whether two successive digits of the representation are equal or not. Using this information the original bit sequence of the key is obtained. This is due to the fact that no matter which BSD representation is chosen for k, the intermediate points computed by a left-to-right or right-to-left binary algorithm will be one of two possible values. This is a property of the BSD representation itself.

- Even when an SPA countermeasure is employed such as in Section A.2.1, exploiting the previous property is still possible using collision attacks [FMPV04]. In a collision attack, the attacker compares two traces of, say, a point doubling, where the device computes $2 \times A$ and $2 \times B$. He is not able to discover the value of A nor B, but he is able to check whether $A = B$ or not. For each digit k'_i of the BSD representation, by detecting the collisions on—two successive—intermediate points, the attacker can subdivide the traces into three groups each corresponding to one of the digits $\{0, \pm 1\}$, but he does not need to know which digit corresponds to which group. If the number of traces in one group is more than half the traces, then $k'_i = 0$ and $k_i = k_{i+1}$, otherwise, if the traces are fairly distributed among the three groups,

then $k_i \neq k_{i+1}$.

Window method with random window size Liardet and Smart [LS01] proposed a DPA countermeasure where the idea is to recode k as in window method, but the window size is randomly selected from the range $[1, R]$ for each recoded digit, and negative digits are included in the digit set. The key k is first recoded from right to left into pairs (b_i, e_i) where b_i is the digit and e_i is the window size. This recoded key is then used in the window method scalar multiplication where the number of precomputed points is 2^{R-2}.
Drawbacks: The number of consecutive point doubling operations depends on the window size. Hence, an attack on this method was presented by Walter in [Wal02a] exploiting the irregularity of the AD sequence. With 18 collected traces, the computational effort to recover a 192-bit key is $O(2^{10})$.

Another similar algorithm was proposed by Ahn *et al.* [AHLM03] where $R = 3$, with the following probabilities: $p(w = 3) = \frac{1}{2}$, $p(w = 3) = \frac{1}{3}$ and $p(w = 3) = \frac{1}{6}$. The digit set is $\{0, 1, \ldots, 7\}$. In order to make the method resistant to SPA attacks and to the afore mentioned attack by Walter, point doubling is performed 3 times in every iteration irrespective of the window size.

Drawbacks:

- The presence of dummy doubling operations make the method susceptible to safe-error attack as before. If the dummy operations are identified, then Walter's attack [Wal02a] becomes applicable.
- Performance penalty. According to the authors, for $n = 160$, the algorithm's ex-

pected running time is

$$(1.29n + 3)D + (0.43n + 3)A$$

We should note that in both methods the number of additions is not constant, which may not directly reveal any information, but may not be desirable.

Overlapping window method (O-WM) This method and the following two were proposed by Itoh *et al.* [IYTT02]. In this method, a window width, w, and an overlapping between windows, h, are selected. The overlapping between windows can vary throughout the key recoding but the authors advise to fix it in order to resist SPA attacks. Also they advise to choose $h \geq \frac{w}{2}$ to prevent a bias in the distribution of the windows.

The exponent windows are recoded from left to right and the digit set is $\{0, 1, \ldots, 2^w - 1\}$. In the recoding procedure, the value of the current window is replaced by another random value and their difference is added the next window—which is overlapping the current window by h bits. Hence, the difference between the original value and the new value must be in the range $[0, 2^h - 1]$. If the overlapping is small, the range of the random values is small. Otherwise, if it is large, the number of recoded windows, and hence, the number of point additions is large. The precomputation amount is the same as the ordinary window method.

Storage: 2^{w-1} points.

Precomputation: $(w - 1)\ D + (2^w - w - 1)\ A$.

Running time: $(n - 1)\ D + \frac{n}{w-h}\ A$.

Randomized table window method (RT-WM) In this method, a random b-bit positive integer r is subtracted from all overlapping windows. The overlapping amount

is $b-w$. The precomputed points are $P[i] = (i \times 2^b + r)\ P$. The precomputation amount involves more additions than ordinary window methods and needs to be performed at the beginning of every ECSM execution as well as the key recoding.

Storage: 2^{w-1} points.

Precomputation: $b\ D + (2^w - 1)\ A$ (this count differs from the authors' count.)

Running time: $(n-b)\ D + \frac{n}{w}\ A$.

Hybrid randomization window method (HR-WM) This method combines the previous two methods. There is no constraint on the overlapping amount h.

The authors analyzed the AR for ECC with $n = 160$:

For affine coordinates, the authors suggest using RT-WM with AR $= 2^{-b}$ (for $b = 20$, 279 additions are needed).

In the case of projective coordinates, assume that cP is an intermediate point that was computed twice by the device, but the windows that were used in the first computation contain different values than those that were used in the second one. In this case, it is very unlikely that the point will be internally represented with the same coordinates in both cases. Therefore the authors recommend the O-WM in this case with AR $= 2^{-hn/(w-h)}$ when h is fixed.

Self-Randomized Exponentiation This algorithm was presented by Chevallier-Mames [CM04] to protect RSA-like systems. It can be also applied to ECC keys. Let $k^{(i_j)} = k_{n-1 \to i_j} = (k_{n-1} \dots k_{i_j})_2, 1 \le j \le t$ and $i_1 > i_2 > \cdots > i_t$. Then, we have

$$\begin{aligned} kP &= k^{(0)}P, \\ &= (k^{(0)} - k^{(i_1)})P + k^{(i_1)}P, \\ &= (k^{(0)} - k^{(i_1)} - k^{(i_2)})P + k^{(i_1)}P + k^{(i_2)}P, \end{aligned}$$

$$= \ldots$$
$$= (k^{(0)} - k^{(i_1)} - k^{(i_2)} - \cdots - k^{(i_t)})P + k^{(i_1)}P + k^{(i_2)}P + \cdots + k^{(i_t)}P.$$

That is, as the key bits are processed from left to right in the ECSM algorithm, at a randomly chosen iteration i_j, the value of the upper bits of the key, $k_{n-1\to i_j}$ is subtracted from $k_{i_j-1\to 0}$, *i.e.*, from the lower bits. The current point in the accumulator $Q[0] = k_{n-1\to i_j}P$ is added to another register, say, $Q[2]$ (initialized with $\mathcal{O}$). This process is repeated as long as $k_{i_j-1\to 0} \geq k_{n-1\to i_j}$. At the end, $Q[2]$ is added to $Q[0]$.

The previous idea is Algorithm I by the authors, it randomizes only the lower half of the key bits. They improve it in Algorithm II by subtracting $k_{n-1\to i_j}$ from the adjacent block of bits of the same length $c = n - i_j$, that is perform $k_{i_j-1\to i_j-c} - k_{n-1\to i_j}$ if it will not yield a negative value. In this case, at iteration i_j, the accumulator point $Q[0] = k_{n-1\to i_j}P$ is stored in $Q[2]$ which is added back to $Q[0]$ at iteration $i_j - c$. In Algorithm III, the length of the adjacent block c can be chosen at random from the range $[n - i_j, i_j]$.

Because of the extra addition that occur occasionally in the three algorithms, the authors show how to combine their algorithms with the side-channel atomicity concept. This is based on considering the multiplication and the squaring the same operation in RSA systems (multiplicative groups). This means that to use this concept, the adding and doubling in ECC should be indistinguishable from a side-channel point of view.

According to the authors, in Algorithms II and III, the maximum number of extra point additions will be $\log_2 n$. They also conclude that the number of possible randomizations (AR^{-1}) is $\binom{n/2}{\log_2 n}$. For example, for $n = 160$, $\mathrm{AR} = 2^{-28}$ and for $n = 256$, $\mathrm{AR} = 2^{-33}$.

The MIST Algorithm This algorithm was presented by Walter [Wal02b] to protect RSA-like systems, but can be adapted to any group-based cryptographic system. The basic idea is to recode the key from right to left with a mixed radix number system, where, at each recoding step, the radix is chosen at random. The possible radices are 2, 3 and 5 and they are selected according to a non-uniform distribution. At the same time, small addition chains are selected for the different part of the computation which are composed from similar sub-blocks of group operations.

Drawbacks:

The security analysis of the algorithm was also presented by Walter [Wal02c]: if point addition and point doubling are distinguishable, then the key search space is about $2^{3n/5}$. If operand reuse can be detected, the key search space is about $2^{n/3}$. Another adaptive chosen ciphertext attack by Sim *et al.* [SPL04] decreases the search space to $2^{0.0756n}$.

Randomizing the Order of the Key Bits

Random Starting Point This method was proposed by Messerges *et al.* [MDS99b]. The idea is to randomly choose a bit position j in the key. Then, the ECSM is executed from right-to-left starting with the bit k_j and ending at the most significant bit k_{n-1}. Using the result of this calculation as initial point in the accumulator, the ECSM is then executed from left-to-right starting at the bit k_{j-1} and ending at the least significant bit k_0. In this method $AR = 1/n$, which may not be small enough for ECCs.

Random Permutation of the Key Bits This method was suggested by Trichina and Bellezza [CQS03], if the device is not memory constrained. The idea is to precompute the points $2^i P,\ 0 \leq i < n$. Before each ECSM execution, the key bits are permuted, by recursive split-and-rotate method, along with the indices of the precomputed points. We

note that AR $= 1/n!$ if it is a pure permutation.

Randomizing the Key Value

This approach is also called *key blinding* technique. In the context of ECC cryptosystems, this countermeasure is referred to as Coron's first countermeasure [Cor99]. Before every scalar multiplication operation, the key is blinded as follows

$$k' = k + r\#E(\mathbb{F}_p),$$

where r is a random random number of length, say, 20 bits, as suggested by Coron. Hence, AR $= 2^{-20}$

Drawbacks:

- It was found by Okeya and Sakurai [OS00] that the bits of k' exhibit some bias that depends on the original bits of k which is a leakage of information.
- It was observed by Ciet [Cie03, Section VIII.6.2] that a part of the bits of the secret scalar may still be in the clear after the randomization. This is due to the following observation. Elliptic curves defined over $\mathbb{F}_p$ that are recommended in the standards have p a generalized Mersenne prime, that is, a prime of the form $2^n \pm 2^l \pm 1$, where l is relatively small, for efficiency reasons. From Hasse theorem we have $|\#E - p - 1| \leq 2\sqrt{p}$. Hence, the binary representation of $\#E(\mathbb{F}_p)$ is likely to be a 1 followed by a long run of 0s. For example, the curve "secp160k1" [SEC2] has

  $$\#E(\mathbb{F}_p) = (0100000000000000000001B8FA16DFAB9ACA16B6B3)_{16}.$$

  As for the curves recommended in [NIST], $\#E(\mathbb{F}_p)$ has long runs of 1s for most of them except for P-256, which has a run of 32 0s. However, many of the binary curves orders satisfy this criteria.

- Performance penalty since the length of the key increases by the number of bits of r.

Key Splitting

For details on this method, we refer the reader to Chapter 6.

A.3.2 Randomizing the Base Point

The countermeasures proposed in this section focus on directly randomizing the intermediate points by randomizing the base point that is input to the ECSM algorithm or the point computed in the first iteration. We present them along with their applicable attacks. We will see that most of the attacks are not possible if the key is randomized. All methods in this section have AR $= 2^{-n}$, unless otherwise stated.

Random Projective Coordinates (RPC)

This countermeasure is known as Coron's third countermeasure [Cor99]. It relies on the idea that the representation of a point using projective coordinates is redundant. That is, in standard (homogeneous) coordinates,

$$(X, Y, Z) = (rX, rY, rZ),$$

for any $r \in \mathbb{F}_p$, $r \neq 0$. The randomization requires $3M$. In Jacobian projective coordinates,

$$(X, Y, Z) = (r^2X, r^3Y, rZ),$$

for any $r \in \mathbb{F}_p$, $r \neq 0$. The randomization requires $4M + 1S$. Hence, the base point P can be stored in projective coordinates and randomized in this manner before every ECSM execution. Better yet, as was noted by Ciet and Joye in [CJ03], the point P can be kept

in affine coordinate—this is equivalent to saying that its Z-coordinate remains equal to 1—and randomize the coordinates of the first intermediate point computed by an ECSM. This first intermediate point is $2P$ if a ECSM method with no precomputation is used. They call this method $2P^*$. This enables the use of mixed Jacobian-affine coordinates in the addition operation. The same observation was made by Izu etal [IMT02] when using Montgomery ladder.

Random Elliptic Curve Isomorphisms (RC)

The idea of this countermeasure is to transfer the base point $P = (x, y) \in E$ to a randomly selected isomorphic curve E', execute the ECSM on that curve and bring the result $Q' = (x_k, y_k)$ back to the original curve. The transferred point is $P' = (r^2x, r^3y)$ and the parameters of the curve E' are $a' = r^4a$ and $b' = r^6b$ (b' is not needed in the ECSM algorithm). To bring the result to the original curve we compute $Q = (x_k/r^2, y_k/r^3)$. The randomization takes $4M + 2S$ at the beginning and $1I + 3M + 1S$ at the end.

Drawbacks: Goubin-type attacks (GRA and ZPA) An attack by Goubin [Gou03] was mounted on both this countermeasure and the previous one when combined with either the double-and-add-always or the Montgomery algorithms (Algorithms A.4 and A.6, respectively). This attack is referred to in the literature as *Goubin's refined attack* (GRA). It is an adaptive chosen ciphertext attack where the attacker chooses as input to the device a multiple of a *special* point $P_0 \in E$. P_0 has one of its coordinates equal to 0, hence, the randomization methods proposed do not affect this coordinate. The attack discovers the bits of the key starting from the most significant bit as follows.

Assume that the attacker knows the most significant bits of the key $k_{n-1 \to j}$, and he is trying to know k_{j-1}. In both algorithms, at iteration $i = j - 2$, the point $(4k_{n-1 \to j} + 1)P$

will be computed by ECADD if $k_{j-1} = 0$ and $(4k_{n-1\to j} + 3)P$ will be computed by ECADD if $k_{j-1} = 1$. Let $c_0 = 4k_{n-1\to j} + 1$ and $c_1 = 4k_{n-1\to j} + 3$. The attacker guesses that $k_{j-1} = 0$ and computes the point

$$P = c_0^{-1} P_0$$

He inputs P to the device e times and collects the corresponding traces $C_i(t)$, $1 \leq i \leq e$. He then computes the mean of the traces

$$\mathcal{M}(t) = \frac{1}{e} \sum_{i=1}^{e} C_i(t).$$

If his guess is incorrect, then $\mathcal{M}(t) \simeq 0$, since the intermediate points are correctly randomized. But if his guess is correct, there will be noticeable peaks in $\mathcal{M}(t)$ at the time corresponding to the iteration $i = j - 2$ due to the processing of the zero coordinate in the device.

The points $(x, 0)$ have order 2 and, hence, do not exist on the curves recommended in the standards. However, the point $(0, y)$ was found on four of the curves recommended in [NIST] but not on the P-224 curve. This point was also found on curves recommended in other standards (WTLS, ANSI X9.62, draft of ISO/IEC 15946-4).

