Université de Limoges
DMI - Département Mathématiques Informatique

Implémentations Efficaces de Crypto-systèmes Embarqués et Analyse de leur Sécurité

Efficient Embedded Implementations of Cryptosystems and Tamper Resistance Studies

THESE

présentée et soutenue publiquement le 16 décembre 2013
pour l'obtention du

Doctorat de l'Université de Limoges
(Spécialité : Informatique)

par

Benoît FEIX

Composition du jury

Directeur:	Christophe CLAVIER	Université de Limoges
Rapporteurs:	Louis GOUBIN	Université Versailles Saint-Quentin-en-Yvelines
	Colin WALTER	Royal Holloway University
Examinateurs:	Jean-Christophe COURREGE	Thales ITSEF
	Jean-François DHEM	Atos Worldline
	Jean-Louis LANET	Université de Limoges
	Pascal PAILLIER	CryptoExperts
	Guillaume POUPARD	Ministère de la défense, DGA/DT/ST

Résumé

La cryptographie est désormais un terme quasi omniprésent dans notre quotidien quel que soit l'intérêt que tout un chacun puisse porter à cette science. Elle représente aujourd'hui un rempart entre nous et les intrusions des pirates ou des institutions sans retenues qui ne se préoccupent guère du respect de notre vie privée. La cryptographie peut protéger nos données personnelles que nous stockons sur de multiples support numériques solides, voire nuageux pour les plus téméraires. Mais utiliser des mécanismes cryptographiques ne suffit pas. Il faut également les implémenter de telle sorte que leur utilisation soit résistante à une catégorie d'attaques particulière nommée *les attaques physiques*. Depuis 1996, date de leur divulgation dans le domaine public, ces techniques d'attaques se sont diversifiées et continuellement améliorées donnant notamment lieu à de nombreuses publications et brevets.

Nous présentons dans les travaux qui suivent, de nouvelles techniques d'attaques physiques que nous avons pu valider et tester de manières théorique et pratique. Nous introduirons des techniques d'attaques par canaux auxiliaires innovantes tirant parti au maximum de l'information fournie par une seule execution d'un calcul cryptographique. Nous détaillerons également de nouvelles attaques CoCo (Collision Correlation) appliquées à deux des standards cryptographiques les plus utilisés: l'AES et le RSA. Nous utiliserons les techniques d'injection de fautes pour monter de nouvelles attaques combinées sur des implémentations de l'AES et du RSA.

Nous introduirons ensuite des méthodes de génération de nombres premiers dites générations prouvées qui s'avèrent efficaces et propices à un usage dans des composants de type carte à puce. Et enfin nous conclurons ce mémoire par la première méthode d'exponentiation sécurisée *Carré Toujours*.

Mots clefs: cryptographie embarquée, analyse par canaux auxiliaires, analyse par injection de fautes, analyse horizontale, nombres premiers prouvés, PACA et ROSETTA.

Abstract

Cryptography has became a very common term in our daily life even for those that are not practising this science. It can represent today an efficient shield that prevent us from hackers' or other non respectable entities' intrusions in our privacy. Cryptography can protect the personal data we store on many physical numerical supports or even cloudy ones for the most intrepid people. However a secure usage of cryptography is also necessary. Cryptographic algorithms must be implemented such that they contain the right protections to defeat the category of *physical attacks*. Since the first article has been presented on this subject in 1996, different attack improvements, new attack paths and countermeasures have been published and patented.

We present in the next pages the results we have obtained during the PhD. New physical attacks are presented with practical results. We are detailing innovative side-channel attacks that take advantage of all the leakage information present in a single execution trace of the cryptographic algorithm. We also present two new CoCo (Collision Correlation) attacks that target first order protected implementations of AES and RSA algorithms. We are in the next sections using fault-injection techniques to design new combined attacks on different state of the art secure implementation of AES and RSA.

Later we present new provable prime number generation method well suited to embedded products. We show these new methods can lead to faster implementations than the probabilistic ones commonly used in standard products. Finally we conclude this report with the secure exponentiation method we named *Square Always*.

Keywords: embedded cryptography, side-channel analysis, fault injection analysis, horizontal attacks, provable prime numbers, PACA and ROSETTA.

Remerciements

Je remercie Christophe Clavier, mon directeur de thèse et ami hors pair sans qui je n'aurais jamais osé franchir ce pas.

Merci également aux membres du jury qui ont accepté d'évaluer mon travail et tout particulièrement un grand merci à mes rapporteurs Louis Goubin et Colin Walter. Merci à Jean-Christophe Courrège, Jean-Francois Dhem, Jean-Louis Lanet, Pascal Paillier et Guillaume Poupard.

Cette thèse représente le travail réalisé durant ces trois dernières années. Elle n'aurait cependant pas été la même sans les travaux que j'ai pu mener auparavant lors de mes différentes expériences professionnelles. Je remercie donc mes anciens collègues d'Oberthur, Thales, Gemalto et Inside Secure pour les discussions enrichissantes et amusantes que j'ai pu avoir avec eux. Merci également à mes nouveaux collègues chez UL pour cette nouvelle aventure ainsi que les discussions et l'expertise qu'ils me partagent au quotidien.

Merci à mes co-auteurs !

Merci à l'Aveyronnais de Toulouse, au Corrèzien du Nord à la Pomme, à l'Anglais du Périgord et sa tequila piment et à Sir Blue-Blood de Maury Boulaouane. Merci à Mylène, Vincent et Georges pour ce qu'on a réalisé ensembles pendant ces années extraordinaires rythmées par votre bonne humeur et votre chaleur humaine (et certes agrémentées de quelques viennoiseries et autres patisseries). Merci aux célèbres et illustres NAG. Merci à mes amis.

Enfin et surtout je remercie ma famille. Plus particulièrement mes parents pour toutes les choses inestimables qu'ils m'ont apprises et données et qui me servent au quotidien; ainsi que mon épouse Camille pour son amour et pour notre merveilleuse fille Faustine, le soutien et la patience qu'elle a déployée pour m'accompagner et me supporter durant ces années de thèse. Merci à ma fille Faustine d'avoir rapidement fait ses nuits.

Mouh.

à Camille et Faustine.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

When Cryptology meets Physical Resistance

The price of freedom is eternal vigilance.
Thomas Jefferson.

This chapter introduces the Cryptology, often said the *science of secret*, and the *physical resistance* of products like smart cards or smart-phones.
It is dedicated to my family and friends. It is written in English (this chapter) and French (next chapter) languages. They have so many times asked me what my job consisted in that I decided my PhD was the occasion to answer once for all to this question.

1.1 Cryptology - The Science of Secret

Cryptology can be split in two activities: cryptography and cryptanalysis. First one consists in building algorithms and techniques to protect messages in confidentiality from unauthorized persons. Second term regroups the techniques that target to defeat the cryptographic techniques and recover encrypted messages without knowing the secret part (most of the time the secret is the key) of the method.

What is the exact definition of Cryptology? A definition is given in the Handbook of Applied Cryptography [**?**]: *Cryptography is the study of mathematical techniques related to aspects of information security, such as confidentiality, data integrity, entity authentication and data origin authentication.*

This sentence summarizes the services offered by the cryptography:

- Confidentiality: gives assurance that only authorized persons have access to information. In the digital word confidentiality can be obtained thanks to encryption methods. For instance Alice and Bob encrypt data they want to exchange on a insecure network (i.e. internet) by sharing a secret they use with an encryption algorithm.

- Integrity: gives assurance that information has not be modified by an unauthorized person. It guaranties the data has not been modified by an attacker during the transmission.
- Authentication of data: gives assurance on the origin of the data. It allows the receiver of the data to know its origin and its author(s). Data authentication also provides data integrity.
- Identification: identification is synonym of *entity authentication*. Identification of a first person A gives assurance to a second person B the first person he is dealing with is effectively A.
- Signature: gives assurance on the non repudiation of the document signed. It prevents the signer entity to deny he is at the origin of the signed document and to refuse to complete the engagements he has signed. It is much stronger than the former and commonly used paper signature. A properly implemented digital signature cannot be reproduced by anyone else than the unique owner of the secret key involved in this mathematical operation.

Although cryptology was initially mainly limited to military activities, it has become in this new century a *key* ingredient to secure our privacy in our more and more connected society.

1.1.1 From Ancient Ages to the Modern Cloudy World

Cryptology has been associated for centuries to military conflicts and strategic games far from the generic concerns of other sciences. The former cryptographic techniques used in the first centuries seem very simple to us. At this time most of them were built without real deep theoretical basis and every new cryptographic technique was broken in short time by the cryptanalyst. This game between clever secret code designers and cryptanalyst became much more serious in the 20th century with the apparition of the computer science. Nowadays cryptology involves many different sciences like mathematics, computer science and micro-electronic.

Our society has changed a lot during the last decade. With the massive development of fast internet and smart multimedia devices like smart-phones the world has become a *connected world*. Communications have become a key element of our daily life. More recently the development of cloud technology and services on the web has contributed to the spreading of our personal informations on millions of servers in the world. Recent scandals on the PRISM activity from the NSA has highlighted the fact that information circulating on these networks are the heart of spy activities.

Confidentiality through encryption, authentication and signature services are efficient solutions to protect our privacy. Cryptography offers these services. For years now these techniques have been used in public domain and are no more limited to spy agencies and military services. Their use will continue to become more and more important in future.

1.1.2 Ancient Ages

Cryptography has been used for centuries. We can consider it was already existing 1900 B.C when some Egyptians were using non "standard" hieroglyph. Another very old example is the ATBASH used by the Hebraic scripts 500 years before J.C. The Atbash uses the alphabet in reverse order to transcript the Jeremiah book. At the same time Greeks were using the scytale to encrypt their messages. It is known as the first military encryption process with the CAESAR one. The Lacedemonian scytale is a simple transposition of the letters of a plain text (message). The classical method consisting in using a cylinder as depicted in figure **??**. The secret key is the diameter of the cylinder. The message could be written on a belt out of leather that has been rolled up on the cylinder. The messenger just wear the belt on the reverse side to bring the encrypted message.

Figure 1.1: Scytale principle.

But the most famous cryptosystem from this period is the CAESAR cipher. It is one of the first military known encryption process and is named after the famous emperor Julius Caesar designed and used it for his correspondence. It consists in replacing the alphabet by the alphabet rotated of k letters. For instance in figures **??** and **??** the key k is equal to 3. Each letter of the alphabet of the plain message is then replaced by the corresponding rotated letter to design the cipher text. The reverse operation is used to decrypt. One can remark here that once the technique is known the number of possible keys is only equal to 25.

This technique can be simply improved to increase the decryption complexity by using the mono-alphabetic substitution. Instead of shifting the alphabet letter it consists in replacing (substituting) a letter by another one. The number of potential new alphabets is then the number of permutations of the 26 alphabet letters, so $Factorial(26)$ that corresponds to about 2^{88} possibilities. It seems this technique was described for the first time in the Kamasutra (before the CAESAR cipher). It was the most used encryption

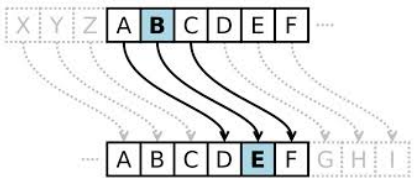


Figure 1.2: CAESAR encryption with key equal to 3 (right rotations)

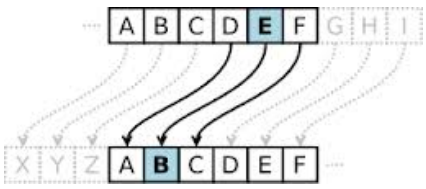


Figure 1.3: CAESAR decryption with key equal to 3 (left rotations)

method during the first millenary. Many scientists from this period were convinced it was unbreakable. At this time cryptosystems were resistant to cryptanalysis. Indeed the mono-alphabetic substitution led to a huge number of possibilities for the secret key (that would be still today a big number for a space of keys).

But finally in the 9th century Arabs succeeded first to break mono-alphabetic encryption. We could say they are at the origins of cryptanalysis.

Abu 'Abd al-Raham al-Khahil ibn Ahmad ibn'Amr ibn Tammam al Farahidi al-Zadi al Yahamadi (Abu-Yusuf Ya'qub ibn Ishaq al-Kindi, called Al-Kindi) published in the 9th century the first book on cryptanalysis. He is the first one to analyse the letter frequency and exploit those results to decrypt messages encrypted with mono-alphabetic substitution. For each language the apparition frequency of letters differs. However each time some letters of the alphabet are more present in texts than others. Those frequencies can be computed and are known. For instance in french the most used letters are "e", then "a", then "s", etc. Therefore by computing for a given ciphertext the frequency of each letter we can guess which one corresponds to "e", "a" and so on. This technique can be also improved by using the frequency of couple of letters.

From this time it became difficult to design "strong" and resistant cryptosystems. Marie Stuart would have may be lived longer if the improved mono-alphabetic substitution cryptosystem she used had been more resistant to cryptanalysis.

Only in the 16th century appeared the first resistant cryptosystem. Blaise de Vigenère designed a clever cryptosystem (named Vigenere) that resisted for 3 centuries to cryptanalysis. He designed a simple and subtle code based on poly-alphabetic (circular) substitution. This technique resisted to cryptanalysts for decades when it was named "The unbreakable cipher". Finally in the 19th century Charles Babbage and Friedrich Wilhelm Kasiski broke it independently in the mid 19's.

These are some of the famous cryptographic methods which have been used at different critical periods of our history in previous centuries. However the latest one, the 20th, has been certainly the most exciting one in term of cryptology's developments. It has been said this was the time of *modern cryptography*.

In between it is interesting to introduce the Enigma machine that was used by Germans during the second world war. It represents one of the most exciting period of cryptography history. The cryptanalysis and new computer science techniques designed at this time have influenced the course of our history.

1.1.3 Enigma

Enigma (cf. figure **??**) [1] was designed by the German engineer Arthur Scherbius in 1919 that makes the first commercial product in 1923. From 1929 it was used by the German army. This is an electro-mechanic machine. The power supply is used to turn rotors when the user press a letter of the "keyboard". Two consecutive similar letters are not substituted with the same letter. It is then a poly-alphabetic substitution. The key was the initial positions to apply to the rotors. During the second world war Germans used it to encrypt sensitive messages, there were more than 30000 machines in usage.

[1] http://www.nsa.gov/about/photo_gallery/gallery.shtml

Before Enigma was used cryptanalysts of many countries succeeded to decrypt German communications. Suddenly with the apparition of Enigma all of them were confronted to the impossibility to decrypt any German cipher text.

Figure 1.4: Enigma

In December 1932, Marian Rejewski, working for the secret services of Poland, succeeded to broke the first version of the Enigma encryption machine with his mathematician colleagues Jerzy Rozicki and Henryk Zygalski. They exploited information transmitted by a french officer. At this time Poland was afraid of German invasion. Rejewski and his team succeeded to decrypt thousands of German messages. They succeeded where all other nations failed.

Before the German invasion happened Polish revealed to British their techniques to break Enigma.

British cryptanalysis activities were located and centralized at the place of Bletchley Park. Many thousands of persons were working here with the unique objective of decrypting German ciphered communications. German cryptographers regularly improved this machine to enhance its security resistance. However, even if sometimes and for short periods Allies failed to decrypt messages, Bletchey park succeeded finally to decrypt the majority of the ciphered messages. The most famous contributor to this success is named Allan Turing. With his colleagues, they designed the famous *Bombe* machines (cf. figure **??**)[2] dedicated to Enigma deciphering computations.

Many battles have been won, many German boats and submarine sunk thanks to the cryptanalysis of Enigma. Enigma has been publicly used by organisations until 1974. They though they were protecting their data as no public weakness has been revealed on Enigma. Indeed the Bletchley Park decryption results were only disclosed in 1974.

[2]http://www.nsa.gov/about/photo_gallery/gallery.shtml

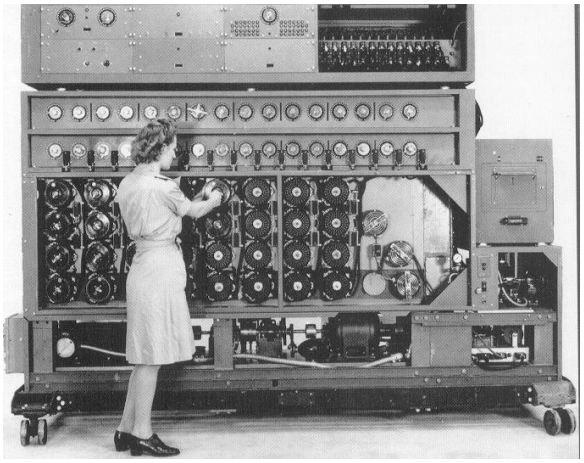


Figure 1.5: Turing Bombe

> "The security of any cryptographic system must not rely on the secrecy of the algorithm used."
>
> A. Kerckhoffs.

1.1.4 The Modern Symmetric Cryptography

The modern symmetric (also said secret-key) cryptography relies on the fundamental principles enounced by Auguste Kerckhoffs [**?**]. One of these most important principles are given in the following:

Symmetric cryptography relies on the fundamental condition that a secret key, a simple sequence of bits, has been initially shared by the communicating people. This secret key is the same secret that is used to encrypt and to decrypt. A conventional symmetric encryption is defined by an algorithm taking as parameter the secret key. It has the property to transform a plain text to a cipher text such that the reversed calculation is easy to who knows the key but unfeasible to others.

Figure **??** illustrates this principle of the symmetric cryptography. Alice and Bob want to share a confidential information (message) m through an insecure channel such as internet for instance. Hence they need initially to share a secret key K with a given symmetric algorithm using K: E_K. Once they have done this key exchange the sender, Alice for instance, can use E_K to encrypt any message m she wants to transmit. The ciphered message $C = E_K(m)$ can then be send to Bob that use the reverse algorithm D_k with the secret K to recover $m = D_K(C)$ from the encrypted message C. This is summarized by figure **??**.

A good encryption algorithm requires the two following properties enounced by Shannon in 1949:

> "Security must rely on the secrecy of the secret key used."
>
> A. Kerckhoffs.

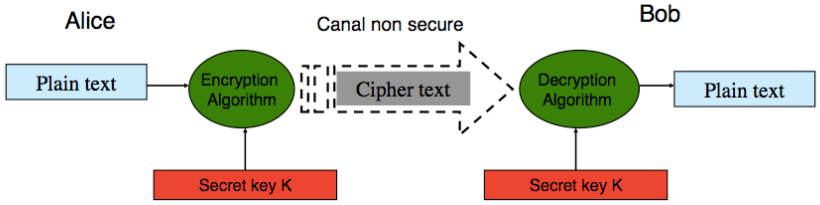


Figure 1.6: Symmetric encryption principle.

- diffusion: 1 bit of the plain text modified will modify the many bits of the cipher text,
- confusion: it must be difficult to obtain statistics on the plain text from statistics of the cipher text.

The most famous symmetric encryption is the Vernam's one. Vernam designed and patented in 1917 the *one-time cipher* also said the *One Time Pad*. It is still today the only cryptosystem with perfect secrecy as it was proven in 1949 by Shannon. However it is not really practical due to the big key length involved in the operations. Indeed it necessitates the key to be as long as the plain text. Moreover a fresh key must be used at each encryption operation.

Symmetric encryption modes includes two cipher families: the stream ciphers and the block ciphers. Stream ciphers allow encryption bit per bit of the plain text and then it can be used for "on the fly" encryption operations. It is the case of the Vernam encryption. Block ciphers encrypt plain texts per block of bits. They are the most used algorithms today.

We are focusing in the rest of this document on block cipher algorithms. Block ciphers allow fast encryption that is necessary for encrypting big data quantities. Another requirement is to have a sufficiently big enough key but small if possible. For instance 128-bit keys are strong enough today (of course the algorithm structure must also be good enough to resist cryptanalysis techniques). It is opposed to the 2048-bit keys that are used in asymmetric cryptography or at least with the huge key that would be necessary for the Vernam cipher.

The major drawback is however the key management that has to be put in place for such encryption. Indeed for each couple of persons willing to communicate encrypted a secret key must be generated and shared. For instance let's consider a 100 persons group where people need to communicate securely two per two. Then each couple of persons needs a unique secret key. In this example 4950 different keys are necessary. As we can see it can quickly lead to a huge number of secret keys.

Many block ciphers have been designed for years, some have been broken (cryptanalysed) when others are still resistant like the DES algorithm in his triple-DES (TDES) operation mode only. Of course, because of the technological evolutions, computational power is increasing every year. It is then necessary to increase key-length of algorithms regularly to ensure techniques remain resistant. Common resistant block cipher have key sizes varying from 112 to 256 bits.

Most famous and used algorithm is the DES or TDES (in its 2 or 3 key version). This algorithm is present in millions of daily life products. You can find it in your bank cards, your mobile or smart phones, your computers and so on... The second one is the AES. It has been selected as Advanced Encryption Standard by the NIST in 2001 [**?**] to progressively replace the (T)DES. We are presenting TDES and AES in the following. Many other algorithms exist but we do not list and detail them in this introduction.

DES Algorithm

This is the most famous symmetric algorithm. DES stands for Data Encryption Standard. It has been selected as international standard by the NIST in January 1977 [**?**]. DES was initially published in 1975 after some modifications made by the NSA on the initial version, named Lucifer and designed by IBM. Since 1977 it has been maintained as standard at many occasions and has resisted for years to many attacks. Due to technology evolutions it has been recommended for use in 2004 in the triple DES form. In this stronger version 2 or 3 keys of 56-bits are necessary. Indeed DES is using a 56-bit key. Such a key length is weak today as recovering a DES key would require "only" 2^{56} DES computations that is achievable today in less than 1 day for less than 50 000 euros of investment. It is then mandatory to use the DES in the triple-DES version as depicted in figure **??**.

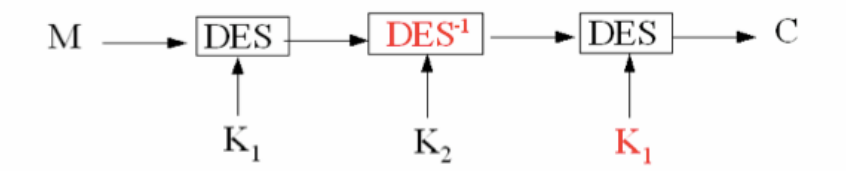


Figure 1.7: Triple-DES principle.

DES is based on the famous Feistel scheme [**?**] that is described with figure **??** for a single Feistel round and figure **??** when many consecutive rounds are chained.

A Feistel round transforms his entry (L_{i-1}, R_{i-1}) to the output (L_i, R_i) by applying the following operations:

- $L_i = R_{i-1}$
- $R_i = L_{i-1} \oplus f_{K_i}(R_{i-1})$

The left part of the output is simply obtained by copying the input right part. The output right part is the result of the XOR operation between the input left part L_{i-1}

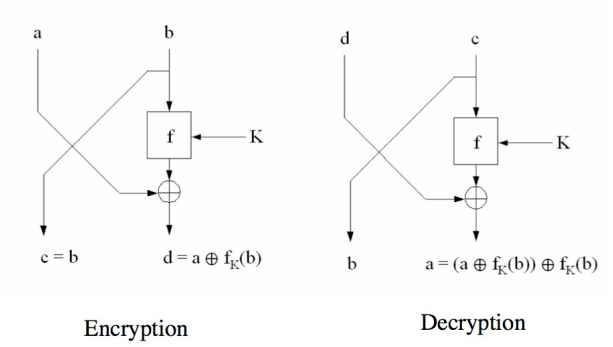


Figure 1.8: Feistel scheme single round.

and the transformation $f_{K_i}(R_{i-1})$ of the right input by a bijective function f. The round function takes as parameters the round keys K_i that are derived from the initial key K.

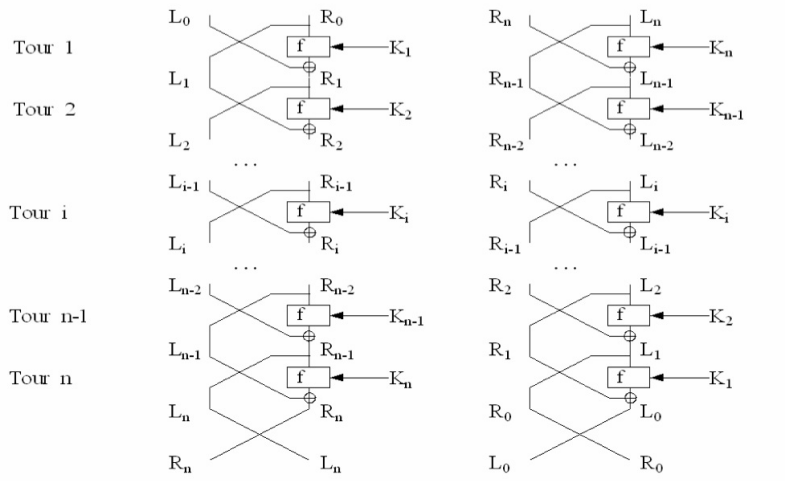


Figure 1.9: Feistel scheme consecutive rounds (said *Tour* here).

DES is based on this scheme. It takes as input parameter a 64-bit key K and a 64-bit plain text M. Each byte of the key containing a parity bit, the actual key size is 56 bits. The number of Feistel's rounds equals 16. At the beginning the plain text is transformed through the *Initial Permutation IP* and split in a 32-left part L_0 and a 32-bit right part R_0 from the most to the least significant bits of $IP(M)$.

The function f is non linear and is made of a substitution, two permutations and a round key addition (XOR). At each round i the round key K_i is obtained from the main key K following a key schedule depicted in figure **??**

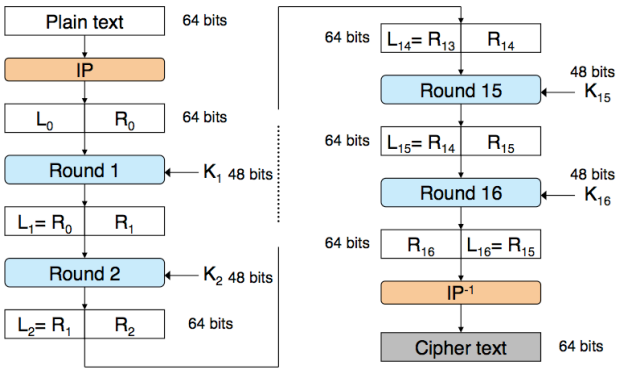


Figure 1.10: DES encryption principle.

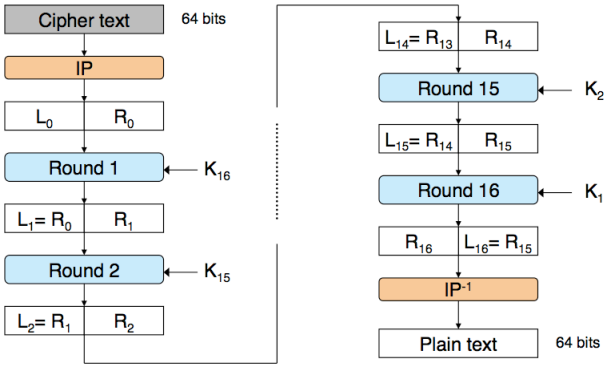


Figure 1.11: DES decryption principle.

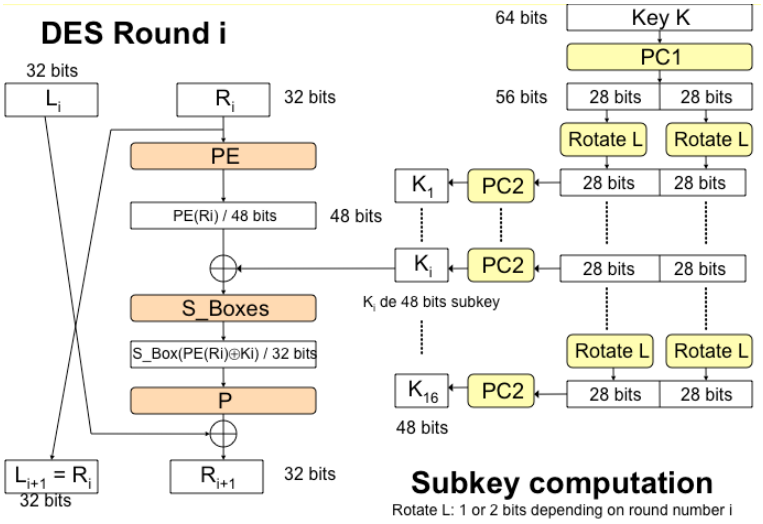


Figure 1.12: DES Round detail.

DES security The security of DES has been scrutinized for three decades. Single DES is weak today for many reasons. The first one is the size of the key. Contrarily to the period it was designed, 56-bit key does not offer enough resistance today to brute force attacks. Trying exhaustive search on all 2^{56} possible keys at a given plaintext/ciphertext pair is feasible for a reasonable cost. Brute force really started in 1997 with the DESCHALL project led by Rocke Verser, Matt Curtin and Justin Dolske. They used idle cycles of thousands of computers accross the internet to broke a message encrypted with DES. They earned $10000 offered by the RSA Security company for the contest. In 1998 a custom DES cracker machine showed it was possible to recover a DES key in only two days. The EFF's[3] $250000 machine contained 1536 custom chips and could brute force a DES key in a matter of days. Seven years later an equivalent machine named COPACOBANA [**?**] was built by Bochum and Kiev universities for less than $10000. It was made of 120 low cost commercial FPGAs. COPACOBANA can brute force a DES key in an average key search of 8.7 days.

Before these brute force practical results, new cryptanalysis techniques have been designed during the DES security studies. The *differential cryptanalysis* was published in 1990 by Eli Biham and Adi Shamir [**?**]. Their analysis required 2^{47} chosen plain texts to recover the full DES key. Their study also highlighted that DES was designed to be resistant to this technique. It made people thinking that NSA and IBM already knew this technique at this time as it has been affirmed by Don Coppersmith [**?**].

In 1993 Mitsuru Matsui discovered the *Linear cryptanalysis* [**?**]. His technique needed 2^{43} known plain texts. It was implemented and was the first experimental cryptanalysis of DES to be reported. There is no evidence that DES was tailored to be resistant to this type if attack.

Today DES must not be used in his single encryption mode. Most of the applications are using it in his triple mode version where it is still a standard. However another standard has been selected by the NIST in 2001: the AES.

AES Algorithm

The Advanced Encryption Standard has been adopted in 2001 by the NIST after the call for candidates this institute had made in 1997. Fifteen candidates were initially proposed by the worldwide scientific community. After a first selection phase, where some candidates were broken and others saw their security reduced, five candidates remained for the second phase. They were MARS, RC6, Rijndael, Serpent, and Twofish. In October 2001 the Rijndael was selected as the new standard to become the AES [**?**]. It was designed by two Belgians named Vincent Daemen and Joan Rijmen. When the DES is not working in a particular mathematical field the AES is processing on elements of the Galois Field GF(2^8). It is not based on the Feistel scheme but more generally it is a Substitution Permutation Network (SPN). This block cipher encrypts 128-bit blocks of message. Three modes in AES are available for different security levels:

- AES-128 with 10 rounds and a 128-bit key,

[3] EFF=Electronic Frontier Fondation, a cyberspace civil rights group

- AES-192 with 12 rounds and a 192-bit key,
- AES-256 with 14 rounds and a 256-bit key.

It must be noticed that the AES is a subset of the possible configurations of the Rijndael algorithm. Indeed Rijndael offered the possibility to encrypt 128 or 192 or 256-bit of plaintext. The NIST has removed the 192 and 256-bit sizes for the plaintext, AES can only encrypt 128-bit plaintext messages.

The operations of this algorithm are mathematical operations in the field GF(2^8). This mathematical structure gives many implementation possibilities. It is then very useful to implement protections and countermeasures against side-channel attacks we will see later. It means an element can be represented in a normal basis or a polynomial basis. If we consider the polynomial representation each byte must be seen as a polynomial. As $GF(2^8)$ is a field of characteristic 2 it makes the addition operation easy to implement as a simple eXclusive OR (XOR). The multiplication corresponds to a polynomial multiplication modulo $F(X)$ where $F(X)$ is $X^8 + X^4 + X^3 + X + 1$ that is the irreducible polynomial used for the field definition.

The operations are performed on two dimensional array of bytes called the *State*. The 16-byte input plain text ($in_0 = m_0, in_1 = m_1, ..., in_{15} = m_{15}$) is transposed in the *state* configuration. Then the operations are processed on the *state* array until the final result is obtained. This is described by figure **??**.

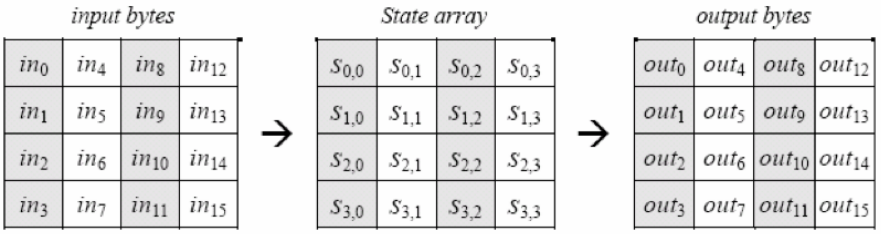


Figure 1.13: AES State array.

AES is composed of five operations if we include the *Key Schedule* as one. Each of them is part of each round except in the final one the MixColumn operation has been removed. These are listed in the following:

- KeySchedule: from the initial master key K, a round key K_i is derived to be used in the key addition at each round i,
- AddRoundKey: it adds the round key value of the current round to the current state array,
- ShifRows: it permutes the byte of the state,
- SubBytes: it is the non linear part of the algorithm. Using a look-up table it simply consists in replacing a byte by the corresponding one in the SubBytes table.

Mathematically it is the composition of the pseudo-inverse operation in $GF(2^8)$ with an affine transformation.

- MixColumn: the most time consuming operation; it corresponds to the multiplication of each row of the state by a degree 3 polynomial in $GF(2^8)$.

An overview of the AES is given in figure **??**.

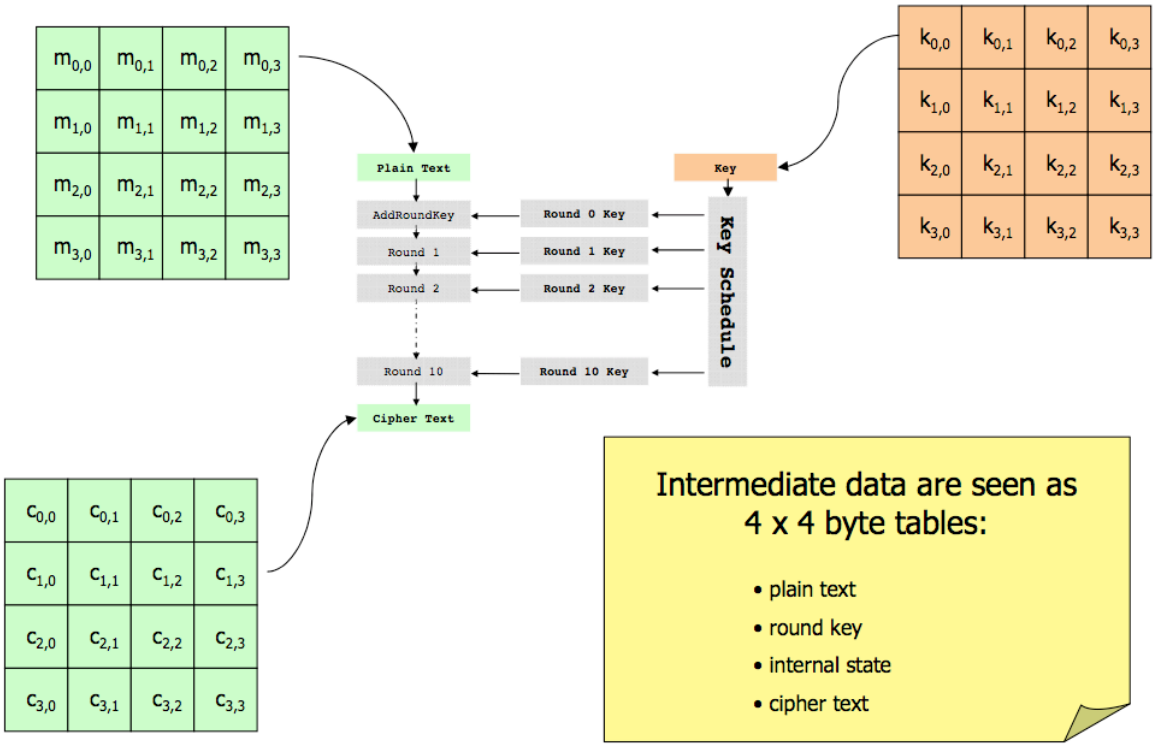


Figure 1.14: AES data structure.

AES security AES has been designed to be resistant to Differential and Linear cryptanalysis.

TDES and AES are the most used symmetric algorithms today. The other final candidates MARS, RC6, SERPENT and TWOFISH are also recommended by NIST as secure algorithms. So there are nowadays many symmetric algorithms that offer fast and secure encryption methods.

However the main drawback of this method is that they require a secret key to be shared between two persons or the members a group of persons. Such key values have to be generated and exchanged before any communication could happen.

Another issue is that signature with non repudiation property are not possible with secret key algorithm. Indeed as the key is shared between two persons at least, it is not

possible to legally prove who has encrypt (signed) what.

To solve these problems, here comes the asymmetric cryptography.

1.1.5 Asymmetric Encryption

This area in the science of secret is very recent. It appeared in 1976. It created a breakthrough in the domain because of the new possibilities it offered. There are two fundamental articles [**?**, **?**].

In both cases the security relies on difficult mathematical problems on long integers: the discrete logarithm problem an the factorization of large integers.

Two difficult mathematical problems

We need in this paragraph to remind the reader what a prime number is.
A *prime* number is a number that is only divisible by 1 and by itself. For instance 2, 3, 5, 7, 65537 are prime numbers. $6 = 2.3$ is not prime.
The public key cryptography is based on the two following difficult mathematical problems.

The factorisation of large integers Consider a big integer value N long of many hundreds of bits (for instance 4096 bits) that is not prime. It's prime factorization is given by $N = p_1 \times p_2 \times \ldots \times p_\ell$ where each p_i is a prime number. For instance the prime decomposition (or factorisation) of 165 is $3 \times 5 \times 11$. The factorisation of 435959 is 547×797. It can easily be done by testing if any number lower to $\sqrt{N}$ divides N. This is the method known as *the sieve of Erastothenes*. But for large numbers it becomes difficult to use such an exhaustive search method. For instance let's consider the following number N:

$N =$
587550236494541478611353138138578005580467762975368463260434976974827347
298863536769702787504358685623450049651464713011105781834657180993917147
795455354319871288463981950412427844745210882196662444451496593704726012
842114427229399542508112651032937251896150838667012906416143534775658474
962257629444310328863641190139357294013078175893340907911676471004226775
314027700238652460322736506279420652528325927819898091797669696076153293
517121898320098306918494234607501965814599521826364630359302267452207241
259184968025844763409493111542369543640881930909114859769081139704680895
741606209683318352990679791906988517067091838982195005818674928151024779
465435454043887613239322575517697128252333590544804532997861726426790342
951218411719461586767769098362520252456939340195279712558343115464915678
615011719542995067793822166076311289149218256749448785510068590506532536
247956724632173820166413319724970565483027969935938175229014362018284558
441113924560532083148647586219397756338884075799242976697423815060755726
319295762837214405179055800023960367088566816087500266222103509654668441
479561825577503878029848567733258508012046001848565070355842154160814816
379752561379604641078225075724022799050477051865207506694881194342567749

101012307

This is a 2048-bit value. The most efficient factorization method can not factorize such a big number as it is composed of the following big prime values p_1 and p_2.

$p_1 =$
31477254128149999252882431464122026131062398267816829757696248451605257182595343437889213643790936770884609238243607356121802426258728265951840063451799792741577696315495986376136574561028899851214854668503318545134321169841136493143839389504257093276823503015785711401900427539491637520589031361763214844396761236324787101772104227433523068550592134532758287701333078515694458831216910370490395453409073567178526675516855577392025342707639963173544397275550854330078091590137349123817512805138696047064022544513284156814307719904177669407542986737480724899666736310586765481726908531755888095687788392063183805457389

$p_2 =$
18665866917822966610681385143632164119897109715992194909806450736375688279856143839473805681104444445663188247693305441111149271214381137505342570528842528625369778778713093251624167470408020705657561268484785179432239435811229635623339962583123656595091617960569541371229058215837331221650955057831317005063133250968865336952253309324090237731010603499836957961507319140650170629338121186309591698850305561181832523760979501958192571151856747960952676586476324558348730028728836888798056966147680674100312292550852159009001408572732634331952673041036569708104152997303361348058677719200047993223805912972693769971263

This illustrates the difficulty to factorize large integer values.

Amongst the different factorization methods the most efficient today are the Elliptic Curve Method (ECM) from Lenstra [**?**] and the Number Field Sieve methods (NFS) from A. Lenstra, H. Lenstra, M. Manasse and J. Pollard [**?**]. ECM can recover small prime factor p_i of a large integer value in shorter time than NFS but it cannot recover as large value as NFS does. For instance ECM biggest factor recovered today is a 83-bit integer[4]. Today the largest integer (factored with NFS algorithm) is the RSA 768-bit (232 digits) value; This value is composed of two same lengths prime factors from the RSA's challenge list. It was achieved in December 2009 by Kleinjung et al. [**?**].

The Discrete Logarithm Problem Let's consider the following parameters: p a prime value, $\alpha < p$ is a generator element of $(Z/pZ)^\star$ and β in $(Z/pZ)^\star$. Recovering the value $0 \leqslant x \leqslant p-1$ such that $\beta = \alpha^x \bmod p$ is difficult.

Diffie-Hellman - DH

The main drawback of the symmetric encryption is the following: Alice and Bob, far from each others, want to exchange sensitive information on internet. In order to protect

[4]http://www.loria.fr/ zimmerma/records/top50.html

this data in confidentiality they want to encrypt it using for instance the AES algorithm. It means they need to share a secret key K to be used with AES.

The question is (and has been for years): "how can they create such a key and be sure the transmitted information on the channel are not exploitable by an attacker?". The solution is given by the Diffie-Hellman key exchange protocol. But let us explain before the notion of public an secret key. Each user owns a key pair made of a public

key A and a private key a. The public key value is shared (on internet for instance) in order to make it available to everyone. The knowledge of the public key A is sufficient to encrypt a message but does not allow to recover the private key a. The decryption of a message requires the knowledge of the private key a to be possible. With this scheme anyone can send encrypted messages to the public key owner. Being the private key unique owner he is the only one able to decrypt the messages encrypted with his public key. Then, to summarize, each person (for instance here Alice) owns a couple of keys (A,a) such that:

- a public key A to encrypt: everybody can use it to encrypt a message to Alice.
- a secret key a to decrypt: only the secret key owner can decrypt the cipher text encrypted with the public key.

It is illustrated in figure **??**. Here the encryption key is different from the decryption key.

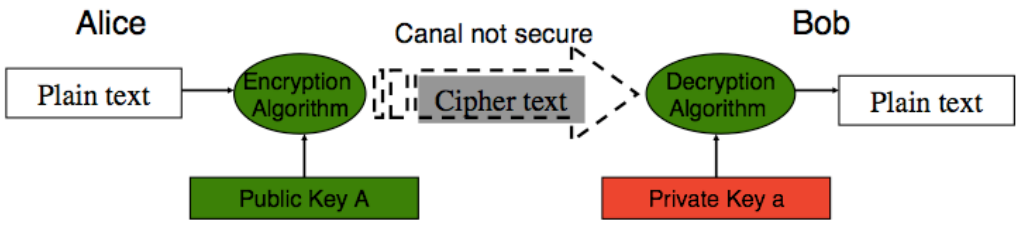


Figure 1.15: Asymmetric encryption principle.

This concept was introduced in 1976 by W. Diffie and M. Hellman in *New directions in Cryptography* [**?**]. They did not succeed to design such a public key encryption system. However they invented the key exchange mechanism based on the mathematical discrete logarithm problem.

For instance, we give in the following an example to illustrate this protocol.

Alice Bidochon has gone in Tazmanie to visit her family. She has discovered a secret information on his family that could make them very rich... she is convinced! She wants to communicate this very important information to her husband Bob Bidochon who stayed in France. (They have of course chosen to use secret names for a better anonymity). Alice decides to use the Diffie-Hellman cryptosystem to exchange a secret key to encrypt this secret message. Alice and Bob proceed as depicted in figure **??**.

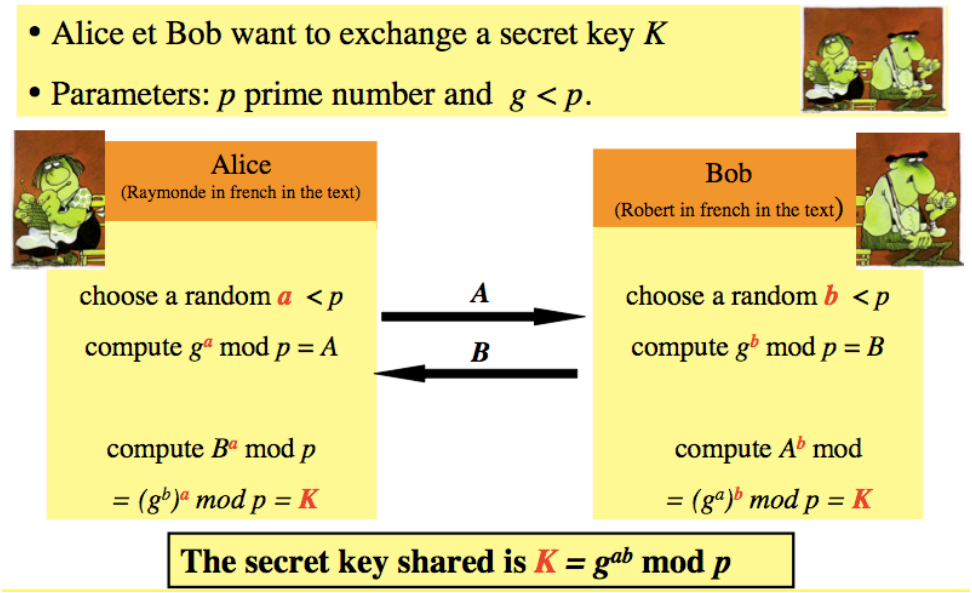


Figure 1.16: Diffie-Hellman key exchange.

1. They use as parameters a long prime integer p and an integer $g < p$.
2. Alice takes a random integer $a < p$
3. She computes $A = g^a \bmod p$
4. She sends the value A to Bob(ert).
5. Bob, on his side, takes a random integer $b < p$
6. He computes $B = g^b \bmod p$
7. He sends B to Alice.
8. By computing $B^a \bmod p$ Alice obtains a value $K = g^{ab} \bmod p$
9. By computing $A^b \bmod p$ Bob obtains the same value $K = g^{ba} \bmod p$

Alice and Bob Bidochon are now sharing a secret key K they can use to exchange encrypted information with a symmetric algorithm. As one can see A and B are public information as sent through a insecure network. But knowing A and B does not allow to recover neither a, b and ab.

This key exchange scheme is very used today.

Rivest-Shamir-Adleman for RSA

In 1977 R. Rivest, A. Shamir and L. Adleman presented their public encryption scheme named RSA [**?**]. It is the first practical and concrete public key cryptography encryption

scheme presented. Its security relies on the difficulty to factorize big integers. The principle of RSA is given in figure **??**.

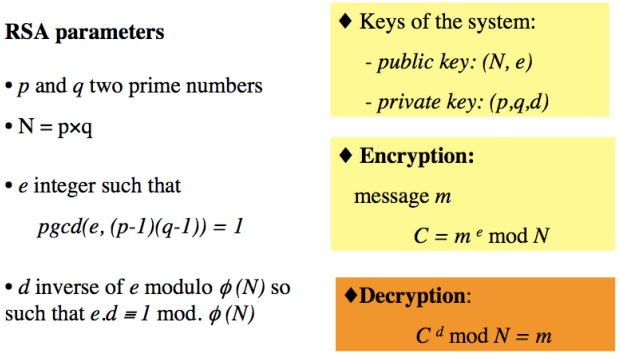


Figure 1.17: RSA Encryption scheme

If we take the previous example, Alice wants to send to Bob(ert) the secret information m she discovered on her family in Tazmanie. She decides to use the public key (N,e) of her husband. She encrypts the secret m by computing $C = m^e \bmod N$ and sends this value C to Bob.

Bob receives the encrypted message C and decrypts it by using the secret key he is the unique owner by computing: $m = C^d \bmod N$.

Explanation:
$C = m^e \bmod N$
The decryption calculates:
$D = C^d \bmod N$
$D = m^{e.d} \bmod N$
$D = m^{1+k.\varphi(N)} \bmod N$ [5]
$D = m^1.1^k \bmod N$
$D = m$

The RSA cryptographic system security relies on the difficulty to recover the secret key d from the public exponent e and the modulus N. Being able to decipher a cipher text is called the RSA problem: *from (n,e) and C, compute* $C^{(1/e)} \bmod N$. The RSA problem and the factorization problem are strongly related. RSA problem may be easier to solve than factorization but this question has been investigated for 35 years now and it is still an open problem...

The Non Secret Encryption - NSE

The story of J. Ellis, C. Cocks and M. Williamson has been revealed by J. Ellis in 1987 with the authorization from the British secret services (CESG). In 1969 J. Ellis was working for the British Government Communication Headquarters (GCHQ). He

[5] $\varphi(N)$ is the order of the multiplicative group $Z/NZ^\star$, then $u^{\varphi(N)} = 1$ for any u in the group.

established the basis of what would be, to his mind, the Non Secret Encryption in [**?**]. But he did not succeed in designing a functional cryptographic system. In 1973 C. Cocks worked on Ellis idea and designed a cryptosystem pretty similar to RSA in [**?**]. He exposed his idea also to M. Williamson. Williamson was trying to cryptanalyse the Cocks system when he discovered the Diffie-Hellman key exchange cryptosystem. In 1975 they had then discovered the fundamentals of Public key cryptography. But this was kept secret by the military service. NSE was the name they used for what is now called PKC.

Cocks Algorithm Alice chooses two large primes p and q and sends $N = p \times q$ to Bob. Bob encrypts his message m by calculating $C = m^N \bmod N$ and sends C to Alice. To decipher Alice finds p' and q' such that:
$p.p' = 1 \bmod (p-1)$
$q.q' = 1 \bmod (q-1)$
then you decrypt by computing:
$m \bmod p = C^{p'} \bmod p$
$m \bmod q = C^{q'} \bmod q$
and you can then recover m.

It is interesting to notice that this NSE scheme is a particular case of RSA for the public exponent being set to N. There are few differences (it is not the same algorithm to encrypt than to decrypt) but the NSE scheme from Cocks could then be the first public key encryption algorithm.

Williamson Schemes In 1974 M. Williamson presented to J. Ellis an encryption scheme that is the following: Alice and Bob share a public prime value, say p. Alice wants to send encrypted a message m to Bob.
Alice chooses a large number a, and sends $A = m^a \bmod p$,
Bob chooses a large number b and sends $A_B = A^b \bmod p$ to Alice. Here the message is encrypted so that $A_B = m^{ab} \bmod p$.
Alice computes a' such that $a.a' = 1 \bmod (p-1)$, and $C = A_B^{a'} \bmod p$ to remove her original encipherment and lets the message encrypted by Bob's key. She sends then $C = m^b \bmod (p-1)$ to Bob.
Bob receives C. He computes b' such that $b.b' = 1 \bmod (p-1)$, and removes his encipherment by computing $C^{b'} \bmod p = m$. It was reported secretly in [**?**].

