Side-Channel Analysis: Countermeasures and Application to Embedded Systems Debugging

by

Carlos Moreno

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

Waterloo, Ontario, Canada, 2013

Author's Declaration

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

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

Abstract

Side-Channel Analysis plays an important role in cryptology, as it represents an important class of attacks against cryptographic implementations, especially in the context of embedded systems such as hand-held mobile devices, smart cards, RFID tags, etc. These types of attacks bypass any intrinsic mathematical security of the cryptographic algorithm or protocol by exploiting observable side-effects of the execution of the cryptographic operation that may exhibit some relationship with the internal (secret) parameters in the device. Two of the main types of side-channel attacks are timing attacks or timing analysis, where the relationship between the execution time and secret parameters is exploited; and power analysis, which exploits the relationship between power consumption and the operations being executed by a processor as well as the data that these operations work with. For power analysis, two main types have been proposed: simple power analysis (SPA) which relies on direct observation on a single measurement, and differential power analysis (DPA), which uses multiple measurements combined with statistical processing to extract information from the small variations in power consumption correlated to the data.

In this thesis, we propose several countermeasures to these types of attacks, with the main themes being timing analysis and SPA. In addition to these themes, one of our contributions expands upon the ideas behind SPA to present a constructive use of these techniques in the context of embedded systems debugging.

In our first contribution, we present a countermeasure against timing attacks where an optimized form of idle-wait is proposed with the goal of making the observable decryption time constant for most operations while maintaining the overhead to a minimum. We show that not only we reduce the overhead in terms of execution speed, but also the computational cost of the countermeasure, which represents a considerable advantage in the context of devices relying on battery power, where reduced computations translates into lower power consumption and thus increased battery life. This is indeed one of the important themes for all of the contributions related to countermeasures to side-channel attacks.

Our second and third contributions focus on power analysis; specifically, SPA. We address the issue of straightforward implementations of binary exponentiation algorithms (or scalar multiplication, in the context of elliptic curve cryptography) making a cryptographic system vulnerable to SPA. Solutions previously proposed introduce a considerable performance penalty. We propose a new method, namely Square-and-Buffered-Multiplications (SABM), that implements an SPA-resistant binary exponentiation exhibiting optimal execution time at the cost of a small amount of storage — $O(\sqrt{\ell}\,)$, where ℓ is the bit length of the exponent. The technique is optimal in the sense that it adds SPA-resistance to an underlying binary exponentiation algorithm while introducing zero computational overhead.

We then present several new SPA-resistant algorithms that result from a novel way of combining the SABM method with an alternative binary exponentiation algorithm where the exponent is split in two halves for simultaneous processing, showing that by combining the two techniques, we can make use of signed-digit representations of the exponent to further improve performance while maintaining SPA-resistance. We also discuss the possibility of our method being implemented in a way that a certain level of resistance against DPA may be obtained.

In a related contribution, we extend these ideas used in SPA and propose a technique to non-intrusively monitor a device and trace program execution, with the intended application of assisting in the difficult task of debugging embedded systems at deployment or production stage, when standard debugging tools or auxiliary components to facilitate debugging are no longer enabled in the device. One of the important highlights of this contribution is the fact that the system works on a standard PC, capturing the power traces through the recording input of the sound card.

Acknowledgements

My first and foremost words of acknowledgement and gratitude go to my advisor, Dr. Anwar Hasan, for his continuing support and encouragement, his invaluable guidance, and his patience.

Secondly, I would like to express my gratitude to all of the instructors in the courses that I attended, especially for the courses officially registered, including my own advisor, Professor Hasan, Professor David Jao, and Professor Gordon Agnew. There are no words to describe the great value in the knowledge that you impart. For those outside our Electrical & Computer Engineering department, including Professor David Jao, Professor Ian Goldberg, Professor Alfred Menezes, and Professor Michele Mosca: it is impossible for me to describe how fortunate I was to find myself sitting in your classes — you are not only such brilliant researchers, but also amazing instructors. Thank you for the opportunity to attend your classes!

My thanks also go to my colleagues and friends for the interesting discussions and conversations — Abdulaziz Alkhoraidly, Nicolas Méloni, Ayad Barsoum, Piotr Tysowski, Peter Dawoud, Diego Aranha, and Patrick Longa: *thank you guys!*

The great experience of all the work leading to the completion of this thesis can not be disconnected from the experience in other courses and areas in which I participated as a teaching assistant or sessional instructor. In that sense, I would like to extend my words of gratitude to those who allowed me the opportunity to work with them in such capacity. First and foremost, Professor Douglas Harder for his support and guidance; also Professor William Bishop, Professor Rodolfo Pellizzoni, Professor Tiuley Alguindigue, Professor Roger Sanderson, and most recently, Professor Vijay Ganesh — it was a great honour to work with each of you! Also thanks to Professor Daniel Davison and Professor James Barby for their support and guidance.