Another similar attack was reported by Akishita and Takagi [AT03]. The attack is referred to as the *zero point attack* (ZPA). The difference from the GRA attack is that the set of special points includes points that cause zero values in the intermediate results of the field operations performed in the ECDBL or the ECADD. For the ECDBL, the points with 0 coordinates that were discussed by Goubin belong to this set. If the binary algorithm is used with the Jacobian coordinates doubling formula, this set includes also the points that satisfy the conditions $3x^2 + a = 0$ or $5x^4 + 2ax^2 - 4bx + a^2 = 0$. Otherwise, if the Montgomery ladder [BJ02; IT02] is used with the x-coordinate only formulas, then the set includes the points that satisfy the conditions $x^2 - a = 0$ and $x^2 + a = 0$.

Those points that result in 0 in the intermediate values computed as part of the ECADD at an iteration $i = j$ depend on $k_{n-1\rightarrow j}$ such as those discussed in Section A.2.2 and can only be found for small $k_{n-1\rightarrow j}$.

We should note that these attacks are not applicable if the key is randomized.

Randomized Linearly-transformed Coordinates (RLC)

This countermeasure proposed by Itoh *et al.* [IIT04] is based on offsetting the coordinates of the points on the curve and accounting for this offset in the addition and doubling formulas. They gave an example on how to do that for the x-coordinate using the addition and doubling formulas of the Jacobian coordinate. This offset, combined with RPC, is chosen at random with every execution. By doing that, the zero coordinates of the special points of Goubin's attack will not be computed in the device and will not be noticeable from the side-channel trace. In their example formulas, AR $= 2^{-2n}$.

Drawbacks:

Performance penalty; their example formulas have the following cost for addition and doubling:

- When combined with Algorithm A.4, left-to-right double-and-add-always, $A = 14M + 4S$ and $D = 8M + 7S$. The cost for doubling is for arbitrary curve parameter a. We believe that it can be optimized for $a = -3$.
- When combined with Algorithm A.5, right-to-left double-and-add-always, $A = 13M + 4S$ and $D = 4M + 4S$.
- When combined with Algorithm A.6, Montgomery ladder, with x-coordinate combined adding and doubling formulas, the cost is $20M+5S$ per iteration for arbitrary a ($1M$ can be saved if $a = -3$).

Random Initial Point (RIP)

This countermeasure was proposed by Itoh *et al.* [IIT04] in order to thwart Goubin-type attacks. It is based on using Algorithm A.5 with the difference that $Q[0]$ is initialized to a random point R rather than $\mathcal{O}$, then R is subtracted at the end. The authors mention that the selection of a new point R before every ECSM execution may require some time. Alternatively, they suggest that some point R_0 be fixed and stored in the device and randomize its projective coordinates using RPC as in Section A.3.2 with every execution.

Binary Expansion with RIP (BRIP, EBRIP, WBRIP)

This countermeasure, proposed by Mamiya *et al.* [MMM04], applies the random initial point idea to the left-to-right algorithms. The basic countermeasure uses Shamir-Strauss method (Sec. 2.1.3) to compute

$$R + kP = (1\underbrace{\overline{1}1\ldots\overline{1}}_{n})_2\, R + (k_{n-1}\ldots k_0)_2\, P.$$

The accumulator is initialized with R and R is subtracted from it at the end.

The authors combine this idea with the comb method (Sec.A.1), which they call *extended-binary-based method with RIP* (EBRIP). In this case, R is subtracted from all precomputed points, the accumulator is initialized with R and R is subtracted from it at the end. If k is written in a $w \times d$ matrix as in Section A.1, where $d = \lceil n/w \rceil$, then we have

Storage: 2^w points.

Precomputation: $(w-1)d\ D + 2^w\ A$.

Running time: $d\ D + d\ A$.

The authors show also how to combine the basic idea with the window method (Sec.2.1.3) (WBRIP). In this case, the point $(2^w-1)R$ is subtracted from all precomputed points. Let $d = \lceil n/w \rceil$, then

Storage: 2^w points.

Precomputation: $w\ D + 2^w\ A$.

Running time: $n\ D + d\ A$.

Hence, the difference in performance, is w more doubling in the window method.

In [YLMH05], Yen *et al.* have shown that their attack—which we discussed in Section A.2.1—is applicable to the BRIP method for curves that have a point P_0 of order 2. This is because the intermediate point computed at iteration i is $-R$ if $k_i = 0$ or $P_0 - R$ if $k_i = 1$. And as we noted before, this attack, when mounted on the EBRIP or WBRIP method will reveal $\lceil n/w \rceil$ bits.

Blinding the Base Point

This method is referred to as Coron's second countermeasure [Cor99]. It was proposed before the Goubin-type and Yen's attacks were published, but they do not apply to it. The idea is to compute $kP = k(P + R) - S$, where $S = kR$. Coron suggested that the points R and S be initially stored inside the device and refreshed before each new ECSM execution as $R \leftarrow (-1)^b 2R$ and $S \leftarrow (-1)^b 2S$, where b is a random bit.

Drawbacks: Doubling Attacks There were two similar attacks reported on this method when combined with Algorithm A.4. One of them is the doubling attack [FV03] as in Section A.2.1. The other one is by Okeya and Sakurai [OS00]. Both attacks exploit the doubling operation that is characteristic of binary algorithms. The input points in the second attack are P, $2P$, ... $2^{e-1}P$ (the first attack uses only P and $2P$ as inputs).

Since the random point R is doubled as well—with a positive sign half the times—before every execution, the correlation between the doubling trace in iteration i of computing kP and iteration $i+1$ of computing $k(2P)$ can reveal the value of the bit k_i.

To avoid these attacks, Avanzi [Ava05] suggested that a set of secret pairs (R_i, S_i) with $S_i = kR_i$ be stored in the device at initialization. Then, before every ECSM execution, both elements of a randomly chosen pair are multiplied by the same small signed scalar and added to the respective elements of another pair. The result is then used to blind the base point.

Also we suggest that the random points be refreshed as $R \leftarrow (-1)^b 3R$ and $S \leftarrow (-1)^b 3S$. Then, there is no sequence of input points that can reveal a correlation between their respective shifted traces.

Point Blinding Combined with Shamir-Strauss method

To thwart Goubin-type and Yen's attacks, Kim *et al.* [KHM+05] proposed to compute kP as

$$kP + \#ER = k(P + R) + sR, \tag{A.2}$$

where $\#E$ is the order of P, R is a random point and $s = \#E - k$. They suggest computing the right-hand side using the Shamir-Strauss method (Algorithm 2.3).

We noticed that the countermeasure contains the same blinding idea as Coron's second countermeasure, the only difference is the way to execute the ECSM. This is obvious since adding sR is mathematically the same as subtracting kR.

Also, based on our reasoning in Lemma 6.1 concerning the additive splitting using subtraction when combined with a multiple point multiplication method, we notice the following. Let $\#E = (e_{n-1} \dots e_0)_2$. Then, $e_{n-1 \to j} = k_{n-1 \to j} - b_j$, where $b_j \in \{0, 1\}$.

Hence, at iteration $i = j$, the intermediate point computed is $k_{n-1\to j}P + (e_{n-1\to j} + b_j)R$. Note that since k and e do not change across the ECSM executions, b_j will also depend only on j and the actual values of k and e. Therefore, it is arguable whether computing the right-hand side of (A.2) using the Shamir-Strauss method is different from computing the left-hand side with the same method when considering the value of intermediate points. The actual difference is in the precomputed points (P, R and $P + R$ for the left-hand side and $P + R$, R and $P + 2R$ for the right-hand side). That is, for the left-hand side, the constant point P will be accessed whenever $k_i = 1$ and $e_i = 0$ which may contribute to detecting operand reuse across ECSM executions as in Section A.4.4.

Yao's method with Random Initialization

In order to make Yao's method (cf. Algorithm A.1) resistant to both SPA and DPA attacks, Möller [Möl02] proposed the following modifications:

- In Step 1 initialize the points $Q_2, \ldots Q_\beta$ to random points rather than the point $\mathcal{O}$, and set $Q_1 \leftarrow -\sum_{j=2}^{\beta} jQ_j$. The author provides an algorithm to compute Q_1 with one doubling and $3\beta - 6$ addition. For efficiency reasons, those points can be chosen once and randomized with RPC (Section A.3.2) before every execution. This makes the AR $= 2^{n(\beta-1)}$.

- Use a digit set that does not contain 0 to represent the key. This is instead of using a dummy variable Q_0 and include it in Step 2.1, since it is prone to safe-error attack (Section A.2.1).

Drawbacks:

There are β RPC operations and β extra addition operations that are added to the

original cost of the algorithm. However, the author proves that, for $w \leq 4$, this algorithm outperforms his proposed window method [IMT02] combined with RPC after each addition (cf. Sections A.2.1 and A.4.3).

A.4 Variants of SPA and DPA Attacks

In this section, we will discuss other forms of SPA and DPA attacks.

A.4.1 Sommer's SPA attack

In [MS00], Sommer provided an experimental evidence that the Hamming weight of secret data can be found from a *single* power trace of a smart card. She studied an algorithm written in the assembly language of the 8-bit processor. The algorithm consisted of a loop, where in each iteration only the data value was changed and was then moved from the accumulator to an internal register or written to output ports. She collected traces of the power consumption of the card where the sampling rate was 50 samples per card clock cycle. Then, for every sample k, she computed the Pearson correlation factor

$$r_k = \frac{\sum_j (v_k(j) - \overline{v_k})(H(j) - \overline{H})}{\sqrt{\sum_j ((v_k(j) - \overline{v_k})^2 (H(j) - \overline{H})^2}}, \tag{A.3}$$

where $v_k(j)$ is the voltage measured over the probing resistor at the moment of the k-th sample during execution of the loop with data argument $j \in [0, 255]$, $\overline{v_k}$ is the average of $v_k(j)$ over all j, $H(j)$ is the Hamming weight of j and $\overline{H}$ is the average Hamming weight. The highest r_k indicated the moment where the correlation between the power consumption and the Hamming weight of the processed data is maximum. This interesting moment occurred during the instruction that moved the data from the accumulator to an internal register. At this moment, the values $v_k(j)$ varied almost

linearly with $H(j)$. The noise was not significant since the device was operated at a sufficiently low frequency and a high supply voltage. Hence, the author could cluster the measured $v_k(j)$ into nine clusters each corresponding to a specific Hamming weight. The author obtained similar results when studying $\Delta H(j) = H(j \oplus (j-1))$ which is the difference in Hamming weight between the data $j-1$ and j. Note that the transition in Hamming weight did not occur in two successive clock cycles, which may be the reason for which Avanzi [Ava05] considers this attack a second-order DPA attack as will be explained in Section A.4.3, though only a single power trace is used. However, for a real attacker, in order to obtain exact information about the Hamming weight of a processed data or the difference in Hamming weight between this data and a previous one from a single trace, the challenge is to find that appropriate sample where the correlation is maximal. This means that he may need to know the characteristics of the device a priori and to compute its correlation factors for known data and known algorithm.

A.4.2 Messerges DPA Attacks

Messerges *et al.* [MDS99b] have mounted the following DPA-like attacks on smart cards running RSA exponentiation algorithms. Their attacks were based on monitoring the power consumption leakage. They were mounted on left-to-right or right-to-left exponentiation algorithms that processed the key one bit at a time, such as Algorithms 2.1 and 2.2, respectively with the doubling replaced by squaring and the addition replaced by multiplication.

Single-exponent, multiple-data attack (SEMD) In this attack, it is assumed that the attacker can make the card exponentiate several random inputs with the secret key and with another known key. The attacker collects the power consumption traces of the

exponentiations that use the secret key and computes their average. He repeats this procedure again but with the known key—$(111\ldots1)_2$ in his attack. He then subtracts the two averaged signals. The resulting signal will show spikes in the iterations where the bits of the two keys differ. The portions of the averaged signals that are data dependent or where the bits of the exponents agree will approach 0.

Messerges' countermeasure to this attack is to blind the exponent before every exponentiation as in Section A.3.1.

Multiple-exponent, single-data attack (MESD) In this attack it is assumed that the card will exponentiate the same input, not necessarily known to the attacker, with several keys of the attacker's choice. The attacker first collects the average power trace of the exponentiation of the input with the secret key.

Now, assuming that the attacker knows the first $j-1$—most significant or least significant, depending on the algorithm—bits of the key, he wants to attack the jth bit. He guesses that this bit is 1 and sets a new key equal to the bits that he knows concatenated with the guessed bit and arbitrary value for the remaining bits. He asks the card to exponentiate several times the constant input with that key and collects the average power trace. He repeats this step with the guessed bit reset to 0. He subtracts each of the collected averaged traces from the original one. For the correct guess, the resulting trace will approach zero through all iterations including iteration j, but for the wrong guess, the resulting trace will depict differences in iteration j.

Zero-exponent, multiple-data attack (ZEMD) This attack is the same DPA attack as explained in Section A.3. Hence, unlike the SEMD and the MESD attacks, it assumes that the attacker knows how the exponentiation algorithm is performed and can simulate

it to compute intermediate points.

The intermediate point computed by Messerges is the result of the multiplication in the iteration processing the guessed bit rather than the result of the squaring of the next iteration as was done by Coron (Section A.3). This is because his attack was on a square-and-multiply algorithm where the multiplication was performed only when the current bit of the key is 1. But Coron's attack was on a double-and-add-always algorithm where the addition was performed regardless of the value of the current bit and its result collected if that bit was 1. Hence, the result of the doubling of the next iteration can reveal the validity of the guess.

Also Messerges used as a partitioning function the Hamming weight of some byte in the intermediate result being 8 or 0 (all-or-nothing 8-bit DPA, cf. Section A.3).

Messerges' countermeasure to the MESD and the ZEMD attacks is to blind the input before every exponentiation and unblind it at the end, as in Section A.3.2. He also suggested the *random starting point* countermeasure that was mentioned in Section A.3.1.

Attacking the ECDSA Algorithm A.8 shows the steps of the signature generation part of the elliptic curve digital signature algorithm (ECDSA). In the algorithm, a curve E with public parameters is agreed upon, $P \in E$ is a public base point of order g, d is the private key of the signer where $0 < d < g$ and the function $H(x)$ is a cryptographic hash function.

Algorithm A.8. Signature Generation in the ECDSA

INPUT: d, P and a message M.

OUTPUT: (r, s), a signature on M.

1. $m \leftarrow H(M)$.
2. $k \leftarrow_R [1, g-1]$. //$\leftarrow_R$: choosing a random integer from that range.

3. $Q \leftarrow kP$.
4. $r \leftarrow x(Q) \bmod g$.
5. $s \leftarrow k^{-1}(m + dr) \bmod g$.
6. Return(r, s).

If the attacker can reveal k, then the signer's private key d is compromised. However, since k is randomly chosen with every signature generation, it is hard to attack Step 3 of the algorithm using DPA (assuming an SPA countermeasure is employed). Nevertheless, Messerges [Mes00] has shown how to mount a DPA (ZEMD) attack on Step 5.