This idea corresponds also to what is known in cryptography as the *double lock* process. Alice and Bob want to share a secret. Alice puts the secret in a box, she locks it and sends to bob. Bob locks the box again with his lock and sends it back to Alice. Alice removes her lock and sent it again to Bob. Then Bob can unlock and open the box to get the secret.

Later Williamson designed a simpler method using exponentiations [**?**]. This idea corresponds to the Diffie-Hellman protocol.

ElGamal

It is the first asymmetric encryption scheme based on the discrete logarithm problem presented in 1983 by T. ElGamal [**?**].

Parameters: g, q such that g is a generator element of the cyclic group G of order q.
Key generation: Alice chooses randomly a value $a \in 1 \dots q-1$ and calculates $A = g^a \bmod q$.
She discloses her public key (G, g, q, A) and keeps secret the private key a.

Encryption: Bob wants to encrypt the message m to send it to Alice. He processes as detailed in the following:
He generates a random value $r \in 1 \dots q-1$ and computes $c_1 = g^r \bmod q$
Computes $s = A^r \bmod q$
Compute $c_2 = m \cdot s \bmod q$
Bob sends the cipher text (c_1, c_2) to Alice.

Decryption: Alice has received (c_1, c_2) from Bob that she wants to decrypt.
Alice computes $w = c_1^a \bmod q$
She computes $u = w^{-1} \bmod q$
and $v = c_2 \cdot u \bmod q$
At this step $v = m$ the original plain text from Bob.

Explanation
$v = c_2 \cdot u$
$v = (m \cdot A^r) \cdot ((g^r)^a)^{-1})$
$v = (m \cdot (g^a)^r) \cdot (g^r)^{-a})$
$v = m$

Practical Concerns

As it can be seen asymmetric cryptography involves calculations on long integers as modular exponentiation which is much time consuming. A first practical solution consists then in using the DH key exchange to share a symmetric key K. This secret key is then used to exchange encrypted information using a symmetric algorithm like TDES or AES that allows fast encryption operations compared to RSA. This solution is very used in practice in many daily life products.

Another key concern of the public key cryptography is the public key certification. Indeed any individual can generate a pair (private a, public A) key and spread the public key A to anybody. But the problem is: "How can I be certain the public key I use belongs to the right person I want to communicate with?". Another concern is the famous attack known as "the man in the middle attack". Alice decides to send by email her public key A_{Alice} to Bob. But she doesn't know that Oscar has hacked her mail account and can intercepts all her emails. Then Oscar get the public key from Alice and send to Bob his public key A_{Oscar} instead of the one from Alice. Bob receives the public key and thinking it was the one from Alice use it to encrypt the message m he wanted to send to her. Oscar intercepts the mail, decrypts it with his private key, reads it and re-encrypts it with the public key from Alice. Neither Alice nor Bob will know the message content is known by Oscar. The key certification solves all these issues. It consists in using

a trusted third party that signs all the keys with her secret key. Then each user can verify the key he uses has been certified by the known third party by verifying the key signature. The public keys management in cryptographic systems defines a Public Key Infrastructure PKI.

If asymmetric schemes are more practical for key exchange than for encryption purposes they become mandatory and are widely used for data signature mechanisms.

1.1.6 Digital Signature and Authentication

The signature permits to authenticate a given message has been sent by a precise emitter. For centuries the signature of any document was the manuscript signature. It is still the most signature used to today. However it is something very easy to reproduce. A more modern and efficient mechanism of our recent world is the digital signature. It is now possible thanks to the Public Key Cryptography. The process is similar to encryption but this time the private key is used to sign a message (and not to decrypt).

The authentication of a person must permit to authenticate a person is the right one. Even today to prove your identity you have to show your identity card (paper), your driving licence card (paper), etc. It is worth to notice that more and more some of these documents are becoming electronic documents but the paper part is still the most used today.

A signature algorithm must respect many properties. The signer must easily be able to produce a signature S for a message m when s must be easily verifiable by anyone. However the verification capability must not allow the verifier to reproduce such a signature value.

The second fundamental property of the signature is the *non repudiability* property. As Alice is the unique owner of the private key she is the only person able to generate the signature s of a given message m. It signifies that Alice cannot deny she has signed this message m.

As it is depicted in figure **??** Alice, the owner of the secret key, is the only person capable to sign a message m to produce the signature value S. But everyone can check with the public key that S is the signature of m by Alice. However the knowledge of the public key A does not allow to reproduce a signature. The requested properties for signature are then fulfilled with the asymmetric cryptography.

RSA and ElGamal schemes can be used to generate digital signatures with respects to the process described in figure **??**. Another method is the Digital Signature Algorithm DSA.

DSA

The Digital Signature Algorithm has been adopted as Digital Signature Standard by the NIST in 1994 [**?**]. It is a variant of the ElGamal signature scheme. It requires the use of the hash function SHA-1 (or SHA-2). Its security relies on the Discrete Logarithm problem. DSA can only be used for signature contrarily to RSA which can be used for

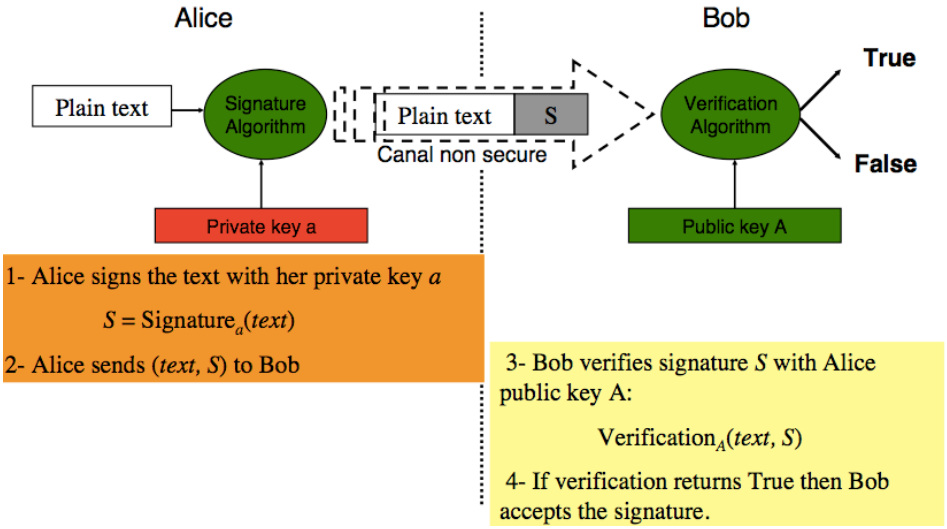


Figure 1.18: Signature principle.

encryption and signature objectives. DSA is not a message recovery function. It means the signature verification will answer 'true: signature is verified' or 'false: signature is refused' but without leading to the recovery of the signed message.

Zero Knowledge

A similar problem to the signature is the person authentication. Electronic authentication is replacing the former authentication with your paper ID document. It requires new authentication mechanisms. The notion of *zero knowledge proof of knowledge* allows a first party to prove its identity to a second party via some secret information... but without leaking any information on its secret to this second person!

This concept of Zero-knowledge was introduced in 1985 by Goldwasser, Micali and Rackoff in [**?**]. They surprisingly showed it was possible to proof a second party your are the owner of a secret without disclosing any information about this secret.

A nice explanation has been given in 1989 by Quisquater and Guillou's families and T. Bergson in the paper titled "How to Explain Zero Knowledge Protocols to Your Children" [**?**].

1.1.7 On the Importance of Random Numbers

Random numbers are the most important element of cryptography. They are required to generate key values, and are at the heart of most of the cryptosystems. Generating random numbers is a science that involves physical and mathematical concepts.

It is then of prior importance to evaluate the quality of random numbers that are generated by a random source in order to decide if they can be used or not for cryptosystem calculations.

Many publications can be found in the literature to design random number generators and to describe statistical tests. Some standards like AIS 20 and AIS31 from BSI [?], FIPS 140-2 [?] and 140-3 (draft) [?] from the NIST describe and specify how to evaluate the security requirements of cryptographically secure random number generations.

Key Lengths

It is obvious the lengths of the cryptographic keys is a fundamental element of the security of a cryptographic algorithm that would be properly designed to thwart cryptanalytic attacks.

Figure **??** gives a brief comparison of the key length involved in AES, RSA and ECC cryptographic algorithm for same security levels.

NIST guidelines for public key sizes for AES

ECC KEY SIZE (Bits)	RSA KEY SIZE (Bits)	KEY SIZE RATIO	AES KEY SIZE (Bits)
163	1024	1 : 6	
256	3072	1 : 12	128
384	7680	1 : 20	192
512	15 360	1 : 30	256

Supplied by NIST to ANSI X9F1

Figure 1.19: Key length comparison

Tables are given by NIST, ANSSI, etc. The site *www.keylength.com* lists these tables. In figure **??** the term "date" corresponds to the period when the listed key length must be used. For instance from 2011 to 2030 NIST recommends tu use 2048-bit keys for RSA operations.

Date	Minimum of Strength	Symmetric Algorithms	Asymmetric	Discrete Logarithm Key	Discrete Logarithm Group	Elliptic Curve	Hash (A)	Hash (B)
2010 (Legacy)	80	2TDEA*	1024	160	1024	160	SHA-1** SHA-224 SHA-256 SHA-384 SHA-512	SHA-1 SHA-224 SHA-256 SHA-384 SHA-512
2011 - 2030	112	3TDEA	2048	224	2048	224	SHA-224 SHA-256 SHA-384 SHA-512	SHA-1 SHA-224 SHA-256 SHA-384 SHA-512
> 2030	128	AES-128	3072	256	3072	256	SHA-256 SHA-384 SHA-512	SHA-1 SHA-224 SHA-256 SHA-384 SHA-512
>> 2030	192	AES-192	7680	384	7680	384	SHA-384 SHA-512	SHA-224 SHA-256 SHA-384 SHA-512
>>> 2030	256	AES-256	15360	512	15360	512	SHA-512	SHA-256 SHA-384 SHA-512

Figure 1.20: NIST Key length recommendations

1.2 Physical Security and Embedded Devices

Don't you know what a secure micro-controller is? Do you know how to implement in a secure way the standard cryptographic algorithms in micro-controllers? What are these attack techniques named side-channel and fault injection attacks? How do the manufacturer proceed to prevent their products from these numerous and strange attacks?

We are discussing these very exiting subjects in the next paragraphs.

You can find many micro-controllers in so many products of your daily life. It can be the one present in your washing machine, your banking card, your smartphone, etc. But amongst these different products the security requirements for the integrated circuits are very different. Nowadays the term security product concerns for instance your electronic passport, your SIM card, your ID card, your banking cards, etc. All these products have been designed with highest security requirements.

Secure micro-controllers have been designed for years in the military and the smart card domains. Since smart cards were created in 1970 these circuits have known many technological improvements. Actually they seem still destined in future to play more and more their role of digital safe box and not only in the classical smart card form factor.

Secure circuits aims at offering confidentiality services to protect the secret assets like cryptographic keys that are stored and manipulated by the product from the numerous attacks that exist. In practice each product manufacturer must submit his product to evaluation laboratories. These experts audit in detail the architecture of the products and of their security protections. In a second practical phase they will apply the state-of-the-art attacks to the product to evaluate its practical resistance to attacks. After many weeks or months of testing the product will obtain its security certification if all the tests are passed with success. The security certificate is issued by the government services, for instance in France by the ANSSI, in Germany the BSI and in the United Kingdom by the CESG.

These security constraints are not really known by the users. We have seen in the previous section of this chapter the most known cryptographic techniques. Modern cryptographic standards are considered to be secure from a theoretical point of view if the keys and parameters are well chosen.

But as far as a cryptosystem can be theoretically secure against actual cryptanalysis, in real implementations (for instance, in a smart card), it faces other threats than the mathematical ones; in particular side-channel attacks and fault injection attacks.

In the following we are first going to deal with *invasive attacks*. As it is not the heart of this report we do not go in these techniques details. Then we discuss in detail the non-invasive attacks including side-channel analysis and fault injection techniques.

But before let's first introduce you the basic concepts of a microcontroller architecture and the way cryptographic algorithms are embedded in such devices.

1.2.1 Embedded Devices

There are many devices that meet the category *Embedded devices*. Smart card is certainly the most famous.

Micro-controller Classical Architecture

Figure **??**[6] illustrates the classical architecture of a smart card microprocessor. Figure **??**[7] gives an example of the structure of a smart card microprocessor when observed with a microscope in 0.25mm^2 technology.

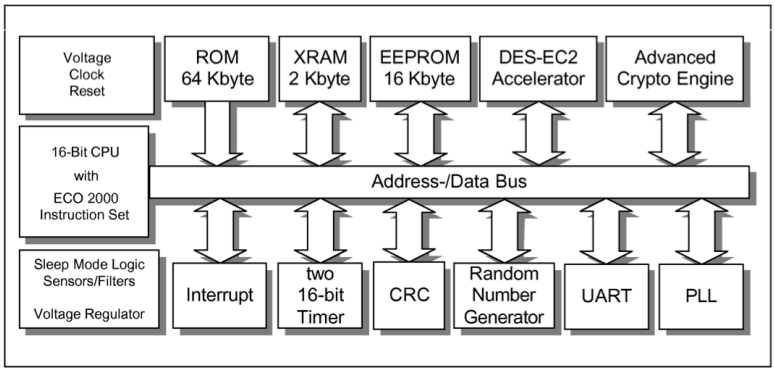


Figure 1.21: Example of microprocessor architecture: Infineon SLEE66 Block Diagram

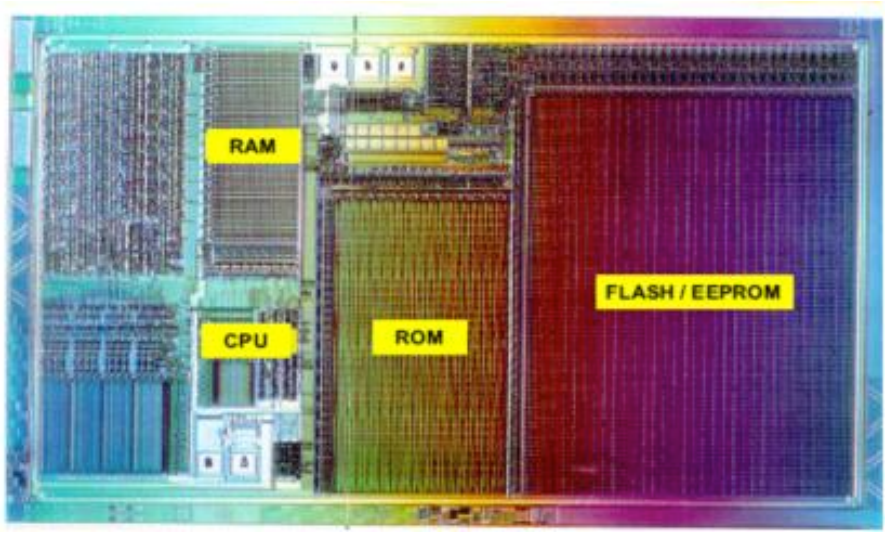


Figure 1.22: Example of smart card chip structure

In the smart card case the microprocessor is located under the chip module. The chip module is the visible part on the card surface that connect the microprocessor to the external world (here the smart card readers). As depicted by figures **??** and **??** it is made of many elements. The Central Processor Unit (CPU) is the core used for calculations

[6]Figure extracted from [**?**]
[7]source; www.eastautomation.com

and manipulations of data. It can be seen as the motor in a car. There are different kinds of memories connected to the core via buses. RAM (Random Access Memory) is a volatile memory that is used for data manipulation and storage. It contains the context data of any operations. It is erased as soon as the chip is no more powered. ROM (Read Only Memory) can only be read and usually contains the program(s) to be executed (for instance test code, Operating system code). It is a non modifiable memory. In between we have the EEPROM or FLASH memories that are non volatile memory that can be read and written. It could be compared to the hard disk of your personal computer. These memories are connected to the CPU through buses that can be 8, 16 or 32 bits. Bigger the bus is faster is the data transfer but higher is the power consumption and die size. For cryptographic purposes some additional hardware blocks must be integrated in the circuit. A hardware random number generator (RNG) is necessary for all cryptographic operations and when random values are required in some functions of the operating system. To perform efficient DES, AES, RSA and ECDSA some dedicated accelerators are often designed and embedded in the integrated circuit. Finally to prevent the product from many attacks some countermeasures and sensors are integrated into the microprocessor.

Symmetric Implementations

The symmetric algorithms embedded in security products are most of the time the standard ones like TDES and AES. Depending on the country where the product is being used other algorithms like FEAL, RC5, SEED ... can also be implemented.

Symmetric implementations can be done in two different forms. The most frequent is the hardware implementation by the use of a cryptographic accelerator. The terms DES (resp. AES) hardware accelerator or TDES (resp. AES) engine is commonly met in the literature. In such a case a specific circuit dedicated to the DES (resp. AES) calculation is added to the microprocessor that makes the calculation processing done in few clock cycles. In this case the whole cryptographic calculation is done by the accelerator. Data to encrypt (or decrypt) and keys are loaded into specific hardware registers and the computation is launched by setting some bits in a configuration register that is used to pilot the engine. The computation time is very short as memory accesses are limited (if there are any) and some calculation can be done in parallel as for instance the 8 substitutions of the DES S-Boxes. For instance a DES computation can be done in 16 clock cycles if one round is computed per clock cycle. Such performances cannot be reached by a software implementation.

The second technique consists in a software implementation of the algorithm using C and/or assembly language. It makes use of the basic set of instructions of the Central Processing Unit (CPU) of the micro-controller. The code is written most of the time in assembly language. Indeed it allows to minimize the read and write operations in memories by taking advantage as much as possible of the core registers instead of RAM local variables.

Asymmetric Implementations

RSA is well known to be currently the most used public key cryptosystem in smart devices. Other public key schemes such as DSA [**?**], Diffie-Hellman key exchange [**?**] protocols, and their equivalent in Elliptic Curve Cryptography (ECC) – namely ECDSA and ECDH [**?**] – are also often involved in security products.

Interestingly, all of them are based on the modular exponentiation or the scalar multiplication and in both cases the underlying operation is modular long integer multiplication. Heavy efficiency constraints thus lie on this operation, especially in the context of embedded devices.

Many methods such as the Montgomery multiplication [**?**] and interleaved multiplication reduction with Knuth, Barrett, Sedlack or Quisquater methods [**?**] can be applied to perform efficient modular multiplications and modular squaring operations. Most of them have in common that the long integer multiplication is internally done with a loop of one (or more) smaller multiplier(s) operating on t-bit words. An example is given in Alg. **??** which performs the schoolbook long integer multiplication using a t-bit internal multiplier giving a $2t$-bit result. The decomposition of an integer x in t-bit words is given by $x = (x_{l-1}x_{l-2}\dots x_0)_b$ with $b = 2^t$ and $l = \lceil log_b(x) \rceil$. Other long integer multiplication algorithms may also be used such as Comba [**?**] and Karatsuba [**?**] methods.

Alg. 1.2.1 Long Integer Multiplication

Input: $x = (x_{\ell-1}x_{\ell-2}\dots x_1x_0)_b, y = (y_{\ell-1}y_{\ell-2}\dots y_1y_0)_b$
Output: multiplication result $\mathsf{LIM}(x, y) = x \cdot y$

1. **for** $i = 0$ **to** $2\ell - 1$ **do**
2. $\quad w_i \leftarrow 0$
3. **for** $i = 0$ **to** $\ell - 1$ **do**
4. $\quad c \leftarrow 0$
5. $\quad$ **for** $j = 0$ **to** $\ell - 1$ **do**
6. $\quad\quad (uv)_b \leftarrow w_{i+j} + x_j \cdot y_i + c$
7. $\quad\quad w_{i+j} \leftarrow v$ and $c \leftarrow u$
8. $\quad w_{i+\ell} \leftarrow c$
9. **return** w

Multiplication with Barrett Reduction. Here a modular multiplication $x \times y \mod n$ is the combination of a long integer multiplication $\mathsf{LIM}(x,y)$ followed by a Barrett reduction by the modulus value n. We use the notation $\mathsf{BarrettRed}(a,n)$ for this reduction, thus $\mathsf{BarrettRed}(\mathsf{LIM}(a,m),n)$ corresponds to the computation of $a \times m \mod n$. We do not detail the Barrett reduction algorithm here, for more details the reader can refer to [**?**] or [**?**].

Montgomery Modular Multiplication. Given a modulus n and two integers x and y, of size v in base b, with $\gcd(n, b) = 1$ and $r = b^{\lceil log_b(n) \rceil}$, $\mathsf{MontMul}$ algorithm computes:

$$\mathsf{MontMul}(x,\ y,\ n) = x \times y \times r^{-1} \mod n$$

Refer to papers [**?**] and [**?**] for details of MontMul implementation.

We denote by ModMul(x,y,n) the operation $x \times y \mod n$, it can be done using Barrett or Montgomery processing.

We do not detail the Barrett reduction algorithm here, for more details the reader can refer to [**?**] or [**?**]. Other techniques can be chosen for processing modular multiplications such as the interleaved multiplication-reduction with Knuth, Sedlak, Quisquater or Montgomery methods [**?**]. Although we have chosen the Barrett reduction our results can also be adapted to these other methods.

Definitions and Notations.

- $x = (x_{\ell-1} \dots x_1 x_0)_b$ corresponds to integer x decomposition in base b, i.e. the x decomposition in t-bit words with $b = 2^t$ and $\ell = \lceil log_b(x) \rceil$.
- LIM(x,y) $= x \cdot y$ long-integer multiplication operation is detailed in the following. Algorithm **??** presents the classical long integer multiplication algorithm.
- BarrettRed(x,n) $= x \bmod n$ using the Barrett reduction method. In this paper we consider reduction operations are done using this algorithm.
- ModMul(x,y,n) $= x \cdot y \bmod n =$ BarrettRed(LIM(x,y),n). It is the combination of a long integer multiplication LIM(x,y) followed by a Barrett reduction by the modulus value n.
- Exp(m,d,n)$= m^d \bmod n$. Algorithm **??** gives more detail on this exponentiation algorithm.

We consider that a modular multiplication ModMul(x,y,n) $= x \cdot y \bmod n$ is performed using a long integer multiplication followed by a Barrett reduction denoted by BarrettRed(LIM(x,y),n).

To perform the modular squaring operation the algorithm **??** is performed and then followed by a Barrett reduction. It is denoted by: ModSquare(x,n) $= x \cdot x \bmod n =$ BarrettRed(LIS(x),n)

The long-integer squaring algorithm complexity is approximately one half the long-integer multiplication algorithm. It means the modular squaring operation is faster than the modular multiplication. In practice we consider a ratio of 0.8, a ModSquare execution timing is about 0.8× ModMult execution timing.

Alg. **??** presents the classical *square and multiply* modular exponentiation algorithm using Barrett reduction. More details on Barrett reduction can be found in [**?**, **?**] and other methods can be used to perform the exponentiation such as *sliding window* techniques [**?**].

Alg. 1.2.2 Long Integer Squaring

Input: $x = (x_{\ell-1}x_{\ell-2}\dots x_1x_0)_b$
Output: multiplication result $\mathsf{LIS}(x) = x^2$

1. **for** $i = 0$ **to** $2\ell - 1$ **do**
2. $\quad w_i \leftarrow 0$
3. **for** $i = 0$ **to** $\ell - 1$ **do**
4. $\quad (uv)_b \leftarrow w_{2i} + x_i \cdot x_i$
5. $\quad w_{2i} \leftarrow v$ and $c \leftarrow u$
6. $\quad$ **for** $j = i + 1$ **to** $\ell - 1$ **do**
7. $\quad\quad (uv)_b \leftarrow w_{i+j} + 2x_j \cdot x_i + c$
8. $\quad\quad w_{i+j} \leftarrow v$ and $c \leftarrow u$
9. $\quad w_{i+\ell} \leftarrow c$
10. **return** w

Alg. 1.2.3 Exponentiation

Input: integers m and n with $m < n$, k-bit exponent $d = (d_{k-1}d_{k-2}\dots d_1d_0)_2$
Output: $\mathsf{Exp}(m,d,n) = m^d \bmod n$

1. $R_0 \leftarrow 1$; $R_1 \leftarrow m$
2. **for** $i = k - 1$ **down to** 0 **do**
3. $\quad R_0 \leftarrow \mathsf{ModSquare}(R_0, n)$
4. $\quad$ **if** $d_i = 1$ **then** $R_0 \leftarrow \mathsf{ModMul}(R_0, R_1, n)$
5. **return** R_0

Exponentiation and RSA. Let p and q be two secret prime integers and $n = p \cdot q$ be the public modulus used in the RSA cryptosystem. Let e be the public exponent and d the corresponding private exponent such that $e \cdot d \equiv 1 \bmod \phi(n)$ where $\phi(n) = (p-1) \cdot (q-1)$. Signing with RSA a message m consists in computing the value $s = m^d \bmod n$. Signature s is then verified by checking that $s^e \bmod n$ is equal to m.

In the case of decryption of signature mechanisms the exponent value is as long as the modulus, for instance consider 2048-bit modulus n and exponent d values. Straightforward implementation of RSA through a modular exponentiation is then time consuming as it require 2048 long-integer squaring modular operations and an average of 1024 long-integer modular multiplications. It is then of strong interest to reduce as much as possible the computational time. A very efficient solution has been proposed by Quisquater and Couvreur in [**?**] based on the Chinese Remainder Theorem. It is commonly known as the RSA CRT algorithm.

Remember the standard RSA signature operation consist in computing: $S = m^d \bmod n$. Now consider the following parameters for the computations:
$n = p \cdot q$
$d_p = d \bmod (p-1)$
$d_q = d \bmod (q-1)$
$i_q = q^{-1} \bmod p$
The RSA CRT computation performs first the exponentiation modulo p and q.

$m_p = m \bmod p$
$s_p = m_p^{d_p} \bmod p$
$m_q = m \bmod q$
$s_q = m_q^{d_q} \bmod q$

Then as $\gcd(p, q) = 1$ the Chinese Remainder Theorem ensures it is possible to recover the unique value s from s_p and s_q such that $s \bmod p = s_p$ and $s \bmod q = s_q$. The value s can be computed thanks to the Garner's algorithm:

$$s = s_q + ((s_p - s_q) \cdot i_q \bmod p) \cdot q$$

The exponentiation modulo n in $m^d \bmod n$ is then replaced by two exponentiations of half size. As the complexity of this operation is in $\mathcal{O}(\ell^{\ni})$ (with ℓ the bit-size of n) a half size exponentiation is 8 times faster. The RSA CRT is then about 4 times faster than the non CRT exponentiation.

It is not possible, even with a 32-bit CPU that offer a multiplication operation to implement efficiently a public key algorithm, CRT or not for the RSA. Indeed a straightforward 2048-bit exponentiation requires an average of 3072 modular multiplication operations. To solve this performance issue most of the chip manufacturers include an arithmetic long integer accelerator– also said public key coprocessor – to compute efficiently modular multiplication operations. Depending on the manufacturer the choice of the modular arithmetic among Montgomery, Barrett, Quisquater, etc. can vary. It allows then to obtain efficient implementation of RSA(CRT), DSA, DH schemes. This coprocessor is also useful to implement efficient elliptic curves operations in ECDSA or ECDH schemes.

1.2.2 Invasive Attacks

Part of this paragraph on invasive attacks is extracted from the chapter 28 [**?**] I wrote with my co-authors for the book [**?**]. Invasive attack techniques originate from the failure analysis domain, but have also been used to attack electronic devices such as smart cards. Equipping a laboratory is very expensive, typically costing several millions of euros. However price is less important as it is possible to rent a full laboratory station. Depending on the complexity of the work involved and the available knowledge of the chip being targeted, it can take many days or even weeks of work in a very specialized laboratory. Moreover the owner of the card should notice these attacks in most cases because of the (partial) destruction of the chip, and warn his provider. Nevertheless it is important to bear in mind that such attacks could fatally undermine an entire security system. For instance, shared master keys should not be present into a card; in that case only the secret proper to the card attacked could be extracted.

Gaining Access to the chip consists in removing the chip module from the card in order to connect it in a test package. With a sharp knife simply cut away the plastic behind the chip until the resin becomes visible. A chemical processing is then applied to remove the resin. It can be later reconnected into another package with fine aluminium wires and a bonding machine.

Reconstitution of the layers consists in reverse engineering the chip layout. It allows to determine the electronic design of a circuit. This type of attack can also be sufficient to provide direct access to sensitive data in the memory. It can also give rise to more complex scenarios such as micro-probing or critical signals or chip reconfiguration. Reverse engineering mainly consists in removing layers sequentially.

Reading the memories consists in directly reading the non-volatile memories such as ROM, EEPROM or FLASH. The remaining effort depends on the kind of memory read, the technology used, and the scrambling of the memory plan.

Probing takes advantage of the information obtained with the previous invasive techniques. Indeed it is possible to observe data flow into internal buses by probing these buses... then one can gain direct access to sensitive data.

FIB and Test engineers scheme flaws make use of focused ion beam (FIB) workstations. This useful tool is a vacuum chamber with an ion gun (usually gallium), which is comparable to a scanning electron microscope. A FIB can generate images of the surface of the chip down to a few nano meter resolution and operates circuit reconfiguration with the same resolution. The FIB is designed initially for IC testability. But the FIB can also be used to modify the internal behaviour of the device by changing the internal connections. Finally FIB is used to facilitate physical analysis of the device by making localized cross sections.

Invasive attacks are very expensive in terms of time, resources and equipment. Moreover these attacks destroy the chip and damage the packaging around it. So it makes

the attacks to be detected by the owner of the card.
It is not true for the second category of attacks named non-invasive attacks. Moreover they are also extremely dangerous because the equipment needed to perform them is relatively inexpensive. Smart cards are nowadays utilized in many security domains and applications: bank cards, mobile communications and secure access, etc. Thus, groups of attackers or illegal organizations could create laboratories in order to process such attacks on the different products that are widely available within our society.

The non-invasive attacks can be split in two categories: the side-channel attacks also said *passive attacks* and the fault injection attacks or *active attacks*.

1.2.3 Side-channel Analysis or Passive Attacks

Side-channel analysis were publicised by P. Kocher in [**?**]. The first attack he presented was the *timing attack*.

Timing attack A timing attack takes advantage of the running time variations in the different executions of an algorithm with different input messages to gain information on the secret key involved. In [**?**] Kocher publicised the first timing attacks done on personal computer implementations of RSA and DSA. It was presented during the conference CRYPTO at Santa Barbara in 1996. The author measured the execution timings to realize a modular exponentiation m^d mod n for different message values m. By analysing these execution timings the attack can recover bit per bit the whole secret exponent value d. Two years later such a timing attack on a smart card was performed by Dhem et al. [**?**].

A simple and efficient countermeasure consists in implementing the computation such that the execution time is constant whatever the key or data manipulated are. A variant of this kind of attacks has been developed and studied. It consists in the *cache attacks*. Indeed when a data is already present into the cache memory the read access is then faster than if it is not into the cache. It means the data access when it is into the cache is faster than when it is not. It creates then variations in the execution timing and this information can be exploited to recover the secret key. Many cache attacks have been published on smart card or more generic embedded devices.

It is worth to notice that timing attacks do not concern only cryptographic calculations. Another famous example is the PIN Verify command present in EMV banking product. Everyone knows he has to enter its PIN (Personal Identification Number) when he wants to pay per his debit bank card. Into your bank card the microprocessor compares the number you have entered with his PIN value that is stored in memory (NVM). If the comparison operations varies depending on the PIN value it can then be exploited to recover the PIN value as we illustrate in the following.

One can observed in step **??** of algorithm **??** that as soon as one byte of the submitted candidate for PIN differs from the correct value the comparison stops and returns the false status. It means the comparison execution time depends on the correctness of the first bytes tested. The execution time can then be ranked in k different executions timings from t_0 to t_{k-1}. Execution timing t_i means that the first i bytes of the PIN candidate are correct: $(CP_0, \dots CP_{i-1}) = (RP_0, \dots RP_{i-1}$. It can be exploited as explained in the

Alg. 1.2.4 PIN Verify comparison

Input: candidate PIN value $PinSubmit = (CP_0, \ldots, CP_{k-1})_8$, correct PIN value $RightPin = RP_0, \ldots, RP_{k-1}$ (stored in smart card memory)
Output: comparison status = $True$ or $False$

```
1. i ← 0
2. while i < k do
3.     if CP_i ≠ RP_i then
4.         return False
5.     i++
6. return True
```

following. The attacker search for the first byte of the PIN. When the guess CP_O is correct then the execution timing of the comparison increases from t_0 to t_1. Once has has recovered the first byte he can guess the second one until timing increases from t_1 to t_2. Then he repeats the same operation byte position per byte position until he recovers the entire exact PIN value RP.

Each of the byte can vary from 0 to 9 as PIN values are most of the time digits and not whole byte values. Exhausting the space for all possible PIN values would require 10^k PIN candidates to be tested in brute force by an attacker. Using the timing attack technique the set of the PIN candidates is reduced to $k \cdot 10$. For instance for a $k = 8$ digits PIN the timing attack requires 80 PIN values to be tested instead of 10^8 for the secret recovery.

A simple countermeasure consists in always comparing the k bytes without any conditional branches code instructions before returning the comparison status.

Power Analysis Another very common side-channel analysis exploits the measure of the power consumption of an electronic device to retrieve information about the secrets inside a tamper proof device. It was publicised by Kocher et al. [**?**] who presented the *Differential Power Analysis* (DPA) that is the first *differential side-channel analysis* (DSCA) technique published. These new attack technique has become fundamental attacks that have changed the smart card security domain.

Today most chips are designed in CMOS technology. An electronic device such as a smart card is made of thousands of logical gates that switch differently depending on the complexity of the operations executed. These communications create power consumption for a few nanoseconds. Thus the current consumption is dependent on the operations of its different peripherals: CPU, cryptographic accelerators, buses and memories, etc. In particular, during cryptographic computations, for the same instruction, the current consumption changes if the value of registers and data processed are different. Monitoring the power consumption, eventually followed by a statistical treatment, one can expect to deduce information on sensitive data when they are manipulated. With some experience and knowledge on the cryptographic algorithms, such analysis can be applied to many smart cards.

Using other side-channels such as the electromagnetic radiations [**?**, **?**] and the radio frequency analysis is quite similar and can lead to a more precise information leakage

depending on the kind of chip analysed and the type of measurement probes and antennas used.

To mount these attacks the basics of the necessary equipment is a numerical oscilloscope, a computer, a card reader to communicate with the card. Antennas, amplifier and EM probes are required for electromagnetic side-channel analysis. Fore more complicated attacks like Differential side-channel analysis (such as DPA and CPA) other softwares are also necessary: to acquire the side-channel traces, to treat the traces (signal processing) and process the attacks. Nowadays the setup required to attack recent devices is still affordable for small organizations.

Simple side-channel analysis SSCA on exponentiation has been introduced by Kocher et al. in [**?**], and one year later improved by Mayer-Sommer in [**?**]. It needs only a single observation of the current side-channel trace. The attacker can find information just by looking carefully at the trace representing the execution of a cryptographic algorithm. This is carried out by a detailed analysis of the trace. The side-channel trace of a microprocessor is different according to the executed instruction and data manipulated. For instance a multiply instruction executed by the CPU needs more cycles than a eXclusive OR (XOR) operation, or in a circular rotation where the value of the carry is either 0 or 1. Most of the algorithm embedded in products are generally standards ones. The implementation variants are not so numerous and depending on the hardware characteristics. For a given algorithm, an attacker can then deduce the structure of the implementation and gain knowledge on the algorithm he is attacking.

The original simple side-channel analysis [**?**] recovered the secret exponent manipulated in an RSA exponentiation from a single consumption trace. Here SPA targets the modular exponentiation implementation when the algorithm used is the classical square-and-multiply method as described in figure **??**. Indeed, when the squaring and the multiplying operations have different recognizable and sizeable patterns the recovery can be done easily because the bits of the secret exponent are directly *read* on the side-channel trace for a classical *Square and Multiply* algorithm. Indeed two consecutive squares on the trace imply the exponent bit is 0 while when a squaring is followed by a multiplication the exponent bit is 1.

Since these two papers very few publications have dealt with simple side-channel analysis on exponentiation. One of them is the zero value side-channel attack from Goubin [**?**]. It was originally presented as a differential analysis but works on a single execution trace of an elliptic curve scalar multiplication. Walter et al. [**?**] showed the power consumption of integrated circuits hardware multipliers is data dependant. Later Yen et al. [**?**] presented a chosen message SSCA defeating some of the common exponentiation algorithms.

SSCA targets to exploit any visible power difference into few traces. It can target any algorithm and any implementation that leaks information.

DPA and statistical side-channel analysis Simple side-channel analysis exploits signal variation in a single or few traces. By comparing these traces he can recover the secret key. However it is not always (and fortunately) efficient when some countermeasures are implemented or because the signal variations are not exploitable for secret

recovering. In that case SSCA can be the starting point for another technique. SSCA is first used by the attacker to obtain as much information as possible. What is the algorithm executed? What is the part of the signal that contain much information that could be useful for another attack technique?

Then the attacker decides to realize some statistical attacks. These attacks require a large number of executions and their associated side-channel traces with different message (input) values. The first statistical attack has been publicised in 1998 by Paul Kocher. Indeed Kocher et al. presented the *Differential Power Analysis* (DPA) in [**?**] on a microprocessor software DES implementation. One year later Messerges et al. published [**?**] the first DPA on the modular exponentiation (so the standard RSA algorithm).

DPA attack uses statistics to amplify the power variations and differences in power traces to reveal the secret key involved in the targeted cryptographic calculation.

Principle of the DPA on an algorithm F Let $F(M, K)$ be cryptographic algorithm with the secret Key K and M the input message. The attacker collects ℓ power consumption traces $C_1 \dots C_\ell$ of the executions $F(M_1, K) \dots F(M_\ell, K)$ on the smart card for ℓ random messages M_i, $i = 1 \dots \ell$. To recover the secret key K the attacker must have the knowledge of the input messages M_i. He proceeds by guessing the key per blocks of bits involved in operations. For instance during the code execution we know there is an operation (for instance a XOR or an addition) between a t-bit part of the message value M_i and a t-bit part of the key value K at an instant I of the computations. This moment I corresponds to a certain number of cycles of operation. Then the attacker focuses only these t bits of the key in opposition to the full key as brute force technique that is not possible on standard algorithm. Indeed he knows that only information on these t bits of the key is strongly related at instant I to the power variations into the chip. It means the power value(s) at instant I that is represented by precise points in the power trace, contain information on the data which is only related to t bits of the key. The attacker wants to guess these t bits of the secret key K. He makes a supposition G on these bits value and uses this guess G to generate some intermediate value of the computation. This intermediate value is the real value manipulated by the chip at instant I when the guess G corresponds to the real t key bits targeted by the attack. At instant I the intermediate data processed by the algorithm is equal to a value $D(M_i, K_{1\dots t})$ with $D(.,.)$ a known function. Depending on the resulting value of this function the attacker decides to separate the power traces in two sets G_0 and G_1 where basically:

- $G_0 = \{C_i$ s.t. at instant I the power consumption is low - because $D(M_i, K_{1\dots t})$ has one (or several) bit(s) to zero$\}$
- $G_1 = \{C_i$ s.t. at instant I the power consumption is low - because $D(M_i, K_{1\dots t})$ has one (or several) bit(s) to one$\}$

Then by subtracting the means of the two groups of traces for each supposition G we obtain 2^t differential traces T_j with $j = 0, \dots, 2^t - 1$.

Basically one can choose a bit of $D(M_i, K_{1\dots t})$ to make the selection in the two groups(if it equals zero (G_0) or 1 (G_1)), but many more evolved methods exist to split

the traces into sets and process the attack. It is important to notice that the bit (value) used for splitting the traces depends on the input message and the t bits guessed on the key. It predicts the real bit value handled by the CPU into the card. For the correct guess on G this bit value will be the same than the one present in the hardware. Only in this case the separation in the two sets G_0 and G_1 will create two groups of high and low average consumptions. Then the differential trace T_u that will show one or many peaks of consumption that will indicate the attacker the guess $G = u$ corresponds to the real key bit values of the secret key K.

So here, to perform the attack we need to know the value of inputs M_i (else it can be done in a similar way with the knowledge of the outputs if we attack by the computation result) and the specification of the algorithm implemented. DPA is also known as the DoM attack for *Difference of Means*.

DPA does not require the knowledge of the power consumption model of the attacked device. But it requires the power variation to be dependant of some bit(s) value(s). This condition is not always true. Indeed the power variation is more related to the gates transitions and not to bit values themselves. Moreover the core registers are generally 8, 16 or 32 bits in CPUs. It means the power variation is not related to a single bit value but to 8, 16 or 32 bits. It renders the one-bit selection for DPA not always efficient and not as optimised as could be statistical side-channel attacks. Another drawback of DPA is the presence of *ghost peaks*. Some peaks not related to the correct key value can appear on differential traces for incorrect guesses. In such cases it is not obvious to recover the secret key value. Ghost peaks have been discussed by Brier, Clavier and Olivier [**?**, **?**] and by Canovas and Clediere [**?**].

The function used to statistically process the side-channel traces is said to be the *distinguisher*. For instance in the DPA attack the distinguisher is the difference of means. A first improvement of DPA was proposed by Coron et al. in [**?**] with the T-test. To solve the *ghost peaks* issue, different statistical techniques have been designed to improve DPA attack and key recovery efficiencies. We can mention the method from Bevan and Knudsen in [**?**].

Amongst these techniques the *Correlation side-channel analysis* remains the most popular and most of the time the most efficient. It has been presented in 2003 by Brier, Clavier and Olivier [**?**, **?**]. The authors consider a power leakage model that is linear in the Hamming weight of the data manipulated. Power consumption W can then be modelized by: $W = a.HW(D) + b$ where D is the word machine value manipulated by the CPU at the instant I of the attack and $HW(D)$ the Hamming weight value of D. Then the Pearson linear correlation factor is computed between the set of collected power traces C_i and the estimated power consumption values W_i. The attacker consider the secret key bits are given by the guess leading to the maximum value of the Pearson correlation factor. This distinguisher is still today one of the most efficient in practice. It is important to notice that the power consumption model presented is linear in the Hamming weight of the data D. It can be also improved by considering the Hamming distance between the data D manipulated at instant I in a register R of the CPU in the power trace and the value contained in register R at instant $I - 1$. In that case we consider the consumption is linear in the Hamming distance model and we can modelized it by $W = A.dH(R, D) + b$ with dH(R,D) the Hamming distance value between R and

D. R can be a fixed value or not. It can be an address value into the code, for a look-up table in the DES for instance. In that case guessing this value for the attack is also necessary. If R is null then it is equivalent to the Hamming weight model. A proposal to improve the CPA has been discussed and given by Le et al. [**?**].

Other distinguishers have been introduced more recently as the Mutual Information Analysis (MIA) [**?**], the Linear Regression Analysis (LRA) [**?**]. Comparison between these different distinguishers have been conducted in different publications and thesis. No universal distinguisher has yet been defined.

DSCA targets any kind of algorithm, symmetric or asymmetric. It fundamentally consists in guessing intermediate value(s) of any algorithm partial computation to statistically process the execution traces in order to reveal t-bit per t-bit blocks which t-bit guess corresponds to the real key bits. This attack is very powerful and has been proven for years efficient on many different products. However it is obvious that applying statistical attacks on recent microprocessors is not so simple. It requires two essential steps in the attack. The signal measurement step is the most crucial. The choice of the probes for EM analysis and the experience and motivation of the attacker to localize on the chip surface the probe position that gives the best exploitable signal. But a second very important step is required: the signal processing operations. On recent devices many countermeasures have rendered the traces noisy and difficult to synchronize. However applying any DPA, CPA or another distinguisher has no sense if the traces used for the attack are not resynchronized or realigned. It is one of the crucial steps of the attack. These two attack steps must not be underestimated as they are the base and a crucial condition for any future success of the statistical attack.

Countermeasures

We consider here there are three categories of existing countermeasures.

First countermeasures are inherently induced by the kind of application using the cryptographic algorithm. For instance the use of (random) padding in a RSA signature prevents the implementation from chosen message attacks; similarly the use of counter value(s) into the data sent to the card does not allow an attacker to sent twice the same data to the cryptographic algorithm.

The second category targets to modify the signal either with hardware security features (noise generators, dummy cycles, clock jitters or power filtering aim at reducing the circuit leakage) or software countermeasures (*e.g.* dummy operations). The aim is to desynchronize the curves and prevent an attacker from correctly exploiting them during their statistical treatment.

The third kind of countermeasures consists in de-correlating the curves with the data related to the algorithm's execution. The principle is to prevent attackers from predicting any intermediate value manipulated during the known algorithm execution. For instance code with constant time execution, masking and randomization techniques on input data and secret key are in this category [**?**, **?**, **?**, **?**].

We focus in this PhD report on the third category of countermeasures. It implies algorithmic and mathematics aspects in order to hide the computational data with randomness

to the attacker during the execution of a known cryptographic algorithm. These techniques are very interesting and funny to implement or to defeat from and a theoretical and a practical point of view. We give in the following an overview of these techniques on symmetric and asymetric cryptographic implementations.

Countermeasures in Asymmetric Algorithms

The first security requirement consists in preventing any implementation from timing and simple side-channel analysis attacks. It requires the developer to implement its algorithm, like the modular exponentiation such that it executes in constant time.

Efficient constant time implementations are often said as *regular* implementations. Two regular exponentiation method are discussed in the side-channel community. The *square-and-multiply always* exponentiation processes both a squaring and a multiplication operation for each bit value. It is achieved by the execution of a "dummy" multiplication when the secret exponent bit equals zero as it is depicted by algorithm **??**.

Alg. 1.2.5 Square-and-multiply always regular exponentiation

Input: integers m and n with $m < n$, k-bit exponent $d = (d_{k-1}d_{k-2} \dots d_1 d_0)_2$
Output: $\mathsf{Exp}(m,d,n) = m^d \bmod n$

1. $R_0 \leftarrow 1$, $R_1 \leftarrow 1$; $R_2 \leftarrow m$
2. **for** $i = k-1$ **down to** 0 **do**
3. $\quad R_1 \leftarrow \mathsf{ModSquare}(R_1, n)$
4. $\quad R_{d_i} \leftarrow \mathsf{ModMul}(R_1, R_2, n)$
5. **return** R_1

The multiplication operation is always performed but the result is only taken into consideration in the next squaring operation if the secret exponent bit is one. Such an implementation prevents the product from simple side-channel analysis techniques even those taking advantage of the possibility to choose the input message values. However the main drawback of this method is the additional cost added to the execution time.

The second regular method is the *Montgomery Ladder* exponentiation [**?**, **?**]. It is presented hereafter in algorithm **??**. It is generally preferred over the square-and-multiply always method since it does not involves dummy multiplications which makes it naturally immune to specific fault attacks named the C safe-error attacks [**?**, **?**].

Such regular algorithms perform one squaring and one multiplication at every iteration and thus require one multiplication and one squaring operation per exponent bit.

One of the most efficient countermeasure against SSCA is the *side-channel atomicity* introduced by Chevallier-Mames et al. [**?**]. In an atomic implementation the code executed during the whole exponentiation loop is the same for a squaring and a multiplication step rendering the attack no more possible. It regroups the exponent bit operation and the modular long integer operation in a same atomic code block.

However such a countermeasure does not prevent the implementation from chosen message side-channel attacks [**?**] because the power consumption of integrated circuits

Alg. 1.2.6 Montgomery Ladder Exponentiation

Input: $m, n \in \mathbb{N}$, $m < n$, $d = (d_{k-1}d_{k-2}\dots d_0)_2$
Output: $m^d \mod n$

1. $R_0 \leftarrow 1$; $R_1 \leftarrow m$
2. **for** $i = k-1$ **to** 0 **do**
3. $\quad R_{1-d_i} \leftarrow \mathsf{ModMul}(R_0, R_1, n)$
4. $\quad R_{d_i} \leftarrow \mathsf{ModSquare}(R_{d_i}, n)$
5. **return** R_0

Alg. 1.2.7 Atomic exponentiation

Input: integers m and n with $m < n$, $\ell \cdot t$-bit exponent $d = (d_{\ell \cdot t-1}d_{\ell \cdot t-2}\dots d_1 d_0)_2$
Output: $\mathsf{Exp}(m,d,n) = m^d \mod n$

1. $R_0 \leftarrow 1$
2. $R_1 \leftarrow m$
3. $i \leftarrow \ell \cdot t - 1$; $\alpha \leftarrow 0$
4. **while** $i \geqslant 0$ **do**
5. $\quad R_0 \leftarrow \mathsf{ModMul}(R_0, R_\alpha, n)$
6. $\quad \alpha \leftarrow \alpha \oplus d_i$;
7. $\quad i \leftarrow i - 1 + \alpha$
8. **return** R_0

hardware multipliers is data dependant as presented by Walter and Samyde in [**?**]. Moreover an atomic exponentiation can also be threatened by DSCA techniques like DPA and CPA as presented by Amiel et al. [**?**]. Another threat is the technique that consists in distinguishing squaring from multiplication operations by Amiel et al. [**?**]. We detail it in the next paragraph.

Distinguishing Squaring from Multiplication Operations Amiel et al. showed in [**?**] that the average Hamming weight of the output of a multiplication $x \times y$ has a different distribution whether:

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

This attack can thus target an atomic implementation such as Alg. **??** where the same multiplication operation is used to perform $x \times x$ and $x \times y$.

First, many exponentiation curves using a fixed exponent but variable data have to be acquired and averaged. Then, considering the average curve, the aim of the attack is to reveal if two consecutive operations are identical – i.e. two squarings – or different – i.e. a squaring and a multiplication. As in the classical SPA, two consecutive squarings reveal that a 0 bit has been manipulated whereas a squaring followed by a multiplication reveals a 1 bit. This information is obtained using the above-mentioned leakage by subtracting

Figure 1.23: Power trace of the RSA exponentiation implementing the atomicity principle

the parts of the average curve corresponding to two consecutive operations: peaks occur if one is a squaring and the other is a multiplication while subtracting two squarings should produce only noise. It is worth noticing that no particular knowledge on the underlying hardware implementation is needed which in practice increases the strength of this analysis.

A classical countermeasure against this attack is the randomization of the exponent[8], i.e. $d^* \leftarrow d + r\varphi(n)$, r being a random value. The result is obtained as $m^d \bmod n = m^{d^*} \bmod n$.

In spite of the possibility to apply the exponent randomization, this attack brings into light an intrinsic flaw of the multiply always algorithm: the fact that at some instant a multiplication performs a squaring $(x \times x)$ or not $(x \times y)$ depending on the exponent. In the rest of this paper we propose new atomic algorithms that are exempt from this weakness.

As these attacks requires either the knowledge or in most powerful attack scheme the control of the input data, the most common additional countermeasure to protect exponentiation (or other public key operations) consists in "hiding" the intermediate data processed during the operation with randomness. The term "blinding" is generally used to refer to this technique. Many blinding techniques exist in the literature. We list in the following the most classical ones.

Message blinding It consist in adding randomness to the input message that is removed at the end of the calculation to obtain the correct computation result. The two common techniques are the multiplicative blinding and the additive one.