I would also like to thank the members of my committee, Dr. Catherine Gebotys, Dr. Hiren Patel, and Dr. Alfred Menezes, for their effort and time, as well as the valuable feedback that they offered and that played an important role in improving this work.

In terms of the studies leading to the completion of this thesis, I would like to acknowledge the fruitful discussions with Dr. Ian Goldberg, who contributed with valuable ideas helping to give a better direction to the first study presented in this thesis as well as several improvements in two of the other studies. I would also like to thank Dr. Alfred Menezes for valuable feedback and comments on several of the studies, including feedback and comments on an earlier version of the manuscript corresponding to the first study presented in this thesis. As well, I would like to acknowledge the valuable feedback and suggestions by Dr. Hiren Patel that contributed to improve the quality of the most recent study presented in this thesis. My thanks of course extend to our co-author for this most recent study, Professor Sebastian Fischmeister, for

his valuable contribution, feedback, guidance, and support in terms of equipment and infrastructure — not to mention his great level of enthusiasm towards this joint work!

I would like to acknowledge the contribution of Summit Sehgal, who offered assistance with the setup and lab equipment for the preliminary tests and experimental phases of the project in the embedded systems lab. Thanks to Dr. Thomas Reidemeister as well, for his valuable assistance and discussions.

All of the works presented in this thesis were supported in part through a grant from the Natural Sciences and Engineering Research Council of Canada (NSERC), awarded to Dr. Hasan.

Table of Contents

List of Tables

List of Figures

List of Acronyms

AES	Advanced Encryption Standard
CDF	Cumulative Distribution Function
CFG	Control Flow Graph
CPU	Central Processing Unit
CRT	Chinese Remainder Theorem
DFT	Discrete Fourier Transform
DPA	Differential Power Analysis
DSP	Digital Signal Processing
ECC	Elliptic Curve Cryptography
EM	Electromagnetic
FFT	Fast Fourier Transform
IDS	Intrusion Detection System
IRQ	Interrupt Request
ISR	Interrupt Service Routine
JSF	Joint Sparse Form
k-NN	k Nearest Neighbours
LDF	Linear Discriminant Function
LSB	Least-Significant Bit
LTR	Left-to-Right (binary exponentiation)
MCU	Microcontroller Unit
MMU	Memory Management Unit
MSB	Most-Significant Bit
NAF	Non-Adjacent Form
NN	Nearest Neighbour(s)
PDF	Probability Density Function

PRNG	Pseudo-Random Number Generator
RFID	Radio-Frequency Identification
RSA	Public-key Cryptosystem named after its inventors, Rivest, Shamir, and Adleman
RTL	Right-to-Left (binary exponentiation)
SAAM	Square-and-Always-Multiply
SABM	Square-and-Buffered-Multiplications
SAM	Square-and-Multiply
SCA	Side-Channel Analysis
SD	Signed-Digits
SHA	Secure Hash Algorithm
SPA	Simple Power Analysis

Chapter 1

Introduction

In Cryptology, the art and science of secret writing, there are two fundamental areas of study and research: Cryptography and Cryptanalysis. In Cryptography, the goal is to create systems, algorithms, and protocols to provide secrecy, integrity, and authentication mechanisms, in general applicable to Information Systems and Telecommunications systems, as well as any system that stores, processes or communicates data with other systems where confidentiality and integrity of the data may be required. Examples of these are medical devices, control systems, home automation systems, smart cards, audio and video devices, etc.

These goals are in general achieved through the use of some secret information, or *key*, which is then used to perform some operations on the data for which it is believed, or assumed, that it is infeasible to perform the inverse operation without knowledge of the key, thus providing a certain intrinsic mathematical security to the method. This key can be a shared secret between the sender and the intended recipient, like in *symmetric key* cryptosystems, in which the same key is needed by the sender and the recipient. In *asymmetric key* or *public-key* cryptosystems, the key is composed of two parts, one part, the *private key* to be kept secret (and not shared, even with the legitimate party in the communication), and the other part, the *public key* to be made publicly available.