The attacker runs Algorithm A.8 several times and collects the corresponding power traces and values of r. He does not need to change or know the input. Then, he starts guessing the bits of d one by one starting from the least significant. For a certain guess for the bit d_j, assuming that the least significant bits of d up to the $j-1$st bit were correctly revealed, the attacker computes the intermediate value $X = dr$ and partitions the traces into two sets according to the bit X_j being 0 or 1. This is possible since the attacker knows r for every execution of the algorithm. If the attacker's guess is correct, the actual values of the least significant bits of X up to the bit X_j will be equal to those computed by the attacker for each value of r and the partitioning would be correctly carried.

As a countermeasure to this attack, Messerges suggested random masking of both m and d, before every signature generation. This is done by selecting an integer ω at random from the range $[0, g-1]$ and multiplying it by each of m and d. At the end, the result of Step 5, s, is multiplied by ω^{-1}.

A.4.3 Second-Order DPA Attack

The definition of a high-order DPA attack was first presented by Kocher *et al.* [KJJ99] as a DPA attack that combines multiple samples from within a trace. The following is the definition according to Messerges [Mes00]:

An n^{th}-order DPA attack makes use of n different samples in the power consumption signal that correspond to n different intermediate values calculated during the execution of an algorithm.

The power leakage model proposed, and experimentally verified, by Messerges [Mes00] assumes that the power consumption of an instruction, $\mathscr{C}(t)$, varies linearly with the Hamming weight $H(\omega)$ of the data ω processed by this instruction at time t. That is

$$\mathscr{C}(t) = \epsilon H(\omega) + l, \tag{A.4}$$

where ϵ and l are some hardware-dependent constants.

Messerges presented a second-order DPA attack on a typical algorithm of a public key cryptosystem. He chose two instructions of the algorithm that are not necessarily consecutive to monitor their power consumption every time the algorithm is executed. The first instruction processes random data and the second one processes—a part of—the secret key XORed with input data XORed with the random data of the first instruction.

To reveal bit j of the secret key, the attacker sets bit j of the input data to 0 and the other ones to random values and gathers the power consumption traces of both instructions with different input data. He repeats the same procedure but with setting bit j of the input data to 1. By averaging the difference in power consumption between these two instructions for both the 0 and 1 set of traces, the attacker could reveal the value of the key bits. The attack here is based on the difference in the average Hamming weight of the input data in the two cases. Messerges has experimentally shown the validity of

his attack.

The concept of this attack was analyzed further by Joye *et al.* in [JPS05]. They define $\omega = I(x, s)$, that is ω is an intermediate value that depends on some known input data x and a small portion of secret data s, *i.e.*, all possible values $\hat{s}$ of s can be exhausted to compute ω. $g(\omega)$ is some bit in ω chosen by the attacker to perform the partitioning of x into one of two sets $\mathfrak{G}_0(\hat{s})$ or $\mathfrak{G}_1(\hat{s})$. That is,

$$\mathfrak{G}_b(\hat{s}) = \{x \mid g(I(x, \hat{s})) = b\} \text{ for } b \in \{0, 1\}.$$

Now let τ_1 be the time when a random data ρ is manipulated and τ_2 be the time when a data $v = f(\omega, \rho)$ is manipulated, *e.g.*, $v = \omega \oplus \rho$. If the attacker knows those two time periods, he can evaluate

$$\overline{\Delta_2}(\hat{s}) = \langle|\mathscr{C}(\tau_2) - \mathscr{C}(\tau_1)|\rangle_{x\in\mathfrak{G}_1(\hat{s})} - \langle|\mathscr{C}(\tau_2) - \mathscr{C}(\tau_1)|\rangle_{x\in\mathfrak{G}_0(\hat{s})}, \tag{A.5}$$

where $\langle\cdot\rangle$ denotes the average operator[5]. Under the power leakage model (A.4), if the average of the absolute difference in Hamming weight between v and ρ—for all possible values ω and ρ— is different when $g(\omega) = 1$ from when $g(\omega) = 0$, then the value $\hat{s}$ for which $\overline{\Delta_2}(\hat{s})$ is maximal is likely $\hat{s} = s$.

If the attacker does not know exactly τ_1 and τ_2 but knows the offset $\delta = \tau_2 - \tau_1$ he can extend (A.5) as follows

$$\Delta_2(\hat{s}, t) = \langle|\mathscr{C}(t + \delta) - \mathscr{C}(t)|\rangle_{x\in\mathfrak{G}_1(\hat{s})} - \langle|\mathscr{C}(t + \delta) - \mathscr{C}(t)|\rangle_{x\in\mathfrak{G}_0(\hat{s})}. \tag{A.6}$$

The authors have derived the peak values for $\overline{\Delta_2}$ for $v = \omega \oplus \rho$ and showed how to maximize it by raising the difference in power consumption to the 3rd power. They also

[5]With the same notation, the first order DPA trace is evaluated as

$$\Delta_1(\hat{s}, t) = \langle\mathscr{C}(t)\rangle_{x\in\mathfrak{G}_1(\hat{s})} - \langle\mathscr{C}(t)\rangle_{x\in\mathfrak{G}_0(\hat{s})},$$

which yields the same trace as (A.1), for the guess of the current key bit $\hat{s} = 0$.

extended their analysis to other logical binary functions such as $v = \omega \wedge \rho$ and to other power leakage models such as the Hamming distance model. Their experimental results on an implementation of RC6 that is resistant to first-order DPA attacks confirmed their analysis. The practicality of the attack decreases as the bit size of the word manipulated by the algorithm at a time increases.

A.4.4 Operand Reuse DPA Attack

Another form of a second-order DPA attack was presented by Okeya and Sakurai [OS02b] on the SPA-resistant method by Moller [Möl01] (Section A.2.1) with window size $w = 4$. In order to be DPA-resistant, the precomputed points of the algorithm are represented in projective coordinates and the representation of the input point is randomized at the beginning of the precomputation step for every ECSM execution.

Their analysis shows that by making use of power consumption traces of two different iterations, they can detect operand (precomputed point) reuse and, hence, digit repetition in the key.

They use the same power leakage model proposed by Messerges. However, they calculate the variance—rather than the mean—of the difference in power consumption in two iterations. They prove that this variance will take value v_1 if the precomputed point that was loaded on the bus is the same in both iterations and v_2 otherwise with $v_1 < v_2$.

For each iteration, they compute the variance with all the following iterations to find which ones result in the value v_1. This reveals the reuse of precomputed points, and, hence, reduces the search space of the key. Note that this attack is neither a chosen- nor a known-plaintext attack, since the attacker needs not to choose the base point or even know it. Even if the attacker knows the input point, he cannot benefit from this knowledge in computing intermediate points since the input point representation is randomized at

the beginning of the ECSM. However, their attack is based on that the digits (or windows) of the key are the same across executions and that many traces are obtained in order to compute the variance accurately.

Their first proposed countermeasure was to increase the window size to $w = 7$, but this incurred about 57% performance penalty. In [OT03], Okeya and Takagi suggested the following countermeasure against a second-order DPA attack. In order to prevent operand reuse detection, after each addition operation in the window method ECSM, the coordinates of the precomputed point involved should be randomized as in Section A.3.2. If the points are stored in Chudnovsky coordinates, then this would increase the cost of the Jacobian-Chudnovsky addition operation by $5M + 2S$, resulting in $16M + 5S$ per addition. If the points are stored in Jacobian coordinates, the cost of the Jacobian-Jacobian addition operation increases by $4M + 1S$, resulting in $16M + 5S$ per addition, as well. Note that the Chudnovsky coordinates store two more coordinates per point than the Jacobian coordinates. The increase in the addition cost is

$$\frac{A - A_0}{A_0},$$

where A_0 is the original cost of addition when Jacobian-affine coordinates are used. If we consider the cost of squaring $S = 0.8M$ as in [HMV04], then $A_0 = 8M + 3S = 10.4M$ and $A = 16M + 5S = 20M$. Hence, the increase in the addition cost is about 92%. The increase in the expected running time based on Section A.2.1 is

$$\frac{A - A_0}{wD + A_0},$$

where $D = 4M + 4S = 7.2M$ for Jacobian coordinates. For $w = 4$, the increase in the running time is about 24.5%.

Note that Avanzi [Ava05] considers that a second-order DPA attack can detect operand reuse from a *single* trace, especially that of an electromagnetic emanation. There-

fore, he recommends that the precomputed data be relocated at some intervals during the execution of the ECSM. This should be done in a way that prevents the attacker from determining when the same operand is transferred more times from one memory location to the other. He says that the optimal way to implement this is not clear.

A.4.5 Address-bit DPA attack

This attack exploits the fact that internal addresses of registers or memory locations are another type of data processed by the CPU. Hence, power traces of two intervals of an algorithm where the same instruction accesses different addresses will be less correlated than if that instruction was accessing the same address. The attack is easier if the number of registers or memory locations accessed during the algorithm depending on bits of the secret key is small.

Itoh *et al.* [IIT02] mounted two address-bit DPA attacks on Montgomery ladder (Algorithm A.7), where the Montgomery-form elliptic curve was used with the x-coordinate only formulas for doubling and addition. The attacks are valid with any formulas since it exploits the correlation between the key bits and addresses of registers in the steps 2.1, 2.3 and 2.4. It is assumed that input points are randomized using RPC as in Section A.3.2, but this is not necessary since the attacker inputs random points, collects their corresponding power traces, and averages them.

The first attack is similar to the SEMD attack in Section A.4.2. The attacker computes the average power trace for a known key, *e.g.*, $(111\ldots1)_2$, and that for the secret key. The difference of the two traces shows spikes where the key bits do not agree. Note that Messerges' attack was demonstrated on a square-and-multiply algorithm where the addition was performed in the iteration where the key bit was 1. Hence, his SEMD attack showed the difference between the average power consumption of squaring and

multiplying when the bits of the two exponents were different. This is not the case in Montgomery ladder since the doubling and addition are performed in every iteration.

The authors say that their second attack is a ZEMD attack, but it involves no partitioning of traces, rather the technique used is very similar to the SEMD attack. The attacker inputs random points to the algorithm, collects the corresponding power traces and computes their average. He subdivides the obtained trace into intervals each corresponding to an iteration of the algorithm. It is assumed that the most significant bit of the key is 1. By subtracting from the average trace interval corresponding to the most significant bit each of those corresponding to the other bits, the 0 bits of the secret key can be identified by the spikes appearing in the difference trace. Even when the addresses of the registers in steps 2.3 and 2.4 are swapped instead of their contents as was mentioned after Algorithm A.7, the authors showed that a more sophisticated attack is still feasible.

The last attack is applicable to ECSM algorithms that use precomputed points provided that the number of such points is small. Digit reuse in the key representation can be detected based on address reuse of the precomputed points.

In [IIT03], Itoh *et al.* proposed a *randomized addressing* countermeasure to address-bit DPA attacks. The countermeasure is based on random masking of the key bits and the addresses. The authors say that this countermeasure is not sufficient to resist data-bit DPA (the original DPA attack as explained in Section A.3). Therefore, it should be combined with RPC (Section A.3.2) or RC (Section A.3.2). The following algorithms are the authors' modified versions of Algorithms A.4 and A.7, where $r = (r_{n-1} \dots r_0)_2$ is an n-bit random integer.

Algorithm A.9. Left-to-Right Double-and-Add-Always with Randomized Addressing

---

INPUT: r, k and $P \in E(\mathbb{F}_p)$.

OUTPUT: kP.

1. $Q[2] \leftarrow \text{RPC}(P)$.
2. for i from $n-2$ down to 0 do
   - 2.1 $Q[r_{i+1}] \leftarrow 2Q[r_{i+1}]$.
   - 2.2 $Q[1-r_{i+1}] \leftarrow Q[r_{i+1}] + Q[2]$.
   - 2.3 $Q[r_i] \leftarrow Q[k_i \oplus r_{i+1}]$.
3. Return($Q[r_0]$).

Algorithm A.10. The Montgomery Ladder with Randomized Addressing

INPUT: r, k and $P \in E(\mathbb{F}_p)$.

OUTPUT: kP.

1. $Q[r_{n-1}] \leftarrow \text{RPC}(P)$; $Q[1-r_{n-1}] \leftarrow 2Q[r_{n-1}]$.
2. for i from $n-2$ down to 0 do
   - 2.1 $Q[2] \leftarrow 2Q[k_i \oplus r_{i+1}]$.
   - 2.2 $Q[1] \leftarrow Q[0] + Q[1]$.
   - 2.3 $Q[0] \leftarrow Q[2-(k_i \oplus r_i)]$.
   - 2.4 $Q[1] \leftarrow Q[1+(k_i \oplus r_i)]$.
3. Return($Q[r_0]$).

Their proposed randomized addressing for ECSM algorithms that use precomputed points was explained by Avanzi [Ava05] as follows. The access to the table containing the precomputed points is done via a base address and an index. In this case, a random integer r of the same bit length as the highest index is chosen before every execution. During both the table initialization phase and the scalar multiplication, the table index is XORed with r.

We argue that address-bit DPA attacks are not feasible if the key is randomized, since then, the address accessed in a certain iteration will be different across executions.

A.5 Conclusion

From this overview over SPA, DPA attacks and their variants we conclude the following:

- SPA attacks (not including Sommer's attack in Section A.4.1) can be prevented by making the ECSM execution uniform over all iterations, preferably with no dummy operations.

- First order DPA attacks are based on the fact that intermediate points computed by the algorithm can be guessed by the attacker. Hence, to prevent them these intermediate points should be randomized. However, randomizing the input point or the intermediate points is not sufficient to resist safe-error, doubling, Goubin-type, Yen *et al.*, operand-reuse and address-bit attacks. To resist those attacks, the key *value* should be randomized before the ECSM execution.

- If the ECSM method used involves access to precomputed values during the execution, even when the key is randomized, it is questionable whether from a single trace, an operand reuse or an address reuse can be detected. The means to detect such a reuse would probably be a collision indicator as in [FV03], which depends on the number of clock cycles in which the power consumption should be identical if the same operand or address is accessed. We argue that the clock cycles in which an address is loaded on the bus would not be enough to indicate a collision. Otherwise, we would need to implement Avanzi's suggestion [Ava05] that the location of a precomputed point should change after each access. As for the number of clock

cycles in which the operand is loaded on the data bus, because of the length of the operands, this number may be significant, in which case we would need to randomize the representation of a precomputed point after it is accessed as explained in Section A.4.4.

- Second-order DPA attacks, as presented in Section A.4.3, require the attacker to know some input to the algorithm, to be able to compute an intermediate value when this input is combined with a part of the secret key, and to detect two instants in the algorithm; one when a random data is processed and the other when this random data is combined with the intermediate value he is guessing. Based on the value of a bit in the representation of the intermediate value he is guessing, he performs the partitioning of the traces and computes the second order DPA signal by computing the average of the difference—or a higher power of that difference—of the power consumption in the two instants for each partition, then computing the difference between the two averages. A possible countermeasure would be to avoid directly combining data that is known to the attacker with secret data, that is, to make it difficult for the attacker to make a successful partitioning of the collected traces.