The multiplicative blinding consist in multiplying to the input message a random value powered to the public exponent e. The message m is transformed to the blinded message $m^\star$ with the following formula:

$$m^\star = r^e \cdot m$$

with r a random integer value. The blinded exponentiation (like in RSA) operation becomes:

$$(m^{\star d} \bmod n) \cdot r^{-1} \bmod n = m^d \bmod n$$

This countermeasures requires the knowledge of the public key (that is not always possible in many products for the private operation) and an inverse computation to be processed at the end in order to remover the randomness added by r.

The additive blinding consists in adding to the message m a random multiple of the modulus n. The message m is here transformed to the blinded message $m^\star$ with the

[8] Notice however that the randomization of the message has no effect on this attack, or even makes it easier by providing the required data variability.

following formula:

$$m^\star = m + r_1 \cdot n \bmod r_2 n$$

with r_1, r_2 two λ-bit random integer values ($\lambda = 16$ or 32 or 64 bits – the larger λ is, the more secure). The blinded exponentiation (like in RSA) operation becomes:

$$(m^{\star d} \bmod r_2 n) \bmod n = m^d \bmod n$$

the final reduction is used to remove the randomness effect of the blinding and to lead to the final expected result. The advantage of this technique compared to the multiplicative one is that it does not require any costly inverse operation to remove the blinding at the end of the exponentiation. Moreover it is less sensitive to zero value or particular input chosen message attacks [**?**]. For these reasons we consider in the following the additive blinding message technique.

We have then seen that a secure exponentiation requires an atomic algorithm and the message blinding countermeasure. However both techniques must be completed with another countermeasure to reach the state of the art in term of secure exponentiation method. Indeed Amiel et al. presented [**?**] a side-channel attack that recovered the secret exponent on a an atomic exponentiation with the additive (or multiplicative) message blinding countermeasure. It highlighted the importance of also implementing the exponent blinding countermeasure.

Exponent blinding The objective of this countermeasure is to randomize the sequence of squaring and multiplication operation all along the exponentiation. There exist different techniques to apply exponent blinding.
Additive exponent randomization: it consists in adding to the private exponent a multiple of the multiplicative group order of $(\mathbb{Z}/n\mathbb{Z})^\star$. This technique has been publicised and patented by Kocher et al. in [] The exponent d is here transformed to the blinded exponent $d^\star$ with the following formula:

$$d^\star = d + r \cdot \varphi(n)$$

with r a λ-bit random integer value and $\varphi(.)$ the Euler Phi function. It can be seen that per design we have $m^{d^\star} \bmod n = m^d \bmod n$. The cost added by this countermeasure is only a computational one that depends on the length λ of r. The blinding adds here an average number of $1.5 \cdot \lambda$ long integer multiplications to the exponentiation computational cost.

Splitting the exponent: this technique has been presented by Joye et al. [] The exponent d is here split in two blinded exponents d_1 and d_2 such that $d = d_1 + d_2$. For instance one can define $d_1 = d - r$ and $d_2 = r$ with r a λ-bit random integer value. The exponentiation operation becomes:

$$(m^{d_1} \bmod n) \cdot (m^{d_2} \bmod n) = m^d \bmod n$$

In that case two exponentiation are computed instead of one. It adds an extra cost that makes the previous blinding more efficient when the value $\varphi(n)$ is known.

Blinded Exponentiation Including atomicity, message and exponent blinding countermeasure lead to the protected blinded exponentiation method we detail in algorithm **??**.

Alg. 1.2.8 Blinded exponentiation

Input: integers m and n with $m < n$, $\ell \cdot t$-bit exponent $d = (d_{\ell.t-1} d_{\ell.t-2} \dots d_1 d_0)_2$, a security parameter λ

Output: $\mathsf{Exp}(m,d,n) = m^d \bmod n$

1. $r_1 \leftarrow random(1, 2^\lambda - 1)$
2. $r_2 \leftarrow random(1, 2^\lambda - 1)$
3. $r_3 \leftarrow random(1, 2^\lambda - 1)$
4. $\bar{n} \leftarrow r_2 \cdot n$
5. $R_0 \leftarrow 1 + r_1 \cdot n \bmod \bar{n}$
6. $R_1 \leftarrow m + r_1 \cdot n \bmod \bar{n}$
7. $\bar{d} \leftarrow d + r_3 \cdot \varphi(n)$
8. $i \leftarrow \ell \cdot t + \lambda - 1$; $\alpha \leftarrow 0$
9. **while** $i \geqslant 0$ **do**
10. $\quad R_0 \leftarrow \mathsf{ModMul}(R_0, R_\alpha, \bar{n})$
11. $\quad \alpha \leftarrow \alpha \oplus d_i$;
12. $\quad i \leftarrow i - 1 + \alpha$
13. $R_0 \leftarrow R_0 \bmod n$
14. **return** R_0

Big Mac Attack

The only known side-channel attack able to threaten such a blinded exponentiation before the results we present in this PhD is the Big Mac attack from Colin-D. Walter [**?**] we are discussing later.

Elliptic Curves Equivalent atomic and blinding techniques applies also to the elliptic curve exponentiation (also said scalar multiplication). We do not detail here these countermeasures are elliptic curves attacks and countermeasures are not investigated in the research results to be presented in this PhD report.

1.2.4 Fault Injection Analysis or Active Attacks

Fault effects and perturbations on electronic devices have been observed since the 1970's in the aerospace industry. However use for analyzing embedded cryptographic implementations has first been done in 1997, when D. Boneh, R. DeMillo and R. Lipton [**?**] published the first theoretical fault attack. This is known as Bellcore attack. The most famous attack in their paper threaten the RSA in CRT implementation mode where only two faulted execution result are required to factorize the modulus. They also introduced other fault attacks that threaten the standard RSA and the Fiat-Shamir and Schnorr identification schemes. In case of computation error, the Bellcore researchers showed how

to recover the secret factors p and q of the public modulus n from two CRT-RSA signatures of the same message: a correct one and a faulty one. Thereafter Lenstra explained in a short memo [?] that only one faulty signature is required when the message value is also known to recover the modulus prime factors. In 1997, E. Biham and A. Shamir [?] presented the first Differential Fault Analysis (DFA) on a symmetric algorithm applied to the Data Encryption Standard (DES). Those attacks are seriously taken into account during the design of secure products since very realistic and easy to mount in practice. Thus any implementation can be threatened by fault injection techniques. As for power analysis, many researches have been conducted in this domain [?, ?, ?, ?, ?, ?].

Countermeasures

There are many ways to protect implementations against fault analysis. The simplest and most direct solutions consist in, either processing operations twice or verifying the decryption by computing consecutively the encryption (similarly with signature and verification), then to compare both results. For instance one can compute an RSA signature and, when the public key is small, compute the verification to be sure the process has not been disturbed. However sometimes in practice, the public key is not available and/or computing twice penalizes too much the execution time of the computation. This is the reason why other solutions have been designed. In 1997 Shamir [?] presented a simple solution to prevent the previous attack: the signer first chooses a (small) random number r relatively prime to n, then he computes $s_{rp} = m^{d \bmod \varphi(rp)} \bmod rp$ and $s_{rq} = m^{d \bmod \varphi(rq)} \bmod rq$. If $s_{rp} \equiv s_{rq} \pmod{r}$, then the computations are assumed correct, and s is computed by applying Chinese remaindering on $(s_{rp} \bmod p)$ and $(s_{rq} \bmod q)$. However because this method does not detect error(s) generated during the CRT recombination this idea has been improved later (also including the Yen's ideas on avoiding decisional tests) in [?, ?, ?].

Passive and Active Combined Attacks

Passive and Active Combined Attacks (PACA in short) have been introduced by Amiel et al. [?].

As already mentioned, there are many existing countermeasures to protect secure products either from active attacks or from passive ones. Usually when products have to be protected from both passive and active attacks, developers add/combine countermeasures. However fault countermeasures often react only at the end of the execution. In such a case we show below that fault injection can be used to successfully realize a passive attack even if active and passive countermeasures have been implemented.

Yen and Joye in [?] have shown that square and multiply always algorithm can help a (certain type of) fault attack. Authors in [?] illustrate once again the need of considering secure implementation as a monolithic block.

The basic principle is to combine active and passive analysis. By injecting a fault disturbing a computation, it becomes possible to realize a passive attack on the perturbed execution. The fault is detected at the end of the command. Unfortunately, this is *too late*, even if the result is not returned. The secret value has already been recovered using a classical power analysis. This is the basic principle of PACA based on the fact

that fault countermeasures are only active after the end of the computation. Moreover, even if the chip kills itself because of fault detection, this is useless since only one curve leakage is necessary.

A kind of combined attack has already been presented in [**?**]. It applies to specific hardware implementations which protection is based on balanced gates. Each successful fault injection is supposed to unbalance the power consumption of a logical balanced gate, the bias is then used to realize a DPA on the hardware cryptographic module. In practice, such attack does not seem so easy to implement due to the difficulty to reproduce thousands of times a successful fault on a protected design and can be prevented by adding classical software countermeasures. In [**?**] the author used similar idea to detect bit change without interfering with the normal device operation.

Chapter 2

Contributions and Outline

We summarize in this chapter the contributions we are presenting in this PhD report. These results have been published through scientific articles in different international conferences in cryptography and embedded security domains.

Advanced Simple Side Channel Analysis on Exponentiation

This section deals with simple side-channel attacks (SSCA) targeting public key implementation, especially those dealing with the cryptosystem RSA. We explain in this section how SSCA can be efficient when selecting particular value for input messages. We also design chosen message techniques that render SSCA very efficient even on state-of-the-art implementations. We also discuss the importance of the base multiplier and the random bit-length on the leakage probability of an implementation. These studies have conducted to a first publication at CARDIS in 2010 [**?**] and a second article [**?**] at the COSADE 2013 conference.

Horizontal Differential Attacks on Exponentiation

We present here a new kind of attacks on Public Key implementations named Horizontal Attacks. In the first part of this section we introduce the first Horizontal Correlation attack. This technique recovers the secret exponent of a RSA exponentiation using the correlation technique on a single side-channel trace. It renders the classical exponent blinding countermeasure inefficient. In a second part we present another Horizontal attack named ROSETTA which defeat message and exponent blinding countermeasure using a single side-channel trace. The studies on this subject have conducted to a first publication at the ICICS conference in 2010 [**?**]. Later a second article [**?**] has been presented at the INDOCRYPT 2012 conference.

Defeating with Fault Injection a Combined Attack Resistant Implementation

PACA attack category has been introduced in 2007. It combines simultaneously side-channel and fault injection techniques in a same attack in order to defeat the most efficient countermeasures. Many algorithms have been designed to prevent implementation from these attacks. Recently Schmidt et al.have presented a combined attack resistant implementation. We explain in this section how this countermeasure can be defeated by using single fault injection attack. We also demonstrates that some PACA can defeat the same algorithm. Finally we update the Schmidt et al.algorithm to make it resistant against our attacks. This study has been published through a scientific article at the COSADE 2013 conference in 2010 [**?**].

Collision Correlation Analysis on First Order Protected AES

First Collision Correlation attacks were introduced by Witteman et al. [**?**] on public key implementation and by Moradi et al. [**?**] on AES implementation. We present here two new attacks taking advantage of this technique. Our attack defeat different first order protected implementations of AES and does not require any leakage model to be mounted. This latter property makes this attack very efficient. Our results have been published [**?**] at CHES 2011.

Combined Attack on First Order Protected AES

We present here a PACA technique that combines a single fault attack with a side-channel attack in order to defeat a first order protected AES implementation. We also discuss countermeasures to be implemented. Results were published at FDTC 2010 conference [**?**].

Efficient Provable Prime Number Generation for Embedded Devices

In this section we present new algorithms to generate provable prime numbers in embedded devices. Contrarily to the known previous algorithms from Shawe-Taylor and Maurer we succeeded in obtaining methods faster than probabilistic ones. We also discuss the countermeasures to make such prime generation algorithm prevented from side-channel attacks. This study lead to the publication of article [**?**] at the PKC 2012 conference.

Square Always Exponentiation

Atomic principle has been used for years now to implement fast and secure exponentiation in smart-cards. However it always uses atomic blocks relying on the modular multiplication operation. We present here new atomic algorithms that relies only on

modular squaring operations. Our algorithm are nearly as efficient as the ones relying on multiplication operations. However when the latter are not resistant to the attack published in [**?**] our Squaring always method are resistant. Results have been published [**?**] and presented at the INDOCRYPT 2011 conference.

Chapter 3

Advanced Simple Side Channel Analysis on Exponentiation

3.1 Introduction

In this chapter we focus our study on the *blinded exponentiation*, the atomic exponentiation algorithm **??** where modulus, message and exponent are blinded. Considering this state-of-the-art implementation we present how to build chosen messages leading to more efficient SSCA when attacking blinded exponentiation on devices and show that, contrarily to common belief, our SSCA can also be successful in some hashed message models. Moreover we introduce a different leakage models for hardware multipliers we know realistic for practical measurements. We then obtain new results when explaining simple side-channel efficiency for different key length, multipliers and the size of the random used for blinding. We highlight then that, even if the hardware multiplier architecture is a 32-bit one, SSCA can be very efficient with a reasonable number of executions to recover the secret exponent manipulated. We discuss then the need for a deep side-channel characterization of hardware multipliers in order to establish the best recommendations for any hardware multiplier being used for secure products. It would allow developers to select with strong assurance the right countermeasures (and algorithm) when implementing for a selected device any public key algorithm.

3.2 Enhanced Simple Power Analysis on Exponentiation

In Yen *et al.* attack and the other techniques introduced in this chapter, SPA does not aim at distinguishing differences in code execution but rather to detect when specific data are manipulated through their specific power signature. Indeed power signatures during an operation $(x \times y)$ or $(x \times y \mod n)$ will depend on values x and y. If x and/or y have very particular Hamming weights then it will lead to a very characteristic power trace for the multiplication. It has been discussed by Walter et al. [**?**].

We present here many values which can generate a recognizable pattern and thus lead to the exponent being recovered from a single power trace.

We illustrate our analysis on the ModMul operation using the Barrett reduction

x_i	y_j	$C(x_i, y_j)$
$x_i \neq 0$	$y_j \neq 0$	C_{High}
$x_i \neq 0$	$y_j = 0$	C_{Medium}
$x_i = 0$	$y_j = 0$	C_{Low}

Table 3.1: Power Signal quantity for $x_i \times y_j$

and especially during the computation of $\mathsf{LIM}(x,y)$. The same analysis can be done in other kind of modular multiplication methods, for instance in the modular Montgomery multiplication method $\mathsf{MontMul}(x,y)$.

3.2.1 Origin of Power Leakage

The power leakage appears during the operation $x_i \times y_j$ of the long integer multiplication $\mathsf{LIM}(x,y)$. Any operation $x_i \times y_j$ has a power consumption related to the number of bit flips of the bit lines manipulated. When one of the operands is null or has very few bits set, for instance is equal to 0, or 2^i with i in $0 \dots t-1$, the t-bit multiplication has a lower power consumption than the average one. We can then distinguish in a long integer multiplication when such a value is manipulated.

If the value of the multiplicand m contains one (or more) of the t-bit word(s) set to 0 or 2^i with i in $0 \dots t-1$, during an atomic exponentiation loop we can recognize each time this value m is manipulated, i.e. each time the exponent bit is 1.

The condition for this SPA to succeed is that one (or more) of the t-bit word(s) of x or y is set to 0 (or in some cases it could be also to 2^i with i in $0 \dots t-1$). We consider here that the leakage appears only for zero values.

We can then quantify the power consumed by the device for computing $x_i \times y_j$. We denote by $C(x_i, y_j)$ this power consumption . As illustrated in Table **??** we can distinguish three categories depending on whether x_i and y_j values are 0 or not.

When $x_i = 0$ and $y_j = 0$ the device only manipulates zero bits. Thus the amount of power consumed by the multiplication is *Low*, we denote it as C_{Low}. A multiplication with non zero values ($x_i \neq 0$ and $y_j \neq 0$) yields a higher power consumption: we consider this as *High* and denote it as C_{High}. Finally when $x_i \neq 0$ and $y_j = 0$ the amount of power consumed by a multiplication is considered as *Medium*: we denote it as C_{Medium}.

In the operation $\mathsf{LIM}(x,y)$ we can graphically estimate the power curve by

$$C_{LIM(x,y)} = \sum_{i=0}^{k-1} \sum_{j=0}^{k-1} C(x_i, y_j) \cdot T(k,i,j)$$

with $C(x_i, y_j)$ being the power consumption of the device for computing $x_i \times y_j$ and T a function which represents the clock cycles corresponding to the exact moments where this operation is executed for the set (k,i,j).

This corresponds to the schematic power curve of Figure **??**.
A graphical estimation of power consumption expected depending on whether we have

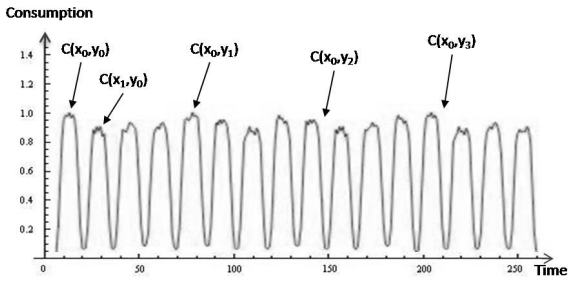


Figure 3.1: $C_{LIM(x,y)}$: power curve representation of operation LIM(x,y) with $k = 4$

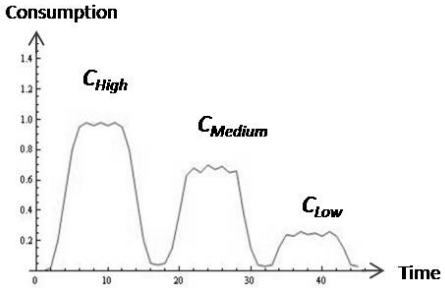


Figure 3.2: Three cases of estimated power curves for $C(x_i, y_j)$

C_{High}, C_{Medium} or C_{Low} is given in Figure **??**.

When an operation $x_i \times y_j$ leads to $C(x_i, y_j)$ being C_{Low} or C_{Medium} we can identify this operation in the curve. This explains why the SPA introduced by Yen *et al.* allows the secret exponent recovering with a single curve for a well chosen message. Indeed when comparing the three possible operations occurring in an exponentiation with the input chosen message $m = n - 1$ we obtain for $k = 3$ the Table **??**. In this table we observe that $C_{LIM(x,y)}$ has different recognizable patterns for each long integer multiplication.

3.2.2 More Chosen Messages.

From this analysis we can enumerate other chosen messages leading to successful SPA on atomic exponentiations such as messages with one or many t-bit word equal to 0 or 2^i with i in $0 \dots t-1$. Messages with a globally low Hamming weight can also lead to a medium or low power consumption and allow to recover the secret exponent in a single power curve.

3.2.3 Experiments and Practical Results

We experimented this attack on many different multipliers processors to confirm our theoretical analysis. In this section we present some results we obtained on two different devices.

First Device.

We implemented a Montgomery Modular exponentiation on a 32×32-bit multiplier, in this case we have t equal to 32. We chose as input messages for exponentiations the following values with $k = 4$: $m_1 = (\alpha, \alpha, 0, 0)$, $m_2 = (\alpha, 0, 0, 0)$ where α is 32-bit random value.

Figure **??** represents a part of the measured exponentiation curves of these two messages. The black curve corresponds to the exponentiation with message m_1 and grey curve with m_2. The multiplication is clearly identifiable by a lower power consumption

$x \times y$	x in base b	y in base b	$C_{LIM(x,y)}$
$(n-1) \times (n-1)$	$a = (a_2, a_1, a_0)_b$	$a = (a_2, a_1, a_0)_b$	$C_H\|C_H\|C_H\|C_H\|C_H\|C_H\|C_H\|C_H\|C_H$
$1 \times (n-1)$	$a = (0, 0, 1)_b$	$m = (m_2, m_1, m_0)_b$	$C_H\|C_M\|C_M\|C_H\|C_M\|C_M\|C_H\|C_M\|C_M$
1×1	$a = (0, 0, 1)_b$	$a = (0, 0, 1)_b$	$C_H\|C_M\|C_M\|C_L\|C_L\|C_M\|C_L\|C_L$

Table 3.2: The three possible power traces for LIM(x,y) in Yen *et al.*'s attack for $k = 3$

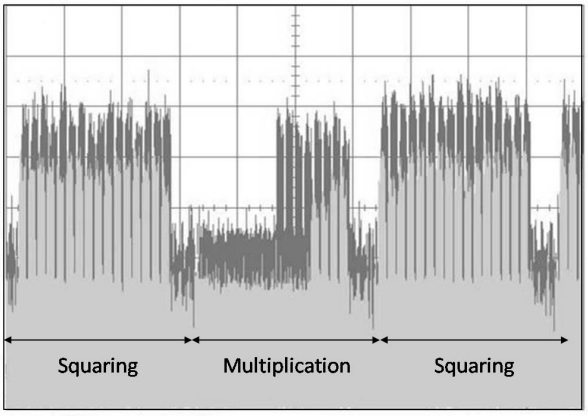


Figure 3.3: Part of exponentiation power curves with messages m_1 and m_2 and $k = 4$, zoom on $\mathsf{LIM}(m_1,a)$ (black) and $\mathsf{LIM}(m_2,a)$ (grey)

compared to the squaring. Message m_2 takes one more 32-bit word equal to 0 than m_1. This results in Figure **??** in a low power consumption longer for grey curve than for black curve during the multiplication.

In this case we also observed that C_{Medium} is close to C_{High} but the two can be distinguished.

Second Device.

We designed an 8×64-bit hardware multiplier with the associated long integer exponentiation. In the multiplication $x \times y$ the operand x is manipulated by 64-bit words when the operand y is taken by 8-bit words. The message is placed in the second operand y for the multiplications during the exponentiation.

We chose several messages containing one or more zero 8-bit words and executed the corresponding long integer exponentiations. We then simulated the power consumption of the synthesized multiplier we have designed. By analyzing these power curves we can observe that a zero byte y_j in operand y produces a lower power consumption curve in the cycles where y_j is manipulated. We are then able to recover the whole secret exponent in an exponentiation when a zero byte is present in the message value.

We have explained here the potential power leakages related to multiplication and exponentiation computations and confirmed our analysis with some practical results. In the next paragraph we study the probability of leakage depending on the multiplier and modulus bit lengths.

3.2.4 Leakage Probability

In this paragraph letters p and q design probabilities.

Probability of leakage during a multiplication.

Let x_i be a t-bit word, and p be the probability for x_i to be null, then we have $P(x_i = 0) = \frac{1}{2^t} = p$ and $P(x_i \neq 0) = 1 - p$.
If Y is the event {None of the t-bit word is null in a k-word integer} with $P(Y) = (1-p)^k$, then we have $\overline{Y}$ which corresponds to the event {at least one of the t-bit words is null in a k-word integer} with probability:

$$q = P(\overline{Y}) = 1 - P(Y) = 1 - (1-p)^k = 1 - (1 - \frac{1}{2^t})^k$$

During a long integer modular multiplication $x \times y$ the leakage appears only if at least one of the k t-bit words of x or/and y is null. The probability for this leakage to appear corresponds to $1 - (1-p)^{2k}$.

Probability of leakage during an exponentiation.

During an exponentiation we focus on the probability of having a leakage in a t-bit multiplication $x \times y$ when only y takes part in the leakage and not x (or the opposite). Indeed during an exponentiation $m^d \mod n$ the message m is used during each multiplication at step **??** of Algorithm **??**, when $\alpha = 1$ ($d_i = 1$). Thus if the value m contains a t-bit word m_i leading to leakage in the operations $m_i \times a_j$ and/or $m_i \times a$ then each multiplication by m_i and thus by m could be identified and the secret exponent d can be recovered from a single power curve.
In this case the probability of having one or many of the t-bit words of m leading to a signing pattern is:

$$q = 1 - (1-p)^k = 1 - (1 - \frac{1}{2^t})^k$$

This is also the probability of having an SPA leaking curve for a single execution of the exponentiation.

In the case of an 8-bit multiplier Figure **??** shows that the probability of having a message with a signing pattern is about 0.394 for a 1024-bit modulus, 0.528 for a 1536-bit modulus and 0.633 for a 2048-bit modulus. When the multiplier is greater than 16 bits this probability decreases for all modulus sizes.
It also obvious that bigger the key length is and smaller the multiplier size (t) is, the higher the probability of recovering the secret exponent d in a single curve is.

Using Poisson law as an approximation of binomial law we have the property that with $1/q$ exponentiation power curves the probability for recovering the secret exponent is $(1 - \frac{1}{exp(1)})$. Thus the probability P_{leak} of recovering the secret exponent using one of the h/q acquired curves is approximated by $P_{leak} = P(h/q) = 1 - (\frac{1}{exp(1)})^h$.

Figures **??** and **??** show how many curves would be needed to have, with a probability close to 1, a message leading to a signing pattern which can be used for our SPA attack. With an 8-bit multiplier (Figure **??**), a very few messages (5 to 10) are necessary to obtain an exploitable leakage with high probability. For a 16-bit multiplier (Figure **??**)

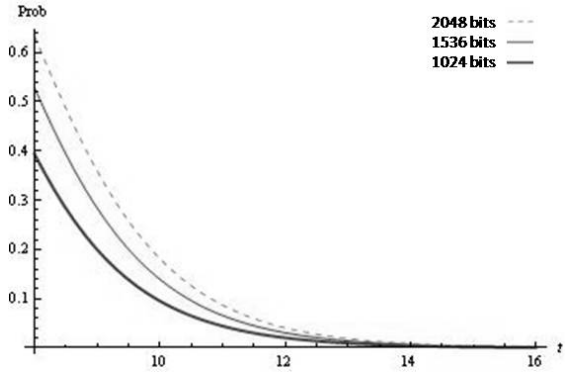


Figure 3.4: Probability of having a message with a signing pattern depending of multiplier size (t) and modulus size (1024, 1536, 2048)

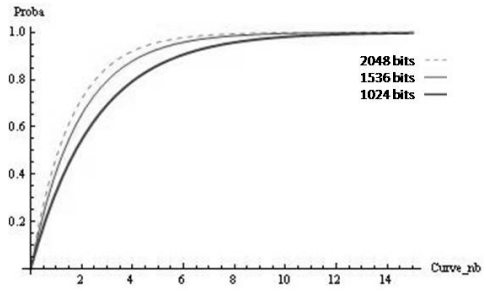


Figure 3.5: Probability P_{leak} for an 8-bit multiplier depending on the number of curves acquired and modulus size (1024, 1536, 2048)

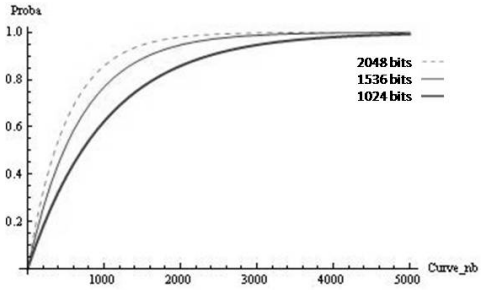


Figure 3.6: Probability P_{leak} for a 16-bit multiplier depending on the number of curves acquired and modulus size (1024, 1536, 2048)

Modulus size	**Multiplier size**		
	8	16	32
1024	12	4720	$\approx 2^{29}$
1536	9	3150	$\approx 2^{29}$
2048	8	2360	$\approx 2^{28}$

Table 3.3: Number of messages needed to have $P_{leak} \geqslant 0.99$

between 3000 and 5000 curves are needed for a success probability close to 1. But for a multiplier size greater than 16 the number of curves needed for recovering the secret exponent makes the attack not practical. The bigger the multiplier, the greater the number of collected curves needed. Examples are given in Table **??**.

3.2.5 Enhanced Simple Power Analysis on Blinded Exponentiations

In this section we consider that the exponentiation is secured using message and exponent blinding.

Exponent Blinding.

This common countermeasure consists in randomizing the secret exponent d by $d^\star = d + r_1 \cdot n \mod \phi(n)$ with r_1 being a random value. However here the exponent blinding has no effect on our analysis since a single curve is used to recover the private exponent and recovering $d^\star$ is equivalent to recovering d.

Randomized Chosen Message.

Now we consider that the message is randomized additively by the classical countermeasure: $m^\star = m + r_1 \cdot n \mod r_2 \cdot n$, with r_1 and r_2 being two l-bit random values. In this case we have $m^\star$ equal to $m + u \cdot n$ with u being a l-bit value equal to $r_1 \mod r_2$. In this case an attack could consist of choosing a message $m^\star$ being 1 or 2^i, guessing a random value u_{guess}, computing message m from guessed randomized message $m^\star$, i.e. $m = m^\star - u_{guess} \cdot n$, and executing at least 2^l exponentiations with input message m. One of the 2^l exponentiation power curves should present leakages and should allow the secret exponent to be recovered with SPA.

However if r_1 and r_2 are effectively chosen in a pure random way we observe that this attack could be done faster. Indeed if we analyze the distribution of values $u = r_1 \mod r_2$ we observe that values do not appear with same probability and that the more frequent are the smallest ones. The most frequent one being $u = 0$. It is illustrated in Figure **??** for $l = 8$. While less pronouced the same phenomenon can also be observed for bigger l values. The best attack method in this case would consist of choosing $u_{guess} = 0$ (or 1 or 2) and executing the exponentiation many times until a leaking power curve is obtained.

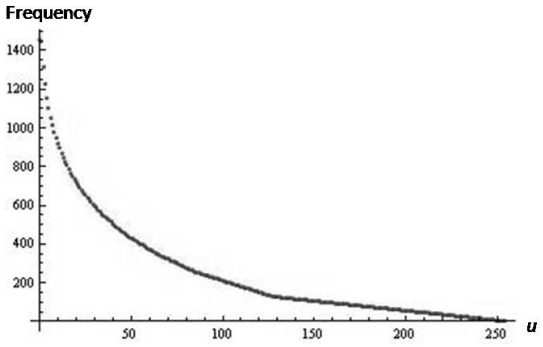


Figure 3.7: Distribution of u for $l = 8$

Unknown Message.

When analyzing the leakage probabilities of Section **??** it appears that the number of curves needed to recover the secret exponent for a fixed multiplier size only depends on the modulus length, even if the message is unknown to the attacker. For instance for a 1024-bit modulus and a 16-bit multiplier by collecting 5000 power consumption curves of exponentiations done with unknown different messages the probability of recovering the secret exponent is close to 1.

Synthesis.

As the additive randomization of the message does not significantly increase message length, the amount of messages needed does not increase either. Thus if the attacker can choose input messages of the blinded exponentiation, he will choose the attack which requires less effort comparing number of chosen message acquisitions needed (when guessing the random) with the number of curves to collect to have $P_{Leak} = 1$.

3.2.6 Countermeasures and Recommendations

Balancing the Power Consumption

The attack presented in this chapter is based on the fact that manipulating zero t-bit values results in low power consumption cycles. Thus a method to prevent this attack would consist in using balanced power consumption technology such as dual rail technique. In this case the manipulation of a value with a low Hamming weight (for instance 0) will no longer have a different power consumption than the one due to the manipulation of other values.

Random Choice for Blinding

As we showed previously the values $r_1 \mod r_2$ are not uniformly distributed when r_1 and r_2 are random. A better solution consists of choosing a fixed value for r_2 and a random value for r_1. From our analysis the best choice for r_2 is to take the biggest l-bit

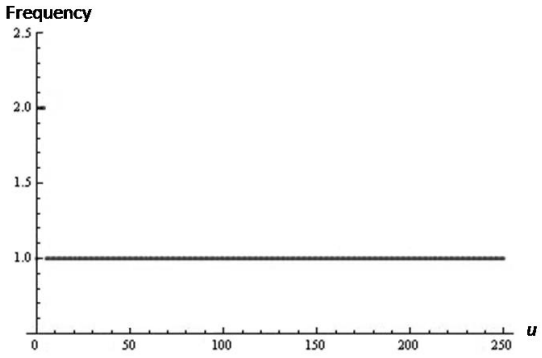


Figure 3.8: Distribution of u for $l = 8$ and $r_2 = 251$

prime number. In that case r_2 will never divide r_1, thus u cannot be null and u values are uniformly distributed as it is showed in Figure **??**.

Another consideration is that the random length choice is also directly related to the multiplier size. In section **??** we have seen that while the number of possible random values u is smaller than the number of messages to test given by the leakage probability analysis, it is easier to test all random values u. Regarding this statistical properties we showed that when the multiplier is small (8 or 16 bits) the quantity of curves needed for a successful Simple Power Analysis is reasonable.

Thus by combining a multiplier with a size of at least equal to 32 bits, with big random number r_1 (longer than 32 or 64 bits) and the biggest prime integer r_2, the feasibility of the attack explained in this chapter is significantly reduced.

3.2.7 Remarks on RSA CRT and ECC

We presented our analysis on exponentiation computations. It corresponds directly to straightforward implementations of RSA signature and decryption algorithms as they simply consist of an exponentiation with the secret exponent. In case of RSA CRT the analysis is a little bit different since the input message is reduced modulo p and q before the exponentiations. Even if data manipulated into the multiplications are twice shorter than the modulus n, similar analysis can be conducted on reduced messages. However the countermeasure which consists in fixing the random value r_2, must not be used in RSA CRT implementations as it would not be protected against the correlation analysis on the CRT recombination presented in [**?**].

ECC are also concerned. The analysis depends on the kind of coordinates and algorithm chosen for the scalar multiplication, anyway implementations using small multipliers and/or small random numbers for coordinates randomization [**?**] have to be avoided.

3.2.8 First Section Synthesis

In this section we have explained the origin of the power leakages during multiplications and presented other ways to mount Simple Power Analysis attacks. Indeed by observ-

ing differences in data power signatures instead of differences in code execution, using some well chosen messages allows the whole secret exponent of RSA cryptosystem to be recovered from a single curve. Moreover we have shown that some improvements in SPA attacks lead to the recovery of the secret exponent on secured exponentiations using blinding countermeasures and with non chosen messages. We analyzed the blinding countermeasures and gave advice to developers to protect their implementations against this enhanced SPA. Judicious choice and large random numbers in blinding countermeasures combined with large size multipliers, especially greater than 32 bits, are recommended for SPA resistance.

3.3 Improving the Previous Simple Side Channel Analysis on Exponentiation

Previously (and [**?**]) we have considered that during a long integer multiplication $R_0 \cdot R_1$, if the multiplicand $R_1 = m$ contains one (or more) of the t-bit words set to 0, it is possible to recognize each time this value m is manipulated all along the exponentiation, i.e. each time the exponent bit is 1.

In that case we say in the following a message m or an operand x are tagged because their manipulation can be distinguished.

Authors considered for their leakage statistical analysis during exponentiation scheme the following side-channel tag model:

[A_0] *Side-channel tag originates when a whole t bit word equals zero in the operand m.*

Notations: We denote by $tag(m^\star)$ the event "the operand m has a t-bit word equal to zero" and by $tag^i(m^\star)$ the event "the operand m has its i^{th} t-bit word equal to zero". For a given ℓ-word operand $x = (x_{\ell-1} \dots x_1 x_0)_b$ we introduce the following notations:

$$\begin{aligned} \overline{x_i} &= x \bmod b^{i+1} = (x_i \dots x_1 x_0)_b \\ \underline{x_i} &= x \bmod b^i = (x_{i-1} \dots x_1 x_0)_b \quad \text{with} \quad \underline{x_0} = 0 \end{aligned}$$

The general principle of the attack is based on the fact that whenever the randomized message $m^\star$ is tagged, this easily detectable event points the attacker to all LIM operations corresponding to multiplications by the message, which thus reveals the private exponent $d^\star$. The probability for a tag to occur is usually quite small so that the attacker has to acquire and analyze many side-channel traces until one of them eventually happens to be tagged.

3.3.1 Improving the Analysis

A first contribution of this section is to observe that an attacker who has control of the non randomized message m is able to further reduce the attack complexity – measured as the number of required side-channel traces – by causing tags on the randomized message $m^\star$ to happen more frequently than by pure chance. More precisely, for any word index

$0 \leqslant i < \ell$, and for any integer $0 \leqslant u^{(i)} < 2^{\lambda} - 1$ which denotes a targeted value for the random $u = r_1 \bmod r_2$ involved in the randomization of m, it is possible to find a message m such that $m^{\star} = m + u^{(i)} \cdot n$ is tagged on word i. This chosen message gives access to the private exponent whenever $u = u^{(i)}$ which may be more probable than would naturally arise, particularly when $\lambda < t$.

We can even do better since we will show that it is possible to build a message which simultaneously verifies such kind of conditional tag property on each of its words. Then in a second study we consider the scenario where the attacker does not have full control on the message which is to be randomized since we assume that this message is the output of a deterministic hash function whose input is chosen by the attacker.

3.3.2 Known Message Scenario

We assume here a known message scenario where the message value to be exponentiated is uniformly distributed over the set of all integers that can be represented on $\ell\, t$ bits.

Theorem 3.3.1 () *Given a message m uniformly distributed over $\{0, \ldots, 2^{\ell t} - 1\}$, the probability that the randomized message $m^{\star} = m + u \cdot n$ is tagged on any of its ℓ least significant words is:*

$$\begin{aligned} \mathrm{Proba}\big(tag(m^{\star})\big) &= 1 - (1 - 2^{-t})^{\ell} \\ &\simeq \ell \cdot 2^{-t} \end{aligned}$$

Proof 3.3.1 *For any $0 \leqslant i < \ell$, and any arbitrary integer $0 \leqslant u < 2^{\lambda} - 1$, letting $s = u \cdot n$, we have:*

$$\begin{aligned} \mathrm{Proba}_m\big(tag^{(i)}(m^{\star})\big) &= \mathrm{Proba}_m\left(m_i^{\star} = 0\right) \\ &= \mathrm{Proba}_m\left(m_i = -\left\lfloor \frac{\overline{s_i} + \overline{m_i}}{b^i} \right\rfloor \bmod b\right) \\ &= 2^{-t} \end{aligned}$$

Now, considering also u as random:

$$\begin{aligned} \mathrm{Proba}_{u,m}\big(tag^{(i)}(m^{\star})\big) &= \sum_{u} \mathrm{Proba}(u)\mathrm{Proba}_m\big(tag^{(i)}(m^{\star})\big) \\ &= 2^{-t} \end{aligned}$$

The proof follows immediately from the independence of the tag on each word.

In the known message only setting, the probability for a side-channel trace to be tagged is close to $\ell\, 2^{-t}$. This result holds whatever the probability distribution of u. In particular it makes no difference whether u is biased – which is the case when r_1 and r_2 are both random – or uniformly distributed.

3.3.3 Chosen Message Scenario

Theorem **??** and Algorithm **??** show how an attacker can build a message whose randomization will be tagged whenever u belongs to a set of ℓ prescribed chosen target values.

Theorem 3.3.2 () *Let $U = \left(u^{(0)}, \ldots, u^{(\ell-1)}\right)$ be an arbitrary set of ℓ targets, with $\forall i, 0 \leqslant u^{(i)} < 2^{\lambda}-1$. The message m returned by Algorithm **??** is such that $m^{\star} = m+u\cdot n$ is tagged on word i whenever $u = u^{(i)}$.*

Proof 3.3.2 *For each i, let $s^{(i)} = u^{(i)} \cdot n$. We have*

$$m_i \;=\; -\left\lfloor \frac{\overline{s_i^{(i)}} + \underline{m_i}}{b^i} \right\rfloor \bmod b$$

so that $(m + s^{(i)})_i = 0$ which implies that $m_i^{\star} = 0$ if $u = u^{(i)}$.

Alg. 3.3.1 Chosen message construction

Input: a ℓ-word modulus n and a set $\left(u^{(0)}, \ldots, u^{(\ell-1)}\right)$ of targeted randoms
Output: a message m whose randomization is tagged for any specified target

1. $m \leftarrow 0$
2. **for** $i = 0$ **to** $\ell - 1$ **do**
3. $\quad s^{(i)} \leftarrow u^{(i)}\, n$
4. $\quad \mu \leftarrow -\left\lfloor \frac{\overline{s_i^{(i)}}+m_i}{b^i} \right\rfloor \bmod b$
5. $\quad m \leftarrow m + \mu\, b^i$
6. **return** m

We now compute the probability that a randomization of the message returned by Algo. **??** is tagged:

$$\begin{aligned} \mathrm{Proba}\left(tag^{(i)}(m^{\star})\right) \;&=\; \mathrm{Proba}(u = u^{(i)}) \cdot 1 + \mathrm{Proba}(u \neq u^{(i)}) \cdot 2^{-t} \\ &\simeq\; \mathrm{Proba}(u = u^{(i)}) + 2^{-t} \\ &\simeq\; \begin{cases} 2^{-t} & \text{if } \lambda > t \\ 2^{-\lambda} & \text{if } \lambda \leqslant t \end{cases} \\ &\simeq\; \max(2^{-\lambda}, 2^{-t}) \end{aligned} \tag{3.1}$$

Equation **??** clearly shows that our chosen message method is particularly interesting when $\lambda \leqslant t$. Indeed, when $\lambda > t$ the randomized message is tagged with same probability than in the known message model. For this reason we consider from now on that $\lambda \leqslant t$. In that case choosing the message according to Algo. **??** changes the complexity of tag probability from $\mathcal{O}(2^{-t})$ to $\mathcal{O}(2^{-\lambda})$. Depending on λ, the attack may now be feasible even on large multipliers (e.g. $t \geqslant 64$) as the tag probability does not depend on t any more.

When u has uniform distribution the choice of the $u^{(i)}$s is not relevant provided they are all distinct. In that case we have:

$$\text{Proba}\big(tag(m^\star)\big) \;\simeq\; \ell\, 2^{-\lambda}$$

When u is biased due to the random choice of both r_1 and r_2 the smaller u the more probable it is. The best strategy for an attacker is then to choose $U = (0, \ldots, \ell - 1)$ which has the largest probability. This results in a tag probability that can be expressed as:

$$\begin{aligned}\text{Proba}\big(tag(m^\star)\big) &\simeq \text{Proba}(u \in U)\\ &\simeq \omega\, \ell\, 2^{-\lambda}\end{aligned}$$

where $\omega \geqslant 1$ is a multiplicative factor which quantifies the gain related to the biased case compared to the uniform one.

Let's now enumerate the three advantages from which our chosen message attack benefits:

1. Considering some given word of the randomized message, the probability that it is tagged is at least $2^{-\lambda}$ instead of 2^{-t} (for $\lambda \leqslant t$). This is by far the more fundamental advantage provided by our method.

2. As it is possible to simultaneously generate a conditional tag on all words, the probability of a tag on $m^\star$ is l times that of a tag on a single word. Note that this gain by a factor l also holds in the known message model.

3. In case of biased randomization – which is more usually implemented than the uniform randomization – the attacker targets the most probable random values u. This results in another gain by a factor ω which is far from being negligible as shown in Table **??**.

Results

For different sets of parameters t, λ we have simulated our attack on a large number of runs by generating a random 1024-bit modulus n, building a message m according to Algo. **??**, computing a randomized message $m^\star$ by applying the classical biased masking procedure, and testing whether $m^\star$ is tagged.

We present in Table **??** the experimental averaged tag probabilities, together with the theoretical ones for comparison. We also mention the resulting mean number of side-channel traces needed, the gain factor ω, as well as the number of simulation runs in each case.

From a practical point of view, the proposed chosen message method allows our tag-based simple side-channel analysis on randomized exponentiation to be feasible in a much wider range of settings. Definitely, the security against our attack cannot be provided by a large multiplier. Also, Table **??** shows that the mean number of traces required to recover the private exponent is small for $\lambda = 16$ and quite practicable for $\lambda = 24$, while these random bit-length values may be considered providing enough security for message blinding purpose. In light of our method, we can say that message blinding must not use random values smaller than 32 bits.

Table 3.4: Simulation results of the chosen message attack for a 1024-bit RSA key with biased randomization

		Tag probability		Number of traces		Gain ω	
		Simu	Theory	Simu	Theory	Simu	Theory
$\lambda = 8$ (10^6 runs)	$t = 16$	$6.50\,10^{-1}$	$6.51\,10^{-1}$	1.54	1.54	2.60	2.60
	$t = 32$	$4.28\,10^{-1}$	$4.28\,10^{-1}$	2.33	2, 33	3.43	3.43
	$t = 64$	$2.63\,10^{-1}$	$2.62\,10^{-1}$	3.80	3.81	4.21	4.20
$\lambda = 16$ (10^7 runs)	$t = 16$	$8.30\,10^{-3}$	$8.30\,10^{-3}$	121	121	8.50	8.50
	$t = 32$	$4.49\,10^{-3}$	$4.48\,10^{-3}$	223	223	9.19	9.18
	$t = 64$	$2.42\,10^{-3}$	$2.41\,10^{-3}$	414	415	9.89	9.86
$\lambda = 24$ (10^8 runs)	$t = 16$	—	—	—	—	—	—
	$t = 32$	$2.77\,10^{-5}$	$2.81\,10^{-5}$	36062	35590	14.5	14.7
	$t = 64$	$1.48\,10^{-5}$	$1.47\,10^{-5}$	67476	68049	15.5	15.4
$\lambda = 32$ (10^9 runs)	$t = 16$	—	—	—	—	—	—
	$t = 32$	—	—	—	—	—	—
	$t = 64$	$8.3\,10^{-8}$	$7.78\,10^{-8}$	$12.0\,10^6$	$12.8\,10^6$	22.3	20.9

3.3.4 Hashed Message Scenario

In this section, we consider a more restricted model where the message is hashed and padded before being randomized and then exponentiated. We still assume that the message m is chosen by the attacker, but the aim is now to obtain a tag on $h^\star$ where:

$$\begin{cases} h &= \mathcal{H}(m) \\ h^\star &= h + u \cdot n \end{cases}$$

We assume that $\mathcal{H}$ is a deterministic hash and pad function – e.g. the full domain hash RSA-FDH [**?**]. Because we do not have control on the hash output, it is not possible to directly set some word of h to that precise value which would create a tag for some given targeted u. Rather we can try to search for some m whose hash has this property. Suppose we want to tag the least significant word of $h^\star$. In order for that word to be tagged for a prescribed target u, we must find a message m such that $h_0 = -s_0 \bmod b$ with $s = u \cdot n$. This allows the attack to necessitate only $\mathcal{O}(2^\lambda)$ side-channel traces as in the chosen message model, but requires an average of $\mathcal{O}(2^t)$ hash computations.

We can do better if we allow any u value to be targeted. Let $S_0 = \{s_0 = (u \cdot n)_0\}$ where $0 \leqslant u < 2^\lambda - 1$. Then we only have to find a message such that $-h_0 \in S_0$. Provided that $\lambda \leqslant t$, the number of distinct values in S_0 is close to 2^λ and the search for a convenient message requires $\mathcal{O}(2^{t-\lambda})$ hash computations and $\mathcal{O}(t\,2^\lambda)$ space storage. We thus found a (time : memory : data) tradeoff – where *data* means the number of side-channel traces required – which achieves $(2^{t-\lambda} : t\,2^\lambda : 2^\lambda)$ complexity.

A further improvement consists in allowing the tag to appear on any word. Defining

$$S = \bigcup_{i=0}^{\ell-1} S_i \quad \text{where } S_i = \{s_i = (u \cdot n)_i\}$$

we now have about $\ell\,2^\lambda$ elements in S so that the tradeoff complexity becomes $(2^{t-\lambda}/\ell : \ell\,t\,2^\lambda : 2^\lambda)$.

This proposed hashed message attack admits three drawbacks compared to the chosen message one:

1. We do not see any means to simultaneously target different u on different words. As a consequence the number of traces required does not benefit from the division by ℓ.

2. Also it seems impossible to provoke a tag for a prescribed u – except if we accept a time complexity $\mathcal{O}(2^{-t})$ instead of $\mathcal{O}(2^{t-\lambda})$. Thus, the number of traces required is not divided by the gain factor ω.

3. The method requires the pre-computation of $\mathcal{O}(2^{t-\lambda})$ hash values and the storage of $\ell\,t\,2^\lambda$ bits.

Despite these drawbacks we think that there are some settings for which the proposed hashed message method can be practically applied while the known message one would be infeasible. For instance when $t = 32$ and $\lambda = 16$ the attack needs 2^{16} traces and a

short pre-computation phase, while it would require 2^{29} traces in the known message model to break a 1024-bit key.

Note that the method described in this section seems restricted to the use of a deterministic padding. It is an open question whether it could be modified to apply also to probabilistic padding schemes such as RSA-PFDH [**?**] or RSA-PSS [**?**].

Those analysis exploits the well-known efficient leakage model $[A_0]$ to design an SSCA efficient chosen message technique which improves the previous results and to propose a hashed message attack. In the following we consider now a relaxed model leakage. Indeed it is also realistic to consider less restrictive leakage models for a side-channel tag to appear in a multiplication calculation. With these new leakage models we give new results that highlight SSCA is still more efficient than said previously to defeat state of the art blinded exponentiations.

3.3.5 Relaxed Side-Channel Leakage Model

We assume here a tag in a message could be due to two following assumptions that are not independent:

$[A_1]$ *Side-channel tag originates from the fact that at least τ consecutive bits in a t-bit word of m are set to 0, with $\tau \leqslant t$.*

$[A_2]$ *Side-channel tag originates from the fact that the Hamming weight h of the t-bit word is lower than a value ν, with $h \leqslant \nu < t$.*

Both assumptions $[A_1]$ and $[A_2]$ are realistic and well suited for hardware implementations of multipliers. The choice of the most relevant model between $[A_1]$ and $[A_2]$ and the best values of parameters τ and ν varies from one integrated circuit to another one, it also depends on t. From our experiments we observed that some integrated circuits are more resistant than others.

In this sequel we separately consider the two leakage models given by both assumptions $[A_1]$ and $[A_2]$[1].

We say that x is A_1-tagged on word i whenever x_i contains at least τ consecutive zero bits. This event will be denoted by $tag_{A_1}^{(i)}(x)$. We also denote by $tag_{A_1}(x)$ the event that x is A_1-tagged on at least one of its words.

In the same way, we say that x is A_2-tagged on word i whenever the Hamming weight of x_i is less than ν, and this event will be denoted by $tag_{A_2}^{(i)}(x)$. We also denote by $tag_{A_2}(x)$ the event that x is A_2-tagged on at least one of its words.

In the following let's denote by p the probability for a t-bit word to be either A_1-tagged or A_2-tagged depending on the considered leakage model.

Theorem 3.3.3 () *Given a message m uniformly distributed over $\{0, \ldots, 2^{\ell t} - 1\}$, the probability that the randomized message $m^\star = m + u \cdot n$ is tagged on any of its ℓ least*

[1] $[A_0]$ leakage model is a particular case of model $[A_1]$ (resp. $[A_2]$) when τ equals t (resp. when ν is null).

significant words is:

$$\begin{aligned} \text{Proba}\big(tag(m^\star)\big) &= 1-(1-p)^\ell \\ &\simeq \ell \cdot p \end{aligned}$$

3.3.6 Tag Probabilities for τ and t Values with $[A_1]$ Leakage Model

Considering the leakage model $[A_1]$ we have computed the different p values for all τ values in the range $[0, \dots t]$. We have then exhausted the number n_τ of existing words which have their longest consecutive zeros sequence being of exact length τ. Knowing this number we compute $p_1(t,\tau)$ the probability for a t-bit word to have its longest consecutive zero sequence to be exactly τ: $p_1(t,\tau) = n_\tau/(2^t)$. Then we have $\text{Proba}(tag^{(i)}_{A_1}(x)) = \sum_{j=\tau}^{t} p_1(t,j)$. Once we obtain these different $tag^{(i)}_{A_1}(x)$ values we compute the $tag_{A_1}(m)$ probabilities for 512, 1024 and 2048 bits long integer messages.