Cryptanalysis aims to overcome, or *break* the cryptographic mechanisms; that is, create techniques or mechanisms that allow us to nullify either one or all of the aspects provided by cryptographic methods; of course, it is implicit in this notion that this goal is to be achieved *without knowledge of the key*. Examples of these cryptanalytic techniques are: methods to recover the presumably "unreadable" information that results from applying an *encryption* technique; methods to forge information or tamper with existing information in the presence of cryptographic methods to enforce integrity, such as *Digital Signatures* or *Message Authentication Codes*; methods to forge or alter the origin or authorship of some information in the presence of cryptographic methods to provide authentication, such as Digital Signatures or any of the various *Origin Authentication*

techniques.

Cryptanalysis has classically attempted to break the intrinsic mathematical security of the cryptographic techniques, usually by breaking some of the premises, or exploiting a defect in implementations that fail to respect the premises on which such mathematical security is based. For example, some cryptographic methods rely on the fact that it is believed that no efficient integer factorization algorithms exist, or in any case, that none is known and it is believed that this will continue to be the case for the foreseeable future. Thus, it can be assumed that no adversary will be able to factor large numbers with large factors. However, if a cryptographic algorithm erroneously uses small numbers, or numbers with small factors, then no efficient algorithm is needed — a cryptanalyst can easily exploit this defect in the implementation, trying all numbers for divisibility, until they obtain the factorization.

Cryptanalysts have started to use more "outside the box" approaches, where they attempt to get around the intrinsic mathematical security, instead of breaking it or breaking some of the premises. Though many of these techniques rely on complex mathematical components to make the technique successful, they still attack the implementation directly, rather than attacking the abstract/mathematical aspects of the design.

1.1 Side-Channel Analysis

One of the most dramatic examples of this class of cryptanalytic techniques, where the implementation is more directly attacked, is the field of *side-channel analysis* or *side-channel attacks*. These types of attacks aim to entirely bypass the intrinsic mathematical security of a cryptographic system, algorithm, or protocol, by observing side-effects of the cryptosystem's implementation that may expose some correlation with internal secret parameters. Cryptographic systems are almost always implemented through electronic circuits (often on general-purpose computers), and thus, observable physical phenomena or observable parameters of the process occur as a side-effect of executing the cryptographic algorithm or protocol. If these observable phenomena or parameters have some correlation with the data — which is to be expected from a normal implementation on a normal electronic circuit — then such correlation constitutes leaked information that could be used to recover secret data and thus break the system.

A typical example is *timing attacks*, in which measurement of the time it takes to complete a decryption operation can lead to recovering the secret key, thus completely breaking the security of the cryptosystem [47]. *Power analysis* is another typical example, usually applied to smart cards and other embedded devices; in this case, the attack relies on the correlation between the data and operations the processor is working with and the power the device consumes — having physical access to the cryptographic device (which is the case with smart cards, hand-held devices, etc.), an attacker can measure

power consumption and recover the secret parameters of the cryptosystem [48]. A similar technique uses electromagnetic (EM) emissions to exploit its correlation with the operations the processor is executing and the data on which it is operating [2, 29, 71]. For both techniques, the statistical characteristics of the noise has been used to recover secret data through the use of statistical detection and estimation techniques [10, 72, 55, 4]. The simultaneous use of multiple side-channels has been also proposed as a means to increase the efficacy of the attacks [3].

Much research work has been done both in devising side-channel attack techniques as well as devising countermeasures — that is, techniques to implement cryptosystems in such a way that makes them resistant to side-channel attacks (see [26] and [25] for a recent survey of existing techniques). Paul Kocher pioneered this area of research, presenting timing attacks [47], power analysis [48], as well as some fundamental countermeasures. Schindler presented a more specific timing attack [74], later refined and demonstrated by Brumley and Boneh [8], presenting a timing attack on OpenSSL [23], a real-world cryptographic system that is widely used for Internet applications — they showed that these attacks can be effective even when applied to a remote system over the Internet. They also presented several countermeasures and showed that these countermeasures are effective against the presented attacks.

A more general form of side-channel analysis has been introduced in the recent years, based on evaluating the *mutual information* between ciphertext bits and the measurement through the side-channel [33]; the study shows that the mutual information reaches a peak for the correct guess of the key (more specifically, a block of key bits of manageable size). This class of side-channel attacks shows to be very powerful, in that it assumes little knowledge about the implementation or a detailed model of the side-channel through which information is being leaked. However, for cases where a model is known, or at least it is known that a linear relationship exists between data and leaked information, it has been shown that correlation analysis provides a more efficient attack mechanism [59, 80].