Appendix B

Finite Markov Chains

A *Markov chain* is a discrete-time discrete-valued random process $\{X_n; n = 0, 1, 2, \ldots\}$. The process is said to be in state i at time n if and only if $X_n = i$, where i belongs to some sample space S. The Markov property for a conditional probability mass function (pmf) is

$$P_{X_{n+1}|X_n,\ldots,X_0}(i_{n+1}|i_n, \ldots, i_0) = P_{X_{n+1}|X_n}(i_{n+1}|i_n).$$

This means that the next state depends only on the present state, but does not depend on the way in which the present state arose from previous states. This process is referred to as a *memoryless* process.

The *transition probability* is defined as

$$P_{ij} \triangleq P(X_{n+1} = j|X_n = i) = P_{X_{n+1}|X_n}(j|i); \qquad i, j \in S$$

and is said to be *stationary* since it does not depend on n.

The transition probabilities are characterized by

$$P_{ij} \geq 0,$$

$$\sum_j P_{ij} = 1.$$

They can be arranged in a matrix known as the *transition matrix*

$$\mathbf{P} = \begin{pmatrix} P_{00} & P_{01} & \dots \\ P_{10} & P_{11} & \dots \\ \vdots & \vdots & \ddots \end{pmatrix}.$$

Also the *n-step transition probability* is defined as

$$P_{ij}^n \triangleq Pr(X_n = j | X_0 = i),$$

and can be computed by multiplying the transition matrix by itself n times, *i.e.*, $\mathbf{P}^n$, and retrieving the i^{th} row and j^{th} column.

The following properties apply to an important class of finite Markov chains

- It can be *irreducible*, this means that every state is reachable from every other state in a finite number of steps.
- It can have *ergodic* states, those are *recurrent* (*i.e.*, not *transient*) *aperiodic* states. A state is said to be recurrent if it will be revisited an infinite number of times in an infinite run of the process. A state is said to be aperiodic if it has a period 1, where the period of a state is the greatest common divisor of the number of times a chain, starting from that state, has a positive probability of returning to it.

The main theorem for Markov theory states that for an irreducible ergodic Markov chain a *steady state* always exists. This is to say that every state in the chain has a

limiting probability defined by

$$\pi_j = \lim_{n\to\infty} P_{ij}^n,$$

which does not depend on i or on n.

Also the vector $\pi = (\pi_1 \ \ldots \ \pi_w)$ for a chain with w states is the unique solution of

$$\begin{aligned} \pi\mathbf{P} &= \pi, \\ \sum_{j=1}^{w} \pi_j &= 1. \end{aligned} \tag{B.1}$$

Appendix C

Grammars, Automata and Generating Functions

C.1 Grammars

A *grammar* G is a quadruple (T, N, S, P) [Gre83; HMU01], where

- T is a *terminal alphabet* (usually small letters),
- N is a *nonterminal alphabet* (usually capital letters),
- $S \in N$ is a *start* symbol,
- P is a set of *productions* (*rewrite rules*) of the form $\alpha \to \beta$, with $\alpha, \beta \in (T \cup N)^*$ and α including at least one nonterminal symbol.

When several productions have a common left hand side, they are conveniently grouped into one production using vertical bars to separate the production possibilities. For example $A \to B$ and $A \to C$ are grouped into $A \to B|C$. The symbol ϵ denotes

the null string.

A grammar is a *context-free* grammar if $\alpha \in N$, *i.e.*, the left hand side of any production is a single nonterminal symbol. A context-free grammar is a *regular* grammar if $\beta \in \{\gamma\eta | \gamma \in T^*, \eta \in N \cup \{\epsilon\}\}$, *i.e.*, the right hand side of any production is a string containing at most one nonterminal as the rightmost symbol.

A *derivation* step consists of matching a substring with the left hand string of a production and replacing it with the right hand string. A string is derived by a certain grammar if it is possible to obtain the string from the start symbol of the grammar with a finite number of derivation steps.

The set of acceptable strings derived from the productions of a grammar form a *formal language*.

C.2 Deterministic Finite Automata

A *deterministic finite automaton* (DFA) A is a quintuple $(Q, \Sigma, s, F, \delta)$ [HMU01], where

- Q is a set of *states*,
- Σ is the alphabet of *input symbols*,
- $s \in Q$ is the *initial state*,
- $F \subset Q$ is the set of *final states*,
- $\delta : Q \times \Sigma \rightarrow Q$ is the *transition function*.

DFAs are represented pictorially as directed graphs where the nodes represent the states and the arcs are labeled by one or more symbols from the alphabet Σ. The initial state is denoted by a short incoming arrow and the final (accepting) states are denoted

by a cross as in [MO90]. For every symbol $x \in \Sigma$ there is an arc labeled x emanating from every state.

Every path in the graph spells out a string over Σ. A string is recognized by an automaton if there exists a path beginning in the initial state and finishing in a final state that spells out the letters of this string by reading the labels of the arcs that form the path. The *language of the automaton* is the set of strings it recognizes.

A regular grammar G that generates the language recognized by a DFA A can be specified by defining its terminal alphabet T as the alphabet Σ of A, its nonterminal alphabet N as the set of states Q, its start symbol S as the start state s and the productions P are derived from the transition function δ as follows

$$\begin{aligned} q &\rightarrow x\delta(q,x) \quad && \forall q \in Q, \forall x \in \Sigma, \\ q &\rightarrow \epsilon && \forall q \in F. \end{aligned}$$

C.3 Analysis of Algorithms using Generating Functions

In [Gre83], Greene has investigated an interesting property of formal languages. That is the productions of a grammar can be treated as equations, with $\rightarrow$ replaced by $=$, $|$ replaced by $+$, ϵ by 1 and the terminal symbols replaced by a dummy variable z. If these equations are solved for the start symbol, the result is a generating function for the number of derivations of the grammar.

Generating functions are an essential tool of combinatorial mathematics; they are used to study sequences of numbers a_n for $n \geq 0$. The a_is are implanted as the Taylor coefficients of a function A of the dummy variable z

$$A(z) = \sum_n a_n z^n,$$

and then the properties of the sequence are studied through the properties of $A(z)$. For instance, let $[z^n]A(z)$ denote the coefficient of z^n in the generating function $A(z)$ obtained for the grammar of a certain automaton language. This coefficient represents the number of strings of length n in the language recognized by the automaton.

To compute the expected cost of an algorithm (studied in the form of an automaton), another variable u can be inserted in the equations that represent the productions to account for the cost of every transition. This is accomplished by making the exponent of this variable in every term of the equations to reflect the cost of the corresponding transition. Hence, a bivariate generating function $A(z, u)$ is obtained

$$A(z, u) = \sum_{n,m} a_{n,m} z^n u^m,$$

where $a_{n,m}$, denoted as $[z^n u^m]A(z, u)$, is the number of strings with n symbols and with cost m.

When the terminal alphabet consists of two equally probable symbols as is the case for Morain and Olivos' automata, we can obtain what is referred to as the *probability generating function* in the Bernoulli uniform case

$$A_p(z, u) = A(\frac{z}{2}, u),$$

from which we deduce the generating function for the *expectation*

$$a_p(z) = \left.\frac{\partial A_p}{\partial u}\right|_{u=1}.$$

The coefficient of z^n in the series $a_p(z)$, *i.e.*, $[z^n]a_p(z)$ is the expected cost of the processing of a string of length n. This can be found using *partial fraction decomposition* [HP] of $a_p(z)$ and Taylor expansion of each term in the resulting expression. For the automata studied in this proposal, we assumed that the most significant bit of the integer

k is 1, thus only $n-1$ bits of k are involved in the processing. Therefore the expected cost is $[z^n]2a_p(z)$. Alternatively, let

$$a(z) = \left.\frac{\partial A}{\partial u}\right|_{u=1},$$

then the expected cost is

$$\frac{1}{2^{n-1}}[z^n]a(z) = [z^n]2a(z/2),$$

where the last equality follows from the *scaling* property of generating functions [SF96]

$$A(\lambda z) = \sum_n \lambda^n a_n z^n.$$

For a deeper study of the use of generating functions in the average-case analysis of algorithms, we refer the reader to Sedgewick and Flajolet's book [SF96] and also to the series they published as INRIA research reports which form the preliminary chapters of their new book Analytic Combinatorics. These reports are available at `http://algo.inria.fr/flajolet/Publications/books.html`.

Appendix D

Examples on Chapter 5

D.1 Examples of Representations

The following tables present the different representations of the "positive" τNAFs on the curve E_0 and their number.

Table D.1: Representations of "positive" τNAFs of length 1.

τNAF k	Representations	$\vartheta(k,1)$
0	0	1
1	1, $\overline{111}$	2

Table D.2: Representations of "positive" τNAFs of length 2.

τNAF k	Representations	$\vartheta(k,2)$
0	0	1
1	1, $\overline{111}$, $101\overline{1}$	3
10	10, $\overline{111}0$	2

Table D.3: Representations of "positive" τNAFs of length 3.

τNAF k	Representations	$\vartheta(k,3)$
0	0	1
1	1, $\overline{11}1$, $111\overline{11}$, $101\overline{1}$	4
10	10, $\overline{11}10$, $101\overline{1}0$	3
$10\overline{1}$	$10\overline{1}$, $\overline{11}10\overline{1}$, $\overline{11}011$, $\overline{1}1\overline{1}1$	4
100	100, $\overline{11}100$	2
101	101, $\overline{11}101$, $\overline{11}$, $111\overline{1}$, $\overline{1}0\overline{1}1\overline{1}$	5

Table D.4: Representations of "positive" τNAFs of length 4.

τNAF k	Representations	$\vartheta(k,4)$
0	0	1
1	1, $\overline{11}1$, $111\overline{11}$, $\overline{1}0\overline{1}1\overline{11}$, $101\overline{1}$, $\overline{11}101\overline{1}$	6
10	10, $\overline{11}10$, $111\overline{11}0$, $101\overline{1}0$	4
$10\overline{1}$	$10\overline{1}$, $\overline{11}10\overline{1}$, $101\overline{1}0\overline{1}$, $\overline{11}011$, 101011, $\overline{1}1\overline{1}1$, $1111\overline{1}1$, $100\overline{11}1$	8
100	100, $\overline{11}100$, $101\overline{1}00$	3
101	101, $\overline{11}101$, $101\overline{1}01$, $\overline{11}$, $111\overline{1}$, $\overline{11}11\overline{1}$, $\overline{1}0\overline{1}1\overline{1}$	7
$10\overline{1}0$	$10\overline{1}0$, $\overline{11}10\overline{1}0$, $\overline{11}0110$, $\overline{1}1\overline{1}10$	4
$100\overline{1}$	$100\overline{1}$, $\overline{11}100\overline{1}$, 1111, $\overline{11}1111$, $\overline{1}0\overline{1}11$, $\overline{1}1$	6
1000	1000, $\overline{11}1000$	2
1001	1001, $\overline{11}1001$, $1\overline{11}1$, $\overline{111}111$, $\overline{1}001\overline{11}$, $\overline{11}001\overline{1}$	6
1010	1010, $\overline{11}1010$, $\overline{11}0$, $111\overline{1}0$, $\overline{1}0\overline{1}1\overline{1}0$	5

Table D.5: Representations of "positive" τNAFs of length 5.

τNAF k	Representations	$\vartheta(k,5)$
0	0	1
1	1, $\overline{1}\overline{1}\overline{1}$, $11\overline{1}\overline{1}$, $\overline{1}\overline{1}\overline{1}1\overline{1}\overline{1}$, $\overline{1}0\overline{1}\overline{1}\overline{1}$, $101\overline{1}$, $\overline{1}\overline{1}\overline{1}01\overline{1}$, $101\overline{1}01\overline{1}$	8
10	10, $\overline{1}\overline{1}\overline{1}0$, $111\overline{1}\overline{1}0$, $\overline{1}0\overline{1}1\overline{1}\overline{1}0$, $101\overline{1}0$, $\overline{1}\overline{1}\overline{1}01\overline{1}0$	6
$10\overline{1}$	$10\overline{1}$, $\overline{1}\overline{1}\overline{1}0\overline{1}$, $111\overline{1}\overline{1}0\overline{1}$, $101\overline{1}0\overline{1}$, $\overline{1}\overline{1}011$, $111\overline{1}011$, 101011, $\overline{1}1\overline{1}1$, $1111\overline{1}1$, $\overline{1}0\overline{1}11\overline{1}1$, $100\overline{1}\overline{1}1$	11
100	100, $\overline{1}\overline{1}\overline{1}00$, $111\overline{1}\overline{1}00$, $101\overline{1}00$	4
101	101, $\overline{1}\overline{1}\overline{1}01$, $11\overline{1}\overline{1}01$, $01\overline{1}01$,$\overline{1}\overline{1}$, $11\overline{1}$, $\overline{1}\overline{1}\overline{1}11\overline{1}$, $01\overline{1}11\overline{1}$, $\overline{1}0\overline{1}1\overline{1}$, $110\overline{1}1\overline{1}$	10
$10\overline{1}0$	$10\overline{1}0$, $\overline{1}\overline{1}\overline{1}0\overline{1}0$, $101\overline{1}0\overline{1}0$, $\overline{1}\overline{1}0110$, 1010110, $\overline{1}1\overline{1}10$, $1111\overline{1}10$, $100\overline{1}\overline{1}10$	8
$100\overline{1}$	$100\overline{1}$, $\overline{1}\overline{1}\overline{1}00\overline{1}$, $101\overline{1}00\overline{1}$, 1111, $\overline{1}\overline{1}\overline{1}111$, $101\overline{1}111$, $\overline{1}0\overline{1}11$, $1110\overline{1}11$, $\overline{1}1$	9
1000	1000, $\overline{1}\overline{1}\overline{1}000$, $101\overline{1}000$	3
1001	1001, $\overline{1}\overline{1}\overline{1}001$, $101\overline{1}001$, $1\overline{1}\overline{1}\overline{1}$, $\overline{1}\overline{1}\overline{1}\overline{1}\overline{1}\overline{1}$, $101\overline{1}\overline{1}\overline{1}\overline{1}$, $\overline{1}001\overline{1}\overline{1}$, $\overline{1}\overline{1}001\overline{1}$, $1010011\overline{1}$	9
1010	1010, $\overline{1}\overline{1}\overline{1}010$, $101\overline{1}010$, $\overline{1}\overline{1}0$, $111\overline{1}0$, $\overline{1}\overline{1}\overline{1}11\overline{1}0$, $\overline{1}0\overline{1}1\overline{1}0$	7
$10\overline{1}0\overline{1}$	$10\overline{1}0\overline{1}$, $\overline{1}\overline{1}\overline{1}0\overline{1}0\overline{1}$, $\overline{1}\overline{1}0110\overline{1}$, $\overline{1}1\overline{1}10\overline{1}$, 10011, $\overline{1}\overline{1}\overline{1}0011$, $1\overline{1}\overline{1}\overline{1}1$, $\overline{1}\overline{1}\overline{1}\overline{1}\overline{1}\overline{1}1$, $\overline{1}001\overline{1}\overline{1}1$, $\overline{1}\overline{1}001\overline{1}1$	10
$10\overline{1}00$	$10\overline{1}00$, $\overline{1}\overline{1}\overline{1}0\overline{1}00$, $\overline{1}\overline{1}01100$, $\overline{1}1\overline{1}100$	4
$10\overline{1}01$	$10\overline{1}01$, $\overline{1}\overline{1}\overline{1}0\overline{1}01$, $\overline{1}\overline{1}01101$, $\overline{1}1\overline{1}101$, $\overline{1}\overline{1}010\overline{1}\overline{1}$, $\overline{1}1\overline{1}0\overline{1}\overline{1}$, $11\overline{1}1\overline{1}$, $\overline{1}\overline{1}\overline{1}1\overline{1}1\overline{1}$, $\overline{1}0\overline{1}\overline{1}1\overline{1}$, $\overline{1}10111\overline{1}$	10
$100\overline{1}0$	$100\overline{1}0$, $\overline{1}\overline{1}\overline{1}00\overline{1}0$, 11110, $\overline{1}\overline{1}\overline{1}1110$, $\overline{1}0\overline{1}110$, $\overline{1}10$	6
$1000\overline{1}$	$1000\overline{1}$, $\overline{1}\overline{1}\overline{1}000\overline{1}$, 10111, $\overline{1}\overline{1}\overline{1}0111$, $\overline{1}\overline{1}11$, $111\overline{1}11$, $\overline{1}0\overline{1}1\overline{1}11$, $1\overline{1}0\overline{1}1$, $\overline{1}\overline{1}\overline{1}1\overline{0}\overline{1}1$, $\overline{1}0010\overline{1}1$	10
10000	10000, $\overline{1}\overline{1}\overline{1}0000$	2