Case $t = 16$

Table **??** gives result examples for a $t = 16$-bit multiplier architecture. [2]

Considering for instance the case $\tau = 12$, we observe there are 28 words which have their longest consecutive zeros sequence being of length 12. The probability for a word to be exactly τ bit A_1 tagged is $p_1(16,12) = 4.27 \times 10^{-4}$. The probability for a word to have at least $\tau = 12$ consecutive zero bits is then $\text{Proba}(tag^{(i)}_{A_1}(x)) = \sum_{i=12}^{16} p_1(16,i) = 7.32 \times 10^{-4}$.

It is then worth to notice the probability a 1024-bit integer is tagged is reduced from 9.76×10^{-4} to 4.58×10^{-2} from model $[A_0]$ to model $[A_1]$ with $\tau = 12$ which can happen in practice. It means that only 22 ($\approx 1/(4.58\,10^{-2})$) messages would be enough for recovering the secret exponent in a 1024-bit blinded exponentiation with probability $1/e \approx 0.63$ instead of 1020 messages when considering $[A_0]$. Finally to reach a leakage probability equal to 0.999 SSCA would require only 140 messages and not 6700 when considering the previous leakage model $[A_0]$.

Case $t = 32$

We processed the same study for a 32-bit multiplier. Table **??** gives result examples.

In [**?**] authors considered that using a 32-bit multiplier counterfeited simple side-channel analysis in blinded exponentiation when random used for blinding were big enough (i.e. $\geqslant 32$ bits). We observe here than it is not exact considering the relaxed but realistic model $[A_1]$. Indeed considering τ equal to 16 we obtain $\text{Proba}(tag^{(i)}_{A_1}(x)) = 1.37 \times 10^{-4}$, it signifies $\text{Proba}(tag_{A_1}(m)) = 4.39 \times 10^{-3}$ for m a 1024-bit integer message. It means that 230 messages would be enough for recovering the secret exponent in a 1024-bit blinded exponentiation with probability $1/e \approx 0.63$ instead of 1.34×10^8 messages when considering $[A_0]$. Moreover to reach a leakage probability equal to 0.999 only 1480 messages are required instead of 8.73×10^8.

[2] The complete result tables of our analysis for $[A_1]$ and $[A_2]$ models, considering all possible τ and ν values in the range $[0,\dots, t]$ are given in the extended version of this analysis [**?**].

Table 3.5: $[A_1]$ Leakage probability examples for some τ values when $t = 16$

τ	t-bit word number	$p_1(t,\tau)$	$P(tag_{A_1}^{(i)}(x))$	$P(tag_{A_1}(m_{512}))$	$P(tag_{A_1}(m_{1024}))$	$P(tag_{A_1}(m_{2048}))$
0	1	$1.53\,10^{-05}$	1	1	1	1
4	1300 8	$1.98\,10^{-01}$	$3.95\,10^{-01}$	1	1	1
8	704	$1.07\,10^{-02}$	$1.95\,10^{-02}$	$4.68\,10^{-01}$	$7.17\,10^{-01}$	$9.20\,10^{-01}$
12	28	$4.27\,10^{-04}$	$7.32\,10^{-04}$	$2.32\,10^{-02}$	$4.58\,10^{-02}$	$8.95\,10^{-02}$
16	1	$1.53\,10^{-05}$	$1.53\,10^{-05}$	$4.88\,10^{-04}$	$9.76\,10^{-04}$	$1.95\,10^{-03}$

τ	t-bit word number	$p_1(t,\tau)$	$P(tag_{A_1}^{(i)}(x))$	$P(tag_{A_1}(m_{512}))$	$P(tag_{A_1}(m_{1024}))$	$P(tag_{A_1}(m_{2048}))$
0	1	$2.33\,10^{-10}$	1	1	1	1
8	111246728	$2.59\,10^{-02}$	$5.02\,10^{-02}$	$5.61\,10^{-01}$	$8.08\,10^{-01}$	$9.63\,10^{-01}$
16	311296	$7.25\,10^{-05}$	$1.37\,10^{-04}$	$2.20\,10^{-03}$	$4.39\,10^{-03}$	$8.75\,10^{-03}$
24	704	$1.64\,10^{-07}$	$2.98\,10^{-07}$	$4.77\,10^{-06}$	$9.54\,10^{-06}$	$1.91\,10^{-05}$
32	1	$2.33\,10^{-10}$	$2.33\,10^{-10}$	$3.73\,10^{-09}$	$7.45\,10^{-09}$	$1.49\,10^{-08}$

Table 3.6: $[A_1]$ Leakage probability examples for some τ values when $t = 32$

We have studied the leakage probabilities for exponentiation with the $[A_1]$ model. Our analysis highlights the risk of SSCA leakage even when the hardware multiplier base size is big, for instance 32-bit contrarily to previous section results. In the following we reproduce the same study for the second ($[A_2]$) model leakage.

3.3.7 Tag Probabilities for ν and t Values with $[A_2]$ Leakage Model

The number of t-bit words which have their Hamming weight being μ is $\binom{\mu}{t}$. The probability for a t-bit word to have its Hamming weight being exactly μ is $p_2(t,\mu) = \binom{\mu}{t} \cdot 2^{-t}$. Thus we obtain the probability for a t-bit word to be ν $[A_2]$ tagged is:

$$\text{Proba}(tag_{A_2}^{(i)}(x)) = \frac{\sum_{\mu=0}^{\nu} \binom{\mu}{t}}{2^t}. \tag{3.2}$$

Using this simple formula we compute in the following the values $\text{Proba}(tag_{A_2}^{(i)}(x))$ and $\text{Proba}(tag_{A_2}(m))$ for t=16 and $t = 32$ bits multipliers and different message bit-length.

Case $t = 16$

Table **??** gives results examples of $\text{Proba}(tag_{A_2}^{(i)}(x))$ and $\text{Proba}(tag_{A_2}(m))$ for t=16.
Considering for instance the case $\nu = 2$, the probability a 1024-bit integer is tagged is $\text{Proba}(tag_{A_2}(m_{1024}) = 1.24 \times 10^{-1}$. It signifies that only 8 messages would be enough for recovering the secret exponent in a 1024-bit blinded exponentiation with a probability of success equal to $1/e \approx 0.63$. Finally to reach a leakage probability equal to 0.999 SSCA it would require only 49 messages (exponentiation executions).

Case $t = 32$

We processed the same study for a 32-bit multiplier.
We consider here a device where the power leakage appears for this $[A_2]$ model when $\nu = 4$, we know by experiments it is a realistic case. The probability a 1024-bit integer is tagged becomes $\text{Proba}(tag_{A_2}(m_{1024})) = 3.09 \times 10^{-4}$. It means that only 3.24×10^3 messages would be enough for recovering the secret exponent in a 1024-bit blinded exponentiation with probability $1/e \approx 0.63$ instead of 1.34×10^8 when considering the $[A_0]$ model. Moreover to reach a leakage probability equal to 0.999 2.1×10^4 messages are required instead of 8.73×10^8.

Synthesis

We have discussed the probability of SSCA leakage for the two relaxed models $[A_1]$ and $[A_2]$ we have introduced. We have shown that the previous model $[A_0]$ is too restrictive and that even for big size multipliers like 32-bit ones it is possible with a reasonable number of executions to recover the private exponent in a blinded exponentiation.

To illustrate our results we gives in Table **??** different leakage probabilities for different models we consider realistic. Of course this table is an example and each integrated circuit will have different leakage characteristic. It is then important to measure the right values τ and ν for each integrated circuit.

ν	t-bit word number	$p_2(t,\nu)$	$P(tag_{A_2}^{(i)}(x))$	$P(tag_{A_2}(m_{512}))$	$P(tag_{A_2}(m_{1024}))$	$P(tag_{A_2}(m_{2048}))$
0	1	$1.53\,10^{-05}$	$1.53\,10^{-05}$	$7.78\,10^{-03}$	$1.55\,10^{-02}$	$3.08\,10^{-02}$
2	120	$1.83\,10^{-03}$	$2.08\,10^{-03}$	$6.43\,10^{-02}$	$1.24\,10^{-01}$	$2.33\,10^{-01}$
4	1820	$2.78\,10^{-02}$	$3.84\,10^{-02}$	$7.14\,10^{-01}$	$9.18\,10^{-01}$	$9.93\,10^{-01}$
8	12870	$1.96\,10^{-01}$	$5.98\,10^{-01}$	1	1	1
12	1820	$2.78\,10^{-02}$	$9.89\,10^{-01}$	1	1	1
16	1	$1.53\,10^{-05}$	1	1	1	1

Table 3.7: $[A_2]$ Leakage probability for some ν values when $t = 16$

ν	t-bit word number	$p_2(t,\nu)$	$P(tag_{A_2}^{(i)}(x))$	$P(tag_{A_2}(m_{512}))$	$P(tag_{A_2}(m_{1024}))$	$P(tag_{A_2}(m_{2048}))$
0	1	$2.33\,10^{-10}$	$2.33\,10^{-10}$	$3.73\,10^{-09}$	$7.45\,10^{-09}$	$1.49\,10^{-08}$
4	35960	$8.37\,10^{-06}$	$9.65\,10^{-06}$	$1.54\,10^{-04}$	$3.09\,10^{-04}$	$6.17\,10^{-04}$
8	10518300	$2.45\,10^{-03}$	$3.50\,10^{-03}$	$5.46\,10^{-02}$	$1.06\,10^{-01}$	$2.01\,10^{-01}$
16	601080390	$1.40\,10^{-01}$	$5.70\,10^{-01}$	1	1	1
24	10518300	$2.45\,10^{-03}$	$9.99\,10^{-01}$	1	1	1
32	1	$2.33\,10^{-10}$	1	1	1	1

Table 3.8: $[A_2]$ Leakage probability for some ν values when $t = 32$

$\tau,\ \nu$	t-bit word number	p	$P(tag_{A_i}(m_{512}))$	$P(tag_{A_i}(m_{1024}))$	$P(tag_{A_i}(m_{2048}))$
$[A_2]\ \nu = 4$	$8.37\,10^{-06}$	$9.65\,10^{-06}$	$1.54\,10^{-04}$	$3.09\,10^{-04}$	$6.17\,10^{-04}$
$[A_1]\ \tau = 16$	$7.25\,10^{-05}$	$1.37\,10^{-04}$	$2.20\,10^{-03}$	$4.39\,10^{-03}$	$8.75\,10^{-03}$
$[A_0]$	$2.33\,10^{-10}$	$2.33\,10^{-10}$	$3.73\,10^{-09}$	$7.45\,10^{-09}$	$1.49\,10^{-08}$

Table 3.9: Leakage probability examples for t=32

$\tau,\ \nu$	m_{512}	m_{1024}	m_{2048}
$[A_2]\ \nu = 4$	$4.22\,10^4$	$2.11\,10^4$	$1.06\,10^3$
$[A_1]\ \tau = 16$	$3\,10^3$	$1.5\,10^3$	750
$[A_0]$	$1.75\,10^9$	$8.73\,10^8$	$4.37\,10^8$

Table 3.10: Number of messages/executions needed for leakage probability at $0,999$, for t=32

It is important to notice that SSCA is much more efficient than previous studies said and particularly can threaten blinded exponentiation implemented with 32-bit cores which are commonly used today. Of course it depends on the kind of hardware selected for the implementation, it is then very important to measure the exact side-channel leakage of the multiplier, for instance the exact values τ and ν in our two models.

3.3.8 Countermeasures and Recommendations

We have shown previously that some mandatory conditions must be respected to prevent any implementations from the enhanced simple side-channel analysis.

Hardware Multiplier Characterization The first consideration to take into account consists in precisely characterizing the leakage characteristics of any designed hardware multiplier. Effectively, contrarily to the previous section results, we have shown that for a t-bit hardware multiplier the leakage probability does not depend only of t but more of the values τ and ν we described previously. It is of particular interest when $t = 32$ as previous studies considered using such a hardware multiplier hardware rendered the SSCA not available if the blinded exponentiation was using big enough random values. But we have shown that it is not true. Indeed, whatever the random size used for blinding and base t value are, if the value ν (resp. τ) is much smaller than t (much bigger than 0) then SSCA can defeat a state of the art blinded exponentiation. It is then important to determine for leakage models $[A_1]$ and $[A_2]$ the values τ and ν leading to a power tag of the selected multiplier in order to determine the exact power leakage of an exponentiation. Once these exact values are determined a developer can select the appropriate algorithm and countermeasure(s) he must use (or not use) for his implementation to be secure enough.

Moreover it is obvious that hardware countermeasures such as jitter, clock divider or in best cases balanced consumption circuits should be also present in the embedded product to enforce the resistance to side-channel analysis and render enhanced SSCA more difficult.

The previous study recommendation still applies: "λ (random bit-length) value must be bigger than 32 bits whatever the value of τ and ν still applies. It is also still recommended to use a constant (rather than random) value for r_2. For instance r_2 could be equal to $2^{\lambda} - 1$."

Exponentiation Algorithm Choice For better resistance we recommend to select an exponentiation algorithm resistant to this analysis. The best solution to us consists in always using right-to-left blinded exponentiation algorithm instead of left-to-right classical ones. As already highlighted by Walter [**?**] and later by Fouque et al. in [**?**] this implementation is much more resistant to the many side-channel attacks than the left-to-right ones. Indeed the square operations being applied on the message value the operands used in multiplications are never the same and it is not possible any more to observe tags on any message multiplication in a same trace. Developers can also decide to apply a new message randomization on the message operand m used in exponentiation after each multiplying (squaring and multiplying) operation, for instance by using for

message the new value $m = m + n \mod r_2 \cdot n$. It is also interesting to notice than in case of Barrett or Montgomery reduction methods, the resistant reduction algorithms given in [**?**] offers a good protection.

3.3.9 Conclusion

In this chapter we have presented some SSCA improvements enhancing simple side-channel analysis to recover of the secret exponent manipulated during state of the art blinded embedded exponentiations, when all the other side-channel techniques are inefficient. We have also demonstrated how to build a chosen message more significantly to reduce the number of needed execution for SSCA attack to succeed with a higher probability. Moreover we have shown that, contrarily to a common belief, simple side-channel analysis can be successful also in some hashed message models. Our results depend on the size of the random values used for blinding and the way they are generated, as well as on the hardware multiplier leakage properties. We have also presented two new side-channel leakage models we consider realistic and well suited for long integer multiplications and exponentiation side-channel analysis. We observe that SSCA remains a very powerful side-channel analysis to defeat blinded exponentiation even when using big random values and big multipliers. Indeed it requires a deep characterization of the hardware multiplier used.

Our new analysis strengthens again the advice previously given by Walter [**?**] at CHES 2001 to use right-to-left exponentiations. Although less often used than left-to-right exponentiation, right-to-left methods appear to be much more resistant against the numerous side-channel attacks.

We then conclude this chapter with the sentence (title) from Fouque and Valette at CHES 2003 [**?**]:

"Upwards is better than downwards!".

Chapter 4

Horizontal Analysis on Exponentiation

4.1 Introduction

Securing embedded products from *Side-Channel Analysis* (SCA) has become a difficult challenge for developers who are confronted with more and more analysis techniques as the physical attacks field is studied. Since the original *Simple* Side-Channel Analysis (SSCA) – which include *Timing Attacks*, SPA, and SEMA – and *Differential* Side-Channel Analysis (DSCA) – including DPA and DEMA – have been introduced by Kocher et al. [**?**, **?**] many improvements and new SCA techniques have been published. Messerges et al. were the first to apply these techniques to public key implementations [**?**]. Later on, original DSCA has been improved by more efficient techniques such as the one based on the *likelihood test* proposed by Bevan et al. [**?**], the *Correlation* Power Analysis (CPA) introduced by Brier et al. [**?**], and more recent techniques like the *Mutual Information Analysis* (MIA) [**?**, **?**, **?**]. A common principle of all these techniques is that they require many power consumption or electromagnetic radiation curves to recover the secret manipulated. Hardware protections and software blinding [**?**, **?**] countermeasures are generally used and when correctly implemented they counteract these attacks.

Among all those studies the so-called *Big Mac attack* is a refined approach introduced by Walter [**?**, **?**] from which our contribution is inspired. This technique aims at distinguishing squarings from multiplications and thus recovering the secret exponent of an RSA exponentiation with a single execution curve. This can be achieved by averaging and comparing the cycles of a device multiplier during long integer multiplications.

We present in this chapter another analysis which uses a single curve. We named this technique *horizontal* correlation analysis, which consists of computing classical statistical treatments such as the correlation factor on several segments extracted from a single execution curve of a known message RSA encryption. Since this analysis method requires only one execution of the exponentiation as the Big Mac attack, it is then not prevented by the usual exponent blinding countermeasure.

The horizontal correlation analysis is presented in Section **??** with some practical results and a comparison between our technique and the Big Mac attack. Known and

new countermeasures are discussed in Section **??**. In Section **??** we deal with horizontal side channel analysis in the most common cryptosystems.

4.1.1 Side-Channel Analysis

We have chosen to introduce in this chapter the terms of *vertical* and *horizontal* side-channel analysis to classify the different known attacks. The present section deals with known vertical and horizontal power analysis techniques. Our contribution, the horizontal correlation analysis on exponentiation is detailed in Section **??**.

Remember, as we said previously, side-channel attacks rely on the following physical property: a microprocessor is physically made of thousands of logical gates switching differently depending on the executed operations and on the manipulated data. Therefore the power consumption and the electromagnetic radiation, which depend on those gates switches, reflect and may leak information on the executed instructions and the manipulated data. Consequently, by monitoring the power consumption or radiation of a device performing cryptographic operations, an observer may recover information on the implementation of the program executed and on the secret data involved.

Simple Side-Channel Analysis

In the case of an exponentiation, original SSCA consists in observing that, if the squaring operation has a different pattern from the one of the multiplication, the secret exponent can be read on the curve. Classical countermeasures consist of using so-called *regular* algorithms like the *square and multiply always* or Montgomery ladder algorithms [**?**, **?**], *atomicity* principle which leads to regular power curves.

Differential Side-Channel Analysis

Deeper analysis such as DSCA [**?**] can be used to recover the private key of an SSCA protected implementation. These analyses make use of the relationship between the manipulated data and the power consumption/radiation. Since this leakage is very small, hundreds to thousands of curves and statistical treatment are generally required to learn a single bit of the exponent. Usual countermeasures consist of randomizing the modulus, the message, and/or the exponent.

Correlation Power Analysis

This technique is essentially an improvement of the Differential Power Analysis. Initially published by Brier et al. [**?**] to recover secrets on symmetric implementations, CPA is also successful in attacking asymmetric algorithms [**?**] with much fewer curves than classical DPA. The power consumption of the device is supposed to vary linearly with $\mathrm{H}(D \oplus R)$, the Hamming distance between the data manipulated D and a *reference state* R. The consumption model W is then defined as $W = \mu \cdot \mathrm{H}(D \oplus R) + \nu$, where ν captures both the experimental noise and the non modelized part of the power consumption. The linear correlation factor on N traces, $\rho_{C,H}$, is then used to correlate each power curve C with $\mathrm{H}(D \oplus R)$.

$$\rho_{C,H} = \frac{cov(C,H)}{\sigma_C \sigma_H} = \frac{t\sum(C_i H_{i,R}) - \sum C_i \sum H_{i,R}}{\sqrt{t\sum C_i^2 - (\sum C_i)^2}\sqrt{t\sum H_{i,R}^2 - (\sum H_{i,R})^2}}, \qquad i = 1 \ldots N$$

The maximum correlation factor being obtained for the right guess of secret key bits, an attacker can try all possible secret bits values and select the one corresponding to the highest correlation value.

In [**?**], Amiel et al. apply the CPA to recover the secret exponent of public key implementations. Their practical results show that the number of curves necessary to an attack is much lower compared to DPA: less than one hundred of curves is sufficient. It is worth noticing that the correlation is the highest when computed on t bits, t being the bit length of the device multiplier.

The authors shows the details [**?**, Fig. 8] of the correlation factor obtained for every multiplicand t-bit word A_i during the squaring operation $A \times A$ using a hardware multiplier. Interestingly a correlation peak occurs for $\mathrm{H}(A_i)$ each time a word A_i is involved in a multiplication $A_i \times A_j$.

We present in the next section our horizontal correlation analysis which takes advantage of this observation.

Collision Power Analysis

The *Doubling attack* from Fouque and Valette [**?**] is the first collision technique published on public key implementations. It is originally presented on elliptic curve scalar multiplication but can be applied on exponentiation algorithms. It recovers the whole secret scalar (exponent) with only a couple of curves. Other collision attacks have been presented in [**?**, **?**, **?**]. They all require at least two power execution curves, therefore the classical exponent randomization (blinding) countermeasure counterfeits those techniques.

Notations Let C^k denote the portion of an exponentiation curve C corresponding to the k-th long integer multiplication, and $C^k_{i,j}$ denote the curve segment corresponding to the internal multiplication $x_i \times y_j$ in C^k.

Big Mac Attack

Walter's attack needs, as our technique, a single exponentiation power curve to recover the secret exponent. For each long integer multiplication, the Big Mac attack detects if the operation processed is either $a \times a$ or $a \times m$. The operations $x_i \times y_j$ – and thus curves $C^k_{i,j}$ – can be easily identified on the power curve from their specific pattern which is repeated l^2 times in the long integer multiplication loop. A template power trace T^1_m is computed (either from the pre-computations or from the first squaring operation) to characterize the message value m manipulation during the long integer multiplication. The Euclidean distance between T^1_m and each long integer multiplication template power trace is then computed. If it exceeds a threshold the multiplication trace is supposed to be a squaring, and a multiplication by m otherwise. An example of such calculation

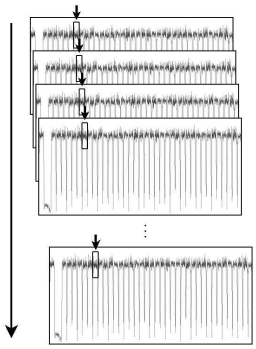


Figure 4.1: Vertical Side Channel Analysis

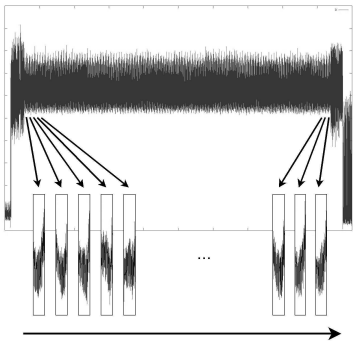


Figure 4.2: Horizontal side-channel analysis

is given in the following: for each t-bit word m_i of the message m, compute $T^1_{m_i} = \frac{1}{l}\sum_{j=0}^{l-1} C^1_{i,j}$ by averaging the l subcurves $C^1_{i,0} \ldots C^1_{i,l-1}$. Then the template curve T^1_m is the concatenation of curves $T^1_{m_0} ... T^1_{m_{l-1}}$. In the exponentiation loop, at each k-th long integer multiplication, the curve T^k is computed in the same manner. The Euclidean distance between T^k and T^1_m is computed. If it exceeds a threshold the multiplication is supposed to be a squaring, and a multiplication by m otherwise. The attack is innovative and has been presented by Walter with theoretical and simulation results. The efficiency of the attack increases with the key length and decreases with the multiplier size.

Cross-Correlation

Cross correlation technique has been used in [**?**] to try to recover the secret exponent in a single curve. However the cross correlation curve obtained by the authors did not allow distinguishing a multiplication from a squaring. More generally no successful practical result for cross correlation using a single exponentiation power curve has been yet published.

4.1.2 Vertical and Horizontal Attacks Classification

We refer to the techniques analysing a same time sample in many execution curves – see Fig. **??** – as *vertical* side-channel analysis. The classical DPA and CPA techniques thus fall into this category. We also include in the vertical analysis class the collision attacks mentioned above. Indeed even if many points on a same curve are used by those techniques, they require at least two power execution curves and manipulate them together. All those attacks are avoided with the exponent blinding countermeasure presented by Kocher [**?**, Section 10].

We propose the *horizontal* side-channel analysis denomination for the attacks using a single curve. First known horizontal power analysis is the classical SPA. Single curve Cross-correlation and Big Mac attacks are also horizontal techniques.

Our attack, we present in the next section, computes the correlation factor on many curve segments extracted from a single consumption/radiation curve as depicted in Fig. **??**. It thus contrasts with vertical attacks which target a particular instant of the

execution in several curves. The exponent blinding is not an efficient countermeasure against horizontal attacks.

4.2 Horizontal Correlation Analysis

We present hereafter our attack on an atomically protected RSA exponentiation using Barrett reduction.

Alg. 4.2.1 Multiply Always Barrett Exponentiation

Input: integers m and n such that $m < n$, v-bit exponent $d = (d_{v-1}d_{v-2}\dots d_0)_2$
Output: $\text{Exp}(m, d, n) = m^d \bmod n$

1. $a \leftarrow 1$
2. Process Barrett reduction precomputations
3. **for** i **from** $v-1$ to 0 **do**
4. $\quad a \leftarrow \mathsf{BarrettRed}(\mathsf{LIM}(a,a),\ n)$
5. $\quad$ **if** $d_i = 1$ **then**
6. $\quad\quad a \leftarrow \mathsf{BarrettRed}(\mathsf{LIM}(a,m),\ n)$
7. **Return**(a)

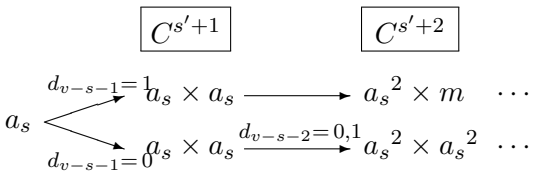
Alg. **??** presents the classical *square and multiply* modular exponentiation algorithm using Barrett reduction as previously discussed in this manuscript. We assume in the following of this paper that Alg. **??** is implemented in an SPA resistant way, for instance using the *atomicity* principle [**?**].

While we have chosen to consider modular multiplication using Barrett reduction, and square and multiply exponentiation, the results we present in this paper also apply to the other modular multiplication methods, long integer multiplication techniques and exponentiation algorithms mentioned above.

4.2.1 Recovering the Secret Exponent with One Known Message Encryption

As in vertical DPA and CPA on modular exponentiation, the horizontal correlation analysis reveals the bits of the private exponent d one after another. Each exponent bit is recovered by determining whether the processing of this bit involves a multiplication by m or not (cf. Alg. **??**). The difference with classical vertical analysis lies in the way to build such hypothesis test. Computing the long integer multiplication $x \times y$ using Alg. **??** requires l^2 t-bit multiplier calls. The multiplication side-channel curve thus yields l^2 curve segments $C^k_{i,j}$ available to an attacker.

Assuming that the first s bits $d_{v-1}d_{v-2}\dots d_{v-s}$ of the exponent are already known, an attacker is able to compute the value a_s of the accumulator in Alg. **??** after processing the s-th bit. The processing of the first s bits corresponds to the first s' long integer multiplications with $s' = s + \text{H}(d_{v-1}d_{v-2}\dots d_{v-s})$ known from the attacker. The value of the unknown $(s+1)$-th exponent bit is then equal to 1 if and only if the $(s'+2)$-th long integer multiplication is ${a_s}^2 \times m$.



At this point there are several ways of determining whether the multiplication by m is performed or not.

First, one may show that the series of consumptions in the set of l^2 curve segments is consistent with the series of operand values m_j presumably involved in each of these segments. To this purpose the attacker simply computes the correlation factor between the series of Hamming weights $\mathrm{H}(m_j)$ and the series of curve segments $C_{i,j}^{s'+2}$ – i.e. taking $D = m_j$ and $R = 0$ in the correlation factor formula. In other words we use the curve segments as they would be in a vertical analysis if they were independent aligned curves. A correlation peak reveals that $d_{v-s-1} = 1$ since it occurs if and only if m is actually handled in this long multiplication.

Alternatively one may correlate the curves segments with the intermediate results of each t-bit multiplication $x_i \times y_j$, cf. Alg. **??**, with $x = a_s$ and $y = m$, or in other words take $D = a_i \times m_j$. This method may also be appropriate since the words of the result are written in registers at the end of the operation. Moreover in that case l^2 different values are available for correlating the curve segments instead of l previously. This diversity of data may be necessary for the success of the attack when l is small. Note that other intermediate values may also lead to better results depending on the hardware leakages.

Another method consists of using the curve segments $C_{i,j}^{s'+3}$ of the next long integer multiplication and correlating them with the Hamming weight of the words of the result ${a_s}^2 \times m$. If the $(s'+2)$-th operation is a multiplication by m then the $(s'+3)$-th operation is a squaring ${a_{s+1}}^2$, manipulating the words of the integer ${a_s}^2 \times m$ in the t-bit multiplier. As pointed out by Walter in [**?**] for the Big Mac attack, the longer the integer manipulated and the smaller the size t of the multiplier, the larger the number l^2 of curve segments. Thus longer keys are more at risk with respect to horizontal analysis. For instance in an RSA 2048 bit encryption, if the long integer multiplication is implemented using a 32-bit multiplier we obtain $(2048/32)^2 = 4096$ segments $C_{i,j}^k$ per curve C^k. *Remark* The series of Hamming weights $\mathrm{H}(m_j)$ is not only correlated with the series of curve segments in $C^{s'+2}$ (provided that $d_{v-s-1} = 1$), but also with the series of curve segments in each and any C^k corresponding to a multiplication by m. Defining a *wide segment* $C_{i,j}^*$ as the set of segments $C_{i,j}^k$ for all k on the curve C and correlating the series of $\mathrm{H}(m_j)$ with the series of wide segments $C_{i,j}^*$ (instead of the series of segments $C_{i,j}^{s'+2}$) will produce a wide segment correlation curve with a peak occurring for each k corresponding to a multiplication by the message. It is thus possible to determine in one shot the exact sequence of squarings and multiplications by m, revealing the whole private exponent with only one curve and only one correlation computation.

4.2.2 Practical Results

This section presents the successful experiments we conducted to demonstrate the efficiency of the horizontal correlation analysis technique. We used a 16-bit RISC microprocessor on which we implemented a software 16×16 bits long integer multiplication to simulate the behavior of a coprocessor. We aim at correlating a single long integer multiplication with one or both operands manipulated – i.e. y_j or $x_i \times y_j$.

The measurement bench is composed of a Lecroy Wavepro oscilloscope, and homemade softwares and electronic cards were used to acquire the power curves and process the attacks.

Firstly we performed a classical vertical correlation analysis to characterize our implementation and measurement bench, and to validate the correlation model; then we processed with the horizontal correlation analysis previously described.

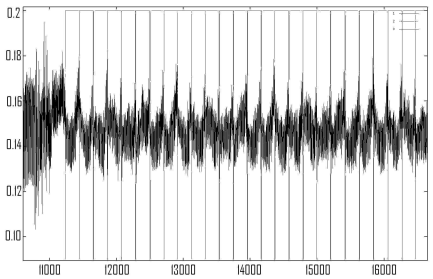


Figure 4.3: Beginning of a long integer multiplication power curve, lines delimitate each $C_{i,j}^k$

Vertical Correlation Analysis

This analysis succeeded in two cases during the operation $x \times y$. We obtained correlation peaks by correlating power curves with values x_i and y_j and also by correlating the power curves with the result value of operation $x_i \times y_j$. Fig. **??** and Fig. **??** show the correlation traces we obtained for both cases with 500 power curves.

This suggests that one can perform horizontal correlation as explained previously either using y_i values or using result values $x_i \times y_j$ for correlating with segment curves

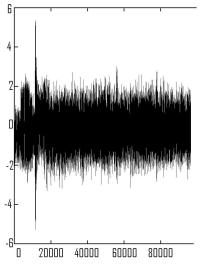


Figure 4.4: Vertical CPA on value y_j.

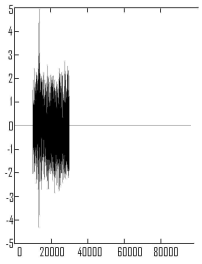


Figure 4.5: Vertical CPA on value $x_i \times y_j$.

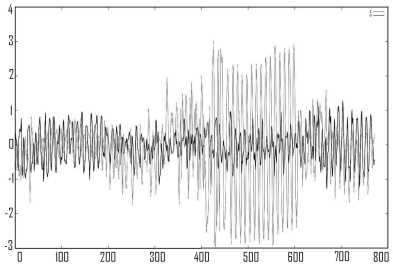


Figure 4.6: Horizontal CPA on value $a_i \times m_j$.

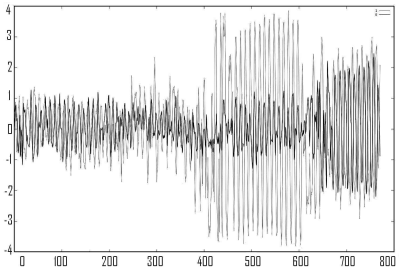


Figure 4.7: Horizontal CPA on value m_j.

of the long integer multiplication.

Horizontal Correlation Analysis

We have chosen to test our technique within a 512-bit multiplication LIM(x, y). This allows us to obtain 1024 curve segments $C^k_{i,j}$ of 16-bit multiplications to mount the analysis, which should be enough for the success of our attack regarding the vertical analysis results. From the single power curve we acquired, we processed the signal in order to detect each set of cycles corresponding to each t-bit multiplication $x_i \times y_j$ and divide the single power curve in 1024 segments $C^k_{i,j}$ as depicted in Fig. **??**.

We performed horizontal correlation analysis as explained in Section **??** for the two cases $D = a_i \times m_j$ and $D = m_j$ and recovered the operation executed as shown in Fig. **??** and Fig. **??**. In each figure, the grey trace shows a greater correlation than the black one and thus corresponds to the correct guess on the operation.

Since our attack actually enabled us to distinguish one operation from another, it is then possible to identify a squaring $a \times a$ from a multiplication $a \times m$ in the Step 3 of Alg. **??**. The secret exponent d used in an exponentiation can thus be recovered by using a single power trace, even when the exponentiation is protected by an atomic implementation.

We have presented here a technique to recover the secret exponent using a single curve when the input message is known and have proven this attack to be practically successful. Although the attack is tested on a software implementation, results obtained by Amiel et al. [**?**, Fig. 8] prove that correlation techniques are efficient on hardware coprocessors (with multiplier size larger than 16 bits), and enable to locate each little multiplication involved in a long integer multiplication. We thus consider that our attack can also threaten hardware coprocessors.

4.2.3 Comparing our Technique with the Big Mac Attack

We now compare our proposed horizontal CPA on exponentiation with the Big Mac attack which is the most powerful known horizontal analysis to recover a private exponent. A common property is that both techniques counteract the randomization of the exponent.

A first difference between both methods is that the Big Mac templates are generated by averaging the leakage dependency from a not targeted argument. It is thus implicitly

accepted to lose the information brought by this auxiliary data. On the other hand, horizontal correlation exploits the knowledge of both multiplication operands a and m (under assumption on the exponent bit) to correlate it with all l^2 segments $C_{i,j}^k$. This full exploitation of the available information included in the l^2 curve segments tends us to expect a better efficiency of the correlation method particularly when processing noisy observations.

But the main difference is not there. What fundamentally separates the Big Mac and correlation methods is that the former deals with templates – which the attacker tries to identify – while the later rather consider intermediate results – whose manipulation validates a secret-dependent guess. With the Big Mac technique an attacker is able to answer the question *Is this operation of that particular kind?* (squaring, multiplication by m or a power thereof) while the correlation with intermediate data not only brings the same information but also answers the more important question *Is the result of that operation involved in the sequel of the computation?* The main consequence is that horizontal CPA is effective even when the exponentiation implementation is *regular* with respect to the operation performed. This is notably the case of the *square and multiply always*[1] and the Montgomery ladder exponentiations which are not threaten by the Big Mac attack. In this respect we can say that our horizontal CPA combines both the advantage of classical CPA which is able to validate guesses based on the manipulation of intermediate results (but which is defeated by the randomization of the exponent) and that of horizontal techniques which are immune to exponent blinding.

On the other hand the limitation of the Big Mac attack – its ignorance of the intermediate results – is precisely the cause of its noticeable property to be applicable also when the base of the exponentiation is not known from the attacker. The Big Mac attack thus applies when the message is randomized and/or in the case of a Chinese Remainder Theorem (CRT) implementation of RSA. While the horizontal correlation technique does not intrinsically deals with message randomization, we give in the next section some hints that allow breaking those protected implementations when the random bit-length is not sufficiently large.

4.2.4 Horizontal Analysis on Blinded Exponentiation

To protect public key implementations from SCA developers usually include blinding countermeasures in their cryptographic codes. The most popular ones on RSA exponentiation are:

- Additive randomization of the message and the modulus: $m^\star = m + r_1 \cdot n \bmod r_2 \cdot n = m + u \cdot n$ with r_1, r_2 being λ-bit random values different each time the computation is executed, and $u = r_1 \bmod r_2$.
- Multiplicative randomization of the message: $m^\star = r^e \cdot m \bmod n$ with r a random value and e the public exponent,
- Additive randomization of the exponent: $d^\star = d + r \cdot \phi(n)$ with r a random value.

[1] Referring to the description given in **??** the method using the curve segments $C_{i,j}^{s'+3}$ validates that the value produced by the multiplication by m is involved or not in the next squaring operation. A similar technique also applies to the Montgomery ladder.

All these countermeasures prevent from the classical vertical side-channel analysis but the efficiency of the implementations is penalized as the exponent and modulus are extended of the random used bit lengths.

It is obvious here the additive randomization of the exponent is defeated by the horizontal analysis. This technique allows to recover the complete value of the secret blinded exponent in a unique side-channel trace.

Guessing the randomized message $m^\star$

In this paragraph we consider that the message has been randomized by an additive (or multiplicative) method, the secret exponent has also been randomized and the message is encrypted by an atomic multiply always exponentiation. We analyze the security of such implementation against horizontal CPA. The major difference with vertical side-channel analysis is that the exponent blinding has no effect since we analyze a single curve and recovering $d^\star$ is equivalent to recovering d.

Assuming that the entropy of u is λ bits, there are 2^λ possible values for the message $m^\star$ knowing m and n. The first step of an attack is to deduce the value of the random u. This is achieved by performing one horizontal CPA for each possible value of u on the very first multiplication which computes $(m^\star)^2$. Since this multiplication is necessarily computed, the value of u should be retrieved as the one showing a correlation peak. Once u is recovered, the randomized message $m^\star$ is known and recovering the bits of the exponent d is similar to the non blinded case using $m^\star$ instead of m. Consequently, the entropy of u must be large enough (e.g. $\lambda \geqslant 32$) to make the number of guess unaffordable and prevent from horizontal correlation analysis.

In case the message were randomized by the multiplicative masking technique the same analysis can be conducted when considering that λ is the bit length of the integer r.

The actual entropy of the randomization

In the case of additive randomization of the message, $m^\star$ depends on two λ-bit random values r_1 and r_2. Obviously, the actual entropy of this randomization is not 2λ bits, and interestingly it is even strictly less than λ bits. The reason is that $m^\star = m + u \cdot n$ with $u = r_1 \bmod r_2$, and thus smaller u values are more probable than larger ones.

Assuming that r_1 and r_2 are uniformly drawn at random in the ranges $\left[0, \ldots, 2^\lambda - 1\right]$ and $\left[1, \ldots, 2^\lambda - 1\right]$ respectively, statistical experiments show that the actual entropy of u is about $\lambda - 0.75$ bits[2].

A consequence of this bias on the random u is that an attacker can exhaust only a subset of the smaller guesses about u. If the attack does not succeed, then he can try again on another exponentiation curve. For $\lambda = 8$ guessing only the 41 smaller u will succeed with probability $\frac{1}{2}$.

An extreme case, which optimizes the average number of correlation curve computations, is to guess only the value $u = 0$ [3]. This way, only 38 and 5352 correlation curve

[2] The loss of 0.75 bits of entropy is nearly independent of λ for typical values ($\lambda \leqslant 64$).

[3] Or $u = 1$ if the implementation does not allow $u = 0$.

computations are needed in the mean when λ is equal to 8 and 16 respectively.

These observations demonstrate that the guessing attack described in the previous paragraph is more efficient than may be trivially expected. This confirms the need to use a large random bit length λ.

4.2.5 Countermeasures

Having detailed the principle and the threats of our horizontal side-channel analysis on exponentiation, We now study the real efficiency of the classical side channel countermeasures and propose new countermeasures.

Hardware Countermeasures

Classical countermeasures consisting in perturbing the signal analysis e.g. clock jitters, frequency clock dividers or dummy cycles, may considerably complicate the analysis but should not be the only countermeasures since efficient signal processing could bypass them depending on their real efficiency.

Techniques consisting in balancing the power consumption of the chip with dual rail, precharge logics or other methods, if really efficient, could be a better solution. However they are expensive countermeasures from the chip surface point-of-view.

Blinding

All SSCA resistant algorithms that can be used to implement the exponentiation – either those protected with atomicity principle or regular ones as *square and multiply always* and Montgomery ladder – are threatened by the horizontal analysis. It is then necessary to randomize the data manipulated during the computation. As said previously the blinding of the exponent is not an efficient countermeasure here, it is thus highly recommended to implement a resistant and efficient blinding method on the data manipulated, for instance by using additive or multiplicative message randomization with random values larger or equal to 32 bits. As regard to the previous analysis on the actual entropy of u, an additional solution consists in eliminating the bias on u by setting r_2 to a constant value, for instance $2^\lambda - 1$.

New Countermeasures

We suggest protecting sensitive implementations from this analysis by introducing blinding into the t-bit multiplications, by randomizing their execution order or by mixing both solutions. These countermeasures are presented on modular multiplication using the Barrett reduction.

Blind Operands in LIM A full blinding countermeasure on the words x_i and y_j consists in replacing in Alg. **??** the operation $(w_{i+j} + x_i \times y_j) + c$ by $(w_{i+j} + (x_i - r_1) \times (y_j - r_2)) + r_1 \times y_j + r_2 \times x_i - r_1 \times r_2 + c$ with r_1 and r_2 two t-bit random values. For efficiency purposes, the values $r_1 \times x_i$, $r_2 \times y_j$, $r_1 \times r_2$ should be computed once and stored. Moreover, these precomputations must also be protected from correlation analysis. For

example, performing them in a random order yields $(2l+1)!$ different possibilities. In this case the LIM operation requires l^2+2l+1 t-bit multiplications and necessitates $2(n+2t)$ bits of additional storage.

In the following we improve this countermeasure by mixing the data blinding with a randomization of the order of the internal loops of the long integer multiplication.

Randomize One Loop in LIM and Blind This countermeasure consists in randomizing the way the words x_i are taken by the long integer multiplication algorithm. In other words it randomizes the order of the lines of the schoolbook multiplication. Then computing correlation between x_i and $C^k_{i,j}$ does not yield the expected result anymore. On the other hand it remains necessary to blind the words of y. An example of implementation is given in Alg. **??**.

The random permutation provides $l!$ different possibilities for the execution order of the first loop. For example, using a 32-bit multiplier, a 1024-bit long integer multiplication has about 2^{117} possible execution orders of the first loop and with 2048-bit operands it comes to about 2^{296} possibilities.

Algorithm 4.2.1 LIM with lines randomization and blinding

INPUT: $x=(x_{l-1}x_{l-2}\dots x_1x_0)_b, y=(y_{l-1}y_{l-2}\cdots y_1y_0)_b$
OUTPUT: LinesRandLIM(x,y) $= x\times y$

Step 1. Draw a random permutation vector $\alpha=(\alpha_{l-1}\dots\alpha_0)$ in $[0,l-1]$
Step 2. Draw a random value r in $[1,2^t-1]$
Step 3. **for** i from 0 to $2l-1$ **do** $w_i=0$
Step 4. **for** h from 0 to $l-1$ **do**
    $i\leftarrow\alpha_h$, $r_i\leftarrow r\times x_i$ and $c\leftarrow 0$
    for j from 0 to $l-1$ **do**
        $(uv)_b\leftarrow(w_{i+j}+x_i\times(y_j-r)+c)+r_i$
        $w_{i+j}\leftarrow v$ and $c\leftarrow u$
    while $c\neq 0$ **do**
        $uv\leftarrow w_{i+j}+c$
        $w_{i+j}\leftarrow v$, $c\leftarrow u$ and $j\leftarrow j+1$
Step 5. **Return**(w)

Compared to the previous countermeasure, Alg. **??** requires only l^2+l t-bit multiplications and $2t$ bits of additional storage.

Remark One may argue that in the case of very small l values such a countermeasure might not be efficient. Remember here that if l is very small, the horizontal correlation analysis is not efficient either because of the small number of curve segments.

Randomize the Two Loops in LIM We propose a variant of the previous countermeasure in which the execution order of the both internal loops of the long integer multiplication are randomized. This means randomizing both lines and columns of the schoolbook multiplication. The main advantage is that none of the operands x_i or y_j needs to be blinded anymore. The number of possibilities for the order of the l^2 internal multiplication is increased to $(l!)^2$. An example of implementation is given in Alg. **??**.

Unlike the two previous countermeasures, Alg. **??** requires no extra t-bit multiplication compared to LIM. It is then an efficient and interesting countermeasure, while the remaining difficulty for designers consists in implementing it in hardware.

Algorithm 4.2.2 LIM with lines and columns randomization

INPUT: $x = (x_{l-1}x_{l-2}\dots x_1x_0)_b, y = (y_{l-1}y_{l-2}\cdots y_1y_0)_b$
OUTPUT: MatrixRandLIM(x,y) $= x \times y$

Step 1. Draw two random permutation vectors α, β in $[0, l-1]$
Step 2. **for** i from 0 to $2l-1$ **do** $w_i = 0$
Step 3. **for** h from 0 to $l-1$ **do**
  $i \leftarrow \alpha_h$
  for j from 0 to $2l-1$ **do** $c_j = 0$
  for k from 0 to $l-1$ **do**
    $j \leftarrow \beta_k$
    $(uv)_b \leftarrow w_{i+j} + x_i \times y_j$
    $w_{i+j} \leftarrow v$ and $c_{i+j+1} \leftarrow u$
  $u \leftarrow 0$
  for s from $i+1$ to $2l-1$ **do**
    $(uv)_b \leftarrow w_s + c_s + u$
    $w_s \leftarrow v$
Step 4. **Return**(w)

4.2.6 Concerns for Common Cryptosystems

We presented our analysis on straightforward implementations of the RSA signature and decryption algorithms which essentially consist of an exponentiation with the secret exponent. In the case of an RSA exponentiation using the CRT method our technique cannot be applied since the operations are performed modulo p and q which are unknown to the attacker. On the other hand DSA and Diffie-Hellman exponentiations were until now considered immune to DPA and CPA because the exponents are chosen at random for each execution. Indeed it naturally protects these cryptosystems from vertical analysis. However, as horizontal CPA requires a single execution power trace to recover the secret exponent, DSA and Diffie-Hellman exponentiations are prone to this attack and other countermeasures must be used in embedded implementations. It is worth noticing that ECC cryptosystems are theoretically also concerned by the horizontal side-channel analysis. However since key lengths are considerably shorter – for instance ECC 224 bits is considered having equivalent mathematical resistance than RSA 2048 – very few curves per scalar multiplication will be available for the attack. On the other hand, scalar multiplication involves point doublings and point additions instead of field multiplications and squaring operations. Each point operation requires about 10 modular multiplications and thus correlation computation could take advantage of all the corresponding curves. Nevertheless, a factor of about 10 should not balance the key length reduction which has a quadratic influence on the number of available curve segments.

4.2.7 Conclusion

We presented in this chapter a way to apply classical power analysis techniques such as CPAon a single curve to recover the secret key in some public key implementations – e.g. non CRT RSA, DSA or Diffie-Hellman – protected or not by exponent randomization. We also applied our technique in practice and presented some successful results obtained on a 16-bit RISC microprocessor. However even with bigger multiplier sizes (32 or 64 bits) this attack can be envisaged depending on the key size, cf. Section **??**. We discussed the resistance of some countermeasures to our analysis and introduced three secure multiplication algorithms.

Our contribution enforces the necessity of using sufficiently large random numbers for blinding in secure implementations and highlights the fact that increasing the key lengths in the next years could improve the efficiency of some side-channel attacks. The attack we presented threatens implementations which may have been considered secure up to now. This new potential risk should then be taken into account when developing embedded products.

Further work could target the use of other values and distinguishers for the horizontal correlation analysis and then improve its efficiency. Possible ideas include: using more intermediate values, some likelihood tests, guessing simultaneously many bits of the secret exponent to increase the number of available curves for the analysis, using different models like the bivariate one for correlation factor computation on curves. Non CRT RSA cannot be threatened by the horizontal CPA (but still by the Big Mac Attack) as the moduli values used in both exponentiation are unknown to the attacker. Another further improvement would target to apply horizontal CPA to CRT RSA.

4.3 ROSETTA for Single Trace Analysis

4.3.1 RSA Implementation

The standard way of computing an RSA signature S of a message m consists of a modular exponentiation with the private exponent: $S = m^d \bmod N$. The corresponding signature is verified by comparing the message m with the signature S raised to the power of the public exponent: $m \stackrel{?}{=} S^e \bmod N$.

In order to improve its efficiency, the signature is often computed using the Chinese Remainder Theorem (CRT). Let us denote by d_p (resp. d_q) the residue $d \bmod p-1$ (resp. $d \bmod q-1$). To compute the signature, the message is raised to the power of d_p modulo p then to the power of d_q modulo q. The corresponding results S_p and S_q are then combined using Garner's formula [**?**] to obtain the signature: $S = S_q + q(q^{-1}(S_p - S_q) \bmod p)$.

If used exactly as described above, RSA is subject to multiple attacks from a theoretical point of view. Indeed, it is possible under some assumptions to recover some information on the plaintext from the ciphertext or to forge fake signatures. To ensure its security, RSA must be used according to a protocol which mainly consists in formatting the message. Examples of such protocols are the encryption protocol OAEP and the signature protocol PSS, both of them being proven secure and included in the

standard PKCS #1 V2.1 [**?**]. Note that, as they do not require the knowledge of the exponentiated value, the new attacks described in this paper also apply when either OAEP or PSS scheme is used.

From a practical point of view, the RSA exponentiation is also subject to many attacks if straightforwardly implemented. For instance, SSCA, DSCA or collision analysis can be used to recover the RSA private key. SSCA aims at distinguishing a difference of behavior when an exponent bit is a 0 or a 1.

DSCA allows a deeper analysis than SSCA by exploiting the dependency which exists between side-channel measurements and manipulated data values [**?**]. To this end, thousands of measurements are generally combined using a statistical distinguisher to recover the secret exponent value. Nowadays, the most widespread distinguisher is the Pearson linear correlation coefficient [**?**].

Finally, collision analysis aims at identifying when a value is manipulated twice during the execution of an algorithm.

Algorithm **??** presents the classical atomic exponentiation which is one of the fastest exponentiation algorithms protected against the SPA.