In terms of countermeasures, one obvious approach would be to entirely avoid any side-channels; for example, one could shield or isolate the electronic circuits from the outside environment in every conceivable way. If the circuit is physically contained in, say, a metallic box or some shielding material, then the EM side-channel would be closed, or at least hidden below noise level to a point that makes attacks extremely difficult; empirical evidence shows that this may be the case with modern devices, as suggested by Gebotys [31]. However, the effectiveness and resolution of attacks has also increased (see for example [53] and [42]), making the argument less convincing as time progresses. A large capacitor or internal battery could be used to "buffer" power consumption and thus avoid any correlation with the internal operations of the processor, effectively closing this side-channel.

In [48], Kocher et al. suggest the reduction of leaked signals through measures like increased bus sizes, attempting to make Hamming weights for operands constant through-

out the execution, and balancing changes from bit values. They somewhat dismiss these measures arguing that it can be extremely complex to implement them and ensuring that they are and remain effective. Gebotys [30] showed that instruction-level parallelism and large bus sizes in modern embedded processors (2004) are indeed helpful in making DPA attacks increasingly difficult. Still, this does not entirely negate Kocher et al. previous claim.

Work has also been done both in developing techniques to design hardware in a way that it reduces leaked signals (mostly EM emissions and power consumption) and in developing attacks that circumvent these countermeasures [11, 49, 57, 52, 32]. The main disadvantage with these approaches is that they would restrict cryptographic systems to be implemented on specially designed hardware, instead of on general-purpose computers or standard electronic circuits (e.g., FPGA).

We can observe that an important limitation with these engineering/technological countermeasures is the fact that one has to *very carefully* design each new cryptosystem, or each new implementation of a cryptosystem, to ensure that these countermeasures are correctly implemented and properly operational. Also, there is the fact that these countermeasures are specifically designed for known and specific classes of side-channel attacks; this means that systems could be vulnerable to other side-channels, perhaps not known at the time of implementation of the system.

An immediately obvious challenge related to this area is that of devising algorithms to implement cryptographic protocols or cryptographic primitives in ways that are secure in the presence of these types of attacks — that is, assuming that the side-channel is present and accessible to an attacker. This is clearly a challenge, as side-effects of the execution are inevitable and, as discussed above, very difficult to isolate through engineering/technological means. Thus, a secure implementation must ensure that these measurable side-effects occur either in identical ways regardless of the values of the secret data or in ways that are unrelated to the data.

Since some of the vulnerabilities to side-channel attacks arise from data-dependent optimizations in the algorithms, an obvious way to implement them in a secure way is to remove those optimizations. Thus, an additional challenge arises — devising implementations that are both efficient and secure against side-channel attacks. This is the main theme in the contributions presented in this thesis, which we now present.

1.2 Our Contributions

The contributions presented in this thesis relate to side-channel analysis from two distinct perspectives: three of the contributions attempt to improve upon existing countermeasures against side-channel attacks; specifically, presenting faster or more computationally efficient (and thus more power-efficient) countermeasures with respect to prior work. As

a related contribution, we present a novel approach for non-intrusive program tracing in which some of the main ideas behind power analysis are applied in a different context and for a different purpose.

We briefly describe these four contributions:

- We proposed and evaluated an efficient countermeasure against timing attacks through idle-wait after the cryptographic operation is completed, to hide any useful timing patterns that an attack could exploit. The main aspect of this contribution is that the idle-wait is adaptive, with the goal of reducing the overhead introduced by the countermeasure. We also explore some aspects that have been overlooked in previous studies related to this technique.

- We proposed an efficient exponentiation technique that is resistant to some forms of power analysis, at the cost of a small amount of storage. The main idea in this technique can be combined with several underlying exponentiation algorithms, adding resistance to power analysis while introducing zero computational overhead. As part of the study, the technique was combined with alternative exponentiation algorithms, showing that we can further improve the efficiency of the method in terms of computational cost.

- In a follow-up contribution, we further improved our power analysis countermeasure by combining it with a modified form of an existing exponentiation algorithm (the modification being part of our contributions for this work), which in turn led to several new algorithms resistant to power analysis while exhibiting increased computational efficiency. This modification that we proposed is only feasible when combining that method with our technique, and it is not applicable to the method in its original form.

- We proposed, implemented and evaluated a novel approach for non-intrusive program tracing of embedded devices through side-channel analysis. This technique has obvious applications in the area of embedded systems, as it can assist in the task of debugging at advanced stages of development or even after deployment. It can also have potential applications in the context of embedded systems security; in particular, it can increase the efficiency of existing attacks. It could also have applications as a monitoring system that would detect anomalies in the execution, thus acting as an intrusion detection system (IDS) [7] for embedded devices. This approach would have an important advantage over existing IDS in that it is external to the system being monitored, making it a tamper-proof device from the point of view of remote attacks that operate by injecting unwanted code after the system is operating.