τNAF k	Representations	$\vartheta(k,5)$
10001	10001, $\overline{1}\overline{1}\overline{1}0001$, $10\overline{1}\overline{1}\overline{1}$, $\overline{1}\overline{1}\overline{1}0\overline{1}\overline{1}\overline{1}$, $\overline{1}\overline{1}011\overline{1}\overline{1}$, $\overline{1}1\overline{1}1\overline{1}\overline{1}$, $1101\overline{1}$, $\overline{1}\overline{1}\overline{1}101\overline{1}$, $\overline{1}0\overline{1}01\overline{1}$	9
10010	10010, $\overline{1}\overline{1}\overline{1}0010$, $1\overline{1}\overline{1}\overline{1}0$, $\overline{1}\overline{1}\overline{1}\overline{1}\overline{1}\overline{1}0$, $\overline{1}001\overline{1}\overline{1}0$, $\overline{1}\overline{1}001\overline{1}0$	6
$1010\overline{1}$	$1010\overline{1}$, $\overline{1}\overline{1}\overline{1}010\overline{1}$, $\overline{1}\overline{1}0\overline{1}$, $111\overline{1}0\overline{1}$, $\overline{1}0\overline{1}1\overline{1}0\overline{1}$, $\overline{1}011$, 111011, $\overline{1}0\overline{1}1011$, $1\overline{1}1\overline{1}1$, $\overline{1}\overline{1}\overline{1}\overline{1}1\overline{1}1$, $\overline{1}001\overline{1}\overline{1}1$, $110\overline{1}\overline{1}1$, $\overline{1}0\overline{1}0\overline{1}\overline{1}1$	13
10100	10100, $\overline{1}\overline{1}\overline{1}0100$, $\overline{1}\overline{1}00$, $111\overline{1}00$, $\overline{1}0\overline{1}1\overline{1}00$	5
10101	10101, $\overline{1}\overline{1}\overline{1}0101$, $\overline{1}\overline{1}01$, $111\overline{1}01$, $\overline{1}0\overline{1}1\overline{1}01$, $100\overline{1}\overline{1}$, $\overline{1}\overline{1}\overline{1}00\overline{1}\overline{1}$, $1111\overline{1}$, $\overline{1}\overline{1}\overline{1}111\overline{1}$, $\overline{1}0\overline{1}11\overline{1}$, $\overline{1}1\overline{1}$	11

Table D.6: Representations of "positive" τNAFs of length 6.

τNAF k	Representations	$\vartheta(k,6)$
0	0	1
1	1, $\overline{1}\overline{1}\overline{1}$, $111\overline{1}\overline{1}$, $\overline{1}\overline{1}\overline{1}11\overline{1}\overline{1}$, $101\overline{1}11\overline{1}\overline{1}$, $\overline{1}0\overline{1}1\overline{1}\overline{1}$, $1110\overline{1}1\overline{1}\overline{1}$, $101\overline{1}$, $\overline{1}\overline{1}\overline{1}01\overline{1}$, $111\overline{1}\overline{1}01\overline{1}$, $101\overline{1}01\overline{1}$	11
10	10, $\overline{1}\overline{1}\overline{1}0$, $111\overline{1}\overline{1}0$, $\overline{1}\overline{1}\overline{1}11\overline{1}\overline{1}0$, $\overline{1}0\overline{1}1\overline{1}\overline{1}0$, $101\overline{1}0$, $\overline{1}\overline{1}\overline{1}01\overline{1}0$, $101\overline{1}01\overline{1}0$	8
$10\overline{1}$	$10\overline{1}$, $\overline{1}\overline{1}\overline{1}0\overline{1}$, $111\overline{1}\overline{1}0\overline{1}$, $\overline{1}0\overline{1}1\overline{1}\overline{1}0\overline{1}$, $101\overline{1}0\overline{1}$, $\overline{1}\overline{1}\overline{1}01\overline{1}0\overline{1}$, $\overline{1}\overline{1}011$, $111\overline{1}011$, $\overline{1}0\overline{1}1\overline{1}011$, 101011, $\overline{1}\overline{1}\overline{1}01011$, $1\overline{1}\overline{1}1$, $1111\overline{1}1$, $\overline{1}\overline{1}\overline{1}111\overline{1}1$, $\overline{1}0\overline{1}11\overline{1}1$, $100\overline{1}\overline{1}1$, $\overline{1}\overline{1}\overline{1}00\overline{1}\overline{1}1$	17
100	100, $\overline{1}\overline{1}\overline{1}00$, $111\overline{1}\overline{1}00$, $\overline{1}0\overline{1}1\overline{1}\overline{1}00$, $101\overline{1}00$, $\overline{1}\overline{1}\overline{1}01\overline{1}00$	6
101	101, $\overline{1}\overline{1}\overline{1}01$, $111\overline{1}\overline{1}01$, $\overline{1}0\overline{1}1\overline{1}\overline{1}01$, $101\overline{1}01$, $\overline{1}\overline{1}\overline{1}01\overline{1}01$, $\overline{1}\overline{1}$, $111\overline{1}$, $\overline{1}\overline{1}\overline{1}11\overline{1}$, $111\overline{1}\overline{1}111\overline{1}$, $101\overline{1}11\overline{1}$, $\overline{1}0\overline{1}1\overline{1}$, $1110\overline{1}1\overline{1}$, $\overline{1}0\overline{1}10\overline{1}1\overline{1}$	14
$10\overline{1}0$	$10\overline{1}0$, $\overline{1}\overline{1}\overline{1}0\overline{1}0$, $111\overline{1}\overline{1}0\overline{1}0$, $101\overline{1}0\overline{1}0$, $\overline{1}\overline{1}0110$, $111\overline{1}0110$, 1010110, $\overline{1}1\overline{1}10$, $1111\overline{1}10$, $\overline{1}0\overline{1}11\overline{1}10$, $100\overline{1}\overline{1}10$	11

τNAF k	Representations	$\vartheta(k,6)$
$100\bar{1}$	$100\bar{1}$, $\bar{1}\bar{1}100\bar{1}$, $111\bar{1}\bar{1}00\bar{1}$, $101\bar{1}00\bar{1}$, 1111, $\bar{1}\bar{1}111$, $111\bar{1}\bar{1}111$, $101\bar{1}111$, $\bar{1}0\bar{1}11$, $1110\bar{1}11$, $\bar{1}0\bar{1}10\bar{1}11$, $\bar{1}1$	12
1000	1000, $\bar{1}\bar{1}\bar{1}000$, $111\bar{1}\bar{1}000$, $101\bar{1}000$	4
1001	1001, $\bar{1}\bar{1}\bar{1}001$, $111\bar{1}\bar{1}001$, $101\bar{1}001$, $1\bar{1}\bar{1}\bar{1}$, $\bar{1}\bar{1}\bar{1}\bar{1}\bar{1}\bar{1}$, $111\bar{1}\bar{1}\bar{1}\bar{1}\bar{1}$, $101\bar{1}\bar{1}\bar{1}\bar{1}$, $\bar{1}001\bar{1}\bar{1}$, $11100\bar{1}\bar{1}$, $\bar{1}\bar{1}001\bar{1}$, $111\bar{1}001\bar{1}$, $1010001\bar{1}$	13
1010	1010, $\bar{1}\bar{1}\bar{1}010$, $111\bar{1}\bar{1}010$, $101\bar{1}010$, $\bar{1}\bar{1}0$, $111\bar{1}0$, $\bar{1}\bar{1}\bar{1}11\bar{1}0$, $101\bar{1}11\bar{1}0$, $\bar{1}0\bar{1}1\bar{1}0$, $1110\bar{1}1\bar{1}0$	10
$10\bar{1}0\bar{1}$	$10\bar{1}0\bar{1}$, $\bar{1}\bar{1}\bar{1}0\bar{1}0\bar{1}$, $101\bar{1}0\bar{1}0\bar{1}$, $\bar{1}\bar{1}0110\bar{1}$, $1010110\bar{1}$, $\bar{1}1\bar{1}10\bar{1}$, $1111\bar{1}10\bar{1}$, $100\bar{1}\bar{1}10\bar{1}$, 10011, $\bar{1}\bar{1}\bar{1}0011$, $101\bar{1}0011$, $1\bar{1}\bar{1}\bar{1}1$, $\bar{1}\bar{1}\bar{1}\bar{1}\bar{1}\bar{1}1$, $101\bar{1}\bar{1}\bar{1}\bar{1}1$, $\bar{1}001\bar{1}\bar{1}1$, $\bar{1}\bar{1}001\bar{1}1$, $10100\bar{1}\bar{1}1$	17
$10\bar{1}00$	$10\bar{1}00$, $\bar{1}\bar{1}\bar{1}0\bar{1}00$, $101\bar{1}0\bar{1}00$, $\bar{1}\bar{1}01100$, 10101100, $\bar{1}1\bar{1}100$, $1111\bar{1}100$, $100\bar{1}\bar{1}100$	8
$10\bar{1}01$	$10\bar{1}01$, $\bar{1}\bar{1}\bar{1}0\bar{1}01$, $101\bar{1}0\bar{1}01$, $\bar{1}\bar{1}01101$, 10101101, $\bar{1}1\bar{1}101$, $1111\bar{1}101$, $100\bar{1}\bar{1}101$, $\bar{1}\bar{1}010\bar{1}\bar{1}$, $10101010\bar{1}\bar{1}$, $\bar{1}1\bar{1}0\bar{1}\bar{1}$, $1111\bar{1}0\bar{1}\bar{1}$, $100\bar{1}\bar{1}0\bar{1}\bar{1}$, $11\bar{1}1\bar{1}$, $\bar{1}\bar{1}\bar{1}1\bar{1}1\bar{1}$, $101\bar{1}1\bar{1}1\bar{1}$, $\bar{1}0\bar{1}\bar{1}1\bar{1}$, $1110\bar{1}\bar{1}1\bar{1}$, $\bar{1}1011\bar{1}$, $11110111\bar{1}$, $100\bar{1}011\bar{1}$	21
$100\bar{1}0$	$100\bar{1}0$, $\bar{1}\bar{1}\bar{1}00\bar{1}0$, $101\bar{1}00\bar{1}0$, 11110, $\bar{1}\bar{1}\bar{1}110$, $101\bar{1}1110$, $\bar{1}0\bar{1}110$, $1110\bar{1}110$, $\bar{1}10$	9
$1000\bar{1}$	$1000\bar{1}$, $\bar{1}\bar{1}\bar{1}000\bar{1}$, $101\bar{1}000\bar{1}$, 10111, $\bar{1}\bar{1}\bar{1}0111$, $101\bar{1}0111$, $\bar{1}\bar{1}11$, $111\bar{1}11$, $\bar{1}\bar{1}\bar{1}11\bar{1}11$, $\bar{1}0\bar{1}1\bar{1}11$, $1\bar{1}0\bar{1}1$, $\bar{1}\bar{1}\bar{1}\bar{1}0\bar{1}1$, $101\bar{1}\bar{1}0\bar{1}1$, $\bar{1}0010\bar{1}1$	14
10000	10000, $\bar{1}\bar{1}\bar{1}0000$, $101\bar{1}0000$	3
10001	10001, $\bar{1}\bar{1}\bar{1}0001$, $101\bar{1}0001$, $10\bar{1}\bar{1}\bar{1}$, $\bar{1}\bar{1}\bar{1}0\bar{1}\bar{1}\bar{1}$, $101\bar{1}0\bar{1}\bar{1}\bar{1}$, $\bar{1}\bar{1}011\bar{1}\bar{1}$, $10101\bar{1}\bar{1}\bar{1}$, $\bar{1}1\bar{1}\bar{1}\bar{1}\bar{1}$, $1111\bar{1}\bar{1}\bar{1}\bar{1}$, $100\bar{1}\bar{1}\bar{1}\bar{1}\bar{1}$, $1101\bar{1}$, $\bar{1}\bar{1}\bar{1}101\bar{1}$, $101\bar{1}101\bar{1}$, $\bar{1}0\bar{1}01\bar{1}$, $1110\bar{1}01\bar{1}$	16