Alg. 4.3.1 Atomic Multiply-Always Exponentiation

Input: $x, n \in \mathbb{N}$, $d = (d_{v-1}d_{v-2}\dots d_0)_2$
Output: $x^d \bmod n$

1. $R_0 \leftarrow 1$
2. $R_1 \leftarrow x$
3. $i \leftarrow v - 1$
4. $k \leftarrow 0$
5. **while** $i \geq 0$ **do**
6. $\quad R_0 \leftarrow R_0 \times R_k \bmod n$
7. $\quad k \leftarrow k \oplus d_i$ [$\oplus$ stands for bitwise X-or]
8. $\quad i \leftarrow i - \neg k$ [$\neg$ stands for bitwise negation]
9.
10. **return** R_0

When correctly implemented, Alg. **??** defeats SSCA since squarings cannot be distinguished from other multiplications on a side-channel trace, as depicted by Fig. **??**.

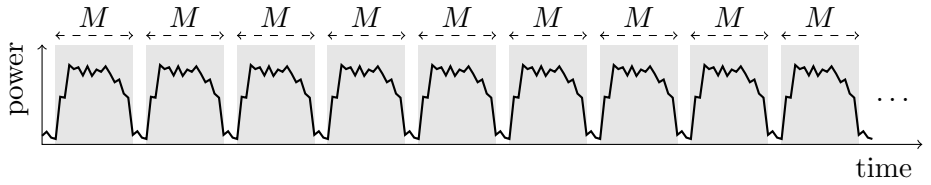


Figure 4.8: Atomic multiply-always side-channel leakage

To prevent the implementation of RSA exponentiation from DSCA, the two main countermeasures published so far are based on message and exponent blinding [**?**, **?**]. Instead of computing straightforwardly $S = m^d \bmod n$, one rather computes $\tilde{S} = (m +$

$k_0 \cdot n)^{d+k_1 \cdot \varphi(n)} \bmod 2^\lambda \cdot n$ where φ denotes the Euler's totient and k_0 and k_1 are two λ-bit random values, then finally reduce $\tilde{S}$ modulo N to obtain S. Using such a blinding scheme with a large enough λ (32 bits are generally considered as a good compromise between security and cost overhead), the relationship between the side-channel leakages occurring during an exponentiation and the original message and exponent values is hidden to an adversary, therefore circumventing DSCA.

As the modular exponentiation consists of a series of modular multiplications, it relies on the efficiency of the modular multiplication. Many methods have been published so far to improve the efficiency of this crucial operation. Amongst these methods, the most popular are the Montgomery, Knuth, Barrett, Sedlack or Quisquater modular multiplications [**?**, **?**]. Most of them have in common that the long-integer multiplication is internally computed by repeatedly calling a smaller multiplier operating on t-bit words. A classic example is given in Alg. **??** which performs the schoolbook long-integer multiplication using a t-bit internal multiplier giving a $2t$-bit result. The decomposition of an integer x in t-bit words is given by $x = (x_{\ell-1}x_{\ell-2}\dots x_0)_b$ with $b = 2^t$ and $\ell = \lfloor \log_b(x) \rfloor + 1$.

Alg. 4.3.2 Schoolbook Long-Integer Multiplication

Input: $x = (x_{\ell-1}x_{\ell-2}\dots x_0)_b, y = (y_{\ell-1}y_{\ell-2}\dots y_0)_b$
Output: $x \times y$

1. **for** $i = 0$ **to** $2\ell - 1$ **do**
2. $\quad z_i \leftarrow 0$
3. **for** $i = 0$ **to** $\ell - 1$ **do**
4. $\quad R_0 \leftarrow 0$
5. $\quad R_1 \leftarrow x_i$
6. $\quad$ **for** $j = 0$ **to** $\ell - 1$ **do**
7. $\quad\quad R_2 \leftarrow y_j$
8. $\quad\quad R_3 \leftarrow z_{i+j}$
9. $\quad\quad (R_5R_4)_b \leftarrow R_3 + R_2 \times R_1 + R_0$
10. $\quad\quad z_{i+j} \leftarrow R_4$
11. $\quad\quad R_0 \leftarrow R_5$
12. $\quad z_{i+\ell} \leftarrow R_5$
13.
14. **return** z

In the rest of this section we recall some previously published attacks on atomic exponentiations which inspired our new technique detailed in Section **??**.

4.3.2 Attacks Background

Distinguishing Squarings from Multiplications in Atomic Exponentiation

It has already been observed in [**?**, **?**] that squaring and multiplication output results had different distributions and it lead to attack paths. For instance the probability for a subtraction by the modulus to happen in a Montgomery modular squaring is different than the one for a Montgomery modular multiplication. This property was exploited

by Walter et al. [**?**] to recover the secret exponent involved in a Montgomery modular exponentiation.

In [**?**] Amiel et al. present a specific DSCA aimed at distinguishing squaring from other multiplications in the atomic exponentiation. They observe that the average Hamming weight of the output of a multiplication $x \times y$ has a different distribution whether:

- the operation is a squaring performed using the multiplication routine, i.e. $x = y$, with x uniformly distributed in $\left[0, 2^{\ell t} - 1\right]$;
- or the operation is an actual multiplication, with x and y independent and uniformly distributed in $\left[0, 2^{\ell t} - 1\right]$.

Thus, considering a device with a single long-integer multiplication routine used to perform either $x \times x$ or $x \times y$, a set of N side-channel traces computing multiplications with random operands can be distinguished from a set of N traces computing squarings, provided that N is sufficiently large to make the two distribution averages separable. This attack can thus target an atomic exponentiation such as Alg. **??** even in the case of message and modulus blinding. Regarding the exponent blinding, authors suggest that their attack should be extended to success on a single trace but do not give evidence of its feasibility. We thus study this point in the following of the paper.

Horizontal Correlation Analysis

Correlation analysis on a single atomic exponentiation side-channel trace has been published in [**?**] where the message is known to the attacker but the exponent is blinded. This attack called *horizontal* correlation analysis requires only one exponentiation trace to recover the full RSA private exponent.

Instead of considering the whole k-th long-integer multiplication side-channel trace T^k as a block, the authors consider each inner side-channel trace segment corresponding to a single-precision multiplication on t-bit words. For instance, if the long-integer multiplication is performed using Alg. **??** on a device provided with a t-bit multiplier, then the trace T^k of the k-th long-integer multiplication $x \times y$ can be split into ℓ^2 trace segments $T^k_{i,j}$, $0 \leqslant i, j < \ell$, each of them representing a single-precision multiplication $x_i \times y_j$. More precisely, for each word y_j of the multiplicand y, the attacker obtains ℓ trace segments $T^k_{i,j}$, $0 \leqslant i, j < \ell$, corresponding to a multiplication by y_j. The slicing of T^k into trace segments $T^k_{i,j}$ is illustrated on Fig. **??**.

In the horizontal correlation analysis the attacker is able to identify whether the k-th long-integer operation T^k is a squaring or a multiplication by computing the correlation factor between the series of Hamming weights of each t-bit word m_j of the message m and the series of corresponding sets of ℓ trace segments $T^k_{i,j}$, $0 \leqslant i, j < \ell$. This correlation factor is expected to be much smaller when the long-integer operation is a squaring (i.e. $R_0 \leftarrow R_0 \times R_0$ in Alg. **??**) than when it is a multiplication by m (i.e. $R_0 \leftarrow R_0 \times R_1$). The correlation factor can be computed by using the Pearson correlation coefficient $\rho(H, T^k)$ where $H = (H_0, \ldots, H_{\ell-1})$, with $H_j = (\mathrm{H}(m_j), \ldots, \mathrm{H}(m_j))$, $\mathrm{H}(m_j)$ standing for the Hamming weight of m_j and $T^k = (T^k_0, \ldots, T^k_{\ell-1})$ with $T^k_j = (T^k_{0,j}, \ldots, T^k_{\ell-1,j})$.

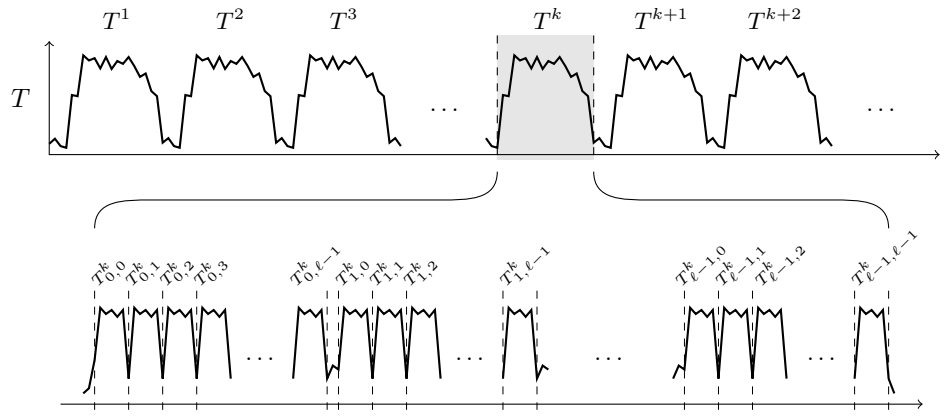


Figure 4.9: Horizontal side-channel analysis on exponentiation

Big Mac Attack

Walter's attack needs, as our technique, a single exponentiation side-channel trace to recover the secret exponent. For each long-integer multiplication, the Big Mac attack detects if the operation performed is either $R_0 \times R_0$ or $R_0 \times m$. The multiplications $x_i \times y_j$ — and corresponding trace segments $T^k_{i,j}$ — can be easily identified on the side-channel trace from their specific pattern which is repeated ℓ^2 times in the long-integer multiplication loop. A template side-channel trace is computed (either from the precomputations or from the first squaring operation) to characterize the manipulation of the message during the long-integer multiplication. The Euclidean distance between the template trace and each long-integer multiplication trace T^k is then computed. If it exceeds a threshold then the attack concludes that the operation is a squaring, or a multiplication by m otherwise.

Walter uses the Euclidean distance but we noticed that other distinguisher could be used. In the following section, we extend the Big Mac attack using a collision-correlation technique.

4.3.3 Big Mac Extension using Collision Correlation

A specific approach for SCA uses information leakages to detect collisions between data manipulated in algorithms. A side-channel collision attacks against a block cipher was first proposed by Schramm et al. in 2003 [**?**]. More recently Moradi et al. [**?**] proposed to use a correlation distinguisher to detect collisions in AES. The main advantage of this approach is that it is not necessary to define a leakage model as points of traces are directly correlated with other points of traces. Later, Clavier et al. [**?**] presented two collision-correlation techniques defeating different first order protected AES implementations. The same year, Witteman et al. [**?**] applied collision correlation to public key implementation. They describe an efficient attack on RSA using square-and-multiply-always exponentiation and message blinding. All these techniques require many side-channel execution traces. In this section, we extend Walter's Big Mac attack using the collision correlation as distinguisher instead of the Euclidean distance.

We consider a blinded exponentiation and use the fact that the second and third modular operations in an atomic exponentiation are respectively $1 * \tilde{m}$ and $\tilde{m} * \tilde{m}$, where $\tilde{m}$ is the blinded message. The trace of the second long-integer multiplication yields ℓ multiplication segments for each word $\tilde{m}_j$ of the blinded message. Considering the k-th long-integer multiplication, $k > 3$, we compute the correlation factor between the series of ℓ trace segments T_j^2 — each one being composed of the ℓ trace segments $T_{i,j}^2$ involved in the multiplication by $\tilde{m}_j$ — and the series of ℓ trace segments T_j^k. Since the blinded value of the message does not change during the exponentiation, a high correlation occurs if the k-th long-integer operation is a multiplication, and a low correlation otherwise. Once the sequence of squarings and multiplications is found, the blinded exponent value is straightforwardly recovered. Notice that recovering the blinded value of the secret exponent is not an issue as it can be used to forge signature as well as its non-blinded value.

This attack also works if we use the trace segments T_j^3 of the third long-integer operation instead of the trace segments T_j^2. One can also combine the information provided by the second and third long-integer operations to improve the attack.

Remark

As the original Big Mac, this attack also applies to the CRT RSA exponentiation since no information is required on either the message or the modulus. This is of the utmost importance since, to the best of our knowledge, this is the first practical attack on a CRT RSA fully blinded (message, modulus and exponent) atomic exponentiation.

4.4 ROSETTA: Recovery Of Secret Exponent by Triangular Trace Analysis

4.4.1 Attack Principle

The long-integer multiplication $\mathrm{LIM}(x, y)$ in base $b = 2^t$ is given by the classical schoolbook formula:

$$x \times y = \sum_{i=0}^{\ell-1} \sum_{j=0}^{\ell-1} x_i y_j b^{i+j}$$

and illustrated, with for instance $\ell = 4$ by the following matrix M:

$$M = \begin{pmatrix} x_0y_0 & x_0y_1 & x_0y_2 & x_0y_3 \\ x_1y_0 & x_1y_1 & x_1y_2 & x_1y_3 \\ x_2y_0 & x_2y_1 & x_2y_2 & x_2y_3 \\ x_3y_0 & x_3y_1 & x_3y_2 & x_3y_3 \end{pmatrix}$$

In the case of a squaring, then $x = y$ and the inner multiplications become:

$$S = \begin{pmatrix} x_0x_0 & x_0x_1 & x_0x_2 & x_0x_3 \\ x_1x_0 & x_1x_1 & x_1x_2 & x_1x_3 \\ x_2x_0 & x_2x_1 & x_2x_2 & x_2x_3 \\ x_3x_0 & x_3x_1 & x_3x_2 & x_3x_3 \end{pmatrix}$$

We consider four observations to design our new attacks, assuming a large enough multiplier size $t \geq 16$:

(Ω_0) LIM(x, y) s.t. $x = y \Rightarrow$ Prob$(x_i \times y_i$ are squaring operations$) = 1 \quad \forall i$

(Ω_1) LIM(x, y) s.t. $x \neq y \Rightarrow$ Prob$(x_i \times y_i$ are squaring operations$) \approx 0 \quad \forall i$

(Ω_2) LIM(x, y) s.t. $x = y \Rightarrow$ Prob$(x_i \times y_j = x_j \times y_i) = 1 \quad \forall i \neq j$.

(Ω_3) LIM(x, y) s.t. $x \neq y \Rightarrow$ Prob$(x_i \times y_j = x_j \times y_i) \approx 0 \quad \forall i \neq j$.

From observations (Ω_0) and (Ω_1) one can apply the attack presented in [**?**] on a single trace as suggested by the authors. The main drawback is that only ℓ such operations are performed during a LIM which represents a small number of trace segments. It is likely to make the attack inefficient for small modulus lengths (with respect to the multiplier size t).

From observations (Ω_2) and (Ω_3) we notice that collisions between $x_i \times y_j$ and $x_j \times y_i$ for $i \neq j$ can be used to identify squarings from other multiplications. Moreover, LIM(x, y) provides $\ell^2 - \ell$ operations $x_i \times y_j$, $i \neq j$, thus $(\ell^2 - \ell)/2$ couples of potential collisions. This represents a fairly large number of trace segments. The principle of our new attack consists in detecting those internal collisions in a single long-integer operation to determine whether it is a squaring or not. Visually, we split the matrix M into an upper-right and a lower-left triangles of terms, thus we call this technique a *triangle trace analysis*.

We present in the following two techniques to identify these collisions on a single long-integer multiplication trace. The first analysis uses the Euclidean distance distinguisher and the second one relies on a collision-correlation technique.

4.4.2 Euclidean Distance Distinguisher

We use as distinguisher the Euclidean distance between two sets of points on a trace as Walter [**?**] in the Big Mac analysis. In order to exploit properties (Ω_2) and (Ω_3) we proceed as follows. For each LIM(x, y) operation we compute the following differential side-channel trace:

$$T_{\text{ED}} = \frac{2}{\ell^2 - \ell} \sum_{0 \leqslant i < j < \ell} \sqrt{(T_{i,j} - T_{j,i})^2}$$

If the operation performed is a squaring then the single-precision multiplications $x_i \times y_j$ and $x_j \times y_i$ store the same value in the result register (or in the memory) at the end of the operation. The side-channel leakage of the result storage of both operations should thus be similar. On the other hand, if $x \neq y$, products differ and the side-channel leakage should present less similarities. Assuming a side-channel leakage function linear in the Hamming weight of the data manipulated, a squaring should result in $E(T_{\text{ED}}) \approx 0$, whereas we should expect a significantly higher value (about $t/2$ for each of the product halves) in the case of a multiplication.

4.4.3 Collision-Correlation Distinguisher

We define the two following series of trace segments, where the ordering of couples (i, j) is the same for the two series:

$$\Theta_0 = \{T_{i,j} \text{ s.t. } 0 \leqslant i < j \leqslant \ell - 1\}$$
$$\Theta_1 = \{T_{j,i} \text{ s.t. } 0 \leqslant i < j \leqslant \ell - 1\}$$

Each set includes $N = (\ell^2 - \ell)/2$ trace segments of base b multiplications.

In order to determine the operation performed by the LIM we compute the Pearson correlation factor between the two series Θ_0 and Θ_1 as described in [**?**]:

$$\begin{aligned}\hat{\rho}_{\Theta_0,\Theta_1}(t) &= \frac{\mathrm{Cov}(\Theta_0(t), \Theta_1(t))}{\sigma_{\Theta_0(t)}\sigma_{\Theta_1(t)}} \\ &= \frac{N\sum(T_{i,j}(t)T_{j,i}(t)) - \sum T_{i,j}(t)\sum T_{j,i}(t)}{\sqrt{N\sum(T_{i,j}(t))^2 - (\sum T_{i,j}(t))^2}\sqrt{N\sum(T_{j,i}(t))^2 - (\sum T_{j,i}(t))^2}}\end{aligned}$$

where summations are taken over all couples $0 \leqslant i < j \leqslant \ell - 1$.

In case of a squaring operation, a much higher correlation value $\hat{\rho}_{\Theta_0,\Theta_1}$ is expected than in case of a multiplication. Computing this correlation value for each LIM operation allows to determine its nature and to recover the sequence of exponent bits.

Remark

Contrary to differential analysis on symmetric ciphers, each exponent bit requires to distinguish one hypothesis out of only two, instead of for instance 256 considering a differential attack on AES. Thus fixing a decision threshold is easier when dealing with the exponentiation. This has already been observed when applying DPA or CPA on RSA [**?**, **?**] compared to DES or AES.

4.4.4 Comparison of the Different Attacks

In order to validate these two techniques, we generated simulated side-channel traces for a classical 32×32-bit multiplier. As generally considered in the literature, we assume a side-channel leakage model linear in the Hamming weight of the manipulated data — here x_i, y_j, and $x_i \times y_j$ — and add a white Gaussian noise of mean $\mu = 0$ and standard deviation σ. We build simulated side-channel traces based on the Hamming weight of the data manipulated in the multiplication operation such that each processed single-precision multiplication generates four leakage points $\mathrm{H}(x_i)$, $\mathrm{H}(y_j)$, $\mathrm{H}(x_i \times y_j \bmod b)$, and $\mathrm{H}(x_i \times y_j \div b)$, where $\div$ stands for the Euclidean quotient.

Besides validating our two Rosetta variants — the Euclidean distance distinguisher (Rosetta ED) and the collision-correlation one (Rosetta CoCo) — we compare Rosetta with other techniques discussed previously, namely the classical Big Mac, the Big Mac using collision correlation (Big Mac CoCo), and the single trace variant of the Amiel et al. attack presented at SAC 2008.

We proceed in the following way: we randomly select two ℓ-bit integers x and y. Then we generate the side-channel traces of the multiplication $\mathrm{LIM}(x, y)$ and of the squaring $\mathrm{LIM}(x, x)$.

Each different attack is eventually applied and we keep trace of their success or failure to distinguish the squaring from the multiplication. Finally, we estimate the success rate of each technique by running 1 000 such experiments. These tests are performed for three different noise standard deviation values[4]: from no noise ($\sigma = 0$) to a strong one ($\sigma = 7$).

Characterisation and Threshold

A threshold for the attack must be selected for each technique to determine whether the targeted operation is a multiplication or a squaring. Using simulated side-channel traces, it was possible to determine the best threshold value for each technique. Without any knowledge on the component, it is more difficult to fix those threshold values. The attacks could be processed with guess on these thresholds, for instance selecting 0.5 for the collision correlation, but it could not reach optimal efficiency or fail. It is then preferable to determine the best threshold values through a characterization phase of the multiplier, either with an access to an open sample or using the public exponentiation calculation as suggested in [**?**].

Results

We obtain the success rates given in tables **??** ($\sigma = 0$), **??** ($\sigma = 2$) and **??** ($\sigma = 7$) for different key lengths ranging from 512 bits to 2048 bits. Figures **??** and **??** present a graphic comparison of these results for $\sigma = 0$ and $\sigma = 7$.

Technique	512 bits	768 bits	1024 bits	1536 bits	2048 bits
Big Mac [**?**]	0.986	0.990	0.993	0.994	0.995
SAC 2008 [**?**]	0.533	0.618	0.734	0.858	0.897
Big Mac CoCo (§**??**)	0.999	1.00	1.00	1.00	1.00
Rosetta ED (§**??**)	1.00	1.00	1.00	1.00	1.00
Rosetta CoCo (§**??**)	1.00	1.00	1.00	1.00	1.00

Table 4.1: Success rate with a null noise, $\sigma = 0$

Results Interpretation

We observe that with no noise (cf. Table **??**) all techniques are efficient when applied to large modulus bit lengths (1536 bits or more). For smaller modulus lengths, the SAC 2008 technique is inefficient (probability of success close to 0.5) as expected since the number of useful operations in that case is too small.

In case of a noisy component, we observe that the original Big Mac and the attack from SAC 2008 are not efficient, their probability of success is about 0.5–0.7. Big Mac

[4]Regarding the standard deviation of the noise, a unit corresponds to the side-channel difference related to a one bit difference in the Hamming weight.

Technique	512 bits	768 bits	1024 bits	1536 bits	2048 bits
Big Mac [**?**]	0.767	0.775	0.807	0.816	0.818
SAC 2008 [**?**]	0.546	0.629	0.717	0.805	0.855
Big Mac CoCo (§**??**)	0.981	0.998	0.999	1.00	1.00
Rosetta ED (§**??**)	1.00	1.00	1.00	1.00	1.00
Rosetta CoCo (§**??**)	1.00	1.00	1.00	1.00	1.00

Table 4.2: Success rate with a moderate noise, $\sigma = 2$

Technique	512 bits	768 bits	1024 bits	1536 bits	2048 bits
Big Mac [**?**]	0.557	0.577	0.621	0.614	0.632
SAC 2008 [**?**]	0.551	0.577	0.623	0.662	0.702
Big Mac CoCo (§**??**)	0.737	0.855	0.909	0.963	0.981
Rosetta ED (§**??**)	0.711	0.821	0.878	0.953	0.992
Rosetta CoCo (§**??**)	0.685	0.816	0.906	0.992	0.997

Table 4.3: Success rate with a strong noise, $\sigma = 7$

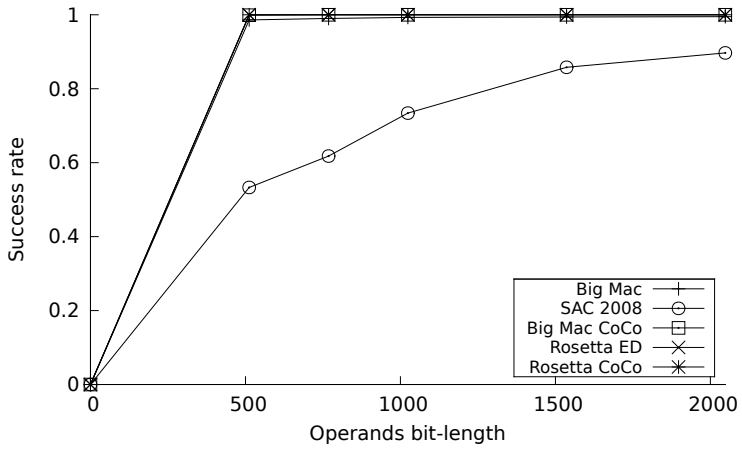


Figure 4.10: Success rate of the different attacks with no noise.

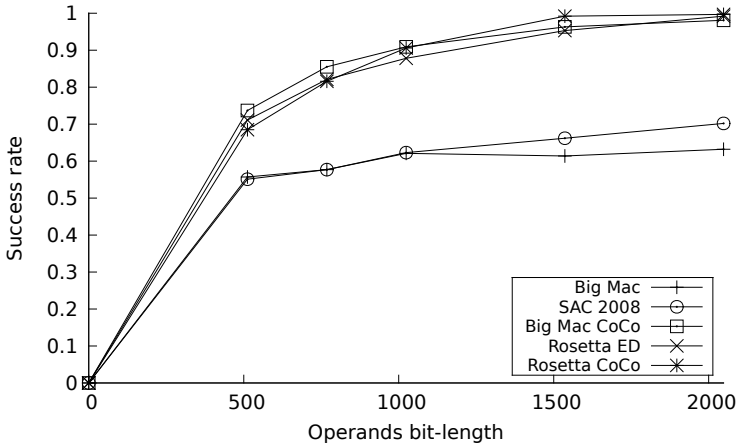


Figure 4.11: Success rate of the different attacks with a strong noise, $\sigma = 7$.

analysis using collision correlation, and both Rosetta techniques start to be efficient from 1024-bit operands and are very efficient for 1536-bit and 2048-bit operands.

Our study demonstrates that these three last techniques are the most efficient ones and represent a more serious threat for blinded exponentiation than the original Big Mac.

From Partial to Full Exponent Recovery

Depending on the component, on the leakage and noise level of the chip, we observe that the success rate of the attack varies and may reveal too few information to recover the whole exponent value. In the case where uncertainty remains on some exponent bits, the attack from Schindler and Itoh [**?**] may help to reveal them. If necessary, Rosetta analysis can thus be advantageously combined with this technique to completely recover the exponent.

4.4.5 Countermeasures

As for the other attacks considered in this paper, both Rosetta techniques we introduced present the following interesting properties: (i) they make use of a single side-channel trace and, (ii) they do not require the knowledge of the message nor of the modulus. As a consequence they are applicable even when the classical set of blinding countermeasures (message, modulus, exponent) is implemented and whatever the size of the random values used.

It is worth to notice that Rosetta won't distinguish a squaring operation from a multiplication if only one of the two operands has the additive countermeasure applied. A first idea to prevent these attacks is to improve the message blinding by randomizing it before each long-integer multiplication, for instance by adding the modulus n or a multiple thereof to the message. At this point, it is worth noticing a specific difference between both Rosetta and other attacks. Rosetta can distinguish a squaring from a

multiplication without using any template or previous leakage. This is not the case with the other techniques — except for the single trace variant of the SAC 2008 attack which we demonstrate not to be efficient in the previous section. The consequence is that Rosetta is still applicable even when this improved blinding is implemented.

We recall hereafter three existing countermeasures that we believe to withstand all the techniques presented in this paper.

Shuffled Long-Integer Multiplication In [**?**], a long integer multiplication algorithm with internal single-precision multiplications randomly permuted is presented. More details are given in [**?**, Sec. 2.7]. This countermeasure makes Rosetta analysis virtually infeasible as indices i, j of multiplication $x_i \times y_j$ are not known anymore.

Always True Multiplication This solution consists in ensuring that multiplication operands are always different (or different with high probability). To achieve this objective, before each multiplication $\mathrm{LIM}(x, y)$, both operands x and y are randomized by $x^\star = x + r_1.n$ and $y^\star = y + r_2.n$. If $r_1 \neq r_2$, two equal operands x and y are traded for $x^\star$ and $y^\star$ with $x^\star \neq y^\star$ and the operation $\mathrm{LIM}(x^\star, y^\star)$ is not a squaring.

Square-Always algorithm The square-always algorithm presented in [**?**] processes any multiplication using two squarings. As for the solution of using multiplications of different terms only, Rosetta does not apply. Regular atomic square always algorithms can be used to prevent SSCA. Exponent blinding countermeasure must be associated with this solution.

4.4.6 Conclusion

We present in this study new side-channel methods — the Big Mac using collision correlation and the two Rosetta techniques — allowing to distinguish a squaring from a multiplication when the same long-integer multiplication algorithm is used for both operations. They can be used to recover an RSA secret exponent — both in standard or CRT mode — with a single execution side-channel trace. We compare our new techniques with other single trace side-channel analyses and demonstrate that they are more efficient than previous ones, especially on noisy measurements. We show that classical combination of message, modulus and exponent blindings is not sufficient to counteract our analysis and we suggest more advanced countermeasures. As a conclusion, we quote Colin Walter [**?**] to recall the very interesting property of these attacks:

"The longer the key length, the easier the attacks."

Chapter 5

Defeating with Fault Injection A Combined Attack Resistant Implementation

5.1 Introduction

Fault Analysis (FA), or *Active Attacks*, consists in perturbing the algorithm process to obtain an abnormal behavior. It can be done by injecting power glitches on the circuit pad or by precise laser light emissions on the device surface (front side or back side). An erroneous computation result is then obtained which can be exploited to recover entirely or partially the secrets. Different active attacks exist: the *Differential Fault Analysis* (DFA), the *Ineffective Fault Analysis* (IFA), the *Collision Fault Analysis* (CFA).

Most of the cryptosystems are nowadays threatened by both techniques like RSA [**?**] and ECC [**?**, **?**] embedded implementations. We focus our study in this chapter on those embedded implementations. In the last decade many countermeasures have been presented to design side-channel resistant algorithm on the first hand and fault injection countermeasures on the other hand. For years implementing those countermeasures separately has never been an issue. But in 1997 Amiel et al. [**?**] present a *combined passive and active attack* on an RSA implementation which is considered at this time resistant to both SCA and FA techniques separately. In 2010 Schmidt et al. [**?**] propose combined-attack resistant algorithms to compute exponentiation and scalar multiplication. Their implementations cleverly include tricks to thwart the Amiel et al. attack.

However in this chapter we present new attacks on their algorithms. The first technique we introduce is a first order fault attack which can recover the whole secret exponent with a practical number of faulted results. Our fault injections benefit from a flaw in the infective computation design of the Schmidt et al. algorithms. The second threat on these algorithms is an attack combining fault injection with differential analysis on many executions. This analysis targets their use of a specific exponentiation technique, *i.e.* left-to-right multiply always, in order to thwart its supposedly resistance against combined attacks.

In Section **??** we remind the reader the necessary background on side-channel and

fault attacks, as well as on combined attack resistant implementation in order to understand the attacks presented in this chapter. In Section **??** we introduce the first order fault attacks which can defeat the combined exponentiation from Schmidt et al. on a simplified and the complete versions of this algorithm. New combined attacks are presented in Section **??**. Section **??** propose an improved version of the Schmidt et al. algorithm which counterfeit the new attacks presented. We conclude in Section **??**.

5.2 Background

We present in this section the combined attack principle and the previous publications on the subject. We also remind the Schmidt et al. algorithms we are attacking in the rest of this chapter.

5.2.1 Combined Attacks on Asymmetric Cryptosystems

Since the publication from Amiel *et al.*, combined attacks have been more and more investigated. This technique exploits leakage information from both a fault analysis (FA) and a classical side-channel attack like SSCA or DSCA. Both symmetric and asymmetric cryptosystems have been shown vulnerable to it. In this paragraph we briefly review the combined attacks proposed in the literature.

The first combined attack publication from Amiel et al. [**?**] combines a fault attack with an SSCA in order to break a modular exponentiation that is supposedly secure against DFA and SSCA. The authors attack a left-to-right multiply always algorithm implementing the *atomicity* principle from Chevallier-Mames et al. [**?**]. Additionally the message and the secret exponent values were randomized to counterfeit DSCA. The first step of the attack consists in injecting a fault in one of the registers (or in the RAM) before (or during) the beginning of the exponentiation. The fault aims at creating a modified message value that will leak in SSCA each time it is manipulated. For instance a low Hamming weight value has been introduced into a part of the message, or the message pointer has been modified to include an erased area of the RAM. This message modification renders the message manipulations visible into a side-channel trace. It becomes then possible to distinguish a squaring operation from a multiplication using SSCA as described in [**?**]. Hence, the FA protection that is present at the end of the algorithm cannot prevent the SSCA leakage that has already occurred during the computation. This attack is very efficient as a single fault applied successfully to the calculation execution will make the SSCA efficient. The principle of the attack of Amiel et al. seems to be applicable to any classic left-to-right atomic algorithm, either exponentiation or scalar multiplication. In [**?**], the authors propose a countermeasure called *Detect and Derive* based on the principle of infective computation. However, it was shown vulnerable in [**?**]. In this paper, Schmidt et al. introduce a new resistant exponentiation algorithm, as well as a scalar multiplication algorithm, also based on infective computation. The idea is to be able to detect a fault as soon as it happens and corrupt the data if necessary so that no relevant information is leaking anymore.

More recently, in [**?**], Fan et al. study the case of combined attacks specially targeting elliptic curve scalar multiplication. Using the properties of elliptic curves, they develop

a powerful attack that can defeat atomic and regular algorithms. In order to perform the attack, one needs to choose a particular input point of the scalar multiplication. By injecting a fault after the initial point verification, the attacker is then able to obtain a point with a small order. During the scalar multiplication, computations with the faulted point will end up on the infinity point which is particularly visible by SSCA in most implementations. The attacker is then able to find information on the secret scalar.

5.2.2 Schmidt et al. Resistant Algorithm

We remind in the following the combined attack resistant implementation from Schmidt et al. [**?**] to give the reader the necessary notions to understand our attacks. We principally consider the exponentiation algorithm in this chapter, however most of our attack paths can be directly applied to the scalar multiplication counterpart.

Fault model considered.

In their paper [**?**], Schmidt et al. deal with the three following fault attack models. The attacker is able with fault injection to:

- randomize data to an unknown value,
- reset data to all zeros or all ones or any given fix value,
- modify opcodes, *i.e.* skip instructions, break loops, etc.

The authors only take into consideration first order fault injections, *i.e.* an attacker injects only one fault per execution of the algorithm. They present two algorithms protected against combined attacks under these fault models. Their first algorithm (Alg. **??**) [**?**, Alg. 3] is a protected exponentiation, and their second one [**?**, Alg. 4] is a protected scalar multiplication. Both algorithms are based on the same principles of countermeasures.

We remind the reader through Algorithm **??** the detailed combined attack resistant algorithm for exponentiation from Schmidt et al. [**?**].

Notations.

In the rest of the chapter, we use the following notations:

- let W be the block length that is generally the size of a processor word, *i.e.* $W = 8$ (resp. $W = 16$ or $W = 32$) for an 8-bit (resp. for a 16-bit or a 32-bit) architecture,
- let d be the t-bit secret exponent and $d = (d_{t-1}, d_{t-2}, \dots d_1, d_0)_2$, with d_i the i-th bit of d, its binary representation,
- let $\bar{d} = (\bar{d}_{t+\lambda-1}, \bar{d}_{t+\lambda-2}, \dots \bar{d}_1, \bar{d}_0)_2$ be the blinded exponent,
- let $\tilde{d}$ be the blinded exponent encoded using the function ψ_α detailed below,
- let $\hat{d}$ be the exponent decoded using ψ_α^{-1},

Alg. 5.2.1 Schmidt et al. [**?**, Alg. 3] left-to-right exponentiation.

Input: $d = (d_{t-1}, \ldots, d_0)_2, m \in \mathbb{Z}_N, N$ and block length W.
Output: $m^d \bmod N$

1. $r_1 \leftarrow \mathsf{random}(1, 2^\lambda - 1)$
2. $r_2 \leftarrow \mathsf{random}(1, 2^\lambda - 1)$
3. $i \leftarrow (r_2^{-1} \bmod N) \cdot r_2$
4. $R_0 \leftarrow i \cdot 1 \bmod N r_2$
5. $R_1 \leftarrow i \cdot m \bmod N r_2$
6. $\bar{d} \leftarrow d + r_1 \cdot \varphi(N)$
7. $[\tilde{d}^{(l-1)}, \ldots, \tilde{d}^{(0)}] \leftarrow [\psi_0(\bar{d}^{(l-1)}), \ldots, \psi_0(\bar{d}^{(0)})]$
8. $k \leftarrow 0$
9. $j \leftarrow \mathsf{bitlength}(\tilde{d}) - 1$
10. **while** $j \geqslant 0$ **do**
11. $\quad R_0 \leftarrow R_0 \cdot R_k \bmod N r_2$
12. $\quad$ **if** $(R_0 = 0)$ **or** $(R_1 = 0)$ **then**
13. $\quad\quad [\tilde{d}^{(l-1)}, \ldots, \tilde{d}^{(0)}] \leftarrow [1, \ldots, 1]$
14. $\quad \hat{d} \leftarrow \psi^{-1}_{(R_0 + R_1 \bmod r_2)}(\tilde{d}^{(\lfloor j/W \rfloor)})$
15. $\quad k \leftarrow k \oplus \mathsf{bit}(\hat{d}, j \bmod W)$
16. $\quad j \leftarrow j - (1 - k)$
17. $c \leftarrow R_0 \bmod N$
18.
19. **return** c

- let $d^{(j)}$ be the j-th W-bit word of d.

The exponent is protected through an encoding function $\psi_\alpha : \mathbb{Z}_{r_2} \times \mathbb{Z}_{r_2} \to \mathbb{Z}_{r_2}$ which is an invertible function defined as:

$$\psi_\alpha(d^{(j)}) = (\alpha + N)^{-1} \cdot d^{(j)} \bmod r_2,$$
$$\psi_\alpha^{-1}(\tilde{d}^{(j)}) = (\alpha + N) \cdot \tilde{d}^{(j)} \bmod r_2,$$

with $\alpha \in \mathbb{Z}_{r_2}$, N the modulus and r_2 a small random value such that $r_2 > 2^W$.

In the next section, we introduce single fault attacks on a simplified version (without exponent/scalar blinding) of Algorithm **??** and on the complete Algorithm **??**.

5.3 Fault Attack on Schmidt et al. Algorithms

We show in this section that a classical single fault attack can still be applied to the exponentiation algorithm proposed by Schmidt et al. [**?**, Alg. 3]. We consider in this section fault attacks based on the modification of an opcode, *i.e.* skip of instruction. We first propose a fault attack on a simplified version of Alg. **??** where the blinding of the exponent is not present (Line 6). Then, based on the same fault attack principle, we propose an attack on the complete version of Alg. **??**.

5.3.1 Fault Attack on a Simplified Algorithm

As we consider no exponent blinding in this section, we have that $\bar{d} = d$, hence the encoded exponent $\tilde{d}^{(k)} = \psi_0(d^{(k)})$ for $0 \leqslant k \leqslant l-1$ where l is the length of d in W-bit words.

To protect their implementation from the combined attack presented in [**?**], the authors introduced at Line 12 of the algorithm an infective operation. The purpose is to corrupt the secret exponent when a fault injection is detected in order to cancel the side-channel leakage that could reveal the secret. More precisely the purpose of the test Line 12 of Alg. **??** is to corrupt the exponent in case one of the registers R_0 or R_1 was erased by fault which could leak simple side-channel information. Hence, the exponentiation would continue its course but using false exponent bits. Schmidt et al. choose to affect the value 1 to all words of the encoded exponent $\tilde{d}$. The decoding of a word of exponent performed Line 15, assuming no faults in registers R_0 or R_1, computes for the k-th word of the exponent:

$$\hat{d} = \psi_0^{-1}(\tilde{d}^{(k)}) = N \cdot \psi_0(d^{(k)}) \bmod r_2 = d^{(k)} \bmod r_2.$$

It is very important for our attacks to notice that if the exponent has been corrupted in Line 13, all the decoded W-bit words of exponent until the end of the exponentiation are equal to the value:

$$\hat{d} = \psi_0^{-1}(1) = N \cdot 1 \bmod r_2 = N \bmod r_2.$$

Moreover, we note from Line 16 that only the W least significant bits of $\hat{d}$ are considered for the exponentiation. It signifies that from the moment a single fault is injected to skip the test at Line 12, all the remaining W-bit words $\hat{d}^{(i)}$ being used for the rest of the exponentiation are equal to this same and unique value $N \bmod r_2$.

We introduce for our analysis two additional notations. Let H be the value $(N \bmod r_2) \bmod 2^W$ and $\tilde{t} = l \cdot W$ be the bit length of $\tilde{d}$.

Now consider that an attacker already knows the v (can be zero) first bits of the exponent and skips Line 12 by fault injection u bits after in the loop of the algorithm. The algorithm outputs the faulted result $\check{S}_u$ that used the following exponent:

$$\check{d}_u = \underbrace{\sum_{i=\tilde{t}-v}^{\tilde{t}-1} 2^i \cdot \tilde{d}_i}_{\text{known part of the exponent}} + \sum_{i=\tilde{t}-v-u}^{\tilde{t}-v-1} 2^i \cdot \tilde{d}_i + \sum_{i=0}^{\tilde{t}-v-u-1} 2^i \cdot H_{(i \bmod W)}. \qquad (5.1)$$

By doing a guess on the next u unknown bits of d and another guess on the value of H, an attacker can compute the guessed result of the exponentiation, denoted $S_g(u, H)$. Then by comparing this value $S_g(u, H)$ with $\check{S}_u$, he can decide if his guesses are correct or not. After an exhaustive calculation for all possible values, when $S_g(u, H) = \check{S}_u$ the attacker recovers the right values $(d_{t-v-1}, \ldots, d_{t-v-u})$ and H.

Complexity.

The computational complexity $\mathcal{C}$ of our fault attack to recover the exponent is:

$$\mathcal{C} = \mathcal{O}\left(\frac{2^{(u+W)} \cdot \tilde{t}}{u}\right) \quad \text{exponentiations.}$$

The number of faulty signatures $\mathcal{F}$ to collect is:

$$\mathcal{F} = \mathcal{O}\left(\frac{\tilde{t}}{u}\right).$$

We have validated our attack on a standard PC using the GMP library[1] for different RSA keys (values and bit-length) with success.

Table **??** gives examples of computational complexity of our attack for $u = 1$ and different values of W and t.

W | Bit-length t	512 bits	1024 bits	2048 bits
8	$\mathcal{C} = 2^{18}$	$\mathcal{C} = 2^{19}$	$\mathcal{C} = 2^{20}$
16	$\mathcal{C} = 2^{26}$	$\mathcal{C} = 2^{27}$	$\mathcal{C} = 2^{28}$
32	$\mathcal{C} = 2^{42}$	$\mathcal{C} = 2^{43}$	$\mathcal{C} = 2^{44}$

Table 5.1: Example of computational complexities for $u = 1$ to recover the exponent on the simplified algorithm.

This attack also applies to the simplified scalar multiplication algorithm of Schmidt et al. [**?**, Alg. 4], *i.e.* with no scalar blinding. However this analysis only works if the attacker can retrieve the exponent u bits at a time using different faulty results. Hence in the presence of exponent blinding, it cannot be applied directly. We present in the following an adaptation of the attack to the blinded exponentiation algorithm.

5.3.2 Fault Attack on the Complete Version of the Algorithm

Based on the attack presented previously, we propose in this section a variation in order to attack Alg. **??** considering the exponent blinding countermeasure. As previously observed by Berzati et al. in [**?**], the blinding using $\varphi(N)$ does not mask homogeneously the exponent. We propose here an attack which exploits this flaw. We do not include the processing of the exponent through the encoding function ψ for easier notation. As seen in the previous section, the output size of the encoding function, *i.e.* the size of the random r_2, has no effect on the attack because the algorithm only considers bits modulo W.

Let $\bar{d}$ be the blinded exponent such that $\bar{d} = d + r_1\varphi(N)$ with r_1 a λ-bit random. Let $\bar{d} = \sum_{i=0}^{t+\lambda-1} 2^i \cdot \bar{d}_i$ be its binary decomposition. We can also write it as:

[1]The GNU Multiple Precision Arithmetic Library, available at urlhttp://gmplib.org/

$$\bar{d} = \sum_{i=t}^{t+\lambda-1} 2^i \cdot (r_1 N)_i + \sum_{i=t/2+\lambda}^{t-1} 2^i \cdot (d + r_1 N)_i + \sum_{i=0}^{t/2+\lambda-1} 2^i \cdot (d + r_1 \varphi(N))_i. \quad (5.2)$$

We observe that the least significant bits of the secret exponent d are randomized with the full mask $r_1\varphi(N)$. On the other hand, the most significant (half upper) bits of d are only masked with $r_1 N$. The attack consists in finding d from its most significant bits to its least significant ones.

We note $\check{S}_u$ the faulty result of an exponentiation where the test Line 12 of Alg. **??** has been skipped by fault after the u-th unknown bit of the exponent has been processed. The faulty exponent $\check{d}_u$ corresponding to $\check{S}_u$ is detailed in Eq. (**??**). We consider that the attacker has already retrieved the v most significant bits of d.

Retrieving the MSB part of d.

We first consider a fault injected after the u-th unknown bit within the range of bits of d being $[(t/2 + \lambda), t]$. We consider then:

$$\bar{d} = \sum_{i=t}^{t+\lambda-1} 2^i \cdot (r_1 N)_i + \sum_{i=t-u}^{t-1} 2^i \cdot (d + r_1 N)_i + \sum_{i=t/2+\lambda}^{t-u-1} 2^i \cdot (d + r_1 N)_i + \sum_{i=0}^{t/2+\lambda-1} 2^i \cdot (d + r_1 \varphi(N))_i. \quad (5.3)$$

Let $\bar{d}_{[u]} = \sum_{i=t-u}^{t+\lambda-1} 2^i \cdot \bar{d}_i$ and $\bar{d}_{<u>} = \sum_{i=0}^{t-u-1} 2^i \cdot \bar{d}_i$.

The faulty exponent $\check{d}_u$ of the result $\check{S}_u$ can be approximated as $\check{d}_u \approx \bar{d}_{[u]} + \bar{d}_{<u>}$, not considering the carry propagation.

Once the fault has been injected, as we observed previously, the least significant part of the encoded exponent is fixed at 1 in Line 13 of Alg. **??** as an infective calculation countermeasure. Hence, we have that after the fault at the u-th bit, $\bar{d}_{<u>} = \sum_{i=0}^{t-u-1} 2^i \cdot H_{(i \bmod W)}$ with $H = (N \bmod r_2) \bmod 2^W$.

In order to find $\bar{d}_{<u>}$, the attacker only needs to guess W bits of H. We note $d_{\mathsf{known}} = \sum_{i=t-v}^{t-1} 2^i \cdot d_i$ the most significant v bits of d already retrieved by the attacker.

From Eq. (**??**) and (**??**), the most significant part of the exponent $\bar{d}_{[u]}$ can be approximated as:

$$\begin{aligned} \bar{d}_{[u]} &\approx \sum_{i=t-u}^{t+\lambda-1} 2^i \cdot (d + r_1 N)_i \\ &\approx d_{\mathsf{known}} + \sum_{i=t-v-u}^{t-v-1} 2^i \cdot d_i + \sum_{i=t-v-u}^{t+\lambda-1} 2^i \cdot (r_1 N)_i + \mathsf{carry} \end{aligned}$$

where carry is the possible carry bit resulting from the addition between the u first bits of r_1 and N. In order to find the value of $\bar{d}_{[u]}$, the attacker needs to guess u bits of d and λ bits of r_1. The possible carry bit only gives an uncertainty on the parity of the guessed value of d. By guessing $2^{(u+W+\lambda)}$ bits, the attacker can construct a guess of the

full exponent $\check{d}_u$. He can then validate his guess by checking if the following relation is verified:

$$\check{S}_u \stackrel{?}{=} m^{\bar{d}_{[u]}+\bar{d}_{<u>}} \bmod N. \tag{5.4}$$

Retrieving the LSB part of d.

Once we have recovered the MSB part of d, we now consider a fault injected after the u-th unknown bit within the range of bits of d being $[0, (t/2+\lambda)]$. Contrary to the MSB case, the bits of d will not be guessable directly as the full mask $r_1\varphi(N)$ is now applied.

We consider then:

$$\bar{d} = \sum_{i=t/2+\lambda}^{t+\lambda-1} 2^i \cdot (d + r_1 N)_i + \sum_{i=t/2+\lambda-u}^{t/2+\lambda-1} 2^i \cdot (d + r_1\varphi(N))_i + \sum_{i=0}^{t/2+\lambda-u-1} 2^i \cdot (d + r_1\varphi(N))_i. \tag{5.5}$$

The least significant part of the faulted exponent is still equal to $\bar{d}_{<u>} = \sum_{i=0}^{t/2+\lambda-u-1} 2^i \cdot H_{(i \bmod W)}$. As previously, in order to find $\bar{d}_{<u>}$, the attacker only needs to guess W bits of H.

We can write the most significant part of the exponent using Eq. (**??**) as:

$$\begin{aligned}
\bar{d}_{[u]} &= \sum_{i=t/2+\lambda-u}^{t+\lambda-1} 2^i \cdot (d + r_1\varphi(N))_i \\
&= \sum_{i=t/2+\lambda-u}^{t+\lambda-1} 2^i \cdot (d + r_1 N - r_1(p+q-1))_i \\
&\approx d_{\mathsf{known}} + \sum_{i=t/2+\lambda-v-u}^{t/2+\lambda-v-1} 2^i \cdot \delta_i + \sum_{i=t/2+\lambda-v-u}^{t+\lambda-1} 2^i \cdot (r_1 N)_i + \mathsf{carry}
\end{aligned}$$

where $\delta_i = (d - r_1(p+q-1))_i$ and carry is the possible carry due to the addition of the u bits of $r_1 N$ with $(d - r_1(p+q-1))$.

As previously, the possible carry bit is not taken into account in the analysis as it only affects the parity of the final guess and is easily checkable. In order to find the value of $\check{d}_u$, the attacker needs to guess $2^{(u+W+\lambda)}$ bits: u bits of δ, λ bits of r_1 and W bits of H. The attacker can then construct a guess of the full exponent and validate this guess by checking if $\check{S}_u \stackrel{?}{=} m^{\bar{d}_{[u]}+\bar{d}_{<u>}} \bmod N$. Contrarily to the MSB case we described previously, recovered bits are not bits of d but u bits of δ_i. This can be solved by using many faulted executions instead of one. Indeed as the values of d and $(p+q-1)$ are fixed between different exponentiations, by faulting at the same time u, the attacker can obtain an additional guess for δ with a different r_1. With two or more faulted exponentiations, he will be able to determine the u bits of d and the u bits of $(p+q-1)$. The validation of the guesses are made, similarly to the MSB case, by comparing the faulted result of exponentiation to the exponentiation with our entire guessed exponent (see Eq. (**??**)).

Complexity.

The computational complexity $\mathcal{C}$ of our fault attack to recover the exponent is:

$$\mathcal{C} = \mathcal{O}\left(\frac{2^{(u+W+\lambda)} \cdot t}{u}\right) \quad \text{exponentiations.}$$

The number of faulty signatures $\mathcal{F}$ to collect is:

$$\mathcal{F} = \mathcal{O}\left(\frac{t}{u}\right).$$

We can note that our fault attack does not require non-faulted results of exponentiations. The complexity of our attack is not impacted by the size of r_2 used in the encoding function ψ but by the size of the window W as only W bits of the output of the encoding are used to perform the exponentiation. This undesirable effect of Alg. **??** implies that the smaller processor words, the easier this fault attack is to perform. As previously, this attack has been validated on a standard PC using the GMP library.

We have presented first order (single) fault injections that defeat the combined resistant implementation with few faulted executions and a reasonable complexity that render this attack practical. Our attacks use a flaw in the design of the infective computation in Schmidt et al. algorithms. In the next section we discuss the resistance of Algorithm **??** against combined attacks and particularly with regards to the combined attacks we introduce.