1.3 Organization of This Thesis

The rest of this thesis proceeds as follows: we first present an overview of the mathematical background and existing techniques related to the contributions presented. That is, the background material presented in Chapter 2 covers all of the contributions. We do not emphasize any limitations on existing techniques as part of Chapter 2.

Then, we include one chapter for each of the contributions being presented. Each of these chapters is organized as follows: we first discuss any limitations on existing techniques as well as any other aspects that justify or motivate the work being presented in that chapter. We then present a summary of the contributions presented in that chapter, including the "main" contribution and any secondary or minor contributions that may be part of the work. After that, we present the details of the work presented in the chapter. Depending on the particular chapter, this may include analytical derivations, proofs, experimental setup, etc. Each chapter closes with a brief discussion and concluding remarks related to the work presented in that chapter.

In Chapter 7, we present a more general discussion related to all of the contributions. As well, we discuss any future work that we believe could derive from our contributions or that we may consider necessary as follow-up work that may confirm or expand the value of our contributions. Finally, we present some conclusions and final thoughts.

Chapter 2

Background

This chapter presents an overview of the mathematical background and existing techniques related to the contributions that we present in this thesis. We will not focus on the limitations or possible improvements to existing techniques, as these will be addressed in each chapter, as part of the justification or motivation for the work being presented.

2.1 Cryptographic Primitives

Historically, cryptography has been seen as a mechanism to communicate in secrecy. Encrypting phrases or numbers used to be the conventional idea of what one can achieve through the use of cryptography. Modern cryptography, however, deals with a more complex landscape in terms of applicability (e.g., telecommunications, electronic commerce, ubiquity of computing devices) and available mechanisms (e.g., electronic circuits and digital computers). Modern cryptographic techniques exist that deal with the goals of secrecy or confidentiality, data integrity, authentication, and non-repudiability. The following cryptographic primitives, or combinations of them, provide techniques addressing the above goals:

- Encryption (and its complementary operation, decryption)
- Cryptographic Hash functions or One-Way Functions
- Message Authentication Codes
- Digital signatures

These cryptographic primitives, at least when used in a practical context requiring security guarantees, typically rely on two fundamental components: a *cryptographic key*,

typically a secret on which the security of the system relies; and a source of random or pseudo-random data. Depending on the application, pseudo-random data may suffice, but usually, we see the notion of *cryptographic quality* pseudo-random numbers generator (PRNG), referring to strict requirements in terms of unpredictability of these pseudo-random values as well as a strict lack of any discernible patterns in the sequence [81].

We will focus on encryption, and in particular, as we will discuss shortly, on *public-key* or *asymmetric* encryption, since this is the area that is relevant to the contributions that we present in this thesis. The interested reader may consult comprehensive references such as [56], [66], or [78] for a more complete discussion on cryptographic primitives and their uses.

Encryption provides secrecy or confidentiality in a context where one entity, the *sender*, has some data that needs to be made accessible — sent — to another entity, the *recipient*. Encryption works by applying a transformation to the input data, the *plaintext*, to obtain the *ciphertext*. This transformation has the following properties:

- It is usually mathematical in nature, and almost always based on binary representation for any data, since modern cryptographic tools are intended for implementation on computers or digital circuits.
- It involves the use of the cryptographic key. This could be seen as the transformation being a function of two arguments, namely, the plaintext and the key; or it could be seen as a family of functions of a single input argument (the plaintext) where each value of the key defines one of the functions.
- The output data or ciphertext should be "gibberish" (more formally, we could say that it is a stream of data that should be indistinguishable from the output of a source of random data) for anyone that does not possess the key. Putting aside the sender, the legitimate recipient is in principle the only entity that possesses the key.
- A related transformation, namely the *decryption*, is similarly defined. The recipient uses this operation to transform the ciphertext back into the plaintext.

One important classification for encryption techniques relates to the keys used for encryption and decryption. In *symmetric key* encryption, the same key is required for both operations; thus, the sender and recipient must possess a shared secret, a piece of data that no-one else has access to. The sender uses the key to encrypt the plaintext, and the recipient uses the same key to decrypt the resulting ciphertext.

In *asymmetric key* or *public-key* cryptography, one key is used for encryption and a different key is used for decryption, avoiding the requirement of a shared secret between sender and recipient. We will focus on providing background for this technique, as our contributions relate specifically to it. We present and discuss the details in the next section.

2.2 Public-Key and Elliptic Curve Cryptography

This section presents an overview of public-key cryptography and some of the important algorithms that are used in practice.