τNAF k	Representations	$\vartheta(k,6)$
10010	10010, $\overline{1}\overline{1}\overline{1}0010$, $101\overline{1}0010$, $1\overline{1}\overline{1}\overline{1}0$, $\overline{1}\overline{1}\overline{1}\overline{1}\overline{1}0$, $101\overline{1}\overline{1}\overline{1}\overline{1}0$, $\overline{1}001\overline{1}\overline{1}0$, $\overline{1}\overline{1}001\overline{1}0$, $10100\overline{1}\overline{1}0$	9
$1010\overline{1}$	$1010\overline{1}$, $\overline{1}\overline{1}\overline{1}010\overline{1}$, $101\overline{1}010\overline{1}$, $\overline{1}\overline{1}0\overline{1}$, $111\overline{1}0\overline{1}$, $\overline{1}\overline{1}\overline{1}11\overline{1}0\overline{1}$, $\overline{1}0\overline{1}\overline{1}\overline{1}0\overline{1}$, $\overline{1}011$, 111011, $\overline{1}\overline{1}\overline{1}11011$, $\overline{1}0\overline{1}1011$, $1\overline{1}\overline{1}\overline{1}$, $\overline{1}\overline{1}\overline{1}\overline{1}\overline{1}\overline{1}$, $101\overline{1}\overline{1}\overline{1}\overline{1}$, $\overline{1}0011\overline{1}\overline{1}$, $110\overline{1}\overline{1}\overline{1}$, $\overline{1}\overline{1}\overline{1}10\overline{1}\overline{1}\overline{1}$, $\overline{1}0\overline{1}0\overline{1}\overline{1}\overline{1}$	18
10100	10100, $\overline{1}\overline{1}\overline{1}0100$, $101\overline{1}0100$, $\overline{1}\overline{1}00$, $111\overline{1}00$, $\overline{1}\overline{1}\overline{1}11\overline{1}00$, $\overline{1}0\overline{1}1\overline{1}00$	7
10101	10101, $\overline{1}\overline{1}\overline{1}0101$, $101\overline{1}0101$, $\overline{1}\overline{1}01$, $111\overline{1}01$, $\overline{1}\overline{1}\overline{1}11\overline{1}01$, $\overline{1}0\overline{1}1\overline{1}01$, $100\overline{1}\overline{1}$, $\overline{1}\overline{1}\overline{1}00\overline{1}\overline{1}$, $101\overline{1}00\overline{1}\overline{1}$, $1111\overline{1}$, $\overline{1}\overline{1}\overline{1}111\overline{1}$, $101\overline{1}111\overline{1}$, $\overline{1}0\overline{1}11\overline{1}$, $1110\overline{1}11\overline{1}$, $\overline{1}1\overline{1}$	16
$10\overline{1}0\overline{1}0$	$10\overline{1}0\overline{1}0$, $\overline{1}\overline{1}\overline{1}0\overline{1}0\overline{1}0$, $\overline{1}\overline{1}0110\overline{1}0$, $\overline{1}1\overline{1}10\overline{1}0$, 100110, $\overline{1}\overline{1}\overline{1}00110$, $1\overline{1}\overline{1}\overline{1}10$, $\overline{1}\overline{1}\overline{1}\overline{1}\overline{1}\overline{1}10$, $\overline{1}001\overline{1}\overline{1}10$, $\overline{1}\overline{1}001\overline{1}10$	10
$10\overline{1}00\overline{1}$	$10\overline{1}00\overline{1}$, $\overline{1}\overline{1}\overline{1}0\overline{1}00\overline{1}$, $\overline{1}\overline{1}011100\overline{1}$, $\overline{1}1\overline{1}100\overline{1}$, $10\overline{1}111$, $\overline{1}\overline{1}\overline{1}0\overline{1}111$, $\overline{1}\overline{1}011111$, $\overline{1}1\overline{1}1111$, $\overline{1}\overline{1}000\overline{1}11$, $\overline{1}\overline{1}0100\overline{1}1$, $\overline{1}1\overline{1}00\overline{1}1$	11
$10\overline{1}000$	$10\overline{1}000$, $\overline{1}\overline{1}\overline{1}0\overline{1}000$, $\overline{1}\overline{1}011000$, $\overline{1}1\overline{1}1000$	4
$10\overline{1}001$	$10\overline{1}001$, $\overline{1}\overline{1}\overline{1}0\overline{1}001$, $\overline{1}\overline{1}011001$, $\overline{1}1\overline{1}1001$, $10\overline{1}\overline{1}\overline{1}\overline{1}$, $\overline{1}\overline{1}\overline{1}0\overline{1}\overline{1}\overline{1}\overline{1}$, $\overline{1}\overline{1}011\overline{1}\overline{1}\overline{1}$, $\overline{1}1\overline{1}1\overline{1}\overline{1}\overline{1}$, $1101\overline{1}\overline{1}$, $\overline{1}\overline{1}\overline{1}10\overline{1}\overline{1}$, $\overline{1}0\overline{1}01\overline{1}\overline{1}$, $10001\overline{1}$, $\overline{1}\overline{1}\overline{1}0001\overline{1}$	13
$10\overline{1}010$	$10\overline{1}010$, $\overline{1}\overline{1}\overline{1}0\overline{1}010$, $\overline{1}\overline{1}011010$, $\overline{1}1\overline{1}1010$, $\overline{1}\overline{1}010\overline{1}\overline{1}0$, $\overline{1}1\overline{1}0\overline{1}\overline{1}0$, $11\overline{1}1\overline{1}0$, $\overline{1}\overline{1}\overline{1}1\overline{1}1\overline{1}0$, $\overline{1}0\overline{1}\overline{1}1\overline{1}0$, $\overline{1}1011\overline{1}0$	10
$100\overline{1}0\overline{1}$	$100\overline{1}0\overline{1}$, $\overline{1}\overline{1}\overline{1}00\overline{1}0\overline{1}$, $11110\overline{1}$, $\overline{1}\overline{1}\overline{1}1110\overline{1}$, $\overline{1}0\overline{1}110\overline{1}$, $\overline{1}10\overline{1}$, 100011, $\overline{1}\overline{1}\overline{1}00011$, $10\overline{1}\overline{1}\overline{1}1$, $\overline{1}\overline{1}\overline{1}0\overline{1}\overline{1}\overline{1}1$, $\overline{1}\overline{1}011\overline{1}\overline{1}1$, $\overline{1}1\overline{1}1\overline{1}\overline{1}1$, $1101\overline{1}1$, $\overline{1}\overline{1}\overline{1}101\overline{1}1$, $\overline{1}0\overline{1}01\overline{1}1$	15
$100\overline{1}00$	$100\overline{1}00$, $\overline{1}\overline{1}\overline{1}00\overline{1}00$, 111100, $\overline{1}\overline{1}\overline{1}11100$, $\overline{1}0\overline{1}1100$, $\overline{1}100$	6
$100\overline{1}01$	$100\overline{1}01$, $\overline{1}\overline{1}\overline{1}00\overline{1}01$, 111101, $\overline{1}\overline{1}\overline{1}11101$, $\overline{1}0\overline{1}1101$, $\overline{1}101$, $1110\overline{1}\overline{1}$, $\overline{1}\overline{1}\overline{1}110\overline{1}\overline{1}$, $\overline{1}0\overline{1}10\overline{1}\overline{1}$, $\overline{1}0\overline{1}\overline{1}$, $101\overline{1}1\overline{1}$, $\overline{1}\overline{1}\overline{1}01\overline{1}\overline{1}\overline{1}$, $\overline{1}\overline{1}\overline{1}1\overline{1}$, $111\overline{1}\overline{1}\overline{1}\overline{1}$, $\overline{1}0\overline{1}1\overline{1}\overline{1}\overline{1}\overline{1}$, $11\overline{1}$	16

τNAF k	Representations	$\vartheta(k,6)$
$1000\bar{1}0$	$1000\bar{1}0$, $\bar{1}\bar{1}\bar{1}000\bar{1}0$, 101110, $\bar{1}\bar{1}\bar{1}01110$, $\bar{1}\bar{1}110$, $111\bar{1}110$, $\bar{1}0\bar{1}1\bar{1}110$, $1\bar{1}0\bar{1}10$, $\bar{1}\bar{1}\bar{1}\bar{1}0\bar{1}10$, $\bar{1}0010\bar{1}10$	10
$10000\bar{1}$	$10000\bar{1}$, $\bar{1}\bar{1}\bar{1}00000\bar{1}$, 100111, $\bar{1}\bar{1}\bar{1}00111$, $1\bar{1}\bar{1}\bar{1}11$, $\bar{1}\bar{1}\bar{1}\bar{1}\bar{1}\bar{1}\bar{1}11$, $\bar{1}001\bar{1}\bar{1}11$, $\bar{1}\bar{1}001\bar{1}11$, $10\bar{1}0\bar{1}1$, $\bar{1}\bar{1}\bar{1}0\bar{1}0\bar{1}1$, $\bar{1}\bar{1}0110\bar{1}1$, $\bar{1}1\bar{1}10\bar{1}1$	12
100000	100000, $\bar{1}\bar{1}\bar{1}00000$	2
100001	100001, $\bar{1}\bar{1}\bar{1}00001$, $100\bar{1}\bar{1}\bar{1}$, $\bar{1}\bar{1}\bar{1}00\bar{1}\bar{1}\bar{1}$, $1111\bar{1}\bar{1}$, $\bar{1}\bar{1}\bar{1}111\bar{1}\bar{1}$, $\bar{1}0\bar{1}11\bar{1}\bar{1}$, $\bar{1}1\bar{1}\bar{1}$, $10101\bar{1}$, $\bar{1}\bar{1}\bar{1}0101\bar{1}$, $\bar{1}\bar{1}01\bar{1}$, $111\bar{1}01\bar{1}$, $\bar{1}0\bar{1}1\bar{1}01\bar{1}$	13
100010	100010, $\bar{1}\bar{1}\bar{1}00010$, $10\bar{1}\bar{1}\bar{1}0$, $\bar{1}\bar{1}\bar{1}0\bar{1}\bar{1}\bar{1}0$, $\bar{1}\bar{1}011\bar{1}\bar{1}0$, $\bar{1}1\bar{1}1\bar{1}\bar{1}0$, $1101\bar{1}0$, $\bar{1}\bar{1}\bar{1}101\bar{1}0$, $\bar{1}0\bar{1}01\bar{1}0$	9
$10010\bar{1}$	$10010\bar{1}$, $\bar{1}\bar{1}\bar{1}00010\bar{1}$, $1\bar{1}\bar{1}\bar{1}0\bar{1}$, $\bar{1}\bar{1}\bar{1}\bar{1}\bar{1}\bar{1}0\bar{1}$, $\bar{1}001\bar{1}\bar{1}0\bar{1}$, $\bar{1}\bar{1}001\bar{1}0\bar{1}$, $1\bar{1}\bar{1}011$, $\bar{1}\bar{1}\bar{1}\bar{1}\bar{1}011$, $\bar{1}001\bar{1}011$, $\bar{1}\bar{1}001011$, $10\bar{1}1\bar{1}1$, $\bar{1}\bar{1}\bar{1}0\bar{1}1\bar{1}1$, $\bar{1}\bar{1}0111\bar{1}1$, $\bar{1}1\bar{1}11\bar{1}1$, $\bar{1}\bar{1}000\bar{1}\bar{1}1$	15
100100	100100, $\bar{1}\bar{1}\bar{1}00100$, $1\bar{1}\bar{1}\bar{1}00$, $\bar{1}\bar{1}\bar{1}\bar{1}\bar{1}\bar{1}00$, $\bar{1}001\bar{1}\bar{1}00$, $\bar{1}\bar{1}001\bar{1}00$	6
100101	100101, $\bar{1}\bar{1}\bar{1}00101$, $1\bar{1}\bar{1}\bar{1}01$, $\bar{1}\bar{1}\bar{1}\bar{1}\bar{1}\bar{1}01$, $\bar{1}001\bar{1}\bar{1}01$, $\bar{1}\bar{1}001\bar{1}01$, $1000\bar{1}\bar{1}$, $\bar{1}\bar{1}\bar{1}000\bar{1}\bar{1}$, $101111\bar{1}$, $\bar{1}\bar{1}\bar{1}01111\bar{1}$, $\bar{1}\bar{1}11\bar{1}$, $111\bar{1}11\bar{1}$, $\bar{1}0\bar{1}1\bar{1}11\bar{1}$, $1\bar{1}0\bar{1}1\bar{1}$, $\bar{1}\bar{1}\bar{1}\bar{1}0\bar{1}1\bar{1}$, $\bar{1}0010\bar{1}1\bar{1}$	16
$1010\bar{1}0$	$1010\bar{1}0$, $\bar{1}\bar{1}\bar{1}010\bar{1}0$, $\bar{1}\bar{1}0\bar{1}0$, $111\bar{1}0\bar{1}0$, $\bar{1}0\bar{1}1\bar{1}0\bar{1}0$, $\bar{1}0110$, 1110110, $\bar{1}0\bar{1}10110$, $1\bar{1}1\bar{1}10$, $\bar{1}\bar{1}\bar{1}\bar{1}1\bar{1}10$, $\bar{1}0011\bar{1}10$, $110\bar{1}\bar{1}10$, $\bar{1}0\bar{1}0\bar{1}\bar{1}10$	13
$10100\bar{1}$	$10100\bar{1}$, $\bar{1}\bar{1}\bar{1}0100\bar{1}$, $\bar{1}\bar{1}00\bar{1}$, $111\bar{1}00\bar{1}$, $\bar{1}0\bar{1}1\bar{1}00\bar{1}$, 101111, $\bar{1}\bar{1}\bar{1}01111$, $\bar{1}\bar{1}111$, $111\bar{1}111$, $\bar{1}0\bar{1}1\bar{1}111$, $1\bar{1}0\bar{1}11$, $\bar{1}\bar{1}\bar{1}\bar{1}0\bar{1}11$, $\bar{1}0010\bar{1}11$, $1000\bar{1}1$, $\bar{1}\bar{1}\bar{1}000\bar{1}1$	15
101000	101000, $\bar{1}\bar{1}\bar{1}01000$, $\bar{1}\bar{1}000$, $111\bar{1}000$, $\bar{1}0\bar{1}1\bar{1}000$	5
101001	101001, $\bar{1}\bar{1}\bar{1}01001$, $\bar{1}\bar{1}001$, $111\bar{1}001$, $\bar{1}0\bar{1}1\bar{1}001$, $101\bar{1}\bar{1}\bar{1}$, $\bar{1}\bar{1}\bar{1}01\bar{1}\bar{1}\bar{1}$, $\bar{1}\bar{1}\bar{1}\bar{1}\bar{1}$, $111\bar{1}\bar{1}\bar{1}\bar{1}$, $\bar{1}0\bar{1}1\bar{1}\bar{1}\bar{1}\bar{1}$, $1\bar{1}\bar{1}$, $\bar{1}001\bar{1}$, $111001\bar{1}$, $\bar{1}0\bar{1}1001\bar{1}$	14

τNAF k	Representations	$\vartheta(k,6)$
101010	101010, $\overline{1}\overline{1}\overline{1}01010$, $\overline{1}\overline{1}010$, $111\overline{1}010$, $\overline{1}0\overline{1}1\overline{1}010$, $100\overline{1}\overline{1}0$, $\overline{1}\overline{1}\overline{1}00\overline{1}\overline{1}0$, $1111\overline{1}0$, $\overline{1}\overline{1}\overline{1}111\overline{1}0$, $\overline{1}0\overline{1}11\overline{1}0$, $\overline{1}1\overline{1}0$	11

D.2 Examples of $k_{max,l}$

Table D.7 presents $k_{max,l}$ and $\vartheta(k_{max,l}, l)$ for $1 \leq l \leq 13$.

Table D.7: "Positive" τNAFs with maximum number of representations

l	$k_{max,l}$	$\vartheta(k_{max,l}, l)$
1	1	2
2	1	3
3	101	5
4	$10\overline{1}$	8
5	$1010\overline{1}$	13
6	$10\overline{1}01$	21
7	$1010\overline{1}01$	34
8	$10\overline{1}010\overline{1}$	55
9	$1010\overline{1}010\overline{1}$	89
10	$10\overline{1}010\overline{1}01$	144
11	$1010\overline{1}010\overline{1}01$	233
12	$10\overline{1}010\overline{1}010\overline{1}$	377
13	$1010\overline{1}010\overline{1}010\overline{1}$	610

Appendix E

Nondeterministic Finite Automata, Directed Graphs and Adjacency Matrices

A *nondeterministic finite automaton* (NFA) Γ is a quintuple $(Q, \Sigma, s_0, F, \delta)$ [HMU01], where

- Q is a set of *states*,
- Σ is the alphabet (set) of *input symbols*,
- $s_0 \in Q$ is the *initial state*,
- $F \subset Q$ is the set of *final* (or *accepting*) *states*,
- $\delta : Q \times \Sigma \to \mathcal{P}(Q)$ is the *transition function*, where $\mathcal{P}(Q)$ is the *powerset* of Q, that is, the set of all subsets of Q (including the empty set).