5.4 Combined Attacks on Schmidt et al. Algorithms

Although the algorithms proposed by Schmidt et al. [**?**] are supposedly resistant to the combined attack published by Amiel et al. [**?**], we explain in the following that Alg. **??** can be threatened by more advanced combined attacks.

Combining Fault Injection with Differential Analysis.

We consider the exponentiation algorithm (Alg. **??**) for the description of this attack, however it directly applies to the scalar multiplication algorithm [**?**, Alg. 4]. Note that the internal registers R_0 and R_1 are randomized at the beginning of the algorithm with a random idempotent element i (Line 6 Alg. **??**). Hence, we can only use attacks that consider unknown plaintexts as the randomization by i cannot be easily removed.

A combined attack that uses an instruction skip fault combined with one of the differential attack using unknown plaintext can be mounted on Schmidt et al. algorithms. If the attacker can skip Line 6 in Alg. **??** by fault injection, then the exponentiation is performed without exponent blinding, *i.e.* $\bar{d} = d$. In case a bit of d is $d_j = 0$, the multiplication Line 11 becomes $R_0 \cdot R_0$, whereas if a bit equals 1, it computes $R_0 \cdot R_1$. More precisely, if $d_j = 0$ the output of the multiplication will have the expected Hamming weight of a squaring which is distinct from the expected Hamming weight of a multiplication output as demonstrated in [**?**, **?**]. Hence the attack of Amiel et al. [**?**]

can be applied. However it requires few thousand curves in order to distinguish correctly squaring from multiplication operations. The fault attack on Line 6 then needs to be repeatable which is demonstrated realistic from recent fault injection techniques [**?**, **?**]. Note that the fault repeatability does not need to be perfect as failed faults are considered as noise in the differential analysis treatment. Hence, it only affects the number of curves necessary to recover the secret.

Combining Fault Injection with Template Analysis.

A template attack[2] using the same principle as Amiel et al. was proposed by Hanley et al. [**?**]. With very few curves, the attacker can recover the full exponent in a template matching phase. If the exponent blinding of Line 6 is removed, this attack can also be applied with less faults and less traces compared to the previous one. Note that without the fault injection, this template attack can be mounted using only one curve. Hence, the recovery of the exponent will most certainly not be complete. Depending on the size of the blinding factor r_1 (Line 1), the size of the modulus N and the success rate of the template attack, the methodology of Schindler and Itoh [**?**] can be applied to recover the full exponent.

5.5 Improved Combined Attack Resistant Algorithms

We propose in this section improvements on the exponentiation algorithm (Alg. **??**) to prevent the attacks presented previously. Our proposed improvements also apply to the scalar multiplication variant.

The fault attack presented in Section **??** exploits a skip of instruction on the conditional test in Line 12 where the infective calculation replaced the entire encoded exponent by 1. A simple and efficient countermeasure consists in replacing this fixed value by random values for each words of the exponent. Another protection could be offered through the classical DFA countermeasure consisting in verifying the calculation with the public exponent e when possible.

To prevent the combined attacks we introduced, it becomes necessary to prevent template and differential side-channel techniques. A possible fix consists in randomizing the internal registers R_0 and R_1 before the multiplication so that even if we have to compute $R_0 \cdot R_0$ the representation of the two operands will be different. The Line 11 of Alg. **??** can be replaced by the following:

1. $r_3 \leftarrow \mathsf{random}(1, 2^\lambda - 1)$
2. $R_2 \leftarrow R_k + r_3 \cdot N \bmod N r_2$
3. $R_0 \leftarrow R_0 \cdot R_2 \bmod N r_2$

This modification adds to the cost of Alg. **??** one more register R_2, one modular multiplication with addition and the selection of a random value r_3 at each turn of the loop. Even if the exponent blinding is removed by fault, none of the attacks presented before can

[2] As the plaintext can be unknown to construct these templates, an open device is not mandatory contrary to the usual definition of a template attack. The attacker only needs to record the power consumption of multiplications and squarings with random inputs.

Alg. 5.5.1 Improved Schmidt et al. left-to-right exponentiation.

Input: $d = (d_{t-1}, \ldots, d_0)_2, m \in \mathbb{Z}_N, N$ and block length W.
Output: $m^d \bmod N$

1. $r_1 \leftarrow \mathsf{random}(1, 2^\lambda - 1)$
2. $r_2 \leftarrow \mathsf{random}(1, 2^\lambda - 1)$
3. $i \leftarrow (r_2^{-1} \bmod N) \cdot r_2$
4. $R_0 \leftarrow i \cdot 1 \bmod N r_2$
5. $R_1 \leftarrow i \cdot m \bmod N r_2$
6. $\bar{d} \leftarrow d + r_1 \cdot \varphi(N)$ (optional)
7. $[\tilde{d}^{(l-1)}, \ldots, \tilde{d}^{(0)}] \leftarrow [\psi_0(\bar{d}^{(l-1)}), \ldots, \psi_0(\bar{d}^{(0)})]$
8. **for** $i = 0$ **to** $l - 1$ **do**
9. $\quad w_i \leftarrow \mathsf{random}(1, 2^W - 1)$
10. $k \leftarrow 0$
11. $j \leftarrow \mathsf{bitlength}(\tilde{d}) - 1$
12. **while** $j \geqslant 0$ **do**
13. $\quad r_3 \leftarrow \mathsf{random}(1, 2^\lambda - 1)$
14. $\quad R_2 \leftarrow R_k + r_3 \cdot N \bmod N r_2$
15. $\quad R_0 \leftarrow R_0 \cdot R_2 \bmod N r_2$
16. $\quad$ **if** $(R_0 = 0)$ **or** $(R_1 = 0)$ **then**
17. $\quad\quad [\tilde{d}^{(l-1)}, \ldots, \tilde{d}^{(0)}] \leftarrow [w_{l-1}, \ldots, w_0]$
18. $\quad \hat{d} \leftarrow \psi^{-1}_{(R_0 + R_1 \bmod r_2)}(\tilde{d}^{(\lfloor j/W \rfloor)})$
19. $\quad k \leftarrow k \oplus \mathsf{bit}(\hat{d}, j \bmod W)$
20. $\quad j \leftarrow j - \neg k$
21. $c \leftarrow R_0 \bmod N$
22.
23. **return** c

be applied now as multiplication and squaring operations are no more distinguishable. A similar modification can be applied to the scalar multiplication algorithm [**?**, Alg. 4] but at a higher cost. One needs to randomize each coordinates of the elliptic curve point which means, in the case of classical projective coordinates, an overhead of 3 modular multiplications, 3 random values and a point buffer. Moreover, this technique might not be sufficient on most normalized curves, *i.e.* NIST curves, as their modulus have very particular forms that can still allow for side-channel leakage on randomized coordinates. A more costly alternative solution consists in using a randomized multi-precision multiplication as proposed in [**?**] and [**?**, Sec. 2.7].

It is important also to notice that in practice the public exponent and the value $\varphi(N)$ can be unknown when computing an exponentiation. In that case, the exponent cannot be blinded and the calculation verified. Although there are alternative solutions, as for instance those proposed by Joye in [**?**], it only applies to particular cases. Hence it could be sometimes impossible to apply the blinding on the exponent. However our improved Algorithm **??** is resistant to combined attacks even when those values are unknown.

To the best of our knowledge, the only other exponentiation algorithm resistant

against combined attacks is the algorithm proposed by Giraud [**?**] based on the Montgomery ladder. However it only protects from a corruption of the data registers, the integrity of the exponent is not assured contrary to Schmidt et al. algorithm.

5.6 Conclusion

We have presented in this chapter two new attacks which threaten the combined attack resistant implementations Schmidt et al. published in [**?**]. Our first technique is a single fault injection technique which can recover with few faulted ciphertexts the secret exponent. This attack was possible due to a flaw in the infective computation countermeasure proposed by the original authors. The second method combines fault injection with differential analysis to reach the same objective. Introducing those new vulnerabilities lead us to propose an improved version of this algorithm which offer better protection against the different attacks based on side channel analysis and fault injection techniques.

Chapter 6

Improved Collision Correlation Analysis on First Order Protected AES

6.1 Introduction

Since side-channel attacks potentially concern any kind of embedded implementations of symmetric or asymmetric algorithms, it is recommended to apply various masking countermeasures (among others) in sensitive products [**?**, **?**]. Second-order or higher-order side-channel analysis can however defeat such countermeasures by combining leakages from different instants of the execution of an algorithm and canceling the effect of a mask [**?**, **?**]. Such attacks are considered very difficult to implement and generally require an important number of power curves.

A specific approach for side-channel analysis is using information leakages to detect collisions between data manipulated in algorithms. Side-channel collision attacks against a block cipher were first proposed by Schramm et al. in 2003 [**?**]. Their attack uses differential analysis to exploit collisions in adjacent S-Boxes of the DES algorithm. In [**?**] an attack against the AES is proposed to detect collisions in the output of the first round `MixColumns`. Later, Bogdanov [**?**] improved this attack by looking for equal S-Boxes inputs in several AES executions. He then studied in [**?**] statistical techniques to detect collisions between power curves. Two recent papers have updated the state-of-the-art by introducing correlation based collision detection: Moradi et al. [**?**] proposed a collision attack to defeat an AES implementation using masked S-Boxes, while Witteman et al. [**?**] applied a cross-correlation analysis to an RSA implementation using message blinding.

In this chapter, we present two collision-correlation attacks on software AES implementations protected against first-order power analysis using masked S-Boxes and practical results on both simulated and real power curves. Our attacks are much more efficient and generic compared to the one presented in [**?**]. Moreover we believe our techniques to be applicable to other embedded implementations of symmetric block ciphers.

The remainder of the chapter is organized as follows: Section **??** presents the two

AES first-order protected implementations targeted by our study. Then in Section **??** we present our attacks and practical results on simulated power curves and on a physical integrated circuit. In Section **??** we compare our technique with second-order power analysis. Section **??** deals with the possible countermeasures and finally we conclude this chapter in Section **??**.

6.2 Targeted Implementations

The AES Algorithm.

For the sake of simplicity in this paper we focus on the AES-128 which includes 10 rounds, each one decomposed into four functions: `AddRoundKey`, `SubBytes`, `ShiftRows` and `MixColumns`. It encrypts a 128-bit message $M = (m_0, \ldots, m_{15})$ using a 128-bit secret key $K = (k_0, \ldots, k_{15})$ and produces a 128-bit ciphertext $C = (c_0, \ldots, c_{15})$. Note however that the techniques presented in this paper are easily applicable to AES-192 and AES-256.

The only non-linear function of the AES is `SubBytes` (also referred to as the S-Boxes S in the following) which is a substitution function defined by the pseudo-inversion I in GF(2^8) and an affine transformation. In this paper, we consider the two following solutions that have been proposed to protect this function against first-order attacks.

6.2.1 Blinded Lookup Table

The first targeted implementation uses a *masked* substitution table as proposed by Kocher et al. [**?**] and Akkar et al. [**?**]. This masked table S' is defined by $S'(x_i \oplus u_i) = S(x_i) \oplus v_i$, with u_i (resp. v_i) the mask of the i-th input byte x_i (resp. output byte) of function `SubBytes`, $x_i, y_i, u_i, v_i \in \mathrm{GF}(2^8)$, $0 \leqslant i \leqslant 15$. This table is usually computed before the AES execution and stored in volatile memory.

We further consider that the same masks u and v are applied on all S-Boxes during one execution (or a round at least) of the algorithm, i.e. $u_i = u$ and $v_i = v$ for $0 \leqslant i \leqslant 15$. We believe that this hypothesis is realistic for embedded security products considering that an expensive recomputation of the 256-byte substitution table S' is necessary for each new pair (u, v) and that the storage of many masked tables is not conceivable in memory constrained devices.

6.2.2 Blinded Inversion Calculation

An alternative solution has been proposed by Oswald et al. [**?**] and improved on by Canright et al. [**?**]. It consists in computing the inversion in GF(2^8) using a multiplicative mask. To do this efficiently it is proposed to decompose the computation using inversions in the subfield GF(2^4) (and possibly in GF(2^2)). Such masking method is well suited for hardware implementations.

We recall some properties of the masked inversion. Let I' denote the masked pseudo-inversion such that $I'(x_i \oplus u_i) = I(x_i) \oplus u_i$. The element $x_i \oplus u_i$ in GF(2^8) is mapped to a couple $(x_{i,h} \oplus u_{i,h}, x_{i,l} \oplus u_{i,l})$ of GF(2^4) such that $x_i \oplus u_i \cong (x_{i,h} \oplus u_{i,h})X + (x_{i,l} \oplus u_{i,l})$. As detailed in [**?**] many calculations occur on these subfield elements to compute the

masked inversion of $x_i \oplus u_i$. The exact details of these computations can be found in [**?**]. Note that in these formulas neither $x_{i,h}$ nor $x_{i,l}$ is directly inversed in GF(2^4) but the following value:

$$d_i \oplus u_{i,h} = x_{i,h}^2 \times 14 \oplus (x_{i,h} \times x_{i,l}) \oplus x_{i,l}^2 \oplus u_{i,h} \ .$$

Then the masked inversion in GF(2^4) of $d_i \oplus u_{i,h}$ gives $d_i^{-1} \oplus u_{i,h}$ and is used to compute $I'(x_i \oplus u_i)$.

The 16 input bytes of `SubBytes` are blinded using different masks u_i, but one can notice that input and output masks of the inversion stage are identical. Therefore another threat to take into consideration is the zero value power analysis. This technique has been introduced in [**?**] and [**?**], and recently implemented on the masked inversion in [**?**]. Finally, note that the technique presented in this paper also applies to the improved version of Canright et al. [**?**] when input and output of the inversion are masked with the same value.

6.2.3 Measurements and Validation of Implementations

Curve Acquisition.

We have developed software implementations on a contact smart card using a 16-bit RISC CPU with low power consumption. Two different methods were used to validate our attacks.

First, we used *simulated curves*: a proprietary tool was used to simulate power curves based on the chip architecture and the code executed. This tool generates ideal power consumption curves without any noise which enables to validate in practice the resistance of an implementation to a set of side-channel attacks leaving aside the acquisition and signal processing problems.

Second, we used *real curves*: we made physical measurements on the chip itself using a MicroPross MP100 reader and a Lecroy WavePro numerical oscilloscope.

First-Order Resistance Validation.

Since our aim was to present techniques able to defeat first-order protected devices, we performed the classical first-order differential and correlation analysis on the two implementations presented above, before testing our collision attacks.

To do so, we applied DPA and CPA on the `AddRoundKey`, `SubBytes` and `MixColumns` functions at the first and the last rounds of our implementations. We also performed detailed SPA for each input byte value using many average curves to detect any noticeable (biased) power traces that would reveal a potential leakage. In any case no leakage were observed. We also verified that both implementations were immune to zero value power analysis and to the attack presented by Moradi et al.

We have thus verified that to the best of our knowledge both considered AES implementations are resistant to known first-order attacks. Nevertheless we present in the next section two new collision-correlation techniques which jeopardize these implementations.

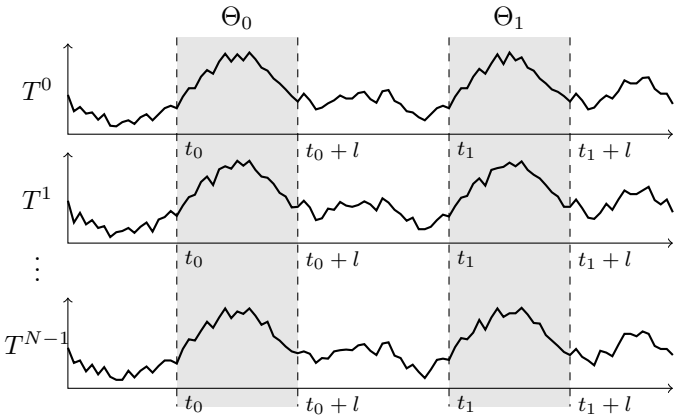


Figure 6.1: General description of the collision-correlation attack

6.3 Description of our Attacks

In this section, we present the general principle of collision-correlation attacks and then detail how it can be applied on the two considered AES implementations.

6.3.1 The Collision-Correlation Method

The principle of the attacks presented in this paper is to detect internal collisions between data processed in blinded S-Boxes on the first round of an AES execution. We demonstrate in the following that if i) we are able to detect that the same data is processed at instants t_0 and t_1, and ii) the S-Boxes are blinded such that either the same mask is applied to all message bytes or the mask is identical at the input and the output of each S-Box, then it is possible to infer information on the secret key with very few curves.

In the following, we will denote $(T^n)_{0 \leqslant n \leqslant N-1}$ a set of N power traces captured from a device processing N encryptions of the same message M. Then we consider two instructions[1] whose processing starts at times t_0 and t_1 and denote l the number of points acquired per instruction processing. As depicted in Fig. **??** we finally consider $\Theta_0 = (T^n_{t_0})_n$ and $\Theta_1 = (T^n_{t_1})_n$ the two series of power consumptions segments at instants t_0 and t_1.

Note that in practice the N power curves should start at the same instant of the encryption and be perfectly aligned. Such conditions generally require signal processing to be performed first. Note also that as the sampling rate is usually such that $l > 1$ points are acquired per instruction, we can generalize the definition of Θ_0 and Θ_1 as being series of l-sample curve segments instead of series of single power consumption samples.

[1] In our attacks we only consider the correlation between two identical instructions, but it may even be possible to detect that two different instructions manipulate identical data, e.g. by spotting a data bus using EMA.

The final stage of the attack consists in applying a statistical treatment to (Θ_0, Θ_1) in order to identify if the same data was involved in $T^n_{t_0}$ and $T^n_{t_1}$ for $0 \leqslant n \leqslant N-1$. Let $\texttt{Collision}(\Theta_0, \Theta_1)$ denote a decision function returning `true` or `false` depending on whether this property is presumed to be fulfilled or not. Such a decision function would usually compare the value of a synthetic criterion with a practically determined threshold. Possible examples of such a criterion include the mean[2] squared difference, the least squared difference with binary or ternary voting [**?**], and the maximum Pearson correlation factor. As we used this latter criterion in our study, we recall that an estimation of the Pearson correlation factor between series of curve segments Θ_0 and Θ_1 at time offset t $(0 \leqslant t \leqslant l-1)$ expressed as

$$\begin{aligned}\hat{\rho}_{\Theta_0,\Theta_1}(t) &= \frac{\mathrm{Cov}(\Theta_0(t), \Theta_1(t))}{\sigma_{\Theta_0(t)}\sigma_{\Theta_1(t)}} \\ &= \frac{N\sum(T^n_{t_0+t}T^n_{t_1+t}) - \sum T^n_{t_0+t}\sum T^n_{t_1+t}}{\sqrt{N\sum(T^n_{t_0+t})^2 - (\sum T^n_{t_0+t})^2}\sqrt{N\sum(T^n_{t_1+t})^2 - (\sum T^n_{t_1+t})^2}}\end{aligned}$$

where summations are taken over $0 \leqslant n \leqslant N-1$, and $\Theta_i(t) = (T^n_{t_i+t})_n$ for $i \in \{0,1\}$.

$\texttt{Collision}(\Theta_0, \Theta_1)$ thus consists in comparing $\max_{0\leqslant t\leqslant l-1}(\hat{\rho}_{\Theta_0,\Theta_1}(t))$ to a given threshold. In our experiments a preliminary characterization of the targeted device enabled us to find proper values for l and the threshold.

Note that in this collision-correlation technique we compute the correlation factor between a set of real power consumptions Θ_0 with another set of real power consumptions Θ_1, rather than with model dependent estimations. As Bogdanov already described in [**?**] about binary and ternary voting techniques, an interesting property of this method is that, unlike Hamming weight based CPA, our criterion does not rely on a particular leakage model. The consequences of this are that i) the attack is more generic and requires much less knowledge of the targeted device, and ii) the secret S-Boxes may be attacked as well as known ones.

As said above, correlating two instants (curve segments) on different traces has already been applied by Moradi et al. [**?**] on a particular AES implementation. However they collect many traces obtained by encrypting random messages and average them according to the value of an S-Box input byte. This results in 2^8 averaged curves for each byte position, from which they try to detect collisions between two bytes. They successfully carried out this attack on their implementation of the Canright et al. [**?**] first-order protected implementation. However as indicated by the authors their implementation presented a remaining first-order leakage based on zero-value attack. We applied Moradi's attack to the first-order protected implementations considered in this study without success. We thus consider that this attack is not applicable to most first-order protected implementations. Indeed averaging different traces implies the use of new random mask values which should spoil the influence of the unmasked data and make the collision of intermediate values undetectable. The technique we develop in this paper improves on Moradi's attack in order to detect data collisions by comparing two instants on a same trace and repeating it on many executions without the destruc-

[2]The mean being taken over the N traces as well as over the l samples.

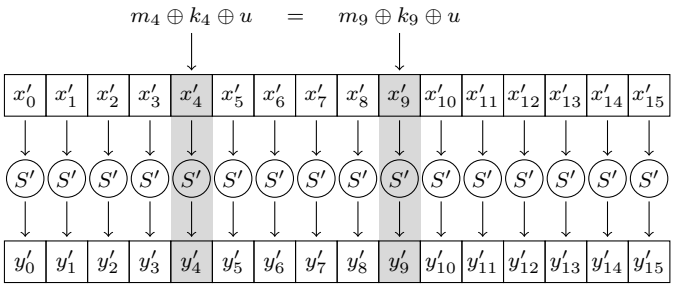


Figure 6.2: Collision between the computation of two S-Boxes on bytes 4 and 9 on the blinded lookup table implementation

tive averaging process. In the following we detail two applications of our attack on two different implementations.

Remark Collision based analyses are also known as *cross-correlation attacks* in [**?**] and *multiple-differential collision attacks* in [**?**]. We prefer the term *collision-correlation attacks* since cross-correlation may be ambiguous depending on the context, and multiple-differential collision attacks seems us too generic for our method.

6.3.2 Attack on the Blinded Lookup Table Implementation

First, we present an application using principle presented above on the implementation described in Section **??**. This attack targets the execution of the first round `SubBytes` function. Each 16 masked input byte $x'_i = x_i \oplus u$ is substituted by a masked output byte $y'_i = y_i \oplus v$ where $y'_i = S'(x'_i)$. We try to detect when two `SubBytes` inputs (and outputs) are equal within the first AES round as depicted on Fig. **??**.

Detecting a collision in the first AES round between bytes i_1 and i_2 yields that $x_{i_1} \oplus u = x_{i_2} \oplus u$ and considering that $x_i = m_i \oplus k_i \oplus u$ implies the following relation of the two involved key bytes:

$$k_{i_1} \oplus k_{i_2} = m_{i_1} \oplus m_{i_2} \ . \tag{6.1}$$

Description.

Practically, we encrypted N times the same message M and collected the N traces corresponding to the first AES round. For each of the N traces we identified the 16 instants t_i corresponding to the beginning of the computation $S'(x_i \oplus u)$. This allowed us to extract 16 segments from each trace and construct the series Θ_i used for collision-correlation as explained in Section **??**.

Performing `Collision`$(\Theta_{i_1}, \Theta_{i_2})$ for all the 120 possible pairs (i_1, i_2) yields a set of relations $(i_1, i_2, m_{i_1} \oplus m_{i_2})$ given by Eq. (**??**). By repeating this process for several random messages M one can accumulate enough relations so that the secret key is recovered up to a guess on one key byte.

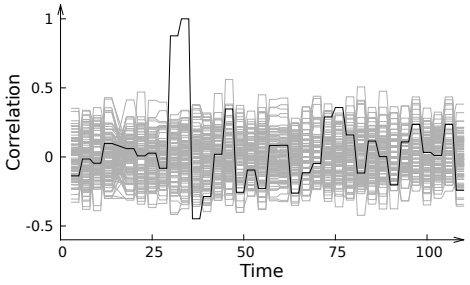


Figure 6.3: Correlation curves obtained for a message giving one collision (black curve)

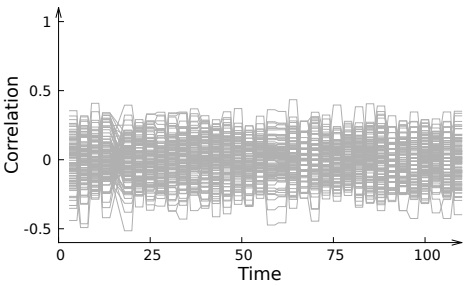


Figure 6.4: Correlation curves obtained for a message giving no collision

Based on 10 000 simulations we observed that on average 59 random messages (each one being encrypted N times) provide enough relations to retrieve the key up to an unknown byte.

Practical Results.

We present hereafter our results on both simulated and real curves.

On simulated curves. The threshold of `Collision` was fixed to having at least one point among the l points correlation curve equal to 1. Under this condition our attack was successful for $N = 16$. Since a mean of 59 different messages are required, then $16 \times 59 = 944$ traces are sufficient on the average for the attack to succeed on simulated curves.

Figures **??** and **??** show the correlation curves obtained for two different messages. Both figures present the 120 outputs of $\hat{\rho}_{\Theta_{i_1},\Theta_{i_2}}(t)$, $i_1 < i_2$ for each message. The black curve on Fig. **??** corresponds to a collision found for the first message, whereas the second message yields no collision.

On real curves. The attack was successful using $N = 25$ so that less than 1 500 traces allow to recover the key. Notice how few traces are needed to detect a collision by correlation. This confirms that the collision-correlation technique is much more efficient than classical model-based CPA which would not obtain high correlation levels with only 25 traces. Figure **??** shows an example of a correlation peak when an equality between two S-Box outputs occurs, while Fig. **??** shows the correlation curve when all S-Box outputs are different.

Note that in the case of real curves the threshold is slightly different. To identify a clear relation between two S-Box outputs the correlation curve must be greater than 0.8 in the interval $[130, 160]$. So only these $l = 30$ points must be considered when computing $\texttt{Collision}(\Theta_0, \Theta_1)$.

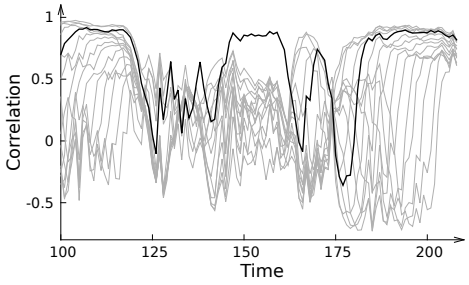


Figure 6.5: Correlation peak on real curves when a collision occurs (black curve)

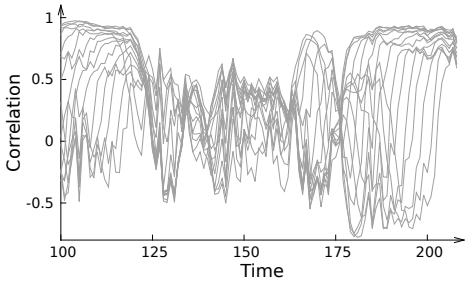


Figure 6.6: No correlation peak occurs on real curves when intermediate data differ

Attack Improvement.

The method for obtaining information about the key as described above basically exploits collision events where a pair (i_1, i_2) of indices gives a high correlation between Θ_{i_1} and Θ_{i_2} revealing the value of $k_{i_1} \oplus k_{i_2}$. While very informative, such collision events occur much less frequently than non-collision ones, that is when Θ_{i_1} and Θ_{i_2} show no significant correlation between each other. Non-collision events individually bring quite few information – namely that $k_{i_1} \oplus k_{i_2}$ is different from $m_{i_1} \oplus m_{i_2}$ – but they are so numerous that it appears worth trying to exploit them also.

As was already noted in [**?**, **?**], the problem of solving a set of equations involving sub-parts of the key can be formulated in terms of a labelled undirected graph. Each vertex i represents a key byte index and the knowledge of the XOR between two key bytes is represented by an edge (i_1, i_2) labelled with $k_{i_1} \oplus k_{i_2}$. At the beginning the graph does not include any edges. Each time a collision occurs between two unrelated key bytes a new edge is put on the graph and results in the merge of two connected components into a single larger one. All key byte values belonging to the same connected component can be derived from each other, and the goal of the attacker is to end up with a fully connected graph.

For a given message, only 0, 1, or 2 from the 120 pairs (i_1, i_2) lead to collisions in most cases. All other pairs reveal some impossible value for each $k_{i_1} \oplus k_{i_2}$. Gathering all the information provided by these non-collisions, for each (i_1, i_2) we maintain a blacklist of impossible values for the XOR of the two key bytes[3].

Given the information provided by previous messages to the current graph and blacklists, we adaptively choose the next message in order to maximize its usefulness which we define as the number of pairs (i_1, i_2) where one can expect new information (either positive or negative) to be obtained. As a first idea we could define the penalty of a candidate message as the number of pairs (i_1, i_2) for which $m_{i_1} \oplus m_{i_2}$ is already blacklisted. Obviously the chosen message should minimize the penalty. Actually this is slightly more complex and the definition of the penalty of a message should be refined. Indeed we must also consider cases where the message is useful for (i_1, i_2) and (i_1, i'_2) – that neither $m_{i_1} \oplus m_{i_2}$ nor $m_{i_1} \oplus m_{i'_2}$ are blacklisted – but the value of $k_{i_2} \oplus k_{i'_2}$ is known to be precisely equal to $m_{i_2} \oplus m_{i'_2}$. In such a case the two usefulness opportunities brought by

[3]Some of these blacklists must also be updated when two connected components are merged.

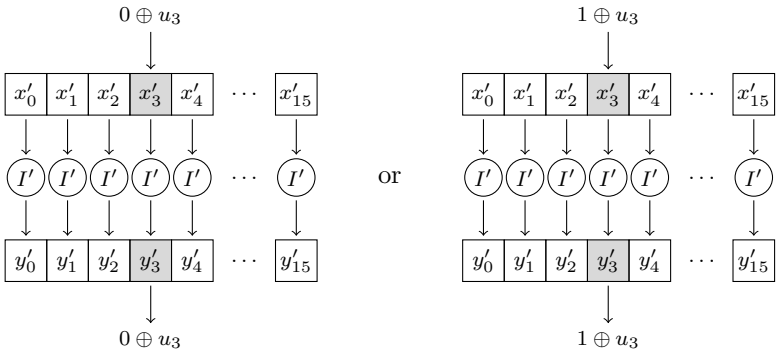


Figure 6.7: Collision between the input and the output on byte 3 of the blinded inversion I' (values 0 and 1 lead to a collision)

the message on pairs (i_1, i_2) and (i_1, i'_2) would bring the same information so that they should count for a single one and the penalty of that message must be increased by one.

In order to find a message with minimal penalty we devised a heuristic which works in two steps. In the first step we consider some random messages (say a few hundred) and select the one with the lowest penalty. This first step ends with a somewhat good candidate. Then in a second step we repeatedly attempt to decrease further the penalty by trying small modifications on this candidate until no more improvements occur by small modifications.

We simulated our method for adaptively choosing the messages. In these simulations we assumed that the attacker is always able to correctly distinguish between collision and non-collision events. Based on 1 000 simulations with random keys, we show that the key is fully recovered (up to the knowledge of one of its bytes) with as few as 27.5 messages instead of 59 messages with the basic method. As distinguishing between a collision and a non-collision necessitates only 25 traces per message, a mere 700 executions would suffice to recover the key by analysing real curves.

6.3.3 Attack on the Blinded Inversion Implementation

The previous attack cannot be applied to the blinded inversion implementation described in Section **??** since the different S-Box input and output bytes are masked with different values u_i. However there may exist a possible leakage leading to what we may call a *Zero & One value attack*.

One can notice that values 0 and 1 produce a collision between the input and the output of the masked pseudo-inversion stage I' as depicted on Fig. **??**. This is due to the following properties of the pseudo-inversion:

$$\begin{aligned} I(0) = 0 \quad &\Rightarrow \quad I'(0 \oplus u_i) = 0 \oplus u_i \\ I(1) = 1 \quad &\Rightarrow \quad I'(1 \oplus u_i) = 1 \oplus u_i \end{aligned}$$

The two cases leading to a collision are indistinguishable from one another. Detecting a collision between the input and the output of a blinded inversion gives either $x'_i = 0 \oplus u_i$

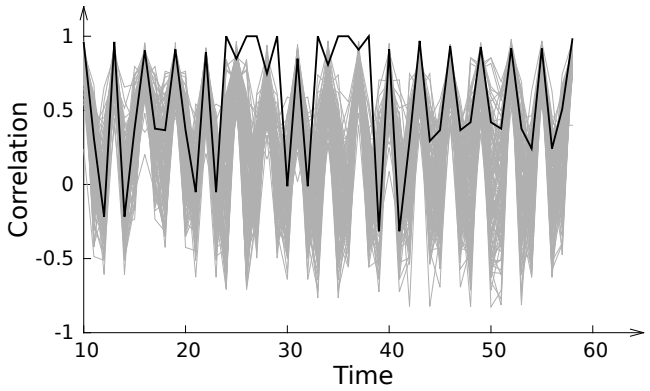


Figure 6.8: Collision-correlation curves in the pseudo-inversion of the first byte in GF(2^8)

or $x'_i = 1 \oplus u_i$ which reveals a key byte except one bit:

$$k_i = m_i \quad \text{or} \quad k_i = m_i \oplus 1 \ .$$

Description.

Assume we want to recover the 7 most significant bits of k_0. For every even byte value g we encrypt N times a single message M with $m_0 = g$ and collect the corresponding power consumption traces $T^{n,g}$, $0 \leqslant n \leqslant N-1$. Note that in this attack we only need to guess the 7 most significant bits because the least significant one is indistinguishable. Let's denote t_0 and t_1 the instants when $x_0 \oplus u_0$ is loaded before the pseudo-inversion I, and when the result is stored respectively. For each of the N traces we extract the two segments $T^{n,g}_{[t_0,t_0+l-1]}$ and $T^{n,g}_{[t_1,t_1+l-1]}$ and construct the series $\Theta^g_0 = (T^{n,g}_{[t_0,t_0+l-1]})_n$ and $\Theta^g_1 = (T^{n,g}_{[t_1,t_1+l-1]})_n$. For this step of our attack it is helpful to have some experience on the targeted implementation identify exactly where these two segments are located.

Applying the decision function $\texttt{Collision}(\Theta^g_0, \Theta^g_1)$ for all the 128 possible values g will reveal two possibilities for k_0. Repeating this step for all key bytes allows the key space to be reduced to 2^{16} values only. Note that a trick which allows to considerably reduce the number of traces is to encrypt the messages $M^g = (g, g, \ldots, g)$ with all bytes equal.

Results on Simulated Curves.

As for previous attack on simulated curves, a relation is established when at least one point among the l points correlation curve is equal to 1. The attack is successful using $N = 16$ curves for each key guess. Figure **??** shows the 128 correlation curves for all possible guesses on k_0. The black curve corresponds to the correct guess for k_0.

The attack on this second implementation has thus been validated on simulated curves. We did not acquire real curves for this implementation. Based on what has been observed on the previous attack (successful results obtained using simulations have led to successful results on the chip in practice), we believe that the attack would be successful

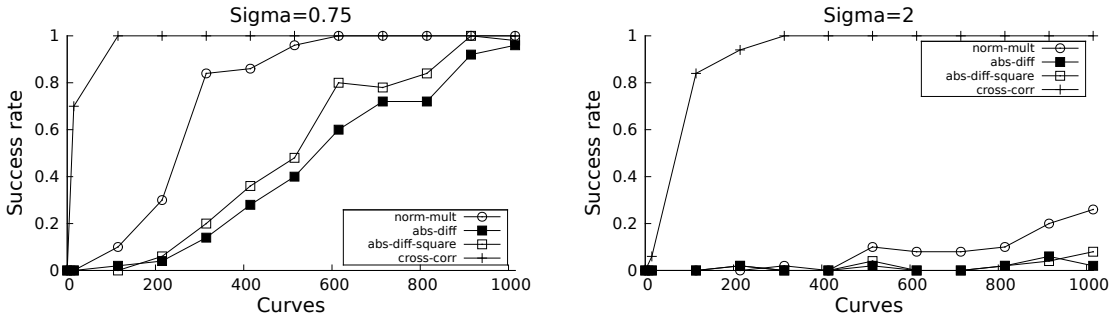


Figure 6.9: Success rates of different simulated second-order attacks

on the real chip too, using a value for N of the same order to what was necessary for the first attack.

6.4 Comparison with Second Order Analysis

In this section, we present a brief comparison between the collision-correlation method and some known second-order attacks. Our analysis was inspired from the recent framework introduced by Standaert et al. in [**?**] and refined later in [**?**]. This comparison gives an overview on the efficiency of these different second-order techniques, and highlights how much the collision-correlation analysis improves on second-order attacks.

Our analysis targets the first implementation only. We compared the collision-correlation analysis with the second-order analysis involving the absolute difference combining function f_1, the squared absolute difference combining function f_2 and the normalized product combining function f_3, when using as distinguisher the Pearson linear correlation factor $\hat{\rho}$. Note that we did not used Mutual Information Analysis, whose results remain less efficient than the classical CPA in practice.

For sake of simplicity, we consider that the power consumption at instant t is the Hamming weight of the intermediate data involved in the computation plus a centered Gaussian noise ω_σ with standard deviation σ. Therefore $\mathrm{H}_n(z)$ corresponds to the handling of the value z for the n-th encryption. We now define θ_0 and θ_1 as:

$$\theta_0 = (\mathrm{H}_n(S(m_i \oplus k_i \oplus u) \oplus v) + \omega_\sigma)_{0 \leqslant n \leqslant N-1}$$

$$\theta_1 = (\mathrm{H}_n(S(m_j \oplus k_j \oplus u) \oplus v) + \omega_\sigma)_{0 \leqslant n \leqslant N-1}$$

Let g_i (resp. g_j) denote a guess on k_i (resp. k_j). We compute the estimated values $w_{g_i,g_j} = \mathrm{H}(S(m_i \oplus g_i) \oplus S(m_j \oplus g_j))$. Considering the N messages we obtain the series $W_{g_i,g_j} = (w^n_{g_i,g_j})_{0 \leqslant n \leqslant N-1}$. Using the combining function f_j, the right key bytes are obtained for the highest correlation value $\hat{\rho}(f_j(\theta_0, \theta_1), W_{g_i,g_j})$.

Then as in [**?**] we execute many times the attack with the different combining functions and calculate the success rate of each one. Figure **??** shows two comparison graphs, one for $\sigma = 0.75$ and the other for $\sigma = 2$. Both graphs plot the success rates on 50 runs with respect to the number of curves used.

We emphasise that in this comparison the second-order attacks are shown in a very favorable light. Indeed the correlation model used here is exactly the one applied to simulate the curves. In practice an attacker would not have such good properties.

6.5 Countermeasures

The attacks presented in this paper defeat first-order protected implementations. Therefore, an obvious countermeasure would be to apply second-order masking. To the best of our knowledge, the best solution should be the countermeasure presented by Rivain et al. [**?**]. It allows the implementation of proven d-order DPA resistant AES for any $d \geq 1$.

Another countermeasure against our first attack may simply consist in executing the `SubBytes` function in a random order. Even if this method is not theoretically perfect, it may be sufficient to practically resist to second-order attacks. Considering the second implementation, we think that its main weakness is the use of a same mask before and after each byte pseudo-inversion. If the result is masked with a different value then the collision-correlation attack is no longer feasible.

It is also necessary to consider that depending on the quality of the hardware countermeasures provided by the device, these attacks can become much more complicated in practice.

6.6 Conclusion

We have presented a new collision-correlation analysis method on first-order secured AES implementations. We highlighted the fact that this kind of attack is more powerful and practicable than previous second-order power analyses, and increases the risk of these implementations being broken in practice. This confirms the necessity for developers to take into account how collisions of masked data may be unsafe in cryptographic implementations. A possible countermeasure could be the use of second (or higher) order resistant schemes.

Though we presented practical results on software implementations, we believe that this technique may also be a threat for hardware coprocessors. Therefore the collision-correlation threat should be taken into consideration by developers and designers during their embedded cryptographic design.

Chapter 7

Combined Attack on First Order Protected AES

7.1 Introduction

In this chapter we present another passive and active combined attack on a state of the art SCA protected AES [**?**]. We combine a particular fault attack technique named *Collision Fault Analysis* (CFA) that was introduced by Blömer and Seifert [**?**] in 2003, with the classic *Correlation side-channel analysis* (CSCA) introduced by Brier et al. [**?**] in 2004.

This chapter is organized as follows. Section **??** gives an overview of active and passive attacks with a focus on the collision fault analysis and the correlation side-channel analysis. We present in Section **??** the AES state of the art implementation chosen for this study and explain why this implementation is resistant to the previously published CFA. In Section **??** we introduce our combined attack and explain how, with the same fault model as the CFA, it can recover the secret key on our AES implementation. We discuss the countermeasures in Section **??**, describe a safe-error variant of our attack which defeats these countermeasures in Section **??**, and conclude the chapter in Section **??**.

7.2 Side Channel and Fault Analysis Background

Passive attacks consist in observing side-channel information, such as the power consumed by the chip while performing sensitive operations during a cryptographic computation. Active attacks consist in perturbing the device when it is processing sensitive data or calculations. Both techniques may result in the recovery of the secrets.

7.2.1 Side Channel Analysis

Most common countermeasures against power analysis, and particularly DPA and CPA, consist in using random values for masking the operations. In this case even if an attacker makes guesses on some secret key bits, he can not predict any intermediate value as another unknown variable, the random mask, is part of any intermediate data

during the computation. However in this case a more complex but realistic attack, named *High Order Differential Power Analysis* (HODPA), presented by Messerges [**?**], is still applicable if the mask values are identical on different bytes, and/or if some different instants on a same power curve can be used to eliminate the random mask effect.

7.2.2 Fault Analysis

Fault effects and perturbations on electronic devices were first observed in the 1970's in the aerospace industry. Later the Differential Fault Analysis, DFA for attacking embedded symmetric cryptosystems was introduced by Biham and Shamir [**?**] in 1997. In this paper the authors explain how to recover the secret key by using between 50 and 200 ciphertexts. For years this threat was considered as only theoretical until the first practical results of light attacks were presented (on an RSA implementation) by Anderson and Skorobogatov [**?**]. The DFA has subsequently been studied and applied on DES in [**?**] where Giraud and Thiebeauld recover the key by means of only 2 faulty ciphertexts. In the case of the AES many attacks have been proposed [**?**, **?**, **?**] that allow the secret key to be recovered by using as few as 2 faulty ciphertexts.

We now present the Collision Fault Analysis technique which is the active component used in our combined attack.

Collision Fault Analysis

In [**?**] Blömer and Seifert first published a CFA on the first XOR of the AES. They assume a fault model where the attacker has the ability to force to zero any chosen bit of the result of this XOR operation. Then they compare a correct and a faulted AES execution for the same message. If both ciphertexts are equal the original value of the result bit is 0, otherwise it is 1. Knowing the message and scanning the different key bits, the whole 128-bit AES key is retrieved with 128 faulty executions. An interesting property is that the classical countermeasures which consist in checking the computation, for instance by executing the AES twice and comparing the results, do not prevent this attack. Indeed whether the card detects the fault or not will provide the attacker with the same information as whether or not the fault corrupted the ciphertext.

Later Hemme [**?**] presented the first CFA on the DES. His attack consists in introducing one bit errors in the first rounds of the algorithm. Then by computing chosen message encryption with the card (without injecting faults) the attacker obtains collisions that he can exploit to recover information on the secret key. With enough collisions he can recover the whole secret key. In this case, verifying the whole DES computation is an efficient countermeasure.

Another CFA analysis on the AES first XOR computation can be done when the fault effect resets a whole byte (or many bytes) instead of a bit. In this attack an induced fault resets the result of a XOR between one message byte M_j and one key byte K_j – with the other key addition byte results not being affected. The attacker stores the faulty ciphertext C' and asks the card to encrypt the 256 messages M with M_j taking all possible byte values. One of these 256 ciphertexts will be equal to C'. This collision is produced for M_j verifying $M_j \oplus K_j = 0$, which indicates that $K_j = M_j$.

Amiel et al. [**?**] adapted this CFA to an AES protected from first order DPA by random masking. In this implementation the same random byte r_1 is used for masking all 16 message bytes and the same random byte r_2 is applied on all 16 key bytes. In that case a single random byte $r = r_1 \oplus r_2$ is applied on all the bytes of intermediate values throughout the computation. The authors succeeded in faulting 2 to 16 bytes of the result of the first XOR. Then by searching collisions they obtained relations between known input bytes and key bytes masked. Exploiting these relations allows them to recover the secret key. This attack needs the precomputation with the card of 2^{23} non faulty ciphertexts and in practice 112 faulty ciphertexts were used. Note that this attack is applicable only if the same random mask r is applied on all of the 16 bytes of the intermediate values. The targeted implementation was not protected against high order differential analysis and in particular against a second order analysis. State of the art implementations are thus not vulnerable to this CFA.

7.3 Targeted AES Implementation

We present here the implementation targeted by our attack. We have chosen a state of the art side channel resistant AES implementation. To prevent DPA and CPA attacks, a 16-byte random mask is used to mask the input message (and another one to mask the key). This random mask is composed of 16 different random bytes that can change at each round. This targeted implementation is designed to resist to the HODPA attack presented in [**?**] and [**?**].

To realize such an implementation it is not possible to use a 256-byte substitution table as randomizing this substitution table for each random byte $r_0, \ldots, r_{15}$ would necessitate precomputing and storing 16×256-byte substitution tables, one for each byte r_i. Moreover these tables would need to be recomputed at each round for changing the mask between each round. The chosen implementation is the one presented by Oswald et al. in [**?**], the inversion is here computed masked in $GF(2^4)$. In this case all the 16 bytes of the message and the intermediate calculations are masked with different random bytes. This implementation is described in Fig. **??**

Note that, as previously stated, the CFA presented by Amiel et al. is not applicable on our targeted implementation.

We have carried out two implementations of a secure AES on an 8-bit microprocessor with different security levels. The first one is resistant to DPA attacks and takes 20 000 cycles (2 ms at 10 MHz). Data are masked by the same byte which requires precomputing only one substitution table for one AES execution. This implementation is not resistant to HODPA attacks and is also vulnerable to Amiel et al. CFA. We also carried out the implementation described above which uses inversion in $GF(2^4)$. All data are masked by different bytes which change between each round. This implementation is resistant to HODPA attacks and takes 51 000 cycles (5.1 ms at 10 MHz). We will refer to both these implementations as AES_{DPA} and $\text{AES}_{\text{HODPA}}$ respectively. The performance and memory footprint figures for both implementations are presented in Table **??**. We introduce the AES_{DPA} implementation here only for comparison purposes to illustrate the cost implied for protecting an AES from HODPA. Only the $\text{AES}_{\text{HODPA}}$ implementation will

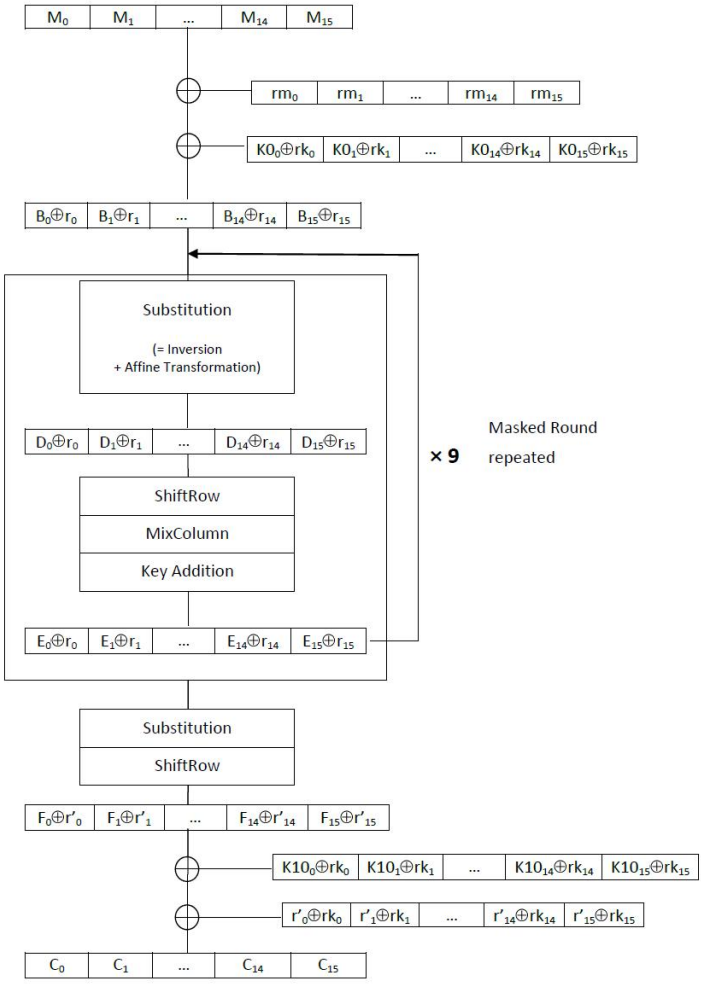


Figure 7.1: Secure HODPA Implementation

Table 7.1: Performance and memory costs for AES_{DPA} and $\text{AES}_{\text{HODPA}}$ implementations

	Cycles	ROM	RAM
AES_{DPA}	20 000	4 000 bytes	256 + 65 bytes
$\text{AES}_{\text{HODPA}}$	51 000	5 500 bytes	75 bytes

be referred to in our attacks.

7.4 Passive and Active Combined Attack on Masked AES

In this section we present an analysis that can be carried out on our $\text{AES}_{\text{HODPA}}$ implementation with the same fault model as in previous publications. Our proposed attack targets the first round calculations of the AES and necessitates choosing the input message and obtaining the ciphertext computed by the card. Compared to previous CFA on masked AES, it does not require a large number of messages to be encrypted by the card.

Notation: We denote by $M = (M_0, \ldots, M_{15})$ the input message and by $K = (K_0, \ldots, K_{15})$ the key used by the card for encrypting M. Given a message M, we also denote by $(M|\,\text{condition})$ the message M modified so that the condition holds. For instance, messages $(M|M_j = 0)$ and $(M|M_j = 1)$ are identical except on the j^{th} byte which is 0 in first case and 1 in the other. We will also refer to a *faulty* computation or a result thereof by means of the superscript symbol $^{\lightning}$. For instance $C^{\lightning}$ will refer to a faulty ciphertext.

7.4.1 Fault Model

We consider the following fault model: the result of an operation XOR can be set to zero – or to a not necessarily known constant value – by the attacker. This model has previously been assumed in several fault analysis papers and can be considered as realistic since practical results were also presented.

In practice such kind of fault effect can be induced in a card by the following events:

- An operation can be bypassed. For instance the instruction to be executed is replaced by a NOP. As explained in [**?**] the opcode fetched by the microprocessor may be replaced by 0x00 that, in some products, corresponds to the NOP operation. In this case the expected computation is not done, and the result register keeps its previous value that may be either 0 or another constant value. If the value is not constant the attack will be not possible.

- A loop counter can be modified, for instance in [**?**] the AddRoundKey operation is bypassed on some bytes by modifying (reducing) a counter value.

- The processing in the ALU can be perturbed and an XOR result can thus be modified to zero or a constant value.