2.2.1 Diffie-Hellman Key Exchange Protocol

In their influential 1976 paper *New Directions in Cryptography* [19], Diffie and Hellman introduced the notion of *public-key cryptography*, a methodology that allows two parties to communicate securely (i.e., using encryption) without having had any prior contact (e.g., to arrange on a secret encryption key to be used for the communication) and without access to an alternate secure channel (possibly with reduced capacity, but enough to securely transmit an encryption key).

The idea is that a cryptographic key is composed of two parts: an encryption or *public* key, which can be (and in principle is) made publicly accessible, and a decryption or *private* key, which must be kept secret by the recipient of the encrypted communications. Data encrypted with the encryption or public key can be decrypted using the corresponding private key, and *only* with the corresponding private key. Clearly, the two components of the key must be related, but this relationship must be such that given the public key, it must be infeasible to efficiently determine the private key.

Though they did not provide any concrete method that implements this idea of public-key cryptography, they did propose a methodology that allows two parties to securely exchange a piece of information that can then be used as the encryption key with any symmetric key cryptosystem, thus achieving the actual goal of allowing secure communication between parties that have not had any prior contact and without access to an alternate secure channel.

The protocol is defined as follows: Let G be a cyclic group with multiplicative notation, and let g be a generator of G. Parties $\mathcal{A}$ and $\mathcal{B}$ would execute the following protocol, namely the *Diffie-Hellman key exchange protocol*, for the purpose of possession of a shared secret:

- $\mathcal{A}$ randomly chooses value a $(0 \leqslant a < |G|)$
- $\mathcal{A}$ computes g^a and transmits it to $\mathcal{B}$ (possibly/presumably over an insecure channel)
- $\mathcal{B}$ randomly chooses value b $(0 \leqslant b < |G|)$
- $\mathcal{B}$ computes g^b and transmits it to $\mathcal{A}$ (possibly/presumably over an insecure channel)

- Upon reception of $\mathcal{A}$'s transmission, $\mathcal{B}$ computes $(g^a)^b = g^{ab}$
- Upon reception of $\mathcal{B}$'s transmission, $\mathcal{A}$ computes $(g^b)^a = g^{ba} = g^{ab}$
- Shared secret is the element $g^{ab} \in G$

For the protocol to be secure, an attacker that observes the values g^a and g^b must not be able to determine g^{ab}. This implies that the attacker must not be able to determine a or b.[1] The security of the protocol therefore relies on the difficulty to solve the discrete logarithm problem for the group G. That is, given the values of $g \in G$ and g^x with $0 \leqslant x < |G|$, no efficient method is known to obtain the value of x. Of course, "efficient" in this context is relative to the magnitude of the values (in particular, the order of the group, $|G|$), and relates to the issue that if $|G|$ is sufficiently small, then the value of x can be easily obtained by trying all possible values in the range (0 to $|G|$). And again, the notion of "sufficiently small" is relative to the computing power available to an adversary — a factor that changes over time. Algorithms to solve the discrete logarithm problem exist that are more efficient than *brute force* search for the solution, but they are still inefficient enough that the above protocol is considered secure. The details are beyond the scope of this thesis. The interested reader may consult [56] for more information.

In a "classical" implementation, the group G would be the set $\mathbb{Z}_p\backslash\{0\}$, where p is a large prime number, and the group's operation $\cdot$ is multiplication modulo p — that is, $a \cdot b \triangleq a \times b \bmod p$, where the symbol $\times$ denotes integer multiplication, and the notation $x \bmod p$ refers to the least nonnegative integer value x_p such that $x = x_p + kp$ for some integer k (in practical terms, $x \bmod p$ is the remainder of the division $x \div p$ — a value between 0 and $p-1$). This "classical" form of the protocol, thus, involves computing (modular) exponentiations with large exponents — a common theme among public-key cryptosystems, as we will see from the next few sections.

Additional measures are needed to make the protocol secure against a variety of other attacks and threats, but these are beyond the scope of this thesis.

2.2.2 RSA Public-Key Cryptosystem

RSA, proposed by Rivest, Shamir and Adleman in 1977 and published in 1978 [73] was the first example of a public-key cryptosystem proposed in the open scientific community. The security of this cryptosystem relies on the hardness of factoring large numbers. More specifically, numbers with prime factorization consisting of only large numbers, such as the product of two large primes.

[1] The careful reader may observe that this is a necessary but not a sufficient condition — one could conceivably determine g^{ab} from g, g^a, and g^b through some other means not involving determining the values of a and b. However, it turns out that no efficient method is known to solve this problem either.

The RSA cryptosystem is defined as follows: Let p and q be two large prime numbers, and let $m = pq$. Let $\phi(m)$ be the Euler function for m, which in this case is given by $\phi(m) = (p-1)(q-1)$ [44].