Let X be a string over the alphabet Σ, and ε be the empty string. Γ *accepts* the string X if there exist both a representation of X of the form $x_1x_2\ldots x_l$, $x_i \in (\Sigma \cup \{\varepsilon\})$, and a sequence of states $s_0, s_1, \ldots, s_l$, $s_i \in Q$, meeting the following conditions:

- s_0 is the initial state,
- $s_i \in \delta(s_{i-1}, x_i)$, for $1 \leq i \leq l$ and
- $s_l \in F$. [Sip97, Section 1.2, pp. 47-63]

An NFA can be represented by a directed graph where the vertices are the states of the set Q, and the directed edges are determined by the function δ. That is, a directed edge exists starting at vertex s_i and ending at vertex s_j iff $s_j \in \delta(s_i, x)$, for any $x \in \Sigma$, and this edge will be labeled as x. The concatenation of directed edges encountered when Γ is reading an accepted string form a *directed path*.

To each directed graph, we can associate the *adjacency matrix*, $M = (m_{ij})$ for $0 \leq i, j \leq |Q|$, such that $m_{ij} = 1$ if there is a directed edge from vertex s_i to vertex s_j in Γ and 0 otherwise. From the definition of matrix multiplication and the concatenation of paths, the l^{th} power of M, *i.e.*, M^l has the number of paths of length l from vertex s_i to vertex s_j as its ij^{th} entry. This is obviously true for $l = 1$. Next observe that any path of length l from vertex s_i to vertex s_j decomposes into the initial path of length $l-1$ starting at s_i (to some intermediate vertex) followed by a path of length 1 ending at s_j, these paths are counted for all possible intermediate vertices by the sum of the vector product of the i^{th} row of M^{l-1} with the j^{th} column of M [Big93, Lemma 2.5].

Moreover, to an NFA Γ, we can associate an adjacency matrix, M_{x_i}, for each input symbol $x_i \in \Sigma$, $1 \leq i \leq |\Sigma|$. Hence the number of directed paths possibly traversed when Γ reads an accepted string $X = x_1x_2\ldots x_l$ can be found as the $(0, f)^{th}$ entry of the

product [Sha; Daw03]

$$M_{x_1} M_{x_2} \cdots M_{x_l},$$

for each $f \in F$ possibly reached when x_l was read.

The following are the adjacency matrices corresponding to the automaton in Figure 5.1.

$$M = \begin{pmatrix}
0&0&0&0&1&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&1&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&1&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&1&0&1&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&1&0&0&0&0&0&0&0&0\\
0&1&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&1&0&0&0&0&0&0\\
0&0&0&1&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&1&0&0&0&0\\
0&1&0&1&0&0&0&0&0&1&0&1&0&0&0&0&0&1&0&1&0&0&0&0\\
0&0&0&0&1&0&1&0&0&0&0&0&1&0&1&0&0&0&0&0&1&0&1&0\\
0&0&0&0&1&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&1&0&0&0\\
0&0&1&0&1&0&0&0&0&0&1&0&1&0&0&0&0&0&1&0&1&0&0&0\\
0&0&0&0&0&1&0&1&0&0&0&0&0&1&0&1&0&0&0&0&0&1&0&1\\
0&0&0&0&0&1&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&1&0&0\\
0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&1\\
0&0&0&0&0&0&0&0&0&1&0&1&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&1&0&1&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&1&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0
\end{pmatrix}$$

$$M_{\bar{1}} = \begin{pmatrix}
0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&1&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&1&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&1&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&1&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0
\end{pmatrix}$$

$$M_0 = \begin{pmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix}$$

$$M_1 = \begin{pmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix}$$

Bibliography

[AARR] D. Agrawal, B. Archambeault, J. R. Rao & P. Rohatgi. *The EM Side-Channel(s): Attacks and Assessment Methodologies*. Internet Security Group, IBM Watson Research Center. `http://www.research.ibm.com/intsec/emf-paper.ps`. 2, 3

[AHLM03] M. Ahn, J. Ha, H. Lee & S. Moon. "A random m-ary method based countermeasure against side channel attacks". In *Computational Science and Its Applications – ICCSA '03*, *LNCS*, vol. 2668, p. 338347. Springer-Verlag, 2003. 175

[AT03] T. Akishita & T. Takagi. "Zero-value point attacks on elliptic curve cryptosystem". In *International Security Conference – ISC '03*, *LNCS*, vol. 2851, pp. 218–233. Springer-Verlag, 2003. 166, 167, 183

[Ava] R. M. Avanzi. Personal communication. March, 2006. 156

[Ava05] R. M. Avanzi. "Side channel attacks on implementations of curve-based cryptographic primitives". Technical report 2005/017, Cryptology ePrint Archive, 2005. Available at `http://eprint.iacr.org/2005/017`. 2, 7, 158, 187, 190, 197, 200, 201

[BCF06] B. Byramjee, J.-C. Courrège & B. Feix. "Practical attacks on smart cards". In H. Cohen & G. Frey, editors, *Handbook of Elliptic and Hyperelliptic Curve Cryptography*, chap. 28. Chapman and Hall/CRC, 2006. 2

[BDL01] D. Boneh, R. A. DeMillo & R. J. Lipton. "On the importance of eliminating errors in cryptographic computations". *Journal of Cryptology*, **14**:101–119, 2001. This is an expanded version of an earlier paper that appeared in *Proc. of EUROCRYPT '97*. 2

[Ber01] D. J. Bernestein. "Pippenger's exponentiation algorithm", 2001. Available at `http://cr.yp.to/papers.html#pippenger`. 16, 154, 156

[BGMW93] E. F. Brickell, D. M. Gordon, K. S. McCurcley & D. B. Wilson. "Fast exponentiation with precomputation". In *Advances in Cryptology – EUROCRYPT '92*, *LNCS*, vol. 658, pp. 200–207. Springer-Verlag, 1993. Extended and more recent (1995) version available at `http://research.microsoft.com/~dbwilson/bgmw`. 154

[Big93] N. Biggs. *Algebraic graph theory*. Cambridge University Press, 1993. 222

[BJ02] É. Brier & M. Joye. "Weierstraß elliptic curves and side-channel attacks". In *Public Key Cryptography – PKC '02*, *LNCS*, vol. 2274, pp. 335–345. Springer-Verlag, 2002. 163, 166, 183

[Cie03] M. Ciet. *Aspects of Fast and Secure Arithmetics for Elliptic Curve Cryptography*. Ph.D. thesis, Université Catholique de Louvain, 2003. 120, 170, 171, 180

[CJ01] C. Clavier & M. Joye. "Universal exponentiation algorithm a first step towards provable SPA-resistance". In *Cryptographic Hardware and Embedded Systems – CHES '01*, *LNCS*, vol. 2162, pp. 300–308. Springer-Verlag, 2001. 4, 24, 120

[CJ03] M. Ciet & M. Joye. "(Virtually) free randomization techniques for elliptic curve cryptography". In *Information and Communications Security – ICICS '03*, *LNCS*, vol. 2836, pp. 348–359. Springer-Verlag, 2003. 4, 25, 122, 164, 181

[CJRR99] S. Chari, C. S. Jutla, J. R. Rao & P. Rohatgi. "Towards sound approaches to counteract power-analysis attacks." In *Advances in Cryptology – CRYPTO '99*, *LNCS*, vol. 1666, pp. 398–412. Springer-Verlag, 1999. 24

[CM04] B. Chevallier-Mames. "Self-randomized exponentiation algorithms". In *Topics in Cryptology – CT-RSA '04*, *LNCS*, vol. 2964, pp. 236–249. Springer-Verlag, 2004. 177

[CMCJ04] B. Chevallier-Mames, M. Ciet & M. Joye. "Low-cost solutions for preventing simple side-channel analysis: Side-channel atomicity". *IEEE Transactions on Computers*, **53**(6):760–768, 2004. 169

[CMO98] H. Cohen, A. Miyaji & T. Ono. "Efficient elliptic curve exponentiation using mixed coordinates". In *Advances in Cryptology – ASIACRYPT '98*, *LNCS*, vol. 1514, pp. 51–65. Springer-Verlag, 1998. 9, 75

[Cor99] J.-S. Coron. "Resistance against differential power analysis for elliptic curve cryptosystems". In *Cryptographic Hardware and Embedded Systems –*

CHES '99, *LNCS*, vol. 1717, pp. 292–302. Springer-Verlag, 1999. 2, 22, 24, 158, 170, 180, 181, 186

[CQS03] M. Ciet, J.-J. Quisquater & F. Sica. "Preventing differential analysis in GLV elliptic curve scalar multiplication". In *Cryptographic Hardware and Embedded Systems – CHES '02*, *LNCS*, vol. 2523, pp. 540–550. Springer-Verlag, 2003. 4, 25, 126, 173, 179

[Daw03] A. Dawar. "Quantum automata, machines and complexity". A talk given at University of Warwick, 24 October 2003. Available at `http://www.cl.cam.ac.uk/users/ad260/talks/warwick.pdf`, 2003. 223

[DOH06] A. Dominguez-Oviedo & M. A. Hasan. "Improved error-detection and fault-tolerance in ECSM using input randomization". CACR Technical Reports CACR 2006-41, University of Waterloo, 2006.

[EH03a] N. Ebeid & M. A. Hasan. "Analysis of DPA countermeasures based on randomizing the binary algorithm". CACR Technical Reports CORR 2003-14, University of Waterloo, 2003. 58, 70, 174

[EH03b] N. Ebeid & M. A. Hasan. "On randomizing private keys to counteract DPA attacks". In *Selected Areas in Cryptography – SAC '03*, *LNCS*, vol. 3006, pp. 58–72. Springer-Verlag, 2003. 230

[EH07] N. Ebeid & M. A. Hasan. "On binary signed digit representations of integers". *Designs, Codes and Cryptography*, **42**:43–65, 2007. This is an expanded and revised version of [EH03b]. 50

[EK94] Ö. Eğecioğlu & Ç. K. Koç. "Exponentiation using canonical recoding". *Theoretical Computer Science*, **129**(2):407–417, 1994. 31, 40

[ElG85] T. ElGamal. "A public key cryptosystem and a signature scheme based on discrete logarithms". *IEEE Transactions on Information Theory*, **31**(4):469–472, 1985. 16

[FMPV04] P.-A. Fouque, F. Muller, G. Poupard & F. Valette. "Defeating countermeasures based on randomized BSD representations". In *Cryptographic Hardware and Embedded Systems – CHES '04*, *LNCS*, vol. 3156, pp. 312–327. Springer-Verlag, 2004. 4, 24, 76, 101, 174

[FV03] P.-A. Fouque & F. Valette. "The doubling attack - *why upwards is better than downwards*". In *Cryptographic Hardware and Embedded Systems – CHES '03*, *LNCS*, vol. 2779, pp. 269–280. Springer-Verlag, 2003. 159, 186, 201

[GG02] C. H. Gebotys & R. J. Gebotys. "Secure elliptic curve implementations: An analysis of resistance to power-attacks in a DSP processor". In *Cryptographic Hardware and Embedded Systems – CHES '02*, *LNCS*, vol. 2523, pp. 114–128. Springer-Verlag, 2002. 168

[GHT05] C. Gebotys, S. Ho & A. Tiu. "EM analysis of Rijndael and ECC on a PDA". CACR Technical Reports CORR 2003-14, University of Waterloo, 2005. 171

[Gor98] D. M. Gordon. "A survey of fast exponentiation methods". *Journal of Algorithms*, **27**(1):129–146, 1998. 103

[Gou03] L. Goubin. "A refined power-analysis attack on elliptic curve cryptosystems". In *Public Key Cryptography – PKC '03*, *LNCS*, vol. 2567, pp. 199–211. Springer-Verlag, 2003. 23, 161, 182

[Gre83] D. H. Greene. "Labelled formal languages and their uses". Tech. Rep. STAN-CS-83-982, Stanford University, 1983. 42, 207, 209

[Has01] M. A. Hasan. "Power analysis attacks and algorithmic approaches to their countermeasures for Koblitz curve cryptosystems". *IEEE Transactions on Computers*, **50**(10):1071–1083, Oct. 2001. 117

[HM02a] J. Ha & S. Moon. "Randomized signed-scalar multiplication of ECC to resist power attacks". In *Cryptographic Hardware and Embedded Systems – CHES '02*, *LNCS*, vol. 2523, pp. 551–563. Springer-Verlag, 2002. 3, 24, 27, 31, 40, 58, 70, 75, 93, 101, 173

[HM02b] Y. Hitchcock & P. Montague. "A new elliptic curve scalar multiplication algorithm to resist simple power analysis". In *Australian Conference on Information Security and Privacy – ACISP'02*, *LNCS*, vol. 2384, pp. 214–225. Springer-Verlag, 2002. 164

[HMU01] J. E. Hopcroft, R. Motwani & J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, second ed., 2001. 42, 207, 208, 221

[HMV04] D. Hankerson, A. Menezes & S. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer-Verlag, 2004. 8, 9, 10, 11, 18, 130, 131, 138, 145, 168, 197

[HP] C. Heuberger & H. Prodinger. Personal communication. August, 2003. 210

[HPB04] M. Hedabou, P. Pinel & L. Bénéteau. "A comb method to render ECC resistant against side channel attacks". Technical report 2004/342, Cryptology ePrint Archive, 2004. Available at `http://eprint.iacr.org/2004/342`. 165

[HVZ02] V. C. Hamacher, Z. G. Vranesic & S. G. Zaky. *Computer Organization*. Boston: McGraw-Hill, fifth ed., 2002. 136

[IIT02] K. Itoh, T. Izu & M. Takenaka. "Address-bit differential power analysis of cryptographic schemes OK-ECDH and OK-ECDSA". In *Cryptographic Hardware and Embedded Systems – CHES '02*, *LNCS*, vol. 2523, pp. 129–143. Springer-Verlag, 2002. 162, 173, 198

[IIT03] K. Itoh, T. Izu & M. Takenaka. "A practical countermeasure against address-bit differential power analysis". In *Cryptographic Hardware and Embedded Systems – CHES '03*, *LNCS*, vol. 2779, pp. 382–396. Springer-Verlag, 2003. 199

[IIT04] K. Itoh, T. Izu & M. Takenaka. "Efficient countermeasures against power analysis for elliptic curve cryptosystems". In *Smart Card Research and Advanced Applications – CARDIS '04*, pp. 99–114. 2004. 184, 185

[IMT02] T. Izu, B. Möller & T. Takagi. "Improved elliptic curve multiplication methods resistant against side channel attacks". In *Advances in Cryptology – INDOCRYPT '02*, *LNCS*, vol. 2551, pp. 296–313. Springer-Verlag, 2002. 182, 189