Figure **??** gives an example of a typical code on an 8-bit microprocessor which can be attacked by what has been presented above. It represents the XOR operation between the masked message and the masked key carried out at the beginning of a secure AES. We can observe in this code that our fault model can be used to model different effects, if a NOP operation is executed, if a XOR result is forced to 0; if the counter value R2 is modified or if the JNZ operation is perturbed, etc...

```
Addition:
  MOV  R0, #address_message_masked
  MOV  R1, #address_key_masked
  MOV  R2, #16

LoopMessageXorKey:
  MOV  A, @R0
  MOV  B, @R1
  XOR  A, B
  MOV  @R0, A
  INC  R0
  INC  R1
  DEC  R2
  JNZ  LoopMessageXorKey
```

Figure 7.2: Example code of masked AES AddRoundKey

7.4.2 Attack on the First Key Addition

As in [**?**] we assume that the key addition before the first round can be perturbed and one or many bytes resulting from this addition can be set to zero. We describe our analysis for the first bytes M_0 and K_0 of the message and the key. The analysis will be identical for the other byte indices.

We denote by $rm = (rm_0, \ldots, rm_{15})$ and by $rk = (rk_0, \ldots, rk_{15})$ the two 16-byte random masks on the message and the key respectively. The resulting mask of the XOR between the message and the key is denoted by $r = rm \oplus rk$.

For a normal execution the first byte of the XOR result is:

$$B_0 = (M_0 \oplus rm_0) \oplus (K_0 \oplus rk_0) = M_0 \oplus K_0 \oplus r_0$$

For a faulty execution we have:

$$B_0^{\nmid} = 0$$

The key observation is that the effect of the fault is to introduce a differential δ on the byte value just before the first round S-Box computation.

$$B_0^{\nmid} = B_0 \oplus \delta, \qquad \text{with} \quad \delta = M_0 \oplus K_0 \oplus r_0$$

The same effect on the execution would have been obtained, without a fault, when using an input message which differs from the initial one by this differential.

Considering without loss of generality an initial message $M = (0, \ldots, 0)$, the differential then reduces to $\delta = K_0 \oplus r_0$, and we have a collision opportunity corresponding to the following equation:

$$C^{\lightning} = AES^{\lightning}(M) = AES(M|M_0 = \delta)$$

Note that a normal execution with $(M|M_0 = \delta)$ will produce a collision with $C^{\lightning}$ whatever the random mask values for this execution.

For any ciphertext $C^{\lightning}$ obtained by injecting a fault we have the following properties:

1. $C^{\lightning}$ is characteristic of the mask value r_0 in the faulty execution.
2. We can recover the value $K_0 \oplus r_0$ of this execution by identifying which input message leads to a collision with $C^{\lightning}$ without fault.

By computing $AES(M|M_0 = u)$ for all $u = 0, \ldots, 255$, we can identify the u value which verifies the relation:

$$u = \delta = K_0 \oplus r_0$$

At this point we are able, for any faulty execution, to recover the value of $\delta = K_0 \oplus r_0$ involved in that execution. It is then possible to reproduce this analysis many times to obtain k ($k \leqslant 256$) such relations for k different values δ_i, $i = 1, \ldots, k$, and store the power consumption curve W_i of the faulty execution corresponding to each δ_i.

Now, observe the following property: for any possible guess g about K_0, $g = 0, \ldots, 255$, we obtain a unique set $S_g = \{r_{0,1}, \ldots, r_{0,k}\}$ of k random mask values. In the HODPA resistant implementation these random values are generated and manipulated in the card at different moments during the inversion in $GF(2^4)$ and during the MixColumn computation applied to the mask in the first round. It is then possible to correlate these random values with the power curves W_i. By computing the linear correlation factor between the set of curves $\{W_1, \ldots, W_k\}$ and the set of random masks $\{r_{0,1}, \ldots, r_{0,k}\}$, the most important correlation peak over the different guesses identifies the correct set of random values manipulated in the card and thus indicates that the corresponding guess g is equal to the secret key byte K_0.

The analysis can be repeated on the next bytes of the key addition operation to recover the other key bytes K_1 to K_{15}.

We summarize the different steps of the attack in Figure **??**. Note that in phase 2 of the attack, the expected number of faulty executions needed to obtain a new informative δ_i grows constantly with i. The expected number of faults N_k required to gather k relations is equal to

$$N_k = \sum_{i=1}^{k} \frac{256}{256 - (i-1)}$$

so that an average of 126 faults generates 100 relations, while the complete set of 256 relations requires $N_{256} = 1\,568$ faults.

Phase 1: dictionnary precomputation

$M = (M_0, \ldots, M_{15}) \leftarrow (0, \ldots, 0)$
for $u = 0$ to 255 **do**
    $C_u \leftarrow AES(M|M_n = u)$

Phase 2: collision search

$\Gamma = \varnothing$
$i \leftarrow 1$
while $(i < k)$ **do**
    $C^{\lightning} = AES^{\lightning}(M)$
    if $C^{\lightning} \notin \Gamma$ **do**
        $\delta_i \leftarrow u$ such that $C^{\lightning} = C_u$ with $u \in \{0, \ldots, 255\}$
        $W_i \leftarrow$ power curve of the faulted execution
        $\Gamma \leftarrow \Gamma \cup \{C^{\lightning}\}$
        $i \leftarrow i + 1$

Phase 3: correlation

for $g = 0$ to 255 **do**
    for $i = 1$ to k **do**
        $r_{n,i} \leftarrow \delta_i \oplus g$
    $\rho_g \leftarrow$ correlation trace between $\{r_{n,1}, \ldots, r_{n,k}\}$ and $\{W_1, \ldots, W_k\}$
$K_n \leftarrow g$ which gives the highest correlation peak

Figure 7.3: Attack algorithm on key byte K_n

Remark: Our attack is also applicable when the fault effect does not result in a 0 value but in an unknown constant c. Instead of recovering the 16-byte key K we recover $K \oplus (c, \ldots, c)$. Then we just have to exhaust all 256 keys until a key matching some correct plaintext/ciphertext pair is found.

7.5 Countermeasures

While data randomization with a full mask (i.e. 16 different bytes) is enough to protect the AES algorithm against the Amiel et al. attack as well as high order differential analysis, it is not sufficient against our combined attack. Below we mention some possible countermeasures.

7.5.1 Inverse computation

A classical and efficient countermeasure used to protect cryptographic algorithms against fault attacks in smartcards is the verification of the computation done. Before returning the ciphertext the card performs the inverse computation on the result. If the value

obtained corresponds to the input message then the computation is considered valid and the card can return the result.
However if the comparison is not successful, it means that a fault was generated during one of the two cryptographic computations. In this case, nothing is returned by the card. As our attack requires faulty ciphertexts, it is not applicable when the inverse computation countermeasure is implemented.

7.5.2 Duplicated Rounds

An alternative to the previous countermeasure consists in duplicating the execution of the rounds exposed to the attacks, for instance the first and the last rounds. The rounds which are duplicated must be performed with two different masks. Their executions are carried out together, and bytes are processed in a random order regardless of their masks. At the end of the round the following property must hold: the addition of the two results must be equal to the addition of the masks. If this property does not hold then a fault is detected. In order to protect against DFA, it is recommended to duplicate the first three and the last three rounds.

7.5.3 Data error

Another way to protect an algorithm may be the deliberate introduction of data errors appearing under some kinds of sequence flow disruptions. This notion of infective countermeasure can be applied to loop counters, round counters,... Some infectious data are supposed to be equal to zero or to a fixed value in a normal execution, but when a fault modifies a loop counter (for instance during the AddRoundKey operation in AES) these values become erroneous when they are XORed with this counter.

7.5.4 Checksums

Data used at the beginning of a cryptographic algorithm (message, key, mask, ...) may be associated with a checksum. After executing sensitive operations a checksum on the obtained data is computed and compared to these values stored in memory. A comparison error implies that a fault occurred during this part of the algorithm. These checksums can be carried out using hardware mechanisms.

We have implemented the first two countermeasures on the HODPA secure algorithm described in Section **??**. Their performances and memory costs are presented in the Table **??**.

7.6 Passive and Active Combined Attack on Masked AES with Safe Errors

Here we present a variant of our combined attack which is not precisely based on collisions but rather on safe errors, also known as ineffective faults.

In this variant, the way to obtain the knowledge of $K_n \oplus r_n$ for some faulty execution differs from the attack described in Section **??**. Instead of comparing a faulty ciphertext

Table 7.2: Performance and memory costs (bytes) for two FA resistant implementations

	Cycles	ROM	RAM
AES_{HODPA} with Inverse Computation	102 000	5 500	75
AES_{HODPA} with 6 Duplicated Rounds	81 000	5 900	91

$C^{\lightning} = AES^{\lightning}(M|M_n = 0)$ with a pool of normal ciphertexts $\{C_u = AES(M|M_n = u)\}$, the attacker repeatedly compares some normal ciphertext $C_u = AES(M|M_n = u)$ with a faulty one $C_u^{\lightning} = AES^{\lightning}(M|M_n = u)$ obtained with the *same* input, until both ciphertexts collide. Once this collision occurs the attacker knows that $K_n \oplus r_n = u$.

At first sight this variant may seem irrelevant since the distributions of the two random variables $C_u^{\lightning}$ and $C_0^{\lightning}$ are the same. Also this variant requires 256 faulty executions on average to obtain one single collision, while the previously described attack requires only one.

The great advantage of the safe errors variant becomes clear when the HODPA resistant implementation is also protected against CFA, either by means of the *inverse computation* countermeasure or by means of the *duplicated rounds* one. In both cases the attacker identifies the collision event each time a result is returned. Indeed the result is returned whenever the fault has not been detected, that is whenever it was safe and had no local effect on the XOR result. Note that it is not possible to distinguish a *safe error* event from a *no fault at all* event. Consequently the safe errors variant may be difficult to perform in practice if the fault injection tool is not highly reliable.

An interesting and unexpected property of the safe errors variant is that it is easier to perform when the computation checking countermeasures are implemented than when they are not. Indeed when either of these countermeasures is present the attack consists in a known message attack, otherwise it is a chosen message attack.

7.6.1 Countermeasures

The countermeasures presented in Section **??** no longer work here as no data has been modified. Then the question is how can we prevent attacks which do not modify data processed by the card? It seems to be an open problem and we do not have any good response. However we can mention several mechanisms which can complicate the attacker's task.

We have not yet addressed hardware mechanisms. As stated previously by Blömer and Seifert [**?**] we must insist on the fact that efficient hardware mechanisms to detect or resist light injections help software implementations and help to prevent many fault

attacks.

The randomization (time or order) during execution is also a good means to destabilize the attacker as he does not exactly know where the targeted data is manipulated.

Due to the somewhat large number of fault injections required to perform the safe errors variant, an efficient means to defeat the attacker could be to limit the number of possible faulty executions. If more than a specified number of faults is detected the card can kill itself or at least refuse to answer except under privileged conditions. Note however that this principle may be difficult to implement in practice depending on the card operating environment and requirements.

The best solution will be to mask the key with a value which is never manipulated during the processing. It is then not possible to correlate power curves with the mask and the only data that an attacker can find would is the masked key.

7.7 Conclusion

Sound countermeasures are known for protecting embedded cryptographic implementations from either high order side-channel analysis or differential and collision fault analysis.

In this chapter we have shown that simply putting together different kinds of countermeasures may not be sufficient. We have presented a Combined Active – CFA – and Passive – CPA – Attack (PACA) which breaks a proposed state of the art side-channel resistant AES implementation with a limited number of faults. We have enumerated some possible countermeasures, but remark that a safe errors variant of our attack can defeat most of them, such as executing and comparing twice a HODPA resistant implementation – though it requires a significant number of fault injections and a highly reliable fault injection tool. Although we have given some hints about how our last attack may be rendered more difficult, it seems to be an open problem how to protect implementations from ineffective faults, which are informative even though they do not alter the computations.

Chapter 8

Efficient Provable Prime Number Generation for Embedded Devices

8.1 Introduction

Nowadays, due to continuous improvements of technology and computing capabilities, larger and larger prime numbers are needed to operate RSA in embedded security devices. Large prime numbers are a basic ingredient of keys in several other standardized primitives such as Digital Signature Algorithm (DSA) [**?**] or Diffie-Hellman key exchange (DH) [**?**]. This chapter precisely addresses the generation of provable prime numbers in embedded, crypto-enabled devices.

When it comes to RSA key generation, two approaches coexist: key pairs may be generated off-board (i.e. out of the device) in a secure environment such as a certified Hardware Security Module (HSM) running in a personalization center, and loaded into devices afterwards. Key pairs may also be generated on-board, that is, by the device itself. In this case the private key cannot be compromised as it is never transmitted to the outside world. This capability also allows the device to generate new keys later on, when deployed in the field. However it implies that the device must be able to generate large primes very efficiently and in a side-channel-secure manner.

Surprisingly enough, in spite of a quite abundant literature on primality testing and on the validation of provable primes, research works that specifically suggest generators for embedded devices are pretty inexistant. Commonly found prime number generators rely on primality (pseudo-)tests to provide a high level of confidence that the output number is prime. It is widely known that this confidence level can be increased arbitrarily by applying sufficiently many iterations of the Miller-Rabin test [**?**]. It is even conjectured that the error probability – the probability under which the tested number is actually composite – is eliminated when a Lucas pseudo-test confirms Miller-Rabin testing [**?**]. This, however, may not be satisfactory because there is no absolute certainty that the generated number is prime. Technical requirements for the generation of prime numbers well-suited for RSA, DSA and ECDSA are described in industry standards such as FIPS 186-3 [**?**]. To ensure compliance, generating a 1024-bit DSA prime number requires as many as 40 Miller-Rabin iterations, which can be reduced to 3 when performing an

additional Lucas test. However carrying out a Lucas test is more costly on an embedded device than a single modular exponentiation, and thus leads to a performance loss. This chapter investigates another approach, namely the application of constructive techniques to achieve truly provable primality.

In this chapter, we introduce two efficient methods for generating provable primes and present fast implementations of these methods on a popular smartcard cryptoprocessor. Our methods rely on Pocklington's theorem and an extended result due to Brillhart, Lehmer and Selfridge. We establish bounds on the entropy of the output distribution of each method and provide evidence that both of them are secure and can be used for cryptographic purposes. Performance measurements are given that demonstrate the efficiency of our algorithms and how they compare with probable prime generation. We also suggest a number of countermeasures against state-of-the-art side-channel and fault-based analysis to ensure security in an untrusted environment.

Roadmap. Section **??** describes the constraints faced by cryptographic implementations on embedded devices and related security aspects. Section **??** recalls the usual methods for primality testing, where we distinguish between probabilistic and true tests. Generation algorithms for provable primes are discussed in Section **??**, where we introduce our two efficient constructive methods. The security of these methods in terms of output entropy is discussed in Section **??**. Practical results are reported in Section **??** together with performance comparisons for smartcard implementations of our probable prime and provable prime generators. Section **??** addresses threats arising from side-channel attacks and shows how to adapt our algorithms to resist these. We conclude in Section **??**.

8.2 Cryptographic Implementations on Embedded Devices

One faces two critical issues when designing an embedded cryptographic library: efficiency and tamper resistance. A reference architecture for embedded devices is found in smartcards: although microcircuits have been subject to significant improvements over the last decades (increase of memory sizes and clock frequencies, technology shrink), developing cryptographic algorithms and more generally any kind of embedded software in a smartcard remains a technical challenge. In the mid-90's, chip manufacturers have started embedding coprocessors in their CPU cores to cope with cumbersome cryptographic operations and achieve user-friendly execution times. Cryptoprocessors are now a must-have feature of most smartcards and provide efficient support for symmetric and asymmetric primitives such as (T)DES, AES, RSA and finite field operations for elliptic curve-based schemes. Public-key accelerators all support modular multiplications in hardware – and often modular additions and subtractions as well – with a considerable speed-up factor compared to what can be done with the CPU alone. Complete algorithms such as RSA or ECDSA are then developed in software on top of the hardware-accelerated operations made available on the chip. Most commonly used arithmetic architectures in these accelerators are based on modular reduction methods due to Barrett, Montgomery, Quisquater or Sedlak. More details on these techniques can be found in [**?**, **?**, **?**].

However, public-key cryptoprocessors often provide little more than just native modular arithmetic. This lack of flexibility can sometimes be dramatic for innovative and alternate programming, as practitioners are left with no other choice than using the (much slower) CPU for performing the desired computations. Typically, right-shifting a large integer by 3 bit positions may be slower than applying a modular operation on it. In such an uncommon computational model, non-modular operations are prohibitive; to come up with an efficient implementation of a non-mainstream algorithm, one must face the challenge of completely rethinking that algorithm in terms of modular arithmetic.

In addition to that, it is also mandatory to protect cryptographic functions against side-channel analysis and fault attacks to defeat cryptanalytic attacks based on information leakage. Since the introduction of the original SPA and DPA attacks by Kocher et al. [**?**], the diversity of attacks to protect embedded devices from has increased considerably [**?**, **?**, **?**, **?**, **?**]. Many attacks have appeared on embedded implementations of RSA and ECC with practical results [**?**, **?**, **?**, **?**, **?**] that are of particular interest in the context of our work. We come back to this issue later in this chapter.

8.3 Prime Number Generation based on Primality Testing

In the broadest possible sense, a primality test $\top$ is a procedure that outputs a guess $\top(n) \in \{\mathsf{true}, \mathsf{false}\}$ as to whether a positive integer n is prime or composite. It can be a pseudo-primality test (also called compositeness test), in which case the guess can be a false positive with some probability, or a true primality test that never fails and provides a proof for primality when positively answered. Once one is given some primality test $\top$, it is natural to derive Algorithm **??** which provides a generic method for generating prime numbers.

Alg. 8.3.1 Generic Prime Number Generation

Input: a primality test $\top$, a constraining property $\mathcal{P}$
Output: a prime integer n

1. generate a random candidate n verifying property $\mathcal{P}$
2. **while** $\top(n) =$ **false do**
3. update n while preserving property $\mathcal{P}$
4. **return** n

Following the naming of Brandt and Damgård [**?**], we refer to the list of tested candidates as the *search sequence*. In the generic prime number generator, each candidate along the search sequence is required to verify some property $\mathcal{P}$. The purpose of this requirement is to reduce the average number of calls to $\top$, which is assumed to be the most time-consuming subroutine of the algorithm, by avoiding candidates known to be composite.

Without this requirement – or equivalently, when $\mathcal{P}$ is satisfied for any n – the average number of calls to $\top$ when generating an ℓ-bit prime is close to $\ln(2^\ell)$. An obvious improvement is to let $\mathcal{P}$ be the property that n is odd and proceed to updating a candidate by adding 2 to it. In that case the average number of calls to $\top$ drops

to $\ln(2^\ell)/2$. A straightforward generalization of this idea is to take for $\mathcal{P}$ the property that n is relatively prime with the t smallest primes $p_1, \ldots, p_t$. The first candidate in the search sequence thus requires the generation of an invertible element modulo $\Pi = \prod_{i=1}^{t} p_i$, which can be done either with trial divisions by each of $p_1, \ldots, p_t$, using Chinese remaindering (*e.g.* Garner [**?**] or Gauss algorithms), or using a technique due to [**?**] based on Carmichael's theorem. Several methods can be applied to update n while preserving $\gcd(n, \Pi) = 1$; Π can simply be added to n, or one can keep track of an array of indicators $\omega_i = n \bmod p_i$ for $i = 1, \ldots, t$ and modular-add 2 to all of those until none is equal to zero. Alternately, an efficient method for preserving $\gcd(n, \Pi) = 1$ for maximally large Π is found in Joye et al. [**?**, **?**]. Overall, the techniques described in [**?**, **?**] provide the most efficient approach on a cryptoprocessor as they generate an invertible element modulo Π faster than the classical trial division method. Irrespective of the chosen methods to implement the different subroutines of Algorithm **??**, the average number of calls to $\top$ is close to

$$N(\ell, \Pi) = \ln(2^\ell) \cdot \frac{\phi(\Pi)}{\Pi}$$

where ϕ is Euler's function. The optimal choice therefore consists in taking the largest possible prime product $\Pi = p_1 \cdot \cdots \cdot p_t$. With $\ell = 512$ for instance, the average number of performed tests is 35.6 with $t = 54$ ($p_t = 251$) instead of 177 tests when the candidates are only required to be odd ($t = 1$). While $N(\ell, \Pi)$ obviously further decreases with larger t, the relative gain rapidly decreases as well as Π becomes larger.

8.3.1 Pseudo-Primality Tests

Pseudo-primality tests may erroneously view a composite number as being prime. Among these, Fermat and Miller-Rabin tests are the most commonly used in embedded applications as they are particularly fast and easy to implement.

The random-base Miller-Rabin test has an error probability $\varepsilon < 1/4$. By iterating this test h times with different random bases this probability is (often quite loosely) upper bounded by $1/4^h$. Practitioners choose the number h of iterations depending on the bitsize of the tested number, the cryptosystem intended to make use of the generated prime, and the specific security requirements imposed by industry standards. Referring to FIPS 186-3, a 1024-bit prime to be used as a DSA parameter requires 40 Miller-Rabin tests (or 3 Miller-Rabin tests followed by a Lucas test). For a 2048-bit RSA key, each 1024-bit prime must pass 4 Miller-Rabin tests, and although applying the Lucas test is not required, it is highly recommended.

The random-base Fermat test has approximately the same efficiency as the random-base Miller-Rabin test while its error probability is higher. However, it is more simple to implement and leads to optimally efficient pseudo-testing when using a base fixed to 2: modular multiplications by 2 can then be replaced with modular additions in the modular exponentiation $2^{n-1} \bmod n$. Fermat testing is usually performed first with $a = 2$, and only when n passes the Fermat test, does it undergo several Miller-Rabin rounds with random bases before being considered to be prime. This leads to the efficient prime number generator referred to as Algorithm **??**, where $\mathsf{F}_a(n)$ and $\mathsf{MR}_a(n)$ respectively denote Fermat and Miller-Rabin tests with base a.

Alg. 8.3.2 Efficient Generation of Probable Primes

Input: a bitsize ℓ, $\Pi = 2 \cdot 3 \cdot 5 \cdot \ldots \cdot p_t$, a confidence parameter h
Output: an ℓ-bit probable prime n

1. generate a random ℓ-bit integer n with $\gcd(n, \Pi) = 1$ and go to **??**
2. update n such that $\gcd(n, \Pi) = 1$
3. if $\mathsf{F}_2(n) =$ **false** then go to **??**
4. **for** $i = 1$ **to** h **do**
5. pick a base a at random from $[2, n-2]$
6. if $\mathsf{MR}_a(n) =$ **false** then go to **??**
7. **return** n

Neglecting the probability that the output prime is a Fermat or a strong pseudoprime, and denoting respectively by T_i, T_u, T_{F_2} and $T_{\mathsf{MR_a}}$ the execution times of the routines for generating the first candidate, updating the current candidate and performing Fermat and Miller-Rabin tests, the average total execution time to generate a probable l-bit prime amounts to

$$T_{\text{probable}}(\ell) = T_i(\ell) - T_u(\ell) + N(\ell, \Pi) \cdot (T_u(\ell) + T_{\mathsf{F}_2}(\ell)) + h \cdot T_{\mathsf{MR_a}}(\ell) \; . \tag{8.1}$$

This generation method is among the most popular ones in use in the embedded security industry at the present time. Section **??** reports practical performance figures for a typical smartcard implementation of this generator.

8.3.2 True Primality Tests

Prime number generators make use of pseudo-primality tests because of their efficiency. However, to fully eliminate the error probability ε, one has to rely on true primality testing a.k.a. primality proving. The asymptotically fastest true primality test is the AKS method [**?**], which is the only known algorithm that runs in polynomial time. However, the preferred general-purpose method for testing large numbers is currently the Elliptic Curve Primality Proving test [**?**] which was used to ascertain the primality of the largest general number, a prime with more than 20′000 decimal digits. Unfortunately the AKS and ECPP methods are way too complex to be of any interest for embedded implementations, where algorithms are preferably based on simple arithmetic operations such as modular exponentiations.

A possible step in this direction relates to a deterministic variant of the Miller-Rabin criterion. Following a result from Ankeny [**?**], Bach [**?**] proved under the Extended Riemann Hypothesis (ERH) that any composite number n has a strong witness[1] upper bounded by $2 \log^2 n$. Thus, verifying that n passes Miller-Rabin testing for all bases smaller than $2 \ln^2 n$ would actually prove that n is prime. The drawback of this approach is the fairly large amount of bases to consider before making sure that n is prime. Proving the primality of a 512-bit number would require more than 250′000 Miller-Rabin rounds. A secondary drawback is that the primality proof only holds under ERH.

[1] A *strong witness* for a composite number n is an integer a such that n does not pass the Miller-Rabin test with base a, thereby proving its compositeness.

Instead of relying on the existence of a small witness, it may be better to rely on the existence of a small set containing at least one witness. Given an upper bound x on candidates, a *reliable set* of witnesses is a set $\mathcal{W}$ such that every odd composite integer $n \leqslant x$ has a witness in $\mathcal{W}$. An interesting result from Alford et al. [**?**] unconditionally proves the existence of a reliable set containing at most $(6/5)\log x$ integers smaller than x. This result does not rely on any conjecture and proves that n is prime with much fewer Miller-Rabin rounds (only 426 rounds for 512-bit numbers). Unfortunately the constructive method put forward by the authors for identifying such a reliable set does not seem to be computationally practical.

8.4 Constructive Generation of Provable Primes

As previously discussed, there does not seem to be any practical true primality test that would suit our context. Rather than testing the true primality of candidates along a search sequence, we revisit Maurer's approach [**?**] wherein provable primes are generated in a *constructive* manner using Pocklington's criterion:

Theorem 8.4.1 (Pocklington's theorem) *Let $n > 3$ be an odd integer, and let $n = rF + 1$ where the factorization of F is known as $F = \prod_{j=1}^{s} q_j^{e_j}$. If there exists an integer a such that*

(i) $a^{n-1} \equiv 1 \pmod{n}$ *and*

(ii) $\gcd(a^{(n-1)/q_j} - 1, n) = 1$ *for each* $j = 1 \ldots s$,

then every prime divisor p of n is congruent to 1 modulo F. In particular, if $F > \sqrt{n} - 1$ then n is prime.

As opposed to Fermat and Miller-Rabin's theorems, Pocklington's theorem isolates sufficient conditions for true primality. Unfortunately it cannot be used to test any given integer since the factorization of $n-1$ must be partially known. Based on Pocklington's theorem, Maurer [**?**] suggested a constructive method for generating provable primes. The main idea there is to construct a prime n such that $n-1$ is divisible by one or more smaller primes. A recursive use of the criterion then allows to generate larger primes at each round starting from small integers whose primality proof is trivial.

Theorem 8.4.2 *Let p be an odd prime, and r an integer such that $r < p$. Let $n = 2rp + 1$.*

(i) If there exists an integer a with $2 \leqslant a < n$ such that $a^{n-1} \equiv 1 \pmod{n}$ and $\gcd(a^{2r} - 1, n) = 1$ then n is prime.

(ii) If n is prime, the probability that a random value a satisfies $a^{n-1} \equiv 1 \pmod{n}$ and $\gcd(a^{2r} - 1, n) = 1$ is $1 - 1/p$.

Maurer's Algorithm Maurer proposed an iterative (and recursive) provable generation method based on this approach [**?**], which is described as Algorithm **??** for completeness.

This iterative method requires precomputing and storing the intermediate bitsize of all provable primes from the highest to the lowest. The number of iterations is variable and depends on a parameter r which is computed in order to provide the best output entropy.

Alg. 8.4.1 Maurer's iterative and recursive generation algorithm for provable primes.

Input: an integer ℓ, a list L of small primes,
Output: ProvablePrimeMaurer(ℓ) = a ℓ-bit provable prime n.

1. if $\ell \leqslant 20$ then
2. generate randomly a ℓ-bit integer n,
3. apply Erathostene sieve on n to determine if n is prime,
4. if n is prime, then **return**(n)
5. **go to** 1.1
6. set $c \leftarrow 0.1$, $m \leftarrow 20$ and $B \leftarrow c \cdot \ell^2$
7. if $\ell > 2m$ then
8. repeat
9. $s \leftarrow$ a real value in $[0, 1]$
10. $r \leftarrow 2^{s-1}$
11. until $(\ell - r \cdot \ell > m)$
12. else $r \leftarrow 0.5$
13. $q \leftarrow$ProvablePrimeMaurer$(\lfloor r \cdot \ell \rfloor + 1)$
14. $I \leftarrow \lfloor \frac{2^{\ell-1}}{2q} \rfloor$
15. status $\leftarrow$ false
16. while (status = false) do
17. $r \leftarrow$ integer in $[I + 1, 2I]$,
18. $n \leftarrow 2r \cdot q + 1$,
19. if no prime in L divides n do the following else **go to** 8.1,
20. $a \leftarrow$ an integer in $[2, n - 2]$
21. $b \leftarrow a^{n-1} (\text{mod } n)$
22. if $(b = 1)$, then do the following:
23. $b \leftarrow a^{2r}$ et $d \leftarrow pgcd(b - 1, n)$
24. if $(d = 1)$ then status $\leftarrow$ true
25. **return**(n)

We ran experimental simulations to analyze the average sizes of the recursive iterations of this algorithm. Based on $100,000$ executions, we collect the results given on Table **??**.

The main drawback of this implementation is that it is not efficient enough and therefore not suited to embedded implementations.

A simpler and faster algorithm can be derived from Theorem **??** (i) by iteratively producing provable primes twice larger at each iteration. We detail thus in the following

ℓ_0	ℓ_1	ℓ_2	ℓ_3	ℓ_4	ℓ_5	ℓ_6	ℓ_7	ℓ_8	ℓ_9	ℓ_{10}	ℓ_{11}	ℓ_{12}
512	363	257	182	129	92	66	38	20	-	-	-	-
768	544	385	273	194	138	98	70	50	28	15	-	-
1024	725	513	363	257	182	129	92	66	38	20	-	-
1536	1087	769	544	385	273	194	138	98	70	50	28	15
2048	1449	1025	725	513	363	257	182	129	92	66	38	20

Table 8.1: Average iterative prime bit sizes in Maurer's method

an efficient generation algorithm relying on this theorem and many implementation and generation tricks.

8.4.1 The Square Root Method

We now show how to generate provable primes more efficiently using Theorem **??** with fixed bitsizes for intermediate primes. We generate a provable prime by doubling at each iteration the size of the current prime p to derive the new prime $n = 2rp + 1$. While the entropy of this approach – estimated later in the chapter – is not as optimized as in Maurer's algorithm, this offers a more suitable and efficient algorithm in embedded environments.

The intermediate prime sizes can be seen as equivalent to those in Maurer's algorithm when fixing $r = 0.5$ at Step **??** of Algorithm **??**. An iterative and recursive method relying on this idea – doubling each time the size of primes – was also proposed by Shawe-Taylor in [**?**] before Maurer's publication and is recommended by the NIST [**?**] to generate provable primes for public key schemes. The first algorithm we propose can therefore be seen as an adaptation of the Shawe-Taylor method, which also relies on Pocklington's theorem. As opposed to Shawe-Taylor, our algorithm is not recursive but directly generates the primes iteratively from the smallest to the largest and many additional optimizations are put forward to improve efficiency.

Initialization.

Before making use of Pocklington's theorem, one starts the generation with a first prime with initial bitsize l_0. In his algorithm, Maurer suggests generating the first prime (which is 20-bit long in the best case) using Erathostene's sieve. Our approach here is different and applies the Miller-Rabin criterion to generate initial primes up to 2^{32}. Indeed, Pomerance et al. [**?**] and Jaeschke [**?**] have proven that any number lesser[2] than 2^{32} is proven prime if it successfully passes the Miller-Rabin test with the three bases 2, 7 and 61. Making use of this trick, we obtain the algorithm $\mathsf{InitGenPrime}(\ell_0)$ **??**. We define the bitsize of the initial prime as

$$\ell_0 = \min_{k>0} \left\{ \left\lceil \frac{\ell_n - 1}{2^k} \right\rceil + 1 \text{ such that } \left\lceil \frac{\ell_n - 1}{2^{k-1}} \right\rceil + 1 > 32 \right\} .$$

As indicated previously, we make use of $\mathsf{InitGenPrime}(\ell_0)$ to generate the initial prime p for any given size ℓ_0 lesser than 32. Though the execution time dedicated to initiate the sequence of primes is negligible compared to generating larger primes later in the sequence, it could be further improved using trial divisions. To illustrate the different steps of our method, Table **??** gives for different bitsizes ℓ_n, the initial prime size ℓ_0, the number k of iterations of Pocklington's theorem, and the intermediate prime sizes ℓ_i at each iteration.

In order to reduce the number of Fermat tests throughout the generation, we apply the same idea as in the generation of probable primes: we get rid of candidates n which are not coprime to a product Π of the smallest primes. By verifying that none of the

[2] More precisely, the exact bound is $4'759'123'141$.

Alg. 8.4.2 Generation of the initial prime based on Miller-Rabin testing.

Input: bitsize $\ell_0 < 32$ of the initial (provable) prime, $\Pi = 2 \cdot 3 \cdot 5 \cdot \ldots \cdot p_t$
Output: $\mathsf{GenInitPrime}(\ell_0)$: a ℓ_0-bit provable prime

1. generate a random ℓ_0-bit integer n with $\gcd(n, \Pi) = 1$ and **go to ??**
2. update n such that $\gcd(n, \Pi) = 1$,
3. if $\mathsf{F}_2(n) =$ **false** then **go to ??**
4. if $\mathsf{MR}_2(n) =$ **false** then **go to ??**
5. if $\mathsf{MR}_7(n) =$ **false** then **go to ??**
6. if $\mathsf{MR}_{61}(n) =$ **false** then **go to ??**
7. **return** n

ℓ_n	k	ℓ_0	ℓ_1	ℓ_2	ℓ_3	ℓ_4	ℓ_5	ℓ_6	ℓ_7
512	5	17	33	65	129	257	512	-	-
768	5	25	49	97	193	385	768	-	-
1024	6	17	33	65	129	257	513	1024	-
1536	6	25	49	97	193	385	769	1536	-
2048	7	17	33	65	129	257	513	1025	2048

Table 8.2: Intermediate bitsizes (ℓ_0 and ℓ_i) and number of iterations (k) for different sizes of the final prime (ℓ_n).

first t small primes divide the candidate n, the number of tests is reduced much further. We thus obtain the provable prime generator presented as Algorithm **??**.

Selection and Update of r and n.

A first solution for finding a suitable r at Step **??** of Algorithm **??** consists in randomly selecting a first value $r \in [I+1, 2I]$, setting $n = 2rp + 1$, and then incrementing r by 1 and n by $2p$ until the modular residues $(\omega_i = n \bmod p_i)_{i=1,\ldots,t}$ are all non zero. Each ω_i is then incremented by $2p \bmod p_i$. An efficient trick consists in obtaining the values $2p \bmod p_i$ by doubling modulo p_i the residues ω_i of the previous iteration since the previous value of n corresponds to the new value of p in the current iteration. At Step **??**, the same incremental update of r and n is applied for generating the next candidate coprime to Π. This leads to Algorithm **??**.

A second solution consists in generating n simultaneously compliant with Pocklington's property (an even multiple of p plus one) and coprime to Π. This is done by first selecting r as $(x - (2p)^{-1} \bmod \Pi)$ where x is randomly selected from $\mathbb{Z}_\Pi^\star$ using the technique of [**?**] based on Carmichael's function. Then r is added to a random multiple of Π so that it lies in $[I+1, 2I]$, and the first candidate n is computed as $2rp + 1$. Doing so, n is constructively coprime to Π. At Step **??**, the next candidate is computed in the same vein from the updated value $x \leftarrow p_{t+1} \cdot x \bmod \Pi$. It leads to algorithm **??**.

Alg. 8.4.3 Efficient-Square-Root-Generation(ℓ_n)

Input: a bitsize ℓ_n, $\Pi = 3 \cdot 5 \cdot \ldots \cdot p_t$
Output: an ℓ_n-bit provable prime n

1. $\ell \leftarrow \ell_n$
2. **while** $\ell > 31$ **do**
3. $\quad \ell \leftarrow \ell/2$
4. $\ell \leftarrow \ell + 1$
5. $n \leftarrow \mathsf{GenInitPrime}(\ell)$ [compute the initial small prime]
6. **while** $\ell < \ell_n$ **do**
7. $\quad p \leftarrow n$
8. $\quad \ell \leftarrow \min(2\ell - 1, \ell_n)$
9. $\quad I \leftarrow \lfloor \frac{2^{\ell-1}}{2p} \rfloor$
10. $\quad$ Select r at random from $[I + 1, 2I]$ such that $n \leftarrow 2rp + 1$ is coprime to Π and **go to ??**
11. $\quad$ Update r in $[I + 1, 2I]$ such that $n \leftarrow 2rp + 1$ is coprime to Π
12. $\quad$ **if** $\ell < 129$ **then**
13. $\quad\quad$ pick an integer a at random from $[2, n - 2]$
14. $\quad$ **else**
15. $\quad\quad a \leftarrow 2$
16. $\quad$ if $a^{n-1} \bmod n \neq 1$ then **go to ??**
17. $\quad$ if $\gcd(a^{2r} - 1, n) \neq 1$ then **go to ??**
18. **return** n

Alg. 8.4.4 EfficientProvablePrimeGen for Square Root method with Trial(ℓ_n,Π)

Input: a targeted bit size ℓ_n, $\Pi = 2 \cdot 3 \cdot 5 \cdot \ldots \cdot p_t$
Output: a ℓ_n-bit provable prime n

1. **while** $\ell > 32$ **do**
2. $\quad \ell \leftarrow \ell/2$
3. $\ell \leftarrow \ell + 1$ [select initial size]
4. $n \leftarrow \mathsf{GenInitPrime}(\ell)$ [compute the initial small prime]
5. $\ell \leftarrow \ell_0$
6. **while** $\ell < \ell_n$ **do**
7. $\quad p \leftarrow n$, $\ell \leftarrow \min(2\ell - 1, \ell_n)$ and $I \leftarrow \lfloor \frac{2^{\ell-1}}{2p} \rfloor$
8. $\quad$ **if** $\ell = \ell_0$ **then**
9. $\quad\quad$ **for** $i = 1$ **to** t **do**
10. $\quad\quad\quad \gamma_i \leftarrow 2p \bmod p_i$
11. $\quad$ **else**
12. $\quad\quad$ **for** $i = 1$ **to** t **do**
13. $\quad\quad\quad \gamma_i \leftarrow 2.\omega_i \bmod p_i$
14. $\quad$ pick an integer r at random from $[I + 1, 2I]$
15. $\quad n \leftarrow 2rp + 1$
16. $\quad$ **for** $i = 1$ **to** t **do**
17. $\quad\quad \omega_i \leftarrow n \bmod p_i$
18. $\quad$ test $\leftarrow$ **true**
19. $\quad$ **for** $i = 1$ **to** t **do**
20. $\quad\quad$ **if** $\omega_i = 0$ **then**
21. $\quad\quad\quad$ test $\leftarrow$ **false**
22. $\quad$ **while not** test **do**
23. $\quad\quad r \leftarrow r + 1$ and $n \leftarrow n + 2p$
24. $\quad\quad$ **for** $i = 1$ **to** t **do**
25. $\quad\quad\quad \omega_i \leftarrow \omega_i + \gamma_i \bmod p_i$
26. $\quad\quad$ test $\leftarrow$ **true**
27. $\quad\quad$ **for** $i = 1$ **to** t **do**
28. $\quad\quad\quad$ **if** $\omega_i = 0$ **then**
29. $\quad\quad\quad\quad$ test $\leftarrow$ **false**
30. $\quad$ **if** $r > 2I$ **then**
31. $\quad\quad$ **go to ??**
32. $\quad$ **if** $\ell < 129$ **then**
33. $\quad\quad$ pick an integer a at random from $[2, n - 2]$
34. $\quad$ **else**
35. $\quad\quad a \leftarrow 2$
36. $\quad$ **if** $a^{n-1} \bmod n \neq 1$ **then**
37. $\quad\quad$ test $\leftarrow$ **false**
38. $\quad\quad$ **go to ??**
39. $\quad$ **if** $\gcd(a^{2r} - 1, n) \neq 1$ **then**
40. $\quad\quad$ test $\leftarrow$ **false**
41. $\quad\quad$ **go to ??**
42. **return** n

Alg. 8.4.5 EfficientSquareGen Constructive(ℓ_n,Π)

Input: a targeted bit size ℓ_n, $p_0 = 3, \ldots, p_t$
Output: a ℓ_n-bit provable prime n

1. **while** $\ell > 32$ **do**
2. $\quad \ell \leftarrow \ell/2$ [select initial size]
3. $\ell \leftarrow \ell + 1$
4. $n \leftarrow \mathsf{GenInitPrime}(\ell)$ [compute the initial small prime]
5. **while** $\ell < \ell_n$ **do**
6. $\quad p \leftarrow n$
7. $\quad \ell \leftarrow \min(2\ell - 1, \ell_n)$
8. $\quad I \leftarrow \lfloor \frac{2^{\ell-1}}{2p} \rfloor$
9. $\quad$ select right $\Pi = 3 \cdot 5 \ldots \cdot p_k$ from ℓ value
10. $\quad J \leftarrow \lfloor \frac{I}{\Pi} \rfloor$
11. $\quad$ Compute x in $\mathbb{Z}^{\star}_{\Pi}$
12. $\quad Inv_p \leftarrow (2p)^{-1} \bmod \Pi$
13. $\quad r \leftarrow x - Inv_p \bmod \Pi$
14. $\quad$ Choose z random in $[J, 2J - 1]$
15. $\quad r \leftarrow r + z\Pi$
16. $\quad$ **if** $r < I$ **then**
17. $\quad\quad r \leftarrow r + \Pi$
18. $\quad$ **if** $r > 2I$ **then**
19. $\quad\quad r \leftarrow r - \Pi$
20. $\quad$ **go to ??**
21. $\quad$ update $x \leftarrow 2x \bmod \Pi$
22. $\quad$ update $r \leftarrow x - Inv_p \bmod \Pi$
23. $\quad$ **go to ??**
24. $\quad n \leftarrow 2rp + 1$
25. $\quad$ **if** $\ell < 129$ **then**
26. $\quad\quad$ pick an integer a at random from $[2, n - 2]$
27. $\quad$ **else**
28. $\quad\quad a \leftarrow 2$
29. $\quad$ **if** $a^{n-1} \bmod n \neq 1$ **then**
30. $\quad\quad$ **go to ??**
31. $\quad$ **if** $\gcd(a^{2r} - 1, n) \neq 1$ **then**
32. $\quad\quad$ **go to ??**
33. **return** n

Fixing $a = 2$ in Fermat testing.

From Theorem **??** (ii), we know that the probability that a random value a rejects a prime n at Step **??** or **??** is $1/p$. Assuming that the fraction of rejected primes does not vary much from one value of a to another, choosing a constant value a has a negligible impact on the distribution of the generated primes when the bitsize ℓ is sufficiently large. For instance when generating a 128-bit prime number $n = 2rp+1$ from a 65-bit provable prime p, less than $1/2^{64}$ of the primes would never be reached. We accept this negligible loss of entropy and use $a = 2$ for the Fermat test when $\ell > 128$. This leads to faster exponentiations for steps **??** and **??** where modular multiplications by the base can be replaced with modular additions.

Estimated Performance.

Denoting respectively by T_{init}, T_I, T_u, T_{F_a} and T_g the execution times taken by the initialization, computing I, updating the candidate n, the Fermat test with base a and the gcd computation, the total average execution time of Algorithm **??** amounts to

$$T_{\text{provable}}(\ell_n) = T_{\text{init}}(\ell_0) + \sum_{i=1}^{k} \left(T_I(\ell_i) + N(\ell_i, \Pi) \cdot (T_u(\ell_i) + T_{\mathsf{F}_a}(\ell_i)) + T_g(\ell_i)\right) . \tag{8.2}$$

We report experimental results from our smartcard implementation of this prime number generator in Section **??**. Note that the value $N(\ell_i, \Pi)$ equals the average number of primality tests in the generation of probable primes for ℓ_i-bit integers coprime to Π. Also, as expected, we observed in our simulations that only one gcd is computed per ℓ_i-bit prime so that its execution time is almost negligible compared to the overall execution time.

8.4.2 The Cube Root Method

Our second method relies on (what we refer to) as the Cube Root Theorem put forward by Brillhart, Lehmer and Selfridge in 1970. More details on this result can be found in [**?**].

Theorem 8.4.3 (Brillhart-Lehmer-Selfridge-Tuckerman-Wagstaff [?]) *Let $n > 3$ be an odd integer, let $n = rF + 1$ where F is completely factored and $gcd(F, r) = 1$. Suppose there exists an integer a such that*

(i) $a^{n-1} \equiv 1 \pmod{n}$,

(ii) $\gcd(a^{(n-1)/q} - 1, n) = 1$ *for each prime factor q of F.*

Let $r = uF + s$, $1 \leqslant s < F$, and suppose $n < 2F^3 + 2F$, $F > 2$. If u is odd, or if u is even and $s^2 - 4u$ is not a perfect square, then n is prime.

As a corollary of Theorem **??**, we derive the following result:

Theorem 8.4.4 (Cube Root Theorem) *Let p be an odd prime, $n = 2rp + 1$ with r an integer such that $r < p^2 + 1$. If there exists an integer a with $2 \leqslant a \leqslant n$ such that*

(i) $a^{n-1} \equiv 1 \pmod{n}$ *and* $\gcd(a^{2r} - 1, n) = 1$,

(ii) $r = up + s$, $1 \leqslant s < p$ *for odd* u,

then n *is prime.*

Theorem **??** makes it possible to put together a prime number generator that iteratively produces provable primes three times larger at each iteration (instead of twice larger in the Square Root method). In order to speed-up the whole generation, we only consider cases where the quotient u is odd. This reduces the output entropy by one bit but has no significant impact on the security of cryptosystems such as RSA and DSA. To generate a provable prime of ℓ_n bits, our algorithm starts with the generation of an initial prime p of ℓ_0 bits, where ℓ_0 is established as follows:

$\ell_0 \leftarrow \ell_n$

while $(\ell_0 > 31)$ $\quad \ell_0 \leftarrow \lfloor \ell_0/3 \rfloor + 1$

The generation of this ℓ_0-bit initial prime is performed as previously using the Miller-Rabin criterion and algorithm InitGenPrime(ℓ_0) **??**. The sizes ℓ_i of intermediate primes are displayed on Table **??**.

ℓ_n	k	ℓ_0	ℓ_1	ℓ_2	ℓ_3	ℓ_4
512	3	20	59	176	512	-
768	3	29	86	257	768	-
1024	4	14	41	122	365	1024
2048	4	26	77	230	689	2048

Table 8.3: Intermediate sizes (ℓ_0 and ℓ_i) and number of iterations (k) for different sizes (ℓ_n) of the final prime.

We then obtain the Cube Root prime number generator described in Algorithm **??**.

Initial Selection and Update of r and n.

A first solution for selecting a suitable r at Step **??** of Algorithm **??** is similar to the one used in the Square Root algorithm **??**. An additional step is necessary that consists in computing u and s in $r = up + s$ in order to avoid candidates for which u is even. We obtain the algorithm **??**.

Our second and most efficient solution for Step **??** consists in generating n in a constructive manner so that n is simultaneously compliant with Pocklington's requirement (an even multiple of p plus one), is coprime to Π and such that the quotient $u = \lfloor r/p \rfloor$ is forced to be odd. To this end, we keep track of an invertible element $x \in \mathbb{Z}_\Pi^\star$ which will serve as the residue of n modulo the prime product Π, and set $r = x - 1/(2p) \bmod \Pi$ to ensure that $n = 2xp \bmod \Pi$ is invertible modulo Π, so that the first two requirements are fulfilled. Now note that letting $r = up + s$, u is odd if and only if r and s have opposite parities. Therefore, if s is set to a fixed odd value throughout the search sequence, it

Alg. 8.4.6 Efficient-Cube-Root-Generation(ℓ_n)

Input: a bitsize ℓ_n, $\Pi = 3 \cdot 5 \cdot \ldots \cdot p_t$
Output: an ℓ_n-bit provable prime n

1. $\ell \leftarrow \ell_n$
2. **while** $\ell > 31$ **do**
3. $\quad \ell \leftarrow \lfloor \ell/3 \rfloor$
4. $\ell \leftarrow \ell + 1$
5. $n \leftarrow \mathsf{GenInitPrime}(\ell)$ [compute the initial small prime]
6. **while** $\ell < \ell_n$ **do**
7. $\quad p \leftarrow n$
8. $\quad \ell \leftarrow \min(3\ell - 1, \ell_n)$
9. $\quad I \leftarrow \lfloor \frac{2^{\ell-1}}{2p} \rfloor$
10. $\quad$ Select r at random from $[I+1, 2I]$ such that $r = up + s$, $1 \leqslant s < p$ for odd u and $n \leftarrow 2rp + 1$ is coprime to Π and **go to ??**
11. $\quad$ Update r in $[I+1, 2I]$ such that $r = up + s$, $1 \leqslant s < p$ for odd u and $n \leftarrow 2rp + 1$ is coprime to Π
12. $\quad$ **if** $\ell < 129$ **then**
13. $\quad\quad$ select a at random from $[2, n-2]$
14. $\quad$ **else**
15. $\quad\quad a \leftarrow 2$
16. $\quad$ if $a^{n-1} \bmod n \neq 1$ then **go to ??**
17. $\quad$ if $\gcd(a^{2r} - 1, n) \neq 1$ then **go to ??**
18. **return** n

Alg. 8.4.7 EfficientProvablePrimeGen for Cube Root method with Trial(ℓ_n,Π)

Input: a targeted bit size ℓ_n, $\Pi = 2 \cdot 3 \cdot 5 \cdot \ldots \cdot p_t$
Output: a ℓ_n-bit provable prime n

1. **while** $\ell > 32$ **do**
2. $\quad \ell \leftarrow \lfloor \ell/3 \rfloor + 1$ [select initial size]
3. $n \leftarrow \mathsf{GenInitPrime}(\ell)$ [compute the initial small prime]
4. $\ell \leftarrow \ell_0$
5. **while** $\ell < \ell_n$ **do**
6. $\quad p \leftarrow n$, $\ell \leftarrow \min(3\ell - 1, \ell_n)$ and $I \leftarrow \lfloor \frac{2^{\ell-1}}{2p} \rfloor$
7. $\quad$ pick an integer r at random from $[I + 1, 2I]$
8. $\quad n \leftarrow 2rp + 1$, $u \leftarrow \lfloor \frac{r}{p} \rfloor$ and $s = r \bmod p$
9. $\quad$ **if** $u \bmod 2 = 0$ **then**
10. $\quad\quad u \leftarrow u + 1$, $r \leftarrow r + p$
11. $\quad\quad$ **if** $r > 2I$ **then**
12. $\quad\quad\quad$ **go to ??**
13. $\quad\quad n \leftarrow n + 2p^2$
14. $\quad$ **if** $\ell = \ell_0$ **then**
15. $\quad\quad$ **for** $i = 1$ **to** t **do**
16. $\quad\quad\quad \alpha_i \leftarrow p \bmod p_i$ and $\xi_i \leftarrow (2\alpha_i)^2 \bmod p_i$
17. $\quad$ **else**
18. $\quad\quad$ **for** $i = 1$ **to** t **do**
19. $\quad\quad\quad \alpha_i \leftarrow \omega_i$ and $\xi_i \leftarrow (2\alpha_i)^2 \bmod p_i$
20. $\quad$ **for** $i = 1$ **to** t **do**
21. $\quad\quad \omega_i \leftarrow n \bmod p_i$
22. $\quad$ test $\leftarrow$ **true**
23. $\quad$ **for** $i = 1$ **to** t **do**
24. $\quad\quad$ **if** $\omega_i = 0$ **then**
25. $\quad\quad\quad$ test $\leftarrow$ **false**
26. $\quad$ **while not** test **do**
27. $\quad\quad u \leftarrow u + 2$ and $r \leftarrow r + 2p$
28. $\quad\quad$ **if** $r > 2I$ **then**
29. $\quad\quad\quad$ **go to ??**
30. $\quad\quad n \leftarrow n + 4p^2$
31. $\quad\quad$ **for** $i = 1$ **to** t **do**
32. $\quad\quad\quad \omega_i \leftarrow \omega_i + \xi_i \bmod p_i$
33. $\quad\quad$ test $\leftarrow$ **true**
34. $\quad\quad$ **for** $i = 1$ **to** t **do**
35. $\quad\quad\quad$ **if** $\omega_i = 0$ **then**
36. $\quad\quad\quad\quad$ test $\leftarrow$ **false**
37. $\quad$ **if** $r > 2I$ **then**
38. $\quad\quad$ **go to ??**
39. $\quad$ **if** $\ell < 129$ **then**
40. $\quad\quad$ pick an integer a at random from $[2, n - 2]$
41. $\quad$ **else**
42. $\quad\quad a \leftarrow 2$
43. $\quad$ **if** $a^{n-1} \bmod n \neq 1$ **then**
44. $\quad\quad$ test $\leftarrow$ **false** and **go to ??**
45. $\quad$ **if** $\gcd(a^{2r} - 1, n) \neq 1$ **then**
46. $\quad\quad$ test $\leftarrow$ **false** and **go to ??**
47. **return** n

is enough to ensure that r is even to force the parity of u to one. We now describe our method in more detail. Focusing on the search sequence associated with the i-th iteration, our generator proceeds as follows:

1. Fetch precomputed values $\Pi \leftarrow \Pi[i]$ and $\Lambda \leftarrow \Lambda[i]$ from code data. $\Pi \approx 2^{\ell_{i-1}-2}$ is a product of small odd primes (thereby excluding 2 from the factorization of Π), and Λ is the Carmichael function of Π.
2. Use [**?**] to generate a random invertible element $x \in \mathbb{Z}_{\Pi}^{\star}$, namely:
   (a) Randomly select x modulo Π
   (b) Compute $t = x^{\Lambda} \bmod \Pi$
   (c) If $t \neq 1$
      i. Randomly select z modulo Π
      ii. Update $x = x + z(1 - t) \bmod \Pi$
      iii. Goto **??**
3. Compute $1/(2p) = (2p)^{\Lambda-1} \bmod \Pi$ and derive $1/p \bmod \Pi$
4. Randomly select an odd value s modulo p
5. Use Chinese remaindering to compute $r \in [0, 2p\Pi]$ such that $r = x - 1/(2p) \bmod \Pi$, $r = s \bmod p$ and $r = 0 \bmod 2$. More precisely:
   (a) Compute $r_{\Pi p} = (((x - 1/(2p) - s)/p) \bmod \Pi) \cdot p + s$
   (b) Compute $r = (r_{\Pi p} \bmod 2) \cdot \Pi \cdot p + r_{\Pi p}$
   (c) Add appropriate multiple of $2p\Pi$ to r to get $r \in [I + 1, 2I]$

This concludes the initialization of the i-th loop *i.e.* the random selection of r at Step **??** at the i-th iteration. Updating r consists in just refreshing x as $x = 2x \bmod \Pi$ and performing a new round of Chinese remaindering as per Step **??** above. It is worthwhile noticing optimizations here: since p and s are fixed throughout the search sequence, the generator can just compute $1/p \bmod \Pi$ and $(-1/(2p) - s) \bmod \Pi$ once and for all and store these values. Step **??** then amounts to a couple of multiplications and additions. Also, modular exponentiations modulo Π are particularly efficient since Λ is small due to the particular form – extreme smoothness – of Π.
We obtain the algorithm **??** given in the following.