Setup phase:

- Recipient chooses at random two large prime numbers p and q of a specified size (for example, 1024-bits each), computes the modulus $m = pq$, and chooses an encryption exponent e relatively prime to $\phi(m)$ (a typical choice is a small prime number, such as $2^{16} + 1 = 65537$)
- Recipient computes the decryption exponent $d = e^{-1} \pmod{\phi(m)}$
- Recipient publishes e and m — that is, the public key is the pair (e, m)
- Private key is d — in particular, recipient never reveals the values of p or q, so he/she may choose to securely discard them.

Operation phase:

- Encryption: The encryption function E for a value x, with $0 \leqslant x < m$, under public key (e, m) is given by $\mathrm{E}(x; e, m) = x^e \bmod m$
- Decryption: The decryption function D for a ciphertext y (which, if it is the result of encrypting with public key (e, m), then we have the guarantee that $0 \leqslant y < m$) is given by $\mathrm{D}(y; d, m) = y^d \bmod m$

The scheme works in that $\mathrm{D}(\mathrm{E}(x; e, m);\ d, m) = x$. We can easily verify this property:

$$\begin{aligned}
\mathrm{D}(\mathrm{E}(x; e, m); d, m) &= \mathrm{D}(x^e \bmod m;\ d, m) \\
&= (x^e \bmod m)^d \bmod m \\
&= (x^e)^d \bmod m \ = \ x^{e \cdot d} \bmod m \\
&= x^{(e \cdot d \bmod \phi(m))} \bmod m \qquad (2.1) \\
&= x \qquad (2.2)
\end{aligned}$$

Equation (2.1) follows from Euler's Theorem (a generalization of Fermat's Little Theorem), from which we know that $x^a \equiv x^b \pmod{m}$ if $a \equiv b \pmod{\phi(m)}$ [44]. The last equality follows directly by construction of the parameters, since d is chosen to be the inverse of e modulo $\phi(m)$, and thus $e \cdot d \ \equiv \ 1 \pmod{\phi(m)}$.

A commonly used optimization takes advantage of the Chinese Remainder Theorem (CRT) to implement the decryption operation as two exponentiations of half the size [44]. Indeed, given ciphertext y, we define the following auxiliary values: let $y_p = y \bmod p$,

$y_q = y \bmod q$, $d_p = d \bmod (p-1)$ and $d_q = d \bmod (q-1)$. With this, we compute $x_p = y_p^{d_p} \bmod p$ and $x_q = y_q^{d_q} \bmod q$ and obtain x using the CRT formula. Evaluating these two exponentiations is more efficient than the original (single) exponentiation since the size of the operands is smaller, and as we will see in §2.3.1, exponentiation involves multiplication and squaring operations, which have runtime above linear time (usually $O(n^2)$ or $O(n^{1.585})$)

The security of RSA relies on the difficulty to factor large numbers; given m (which is publicly available, given that it is part of the public key), if an adversary was able to factor it to obtain p and q, then they could directly compute $\phi(m)$ and thus d, entirely breaking the system.

We observe that, though an entirely different structure, with security relying on an entirely different mathematical premise, RSA also uses modular exponentiation with large exponent values — though e is in general small, d is always a large number (in the same order of m), and even with the CRT optimization, the exponents are in the order of p and q, which are by construction half the bit length of m.

2.2.3 ElGamal Public-Key Cryptosystem

Taher ElGamal [21] proposed a public key encryption scheme closely related to Diffie-Hellman's key exchange protocol. The extension was simple, but somewhat ground breaking, in that it was the first cryptosystem in which the ciphertext is random (in particular, it is not defined as a mathematical function of the plaintext, since each encryption of the same plaintext with the same public key produces a different ciphertext). Its security therefore relies on the difficulty of solving the discrete logarithm problem in the group G, like the Diffie-Hellman protocol.

The setup of the cryptosystem is as follows (using the same notation and parameters for Diffie-Hellman): Recipient chooses a random value α, which is the private key, and computes g^α, which is the public key. To encrypt plaintext x, a random value r is chosen, and the values $c_1 = g^r$ and $c_2 = x \cdot (g^\alpha)^r = x \cdot g^{\alpha r}$ are computed. The ciphertext is the pair (c_1, c_2). To decrypt, the recipient obtains $(g^{\alpha r})^{-1}$ using the private key to compute ${c_1}^\alpha$ (this step can be seen as part of a Diffie-Hellman key exchange protocol with a being r and b being the private key α), and then obtain $x = c_2 \cdot (g^{\alpha r})^{-1}$.