[IT02] T. Izu & T. Takagi. "A fast parallel elliptic curve multiplication resistant against side channel attacks". In *Public Key Cryptography – PKC '02*, *LNCS*, vol. 2274, pp. 280–296. Springer-Verlag, 2002. Extended version available at `http://www.cacr.math.uwaterloo.ca/techreports/2002/corr2002-03.ps`. 23, 142, 143, 158, 162, 163, 183

[IYTT02] K. Itoh, J. Yajima, M. Takenaka & N. Torii. "DPA countermeasures by improving the window method". In *Cryptographic Hardware and Embedded Systems – CHES '02*, *LNCS*, vol. 2523, p. 303317. Springer-Verlag, 2002. 176

[Joy03] M. Joy. "Elliptic curves and side-channel analysis". *ST Journal of System Research*, **4**(1):283–306, 2003. 117

[Joy05] M. Joye. "Defenses against side-channel analysis". In I. F. Blake, G. Seroussi & N. P. Smart, editors, *Advances in Elliptic Curve Cryptography*, chap. 5. Cambridge University Press, 2005. 2, 24

[JPS05] M. Joye, P. Paillier & B. Schoenmakers. "On second-order differential power analysis". In *Cryptographic Hardware and Embedded Systems – CHES '05*, *LNCS*, vol. 3659, pp. 293–308. Springer-Verlag, 2005. 146, 195

[JV02] M. Joye & K. Villegas. "A protected division algorithm". In *Smart Card Research and Advanced Applications – CARDIS '02*, pp. 59–68. Usenix Association, 2002. 122, 136

[Kal85] J. G. Kalbfleisch. *Probability and Statistical Inference. Volume 1: Probability.* Springer-Verlag, 1985. 86, 112

[KHM+05] C. Kim, J. Ha, S. Moon, S.-M. Yen, W.-C. Lien & S.-H. Kim. "An

improved and efficient countermeasure against power analysis attacks". Technical report 2005/22, Cryptology ePrint Archive, 2005. Available at `http://eprint.iacr.org/2005/022`. 187

[KJJ99] P. Kocher, J. Jaffe & B. Jun. "Differential power analysis". In *Advances in Cryptology – CRYPTO '99*, *LNCS*, vol. 1666. Springer-Verlag, 1999. 2, 22, 24, 172, 194

[Knu73] D. E. Knuth. *The Art of Computer Programming/Seminumerical Algorithms*, vol. 2. Addison-Wesley, second ed., 1973. 12

[Kob87] N. Koblitz. "Elliptic curve cryptosystems". *Mathematics of Computation*, **48**:203–209, 1987. 1

[Kob92] N. Koblitz. "CM curves with good cryptographic properties". In *Advances in Cryptology – CRYPTO '91*, *LNCS*, vol. 576, pp. 279–287. Springer-Verlag, 1992. 19, 89

[Koc96] P. Kocher. "Timing attacks on implementations of Diffie-Hellman, RSA, DSS and other systems." In *Advances in Cryptology – CRYPTO '96*, *LNCS*, vol. 1109, pp. 104–113. Springer-Verlag, August 1996. 2

[Kos01] T. Koshy. *Fibonacci and Lucas numbers with Applications*. New York: Wiley, 2001. 50, 65, 87, 109, 150, 152

[KW03] C. Karlof & D. Wagner. "Hidden Markov model cryptanalysis". In *Cryptographic Hardware and Embedded Systems – CHES '03*, *LNCS*, vol. 2779, pp. 17–34. Springer-Verlag, 2003. 47, 174

[Lan06] T. Lange. "Mathematical countermeasures against side-channel attacks". In H. Cohen & G. Frey, editors, *Handbook of Elliptic and Hyperelliptic Curve Cryptography*, chap. 29. Chapman and Hall/CRC, 2006. 2

[LD99] J. López & R. Dahab. "Improved algorithms for elliptic curve arithmetic in GF(2^n) without precomputation". In *Selected Areas in Cryptography – SAC '98*, *LNCS*, vol. 1556, pp. 201–212. Springer-Verlag, 1999. 75

[Lim04] C. H. Lim. "A new method for securing elliptic scalar multiplication against side-channel attacks". In *Australian Conference on Information Security and Privacy – ACISP'04*, *LNCS*, vol. 3108, pp. 289–300. Springer-Verlag, 2004. 128

[LL94] C. H. Lim & P. J. Lee. "More flexible exponentiation with precomputation". In *Advances in Cryptology – CRYPTO '94*, vol. 839, pp. 95–107. Springer-Verlag, 1994. 156

[LS01] P.-Y. Liardet & N. P. Smart. "Preventing SPA/DPA in ECC systems using the Jacobi form". In *Cryptographic Hardware and Embedded Systems – CHES '01*, *LNCS*, vol. 2162, pp. 391–401. Springer-Verlag, 2001. 175

[MDS99a] T. S. Messerges, E. A. Dabbish & R. H. Sloan. "Investigations of power analysis attacks on smart cards". In *USENIX Workshop on Smartcard Technology*, pp. 151–161. May 1999. 2, 24, 172

[MDS99b] T. S. Messerges, E. A. Dabbish & R. H. Sloan. "Power analysis attacks of modular exponentiation in smart cards". In *Cryptographic Hardware and Embedded Systems – CHES '99*, *LNCS*, vol. 1717, pp. 144–157. Springer-Verlag, Aug. 1999. 2, 179, 190

[MDS02] T. S. Messerges, E. A. Dabbish & R. H. Sloan. "Examining smart card security under the threat of power analysis attacks". *IEEE Transactions on Computers*, **51**(5):541–552, May 2002. 22, 24

[Mes00] T. S. Messerges. "Using second-order power analysis to attack DPA resistant software". In *Cryptographic Hardware and Embedded Systems – CHES '00*, *LNCS*, vol. 1965, pp. 238–251. Springer-Verlag, 2000. 145, 193, 194

[Mil86] V. S. Miller. "Use of elliptic curves in cryptography". In *Advances in Cryptology – CRYPTO '85*, *LNCS*, vol. 218, pp. 417–426. Springer-Verlag, 1986. 1

[MMM04] H. Mamiya, A. Miyaji, & H. Morimoto. "Efficient countermeasures against RPA, DPA, and SPA". In *Cryptographic Hardware and Embedded Systems – CHES '04*, *LNCS*, vol. 3156, pp. 343–356. Springer-Verlag, 2004. 185

[MO90] F. Morain & J. Olivos. "Speeding up the computations on an elliptic curve using addition-subtraction chains". *Informatique théorique et Applications/Theoretical Informatics and Applications*, **24**(6):531–544, 1990. 15, 28, 36, 37, 42, 44, 46, 50, 70, 209

[MOC97] A. Miyaji, T. Ono & H. Cohen. "Efficient elliptic curve exponentiation". In *Information and Communication Security, First International Conference – ICICS '97*, *LNCS*, vol. 1334, pp. 282–290. Springer-Verlag, 1997. 15

[Möl01] B. Möller. "Securing elliptic curve point multiplication against side-channel attacks". In *International Security Conference – ISC '01*, *LNCS*,

vol. 2200, pp. 324–334. Springer-Verlag, 2001. Extended version (addendum: "Efficiency improvement" and errata) available at `http://www.informatik.tudarmstadt.de/TI/Mitarbeiter/moeller/ecc-scaisc01.pdf`. 23, 164, 196

[Möl02] B. Möller. "Parallelizable elliptic curve point multiplication method with resistance against side-channel attacks". In *International Security Conference – ISC '02*, *LNCS*, vol. 2433, pp. 402–413. Springer-Verlag, 2002. 154, 188

[Mon87] P.-L. Montgomery. "Speeding the Pollard and elliptic curve methods of factorization". *Mathematics of Computation*, **48**:243–264, 1987. 23, 131, 161

[MS00] R. Mayer-Sommer. "Smartly analyzing the simplicity and the power of simple power analysis on smartcards". In *Cryptographic Hardware and Embedded Systems – CHES '00*, *LNCS*, vol. 1965, pp. 78–92. Springer-Verlag, 2000. 121, 189

[MvOV96] A. J. Menezes, P. C. van Oorschot & S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996. 139

[NIST] National Institute of Standards and Technology. "FIPS 186-2: Digital Signature Standard (DSS)", Jan. 2000. 1, 4, 12, 116, 144, 180, 183

[NSA] National Security Agency (NSA). "Fact sheet NSA suite B cryptography". `http://www.nsa.gov/ia/industry/crypto_suite_b.cfm`, 2005. 1, 153

[OA01] E. Oswald & M. Aigner. "Randomized addition-subtraction chains as

a countermeasure against power attacks". In *Cryptographic Hardware and Embedded Systems – CHES '01*, *LNCS*, vol. 2162, pp. 39–50. Springer-Verlag, 2001. 3, 24, 27, 28, 70, 173

[OH03] K. Okeya & D.-G. Han. "Side channel attack on Ha-Moon's countermeasure of randomized signed scalar multiplication". In *Advances in Cryptology – INDOCRYPT '03*, *LNCS*, vol. 2904, pp. 334–348. Springer-Verlag, 2003. 174

[OS00] K. Okeya & K. Sakurai. "Power analysis breaks elliptic curve cryptosystems even secure against the timing attack". In *Advances in Cryptology – INDOCRYPT '00*, *LNCS*, vol. 1977, pp. 178–190. Springer-Verlag, 2000. 180, 186

[OS02a] K. Okeya & K. Sakurai. "On insecurity of the side channel attack countermeasure using addition-subtraction chains under distinguishability between addition and doubling". In *Australian Conference on Information Security and Privacy – ACISP'02*, vol. 2384, pp. 420–435. Springer-Verlag, 2002. 31, 47, 174

[OS02b] K. Okeya & K. Sakurai. "A second-order DPA attack breaks a window-method based countermeasure against side channel attacks". In *International Security Conference – ISC '02*, *LNCS*, vol. 2433, pp. 389–401. Springer-Verlag, 2002. 196

[Osw05] E. Oswald. "Side-channel analysis". In I. F. Blake, G. Seroussi & N. P. Smart, editors, *Advances in Elliptic Curve Cryptography*, chap. 4. Cambridge University Press, 2005. 2

[OT03] K. Okeya & T. Takagi. "The width-w NAF method provides small memory and fast elliptic scalar multiplications secure against side channel attacks". In *Topics in Cryptology – CT-RSA '03*, *LNCS*, vol. 2612, pp. 328–343. Springer-Verlag, 2003. 23, 128, 164, 197

[Pip76] N. Pippenger. "On the evaluation of powers and related problems (preliminary version)". In *17th annual symposium on foundations of computer science, IEEE Computer Society*, pp. 258–263. 1976. 156

[QS01] J.-J. Quisquater & D. Samyde. "Electromagnetic analysis (EMA): Measures and counter-measures for smart cards". In *Smart Card Programming and Security (E-smart 2001)*, *LNCS*, vol. 2140, pp. 200–210. Springer-Verlag, 2001. 2

[RAA+03] P. Rohatgi, D. Agrawal, B. Archambeault, S. Chari & J. R. Rao. "Power, EM and all that: Is your crypto device really secure?" A talk given as part of the 7th Workshop on Elliptic Curve Cryptography – ECC 2003, 2003. 3

[Rei60] G. W. Reitwiesner. "Binary arithmetic". *Advances in Computers*, **1**:231–308, 1960. 14, 15, 31, 51, 58, 70

[ScKK00] E. Savas & Çetin Kaya Koç. "The Montgomery modular inverse-revisited." *IEEE Transactions on Computers*, **49**(7):763–766, 2000. 138, 139, 140

[SEC2] "Standard for Efficient Cryptography, SEC 2: Recommended elliptic curve domain parameters, ver.1". Certicom Research, 2000. Available at: `http://www.secg.org/index.php?action=secg,docs_secg`. 180

[SF96] R. Sedgewick & P. Flajolet. *An Introduction to the Analysis of Algorithms*. Addison-Wesley, 1996. 42, 211

[Sha] J. Shallit. Personal communication. May, 2006. 223

[Sip97] M. Sipser. *Introduction to the theory of computation*. Boston : PWS Pub. Co, 1997. 222

[Sol00] J. A. Solinas. "Efficient arithmetic on Koblitz curves". *Designs, Codes and Cryptography*, **19**:195–249, 2000. 4, 14, 15, 20, 37, 46, 50, 58, 70, 89, 90, 91, 92, 95, 110, 112

[SPL04] S. G. Sim, D. J. Park & P. J. Lee. "New power analysis on the Ha-Moon algorithm and the MIST algorithm". In *Information and Communications Security – ICICS '04*, *LNCS*, vol. 3269, pp. 291–304. Springer-Verlag, 2004. 101, 179

[Sta93] W. M. O. Staffelbach. "Efficient multiplication on certain nonsupersingular elliptic curves". In *Advances in Cryptology – CRYPTO '92*, *LNCS*, vol. 740, pp. 333–344. Springer-Verlag, 1993. 102

[Str64] E. G. Straus. "Addition chains of vectors (problem 5125)". *American Mathematical Monthly*, **70**:806–808, 1964. 16

[Thé06] N. Thériault. "SPA resistant left-to-right integer recodings". In *Selected Areas in Cryptography – SAC '05*, *LNCS*, vol. 3897, pp. 345–358. Springer-Verlag, 2006. 23, 128, 133, 164

[Wal02a] C. D. Walter. "Breaking the Liardet-Smart randomized exponentiation

algorithm". In *Smart Card Research and Advanced Applications – CARDIS '02*, pp. 59–68. Usenix Association, 2002. 175

[Wal02b] C. D. Walter. "MIST: An efficient, randomized exponentiation algorithm for resisting power analysis". In *Topics in Cryptology – CT-RSA '02*, *LNCS*, vol. 2271, pp. 53–66. Springer-Verlag, 2002. 179

[Wal02c] C. D. Walter. "Some security aspects of the MIST randomized exponentiation algorithm". In *Cryptographic Hardware and Embedded Systems – CHES '02*, *LNCS*, vol. 2523, pp. 276–290. Springer-Verlag, 2002. 179

[Wal04a] C. D. Walter. "Issues of security with the Oswald-Aigner exponentiation algorithm". In *Topics in Cryptology – CT-RSA '04*, *LNCS*, vol. 2964, pp. 208–221. Springer-Verlag, 2004. Also available at `http://eprint.iacr.org/2003/013`. 31, 47, 174

[Wal04b] C. D. Walter. "Simple power analysis of unified code for ECC double and add". In *Cryptographic Hardware and Embedded Systems – CHES '04*, *LNCS*, vol. 3156, pp. 191–204. Springer-Verlag, 2004. 167

[Yao76] A. C.-C. Yao. "On the evaluation of powers". *SIAM Journal on Computing*, **5**:100–103, 1976. 154

[YJ00] S.-M. Yen & M. Joye. "Checking before output may not be enough against fault-based cryptanalysis". *IEEE Transactions on Computers*, **49**(9):967–970, 2000. 159, 165, 168

[YLMH05] S.-M. Yen, W.-C. Lien, S. Moon & J. Ha. "Power analysis by exploiting chosen message and internal collisions–vulnerability of checking mechanism

for RSA-decryption". In *Progress in Cryptology - Mycrypt 2005*, *LNCS*, vol. 3715, pp. 183–195. Springer-Verlag, 2005. 160, 186