8.5 Estimating the Output Entropy

The rule for deriving at each iteration an ℓ_i-bit provable prime from an ℓ_{i-1}-bit other provable prime ($n \leftarrow 2rp+1$) intrinsically generates primes p_i such that $p_i - 1$ is a multiple of a *half-size* prime p_{i-1}. This particular structure is not representative of the majority of prime integers, and obviously does not allow to generate them all. This section establishes the entropy of the output distribution of primes generated by Algorithms **??**

Alg. 8.4.8 EfficientCubeGenConstructive(ℓ_n,Π)

Input: a targeted bit size ℓ_n, $p_0 = 3, \ldots, p_t$
Output: a ℓ_n-bit provable prime n

1. **while** $\ell > 32$ **do**
2. $\quad \ell \leftarrow \lfloor \ell/3 \rfloor + 1$ [select initial size]
3. $n \leftarrow \mathsf{GenInitPrime}(\ell)$ [compute the initial small prime]
4. **while** $\ell < \ell_n$ **do**
5. $\quad p \leftarrow n$
6. $\quad \ell \leftarrow \min(3\ell - 1, \ell_n)$
7. $\quad I \leftarrow \lfloor \frac{2^{\ell-1}}{2p} \rfloor$
8. $\quad$ select right $\Pi = 3 \cdot 5 \ldots \cdot p_k$ from ℓ value
9. $\quad J \leftarrow \lfloor \frac{2^{\ell-1}}{2p.\Pi} \rfloor$
10. $\quad$ Compute x in $\mathbb{Z}^{\star}_{\Pi}$
11. $\quad Inv_p \leftarrow 2p^{-1} \bmod \Pi$
12. $\quad r \leftarrow x - Inv_p \bmod \Pi$
13. $\quad$ Choose z random in $[J, 2J - 1]$
14. $\quad r \leftarrow r + z\Pi$
15. $\quad$ **if** $r < I$ **then**
16. $\quad\quad r \leftarrow r + \Pi$
17. $\quad$ **if** $r > 2I$ **then**
18. $\quad\quad r \leftarrow r - \Pi$
19. $\quad u \leftarrow \lfloor \frac{r}{p} \rfloor$
20. $\quad$ **if** u even **then**
21. $\quad\quad$ **go to ??**
22. $\quad$ **go to ??**
23. $\quad$ update $x \leftarrow 2x - Inv_p \bmod \Pi$
24. $\quad$ **go to ??**
25. $\quad n \leftarrow 2rp + 1$
26. $\quad$ **if** $\ell < 129$ **then**
27. $\quad\quad$ pick an integer a at random from $[2, n - 2]$
28. $\quad$ **else**
29. $\quad\quad a \leftarrow 2$
30. $\quad$ **if** $a^{n-1} \bmod n \neq 1$ **then**
31. $\quad\quad$ test $\leftarrow$ **false**
32. $\quad\quad$ **go to ??**
33. $\quad$ **if** $\gcd(a^{2r} - 1, n) \neq 1$ **then**
34. $\quad\quad$ test $\leftarrow$ **false**
35. $\quad\quad$ **go to ??**
36. **return** n

and **??** [3] and compare the output entropy with that obtained by a perfect generator that outputs uniformly random primes of a given bitsize ℓ_n.

Let us denote by R_{ℓ_i} the number of ℓ_i-bit primes that are attainable by the Square Root method at the end of iteration i. Note that any one of them can be uniquely derived from the sequence $(r_1, \ldots, r_i)$ of the values taken by r at each iteration. Since r is drawn at random, this suggests the heuristic approximation that the distribution of generated primes is uniform and that its entropy is equal to $H_{\ell_i} = \log_2(R_{\ell_i})$. According to Gauss's theorem, the number $\pi(x)$ of primes lesser than x is well approximated by $\frac{x}{\ln(x)}$ for large x. The number of exactly ℓ-bit primes can thus be estimated by

$$S_\ell = \frac{2^\ell}{\ln(2^\ell)} - \frac{2^{\ell-1}}{\ln(2^{\ell-1})} \ .$$

In an initial step, the algorithm randomly generates an ℓ_0-bit prime p_0, so that $R_{\ell_0} = S_{\ell_0}$. For $x \in \left[2^{\ell_{i-1}-1}, 2^{\ell_{i-1}}\right]$, consider an interval of width dx centered on x. Every p_{i-1} in this interval can generate $I = \lfloor \frac{2^{\ell_i-1}}{2 \cdot p_{i-1}} \rfloor \simeq \frac{2^{\ell_i-2}}{x}$ candidates among which $\frac{2^{\ell_i-2}}{x \cdot \ln(2^{\ell_i})}$ are prime numbers[4]. The total number of primes – that can or cannot be reached by the generator – in the considered interval is $\frac{dx}{\ln(x)}$, but only a fraction

$$\frac{R_{\ell_{i-1}} \cdot \ln(2^{\ell_{i-1}})}{2^{\ell_{i-1}-1}}$$

of these can be generated at iteration $(i-1)$, so that the number of primes p_{i-1} to consider in the interval is

$$\frac{R_{\ell_{i-1}} \cdot \ln(2^{\ell_{i-1}}) \cdot dx}{2^{\ell_{i-1}-1} \cdot \ln(x)} \ .$$

Integrating over $\left[2^{\ell_{i-1}-1}, 2^{\ell_{i-1}}\right]$ the number of primes that each p_{i-1} can generate, we obtain

$$\begin{aligned}
\frac{R_{\ell_i}}{R_{\ell_{i-1}}} &\simeq \int_{2^{\ell_{i-1}-1}}^{2^{\ell_{i-1}}} \frac{\ln(2^{\ell_{i-1}}) \cdot 2^{\ell_i-2}}{2^{\ell_{i-1}-1} \cdot \ln(2^{\ell_i})} \cdot \frac{dx}{x \ln(x)} \\
&\simeq \frac{\ell_{i-1} \cdot 2^{\ell_i-2}}{\ell_i \cdot 2^{\ell_{i-1}-1}} \cdot \int_{2^{\ell_{i-1}-1}}^{2^{\ell_{i-1}}} \frac{dx}{x \ln x} \\
&\simeq \frac{\ell_{i-1}}{\ell_i} \cdot 2^{\ell_i-\ell_{i-1}-1} \cdot \big(\ln(\ell_{i-1}) - \ln(\ell_{i-1}-1)\big) \\
&\simeq \frac{\ell_{i-1}}{\ell_i} \cdot \frac{2^{\ell_i-\ell_{i-1}-1}}{\ell_{i-1}-1}
\end{aligned}$$

[3] Note that for efficiency purposes Algorithm **??** only selects r values for which $u = \lfloor \frac{r}{p} \rfloor$ is odd. In the sequel we first derive the entropy of our method when ignoring this trick. We subsequently address the effect of this feature later on.

[4] This derives from a commonly accepted approximation that the Chebotarëv density theorem also stands for large intervals. This theorem actually implies that for any coprime integers a and d, the proportion of primes less than x belonging to the arithmetic progression $\{a + nd\}_n$ tends to $\frac{1}{\phi(d)}$ when x tends to infinity.

whence

$$R_{\ell_n} = S_{\ell_0} \cdot \frac{\ell_0}{\ell_n} \cdot \frac{2^{\ell_n - \ell_0 - k}}{\prod_{i=1}^{k}(\ell_{i-1} - 1)} \tag{8.3}$$

where examples cases for k, ℓ_0 and ℓ_i are given in Tables **??** and **??**.

As mentioned above, Equation (**??**) does not take into account that only half of the values for r are selected as prime candidates in Algorithm **??**. Assuming that even and odd values of u are evenly distributed for r ranging from $I+1$ to $2I$, the effect of ignoring half of potential candidates is that every prime p_{i-1} in the neighborhood of x can generate only $\frac{2^{\ell_i - 3}}{x \cdot \ln(2^{\ell_i})}$ primes. This results in the following expression for the number of l_n-bit primes generated by Algorithm **??** when only odd u values are selected:

$$R_{\ell_n} = S_{\ell_0} \cdot \frac{\ell_0}{\ell_n} \cdot \frac{2^{\ell_n - \ell_0 - 2k}}{\prod_{i=1}^{k}(\ell_{i-1} - 1)} \ . \tag{8.4}$$

The estimated entropies H_{ℓ_n} provided by Algorithms **??** and **??** are given in Table **??** for different output bitsizes ℓ_n together with the entropy $H^*_{\ell_n}$ of a perfectly uniform distribution.

ℓ_n	512	768	1024	1536	2048
$H^*_{\ell_n}$	503	758	1014	1525	2037
H_{ℓ_n} (Alg. **??**, Eq. (**??**))	467	720	968	1476	1980
H_{ℓ_n} (Alg. **??**, Eq. (**??**))	479	733	981	1490	2000

Table 8.4: Entropy loss w.r.t. ideal prime generation

The entropy loss of the proposed prime generation ranges from 36 bits for 512-bit primes to 57 bits for 2048-bit primes for the Square Root method, and only from 24 to 37 bits for the Cube Root method. While somewhat larger than the entropy loss of about 4 bits found in Maurer's method, it is noticeable that it is small enough so that exhaustive search remains infeasible for currently secure bitsizes. We believe that the security of RSA and DSA cryptosystems is not in practice affected by using either Algorithm **??** or **??** for generating provable primes.

8.6 Implementation Results and Practical Aspects

8.6.1 On-board Generation of Probable Primes

Our implementation relies on an AT90SC chip supplied by Inside Secure embedding the Ad-X cryptoprocessor and the 8-bit AVR core both running at 30 MHz. The chip manufacturer provides a cryptographic toolbox for cryptography developers with all basic operations over large integers: modular multiplication, modular exponentiation, GCD, inversion, division, and so forth. The associated documentation provides estimated performances (cycle count) for these operations. Using this information we know the exact cycle count for any step of the generation algorithm. The exact average timings of our prime number generators can then be deduced on this component using Equation **??**.

Using the development kit from IAR running on a chip emulator loaded with the toolbox, the performance of our implementation of the generator for probable primes was experimentally confirmed to coincide perfectly with Equation **??**.

The Fermat test with base 2 runs in 11 ms for a 512-bit integer n while the Miller-Rabin test with a random base is computed in 18 ms. We chose $t = 54$, so that Π is the product of small primes ranging from 2 to 251 and we choose $h = 3$ (the number of Miller-Rabin rounds).

On average, our generator outputs 512-bit probable primes in 580 ms ($N(512, \Pi) = 35.6$), 768-bit probable primes in $2'130$ ms ($N(768, \Pi) = 53.4$) and 1024-bit probable primes in $5'780$ ms ($N(1024, \Pi) = 71.2$).

8.6.2 Generating Provable Primes

Similarly, we deduced from Equation **??** the execution timings for our generator of provable primes on the same smartcard platform. We made use of the base-2 Fermat test when ℓ is greater than 128 bits, and took the same value for Π as in the case of probable primes. We have also implemented Algorithm **??** on the target chip. As a result, using the Square Root method to generate provable primes of respectively 512, 768 and 1024 bits requires on average 810, $2'580$ and $5'940$ ms. The Cube Root method decreases these figures to 760, $2'240$ and $5'700$ ms respectively.

8.6.3 Comparing Generators for Probable and Provable Primes

Given the expressions of $T_{\text{Prob}}(\ell)$ and $T_{\text{Provable}}(\ell)$, a rough guesstimate is that about the same number of modular exponentiations should be required to generate probable and provable primes of the same size, assuming trial divisions and identical values for Π. This is because the extra workload needed to generate the sequence of intermediate primes in the provable case remains fairly small compared to the resources needed to generate the full-length ℓ_n-bit provable prime. Moreover, this extra workload is somewhat compensated by the absence of final Miller-Rabin rounds or the Lucas test. All in all, we observe that the generation of a provable prime is slightly less efficient that the one of a probable prime when only a few Miller-Rabin rounds are required. However, the Cube Root algorithm becomes the fastest option when either a significant amount of Miller-Rabin iterations or a Lucas test is needed.

Even though it is not suitable for a smartcard implementation, we analyze here what would be the exact performances of a prime number generator based on Maurer's algorithm. Algorithm **??** recalls Maurer's approach. Based on these results, we can evaluate the average time needed to compute a prime number. We find then a 1024-bit prime number require 11 iterations starting with a 20-bit prime value. We consider the method is optimised through the use of trial divisions to ensure the candidate n is coprime with Π, and base for the exponentiation is set to value $a = 2$. We then adapt Equation **??**, inserting the additional operations to measure the average execution of an optimized Maurer algorithm.

Figure **??** provides performance measurements for the various generation methods discussed in the chapter.

Bitlength ℓ_n	h	512	768	1024	1536	2048	Lucas test
Algorithm **??**	3	580	1960	5400	24400	71800	no
Algorithm **??**	8	660	2200	5940	26100	75500	no
Algorithm **??**	3	**640**	**2130**	**5780**	**25700**	**74400**	yes
Algorithm **??**	40	1170	3700	9290	36800	98900	no
Algorithm **??**	-	**810**	**2580**	**5940**	**26500**	**75600**	provable
Algorithm **??**	-	**760**	**2240**	**5700**	**24400**	**73550**	provable
Algorithm **??**	-	2060	4140	9280	34500	99900	provable

Figure 8.1: Time (in milliseconds) measurements for various prime number generators.

We find that a Lucas test, as defined in FIPS 186-3, is roughly equivalent to 3.5 Miller-Rabin rounds and is therefore rather efficient on the AT90SC – comparatively to higher ratios found on other architectures.

Graphical Comparison

Figures **??** and **??** graphically compares the performance of Maurer's algorithm with the Square and the Cube root methods. The probabilistic generation has also been represented.

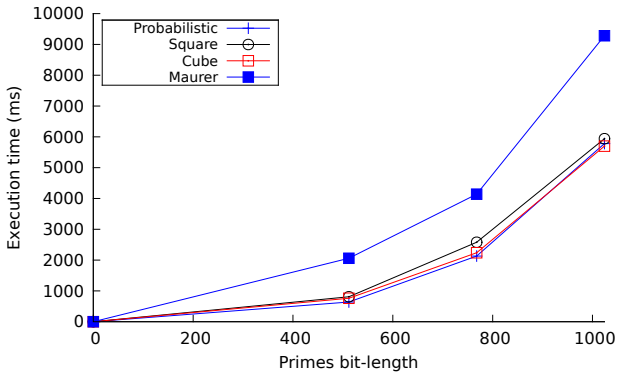


Figure 8.2: Comparison between prime generation techniques from 512 to 1024 bits.

Overall, our experimental validation shows that the Cube Root method is essentially as efficient as the state-of-the-art generation algorithms for probable primes.

8.7 Achieving Leakage-Resistant Prime Number Generation

This section addresses side-channel attacks and ways to protect prime number generation from information leakage. Recent research works [**?**, **?**] have highlighted that prime

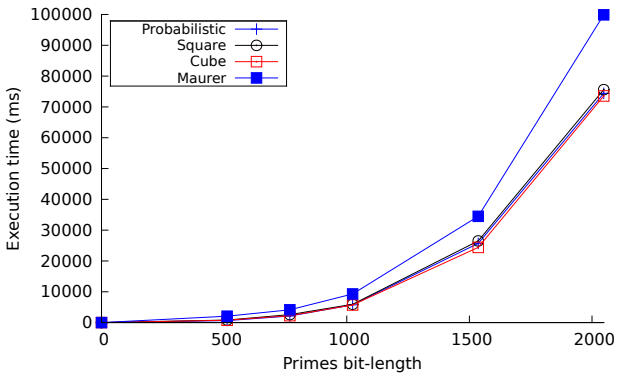


Figure 8.3: Comparison between prime generation techniques from 512 to 2048 bits.

number generation may be subject to power analysis. It is therefore necessary to ensure resistance against side-channels, especially when the device is operated in an untrusted environment. We give in this section a few guidelines for designing a protected implementation.

Assets to be protected are the output prime number as well as the secret elements used throughout its generation, more precisely the random values r and the sequence of intermediate primes reached by each iteration. It is therefore necessary to ensure that the implementation does not leak these values either during their generation or while they are being manipulated by the generation algorithm.

A first information leakage can occur during the generation of the first ℓ_0-bit prime. Since this is done using the Miller-Rabin criterion, the Miller-Rabin test itself has to be protected against side-channel attacks. A typical protection mechanism consists in performing an atomic modular exponentiation in the sense of [**?**] but since the base we use here is small, there is a risk that the exponent $n-1$ leaks at each multiplication as explained in [**?**]. The exponentiation may therefore be computed using a Square and Multiply-Always exponentiation which is a regular algorithm. A second operation to protect is the computation of I. This step involves the manipulation of p which must be kept secret. We therefore suggest to implement a secure division algorithm as described in [**?**].

Finke et al.presented in [**?**] an attack that specifically targets the computation of the next prime candidate (coprime to Π) at Step 2. of Algorithm **??**. The attack is particularly applicable when a trial update operation is done with increments of 2 or Π. This attack does not seem applicable on Step 9 (performed with trial updates) of Algorithm **??** since the value used for next value of n is $n+2p$ and p is unknown to the attacker. We recommend to implement the constructive method which is not sensitive to this attack and resists physical observation if the computation of p is done with the same exponentiation as the one used when applying the Miller-Rabin criterion.

We also note that the exponentiation $a^{n-1} \bmod n$ in Step 11 must be performed securely and that the atomic exponentiation is neither resistant nor efficient when $a = 2$. This part can be computed in a regular way using a Square and Multiply-Always

exponentiation. In this case using $a = 2$ still results in negligible computational time for the multiplication and the computation remains protected against the SPA attack published in [**?**]. However the first squaring and multiplication operations (when the accumulator is still a power of 2 smaller than the modulus n) could leak information. It would then reveal the first bits of the exponent (about 10). It is then recommended to blind the modulus with a random value: in that case the computation would be $(2^{n-1} \bmod r_1 \cdot n) \bmod n$.

The final computations to protect from power analysis lie in Step 12. The exponentiation $2^{2r} \bmod n$ must be protected against the disclosure or r by using, as previously, the Square and Multiply-Always exponentiation technique. Also, the GCD operation $\gcd(2^{2r} - 1, n)$ could reveal the value of p if not implemented in a secure way. Our implementation of the GCD calculation has been carried out in constant time using dummy operations.

Applying these methods, a side-channel protected yet efficient generator for provable primes would lead to the performances given in Figure **??**. When comparing to those with a protected generation of probable primes, we get that performances are quite similar.

Finally, we note that fault-based attacks are not considered as a serious threat for prime number generators at the present time. This is mainly due to the inherently randomized nature of the generation algorithms.

8.8 Giving the Proof of Primality

We showed previously that the Square Root and Cube Root methods generate efficiently provable prime numbers. However it is not possible to verify the generated number $n = rF + 1$ returned by the card respects the Pocklington theorem as the factorization of F (for each iteration) is unknown out of the card. In case it were necessary for testability and certification purposes it could then become problematic. To achieve this objective, both algorithm can be simply modified to return with the generated provable number n at the end all the primes p_i from each pocklington iteration of the algorithm. Knowing all the intermediate prime values it is easy to verify the returned prime n complies with the Pocklington theorem. In that case the algorithm we have designed also gives a proof of primality.

8.9 Conclusion

This chapter introduced two new methods to efficiently generate provable primes in embedded environments. We put forward novel algorithmic solutions and report practical results from our smartcard implementations. We have demonstrated that efficient generators exist for provable primes in constrained environments and compared the new methods with state-of-the-art generators for probable primes. We addressed side-channel analysis to ensure secure implementations of our generation methods. Overall, the study opens the way to embedded generation of provable primes in nearly similar or better performances than current generators.

Bitlength ℓ_n	h	512	768	1024	1536	2048	Lucas test
Secure version of algorithm **??**	8	680	2410	6400	29200	83000	yes
Secure version of algorithm **??**	-	870	2830	6600	29050	84900	provable
Secure version of algorithm **??**	-	815	2460	6340	27400	82700	provable

Table 8.5: Time (ms) measurements of secure implementations

Chapter 9

Square Always Exponentiation

9.1 Introduction

Nowadays most embedded devices implementing public key cryptography use RSA [**?**] for encryption and signature schemes, or cryptographic primitives over $(\mathbb{F}_p, \times)$ such as DSA [**?**] and the Diffie-Hellman key agreement protocol [**?**]. All these algorithms require the computation of modular exponentiations. Since the emergence of the so-called *side-channel analysis*, embedded devices implementing these cryptographic algorithms must be protected against a wider and wider class of attacks.

Moreover, the cost and timing constraints are crucial in many applications of embedded devices (e.g. banking, transport, etc.). This often requires cryptographic implementors to choose the best compromise between security and speed. Improving the efficiency of algorithms or countermeasures generates thus a lot of interest in the industry.

An exponentiation is generally processed using a sequence of multiplications, some of them having different operands and some of them being squarings. In [**?**], Amiel et al. showed that this distinction can provide exploitable side-channel leakages to an attacker. Classical countermeasures consist of using exponentiation algorithms where the sequence of multiplications and squarings does not depend on the secret exponent.

Our contribution is to propose a new exponentiation scheme using squarings only, which is faster than the classical countermeasures. Also, we introduce new algorithms having a particularly low cost when two squarings can be parallelized.

This paper is organized as follow: in Section **??** we recall classical exponentiation algorithms and present some well-known side-channel attacks and countermeasures. Then we propose our new countermeasure in Section **??**. Finally we present some practical results in Section **??** and we conclude in Section **??**.

9.2 Background on Exponentiation on Embedded Devices

We recall in this section some classical exponentiation algorithms. First we present the *square-and-multiply* algorithms upon which are based most of the exponentiation methods. Then we introduce the *side-channel analysis* and in particular the *simple power analysis* (SPA). We present some algorithms immune to this attack, and we finally recall

a particular side-channel attack aimed at distinguishing squarings from multiplications in an exponentiation operation.

9.2.1 Square-and-Multiply Algorithms

Many exponentiation algorithms have been proposed in the literature. Among the numerous references an interested reader can refer for instance to [**?**] for details. Alg. **??** and Alg. **??** are two variants of the classical square-and-multiply algorithm which is the simplest approach to compute an RSA exponentiation.

Alg. 9.2.1 Left-to-Right Square-and-Multiply Exponentiation

Input: $m, n \in \mathbb{N}$, $m < n$, $d = (d_{k-1}d_{k-2}\dots d_0)_2$
Output: $m^d \mod n$

1. $a \leftarrow 1$
2. **for** $i = k-1$ **to** 0 **do**
3. $\quad a \leftarrow a^2 \mod n$
4. $\quad$ **if** $d_i = 1$ **then**
5. $\quad\quad a \leftarrow a \times m \mod n$
6. **return** a

Alg. 9.2.2 Right-to-Left Square-and-Multiply Exponentiation

Input: $m, n \in \mathbb{N}$, $m < n$, $d = (d_{k-1}d_{k-2}\dots d_0)_2$
Output: $m^d \mod n$

1. $a \leftarrow 1$; $b \leftarrow m$
2. **for** $i = 0$ **to** $k-1$ **do**
3. $\quad$ **if** $d_i = 1$ **then**
4. $\quad\quad a \leftarrow a \times b \mod n$
5. $\quad b \leftarrow b^2 \mod n$
6. **return** a

Considering a balanced exponent d, these algorithms require on average $1S + 0.5M$ per bit of exponent to perform the exponentiation – S being the cost of a modular squaring and M the cost of a modular multiplication. It is generally considered in the literature – and corroborated by our experiments – that on cryptographic coprocessors $S \approx 0.8M$.

These algorithms are no longer used in embedded devices for security applications since the emergence of the side-channel analysis.

9.2.2 Side-Channel Analysis on Exponentiation

Side-channel analysis was introduced in 1996 by Kocher in [**?**] and completed in [**?**]. Many attacks have been derived in the following years.

On one hand, *passive* attacks rely on the following physical property: a microprocessor is physically made of thousands of logical gates switching differently depending on the executed operations and on the manipulated data. Therefore the power consumption and

the electromagnetic radiation, which depend on those gates switchings, reflect and may leak information on the executed instructions and the manipulated data. Consequently, by monitoring such side-channels of a device performing cryptographic operations, an observer may infer information on the implementation of the program executed and on the – potentially secret – data involved.

On the other hand, *active* attacks intend to physically tamper with computations and/or stored values in memories. Such effects are generally obtained using clock or power glitches, laser beam, etc.

Finally some works [**?**] have highlighted the fact that passive and active attacks may be combined to threaten implementations applying countermeasures against both of them but not against their simultaneous use.

In the remainder of this section we focus on two passive attacks : the SPA presented hereafter with classical countermeasures, and a particular analysis from [**?**] discussed in Section **??**.

Simple Power Analysis

Simple side-channel analysis [**?**] consists in observing a difference of behavior depending on the value of the secret key on the component performing cryptographic operations by using a single measurement.

In the case of an exponentiation, the original SPA is based on the fact that, if the squaring operation has a different pattern than a multiplication, the secret exponent can be directly read on the curve. For instance, in Alg. **??**, a 0 exponent bit implies a squaring to be followed by another squaring, while a 1 bit causes a multiplication to follow a squaring. Classical countermeasures consist of using *regular* algorithms or applying the *atomicity* principle, as detailed in previous sections.

9.3 Square Always Countermeasure

We present in this section new exponentiation algorithms which simultaneously benefit from efficiency of the atomicity principle and immunity against the aforementioned weakness of the multiply always method.

9.3.1 Principle

It is well known that a multiplication can be performed using squarings only. Therefore we propose the following countermeasure which consists in using either expression (**??**) or (**??**) to perform all the multiplications in the exponentiation. Combined with the atomicity principle, this countermeasure completely prevents the attack described in Section **??** since only squarings are performed.

$$x \times y = \frac{(x+y)^2 - x^2 - y^2}{2} \tag{9.1}$$

$$x \times y = \left(\frac{x+y}{2}\right)^2 - \left(\frac{x-y}{2}\right)^2 \tag{9.2}$$

At the first glance, (**??**) requires three squarings to perform a multiplication whereas (**??**) requires only two. Further analysis reveals however that using (**??**) or (**??**) in Alg. **??** and **??** has always the cost of replacing multiplications by twice more squarings. Indeed, notice that in the multiplication $a \leftarrow a \times m$ of Alg. **??** m is a constant operand. Therefore implementing $a \times m$ using (**??**) yields $y = m$, thus $m^2 \mod n$ can be computed only once at the beginning of the exponentiation. The cost of computing y^2 can then be neglected.

This trick does not apply to Alg. **??** since no operand is constant in step **??**. However $b \leftarrow b^2$ is the following operation. Using equation (**??**) in Alg. **??** then yields to store $t \leftarrow y^2$ and save the following squaring: $b \leftarrow t$. The resulting cost is thus equivalent as trading one multiplication for two squarings.

Remark In our context, (**??**) or (**??**) refer to operations modulo n. Notice however that divisions by 2 in these equations require neither inversion nor multiplication. For example, we recommend computing $z/2 \mod n$ in the following atomic way:

$t_0 \leftarrow z$
$t_1 \leftarrow z + n$
$\alpha \leftarrow z \mod 2$
return $t_\alpha/2$

9.3.2 Atomic Algorithms

Trading multiplications for squarings in Alg. **??** and **??** just requires to apply formula (**??**) or (**??**) at step **??** in Alg. **??** or step **??** in Alg. **??**. However the resulting algorithms would still present a leakage since different operations would be performed when processing a 0 or 1 bit. Hence it is necessary to apply the atomicity principle on these algorithms.

This step is achieved by identifying a minimal pattern of operations to be performed on each loop iteration and rewrite the algorithms using this pattern. For the considered algorithms, the minimal pattern should obviously contain a single squaring since it is the only operation required by the processing of a 0 bit and performing dummy squarings would lessen the performances of the algorithm. An addition, subtraction and division by 2 should also be present to compute formulas (**??**) or (**??**). Finally some more operations are required to manage the loop counter and the pointer on exponent bits.

Algorithm **??** presented hereafter details how to implement atomically the square always method in a left-to-right exponentiation using equation (**??**).

As in [**?**] we use a matrix for a more readable and efficient implementation:

$$M = \begin{pmatrix} 1 & 1 & 1 & 0 & 2 & 1 & 1 & 1 & 2 & 1 \\ 2 & 0 & 1 & 2 & 2 & 2 & 2 & 2 & 3 & 0 \\ 1 & 1 & 3 & 0 & 0 & 0 & 0 & 2 & 0 & 0 \\ 3 & 3 & 3 & 0 & 3 & 3 & 1 & 1 & 3 & 1 \end{pmatrix}$$

The main loop of Alg. **??** can be viewed as a four state machine where each row j of M define the operands of the atomic pattern. The atomic pattern itself is given by the content of the loop, i.e. steps **??** to **??**. An exponent bit d_i is processed by the state

Alg. 9.3.1 Left-to-Right Square Always Exponentiation with (**??**)

Input: $m, n \in \mathbb{N}$, $m < n$, $d = (d_{k-1}d_{k-2}\dots d_0)_2$
Output: $m^d \mod n$

1. $R_0 \leftarrow 1$; $R_1 \leftarrow m$; $R_2 \leftarrow 1$; $R_3 \leftarrow m^2/2 \mod n$
2. $j \leftarrow 0$; $i \leftarrow k-1$
3. **while** $i \geq 0$ **do**
4. $\quad R_{M_{j,0}} \leftarrow R_{M_{j,1}} + R_{M_{j,2}} \mod n$
5. $\quad R_{M_{j,3}} \leftarrow {R_{M_{j,3}}}^2 \mod n$
6. $\quad R_{M_{j,4}} \leftarrow R_{M_{j,5}}/2 \mod n$
7. $\quad R_{M_{j,6}} \leftarrow R_{M_{j,7}} - R_{M_{j,8}} \mod n$
8. $\quad j \leftarrow d_i(1 + (j \mod 3))$
9. $\quad i \leftarrow i - M_{j,9}$
10. **return** R_0

$j = 0$ (resp. $j = 3$) if the previous bit d_{i+1} is a 0 (resp. a 1). This state is followed by the processing of the next bit if $d_i = 0$, or by the states $j = 1$ and $j = 2$ if $d_i = 1$. For more clarity, we present below the four sequences of operations corresponding to each state. The dummy operations are identified by a $\star$.

$j = 0$ ($d_i = 0$ or 1)	
$R_1 \leftarrow R_1 + R_1 \mod n$	$\star$
$R_0 \leftarrow {R_0}^2 \mod n$	
$R_2 \leftarrow R_1/2 \mod n$	$\star$
$R_1 \leftarrow R_1 - R_2 \mod n$	$\star$
$j \leftarrow d_i$	[$\star$ if $d_i = 0$]
$i \leftarrow i - (1 - d_i)$	[$\star$ if $d_i = 1$]

$j = 2$ ($d_i = 1$)	
$R_1 \leftarrow R_1 + R_3 \mod n$	$\star$
$R_0 \leftarrow {R_0}^2 \mod n$	
$R_0 \leftarrow R_0/2 \mod n$	
$R_0 \leftarrow R_2 - R_0 \mod n$	
$j \leftarrow 3$	
$i \leftarrow i - 1$	

$j = 1$ ($d_i = 1$)	
$R_2 \leftarrow R_0 + R_1 \mod n$	
$R_2 \leftarrow {R_2}^2 \mod n$	
$R_2 \leftarrow R_2/2 \mod n$	
$R_2 \leftarrow R_2 - R_3 \mod n$	
$j \leftarrow 2$	
$i \leftarrow i$	$\star$

$j = 3$ ($d_i = 0$ or 1)	
$R_3 \leftarrow R_3 + R_3 \mod n$	$\star$
$R_0 \leftarrow {R_0}^2 \mod n$	
$R_3 \leftarrow R_3/2 \mod n$	$\star$
$R_1 \leftarrow R_1 - R_3 \mod n$	$\star$
$j \leftarrow d_i$	
$i \leftarrow i - (1 - d_i)$	[$\star$ if $d_i = 1$]

We also present in Alg. **??** a right-to-left variant of the square always exponentiation using equation (**??**). This algorithm requires the following matrix:

$$M = \begin{pmatrix} 0 & 0 & 2 & 0 & 0 & 0 & 2 & 1 \\ 2 & 1 & 2 & 2 & 1 & 0 & 1 & 0 \\ 0 & 2 & 1 & 1 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 1 & 2 & 1 & 1 \end{pmatrix}$$

As for the previous algorithm, the main loop of Alg. **??** has four states. Here, the state $j = 0$ corresponds to the processing a 0 bit and the sequence $j = 1$, $j = 2$, and $j = 3$ corresponds to the processing of a 1 bit, as detailed below.

Alg. 9.3.2 Right-to-Left Square Always Exponentiation with (**??**)

Input: $m, n \in \mathbb{N}$, $m < n$, $d = (d_{k-1}d_{k-2}\dots d_0)_2$
Output: $m^d \mod n$

1. $R_0 \leftarrow m$; $R_1 \leftarrow 1$; $R_2 \leftarrow 1$
2. $i \leftarrow 0$; $j \leftarrow 0$
3. **while** $i \leq k-1$ **do**
4. $\quad j \leftarrow d_i(1 + (j \mod 3))$
5. $\quad R_{M_{j,0}} \leftarrow R_{M_{j,1}} + R_0 \mod n$
6. $\quad R_{M_{j,2}} \leftarrow R_{M_{j,3}}/2 \mod n$
7. $\quad R_{M_{j,4}} \leftarrow R_{M_{j,5}} - R_{M_{j,6}} \mod n$
8. $\quad R_{M_{j,3}} \leftarrow {R_{M_{j,3}}}^2 \mod n$
9. $\quad i \leftarrow i + M_{j,7}$
10. **return** R_1

$j = 0$ ($d_i = 0$)	
$j \leftarrow 0$	[$\star$ if j was 0]
$R_0 \leftarrow R_0 + R_0 \mod n$	$\star$
$R_2 \leftarrow R_0/2 \mod n$	$\star$
$R_0 \leftarrow R_0 - R_2 \mod n$	$\star$
$R_0 \leftarrow {R_0}^2 \mod n$	
$i \leftarrow i + 1$	

$j = 2$ ($d_i = 1$)	
$j \leftarrow 2$	
$R_0 \leftarrow R_2 + R_0 \mod n$	$\star$
$R_1 \leftarrow R_1/2 \mod n$	
$R_0 \leftarrow R_0 - R_2 \mod n$	$\star$
$R_1 \leftarrow {R_1}^2 \mod n$	
$i \leftarrow i$	$\star$

$j = 1$ ($d_i = 1$)	
$j \leftarrow 1$	
$R_2 \leftarrow R_1 + R_0 \mod n$	
$R_2 \leftarrow R_2/2 \mod n$	
$R_1 \leftarrow R_0 - R_1 \mod n$	
$R_2 \leftarrow {R_2}^2 \mod n$	
$i \leftarrow i$	$\star$

$j = 3$ ($d_i = 1$)	
$j \leftarrow 3$	
$R_0 \leftarrow R_0 + R_0 \mod n$	$\star$
$R_0 \leftarrow R_0/2 \mod n$	$\star$
$R_1 \leftarrow R_2 - R_1 \mod n$	
$R_0 \leftarrow {R_0}^2 \mod n$	
$i \leftarrow i + 1$	

9.3.3 Performance Analysis

Algorithms **??** and **??** are mostly equivalent in terms of operations realized in a single loop. The number of dummy operations (additions, subtractions and halvings) introduced to fill the atomic blocks are the same in the two versions – it is generally considered that the cost of these operations is negligible compared to multiplications and squarings. Both algorithms require $2S$ per exponent bit on average or $1.6M$ if $S/M = 0.8$ which represents a theoretical 11.1% speed-up over Alg. **??** which is the fastest known regular algorithm immune to the attack from [**?**]. Table **??** compares the efficiency of the multiply always, Montgomery ladder, and square always algorithms when $S = M$ and $S/M = 0.8$.

In addition, our algorithms can be enhanced using the sliding window or m-ary exponentiation techniques [**?**, **?**] while the Montgomery ladder cannot. These techniques are known to provide a substantial speed-up on Alg. **??** when extra memory is available.

Though we did not investigate this path, we believe that a comparable trade-off between space and time can be expected.

9.3.4 Security Considerations

Our algorithms are protected against the SPA by the implementation of the atomicity principle. The analysis from [**?**] cannot apply either since only squarings are involved. As a matter of comparison, notice that the exponent blinding countermeasure does not fundamentally remove the source of the leakage but only renders this attack practically infeasible. Embedded implementations should also be protected against the *differential power analysis* (DPA) which we do not detail in this study. However it is worth noticing that classical DPA countermeasures, like exponent or modulus randomization, can be applied as well. The interested reader may refer to [**?**, **?**].

We recommend implementing Alg. **??** instead of Alg. **??** since left-to-right algorithms are vulnerable to the chosen message SPA and *doubling attack* [**?**], and more subject to combined attacks [**?**]. Besides, Alg. **??** requires one less register than Alg. **??**.

It is well-known that algorithms using dummy operations generally succumb to safe-error attacks. Immunity to C and M safe-errors can be easily obtained by applying the exponent randomization technique, which also prevent the DPA. Nevertheless, special care has been taken in our algorithms to ensure that inducing a fault in any of the dummy operations would produce an erroneous result. For instance, in the following sequence of dummy operations in Alg. **??** ($j = 0$), no operation can be tampered with without corrupting R_0 and thus the result of the exponentiation:

$$\begin{aligned} R_0 &\leftarrow R_0 + R_0 \mod n \\ R_2 &\leftarrow R_0/2 \mod n \\ R_0 &\leftarrow R_0 - R_2 \mod n \end{aligned}$$

Only operations $i \leftarrow i$ and $j \leftarrow 0$, appearing in some instances of Alg. **??** and **??** patterns, have not been protected for readability reasons. It is easy to fix these points: perform $i \leftarrow i \pm M_{j,\cdot} + \alpha$ instead of $i \leftarrow i \pm M_{j,\cdot}$ in Alg. **??** and **??** and add a step $i \leftarrow i - \alpha$ in the loop. The $j \leftarrow d_i(1 + \dots)$ operation should be protected in the same manner. In the end, our algorithms are immune to C safe-error attacks.

Further work may focus on implementing on our algorithms the *infective computation* strategy presented by Schmidt et al. in [**?**] in order to counterfeit the combined attacks.

9.4 Practical Results

In this section, we briefly present practical implementation results of the non-parallelized square always algorithm. As discussed in Section **??** we focused the right-to-left version.

We implemented this algorithm and the Montgomery ladder on an Atmel AT90SC smart card chip. This component is provided with an 8-bit AVR core and the AdvX coprocessor dedicated to long integer arithmetic. We used the Barrett reduction [**?**] to implement modular arithmetic.

Table 9.1: Comparison of the expected cost of SPA protected exponentiation algorithms (including the multiply always which is not immune to the attack from [?])

Algorithm	General cost	$S/M = 1$	$S/M = 0.8$	# registers
Multiply always (Alg. **??**)	$1.5M$	$1.5M$	$1.5M$	2
Montgomery ladder (Alg. **??**)	$1M + 1S$	$2M$	$1.8M$	2
L.-to-r. Square always (Alg. **??**)	$2S$	$2M$	$\mathbf{1.6M}$	4
R.-to-l. Square always (Alg. **??**)	$2S$	$2M$	$\mathbf{1.6M}$	3

We present in Table **??** the memory (code and RAM) and timing figures obtained with the chip and the AdvX running at 30 MHz. The observed speed-up of the square always algorithm over the Montgomery ladder is 5% on average. This is less than the predicted 11% but the difference can be explained by the neglected operations of the atomic pattern. Keep in mind that such results highly depend on the considered device and its hardware capabilities.

We performed careful SPA on both implementations and observed no leakage on power traces.

9.5 Conclusion

In this paper we show that trading multiplications for squarings in an exponentiation scheme together with the atomicity principle provides a new countermeasure against side-channel attacks aimed at distinguishing squarings from multiplications. Moreover, this countermeasure is intrinsically more secure against such analysis than the classical multiply always atomic algorithm with exponent blinding, and provides better performances and flexibility towards space/time trade-offs than regular algorithms such as the Montgomery ladder or the square-and-multiply always.

Table 9.2: On chip comparison of the Montgomery ladder and square always algorithms

Algorithm	Key Length (b)	Code Size (B)	RAM used (B)	Timings (ms)
Montgomery ladder (Alg. **??**)	512	360	128	30
	1024	360	256	200
	2048	360	512	1840
Square Always (Alg. **??**)	512	510	192	28
	1024	510	384	190
	2048	510	768	1740

Conclusion

We have presented in this manuscript some new side-channel and fault injection attacks. We also introduced new efficient methods to generate provable prime numbers in devices like smart cards and to compute exponentiation by only applying squaring operations. We have tested in practice these results on devices.

In the chapter 3 we have first discussed the security of embedded exponentiations methods with regards to the simple side-channel analysis. We showed that depending on the hardware device used, especially the hardware multiplier, some of the common countermeasures can be defeated by side-channel analysis. There is then a strong interest to characterize the hardware multiplier used for such implementations. It is then preferable to apply right-to-left exponentiations instead of left-to-right methods.

In chapter 4 we have introduced a new classification of attacks by using the terms *vertical* and *horizontal* attacks. We have presented two new horizontal attacks that can defeat some exponentiation implementations and compared these methods with the other existing attacks like the Big Mac attack. We also presented some countermeasures to thwart these new threats.

In chapter 5 we have presented a fault attack that defeat the Schmidt et al. combined attack resistant implementation. We also presented some combined attack that threaten this algorithm. Finally we concluded this chapter by proposing an improved version of the Schmidt et al. algorithm resistant to the attacks we presented.

Chapter 6 deals with two new collision correlation attacks defeating some first order protected implementation of the AES algorithm. We have tested these attack in practice and showed their practical efficiency.

In chapter 7 we introduced a new attack that combines fault injection with differential side-channel analysis to defeat a first order implementation of the AES. We also discussed the countermeasures to be implemented.

Chapter 8 discussed efficient embedded generation of provable prime numbers. We showed that it is possible to implement in product like smart cards efficient provable generation methods. We also showed that it can be more efficient than the classical probabilistic methods present in many standards and products. We also gave advices to implement our methods with regards to side-channel resistance.

An original method to compute exponentiation is presented in chapter 9. In opposition to the atomic multiply always exponentiation that only makes use of multiplication operation we have presented atomic square always algorithm. These new methods are

efficient and offer more security than the multiply always exponentiation against the attacks that consist in distinguishing squaring from multiplying operations.

Imagination is more important than knowledge.
A. Einstein.

Tout le monde savait que c'était impossible.
Est arrivé un qui ne le savait pas et qui l'a fait.
Marcel Pagnol.

Mouuuuuh.
Anonyme.

Publications

Conference Articles

[1] J.-C. Courrege, B. Feix, and M. Roussellet. Simple Power Analysis on Exponentiation Revisited. In D. Gollman and J.-L. Lanet, editors, *Ninth Smart Card Research and Advanced Application IFIP Conference - CARDIS 2010*, volume 6035 of *Lecture Notes in Computer Science*, pages 65–79. Springer, 2010.

[2] Christophe Clavier, Benoit Feix, Georges Gagnerot, Mylène Roussellet, and Vincent Verneuil. Horizontal correlation analysis on exponentiation. In Miguel Soriano, Sihan Qing, and Javier López, editors, *ICICS*, volume 6476 of *Lecture Notes in Computer Science*, pages 46–61. Springer, 2010.

[3] C. Clavier, B. Feix, G. Gagnerot, and M. Roussellet. Passive and active combined attacks on aes, combining fault attacks and side channel analysis. In Luca Breveglieri, Marc Joye, Israel Koren, David Naccache, and Ingrid Verbauwhede, editors, *FDTC*, pages 10–19. IEEE Computer Society, 2010.

[4] C. Clavier, B. Feix, G. Gagnerot, M. Roussellet, and V. Verneuil. Improved collision-correlation power analysis on first order protected aes. In Bart Preneel and Tsuyoshi Takagi, editors, *Cryptographic Hardware and Embedded Systems - CHES 2011*, volume 6917 of *Lecture Notes in Computer Science*, pages 49–62. Springer, 2011.

[5] C. Clavier, B. Feix, G. Gagnerot, M. Roussellet, and V. Verneuil. Square always exponentiation. In Daniel J. Bernstein and Sanjit Chatterjee, editors, *INDOCRYPT 2011*, volume 7107 of *Lecture Notes in Computer Science*, pages 40–57. Springer, 2011.

[6] C. Clavier, B. Feix, C. Giraud, G. Gagnerot, M. Roussellet, and V. Verneuil. Rosetta for single trace analysis. In Steven D. Galbraith and Mridul Nandi, editors, *INDOCRYPT*, volume 7668 of *Lecture Notes in Computer Science*, pages 140–155. Springer, 2012.

[7] C. Clavier, B. Feix, L. Thierry, and P. Paillier. Generating provable primes efficiently on embedded devices. In Marc Fischlin, Johannes Buch-

mann, and Mark Manulis, editors, *Public Key Cryptography*, volume 7293 of *Lecture Notes in Computer Science*, pages 372–389. Springer, 2012.

[8] B. Feix and A. Venelli. Defeating with fault injection a combined attack resistant exponentiation. In Emmanuel Prouff, editor, *COSADE*, volume 7864 of *Lecture Notes in Computer Science*, pages 32–45. Springer, 2013.

[9] C. Clavier and B. Feix. Updated recommendations for blinded exponentiation vs. single trace analysis. In Emmanuel Prouff, editor, *COSADE*, volume 7864 of *Lecture Notes in Computer Science*, pages 80–98. Springer, 2013.

[10] B. Feix, and V. Verneuil. There is something about m-ary - Fixed-point scalar multiplication protected against physical attacks. In Goutam Paul and Serge Vaudenay, editors, *INDOCRYPT*, 2013, to appear.

Journal Article

[11] B. Feix. Des bonnes pratiques de programmation de la cryptographie contre les attaques par canaux cachés. *MISC Magazine H.S. 6*, Novembre-Decembre, 2012.

Patents

[12] C. Clavier, B. Feix, G. Gagnerot, M. Roussellet and V. Verneuil. Integrated circuit protected against horizontal side-channel analysis. US2010/07500953.

[13] B. Feix, G. Gagnerot, M. Roussellet and V. Verneuil. Process for testing the resistance of an integrated circuit to a side-channel analysis. US2010/07500846.

[14] B. Feix, S. Nérot, G. Chew and B. Vian. Secure integrated circuit comprising divulgation means of mask countermeasures values. EP2249509A1.

[15] C. Clavier, B. Feix, G. Gagnerot, M. Roussellet and V. Verneuil. Encryption method comprising an exponentiation operation. US2012/0221618A1.

[16] C. Clavier, B. Feix, L. Thierry, and P. Paillier. Provable prime number generation method adapted to smart cards. WO/2013088065/A1

[17] B. Feix and G. Gagnerot. Microprocessor protected from memory dumps. EP/2565810/A1

[18] B. Feix and M. Roussellet. Encryption process protected against side-channel attacks. FR20120050272

[19] C. Clavier, B. Feix, and V. Verneuil. (to be published)

Workshops and Invited Talk Presentations

[20] B. Feix. Design and security of cryptographic algorithms and devices. ECRYPT II workshop. 2011.

[21] B. Feix. Security in embedded public key cryptography. e-Smart 2011.

[22] B. Feix with R. Duclos, G. Gagnerot, S. Nérot, M. Saisse and J. Vasseur. Designing an up-to-date efficient secure platform need hardware and software cohesion. e-Smart 2010.

[23] B. Feix. What's up in PACA - passive and active combined attacks. PASTIS 2010 workshop.