2.2.4 Digital Signatures

Digital signature schemes are closely related to public key cryptosystems, in that a document (seen in this context as a piece of data) is signed using a signing key, which is private, and then can be verified using a verification key, which is publicly accessible.

The operations typically involve exponentiation with large exponents, like operations with public key cryptosystems.

We omit a more detailed description of digital signatures techniques, since it is beyond the scope of this thesis. The interested reader can consult [56] or [39] for more details. We just wish to emphasize the aspect that this important class of cryptographic primitive also involves exponentiation with large exponents as a fundamental operation.

2.2.5 Elliptic Curve Cryptography

Though considered a class of cryptography on its own, elliptic curve cryptography (ECC), concurrently proposed by Koblitz and Miller [46], [58], can be seen as a particular form of public key cryptography, where the operations involved are done in the additive group of points in an elliptic curve defined over an underlying finite field.[2]

The basic group operation (point addition), defined through a special mechanism when the two operands are the same point (point doubling) on the curve E defined over the field $\mathbb{F}$ (with $\text{char}(\mathbb{F}) > 3$) as $E/\mathbb{F} \ : \ y^2 = x^3 + ax + b$ is shown below:

Given points $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ on the curve, with $P + Q = (x_3, y_3)$ and $2P = (x_4, y_4)$, we have:

$$P + Q : \qquad x_3 = \left(\frac{y_2 - y_1}{x_2 - x_1}\right)^2 - x_1 - x_2 \tag{2.3}$$

$$y_3 = \left(\frac{y_2 - y_1}{x_2 - x_1}\right)(x_1 - x_3) - y_1 \tag{2.4}$$

$$2P : \qquad x_4 = \left(\frac{3x_1^2 + a}{2y_1}\right)^2 - 2x_1 \tag{2.5}$$

$$y_4 = \left(\frac{3x_1^2 + a}{2y_1}\right)(x_1 - x_4) - y_1 \tag{2.6}$$

Since elliptic curves use additive notation for the group operation, we don't use the term exponentiation, but rather scalar multiplication as the equivalent operation. The idea and the computational aspects are essentially identical, and as we will see, the implementation issues, both on the efficiency side and on the security side (in particular physical security with respect to side-channel analysis), are essentially identical. Thus, throughout the rest of this thesis, we will refer to exponentiation in general, including the case of scalar multiplication in ECC as an equivalent operation, since the difference is simply notational.

[2] Elliptic curves can be defined over any field, but to be useful as a cryptographic primitive, a finite field has to be used.

ECC is a very attractive alternative in general, but in particular for embedded systems and in general power- and resource-constrained devices, given its shorter key sizes required for a given security level, which leads in general to reduced computations, and thus lower power consumption, as well as reduced storage requirements.

2.3 Binary Exponentiation

As discussed in the previous sections, exponentiation with large exponents — or, in the context of ECC, scalar multiplication with large scalar values — plays a crucial role in cryptographic primitives, especially in public-key and ECC cryptosystems (including digital signatures). Efficient algorithms have been proposed and are in general well studied [34]. In this section, we present some of the basic exponentiation algorithms that exploit the binary representation of the exponent to provide computationally efficient solutions.

2.3.1 Binary Exponentiation Algorithms

Two main types of algorithms have been proposed that exploit the binary representation of the exponent to provide efficient exponentiation. Both algorithms traverse the exponent bits in sequence, and execute a square operation and a conditional multiplication for each exponent bit. One of the algorithms traverses the exponent bits left-to-right (MSB to LSB) using the property that, given the result of x^a, we can easily obtain $x^{a'}$, where a' is obtained by adding bit b at the right (LSB) of the exponent a; we notice that $a' = 2a + b$, and thus

$$x^{a'} = x^{2a+b} = (x^a)^2 \cdot x^b \tag{2.7}$$

We observe that x^b can only be 1 or x, if the value of bit b is 0 or 1, respectively. This means that this term can contribute with either a multiplication by the base, or with a null operation (a multiplication by 1), and leads to the iterative square-and-multiply algorithm (or its equivalent in the context of ECC, double-and-add) shown as Algorithm 1, for an exponent of ℓ bits.

An alternative approach, traversing the exponent bits from right to left, is shown as Algorithm 2. This technique, known simply as *right-to-left binary exponentiation*, takes advantage of the property shown in Equation (2.8), for an ℓ-bits exponent e with binary representation $b_{\ell-1}\, b_{\ell-2} \cdots b_2\, b_1\, b_0$:

$$x^e = x^{\left(\sum_{i=0}^{\ell-1} b_i \cdot 2^i\right)} = x^{\left(\sum_{\substack{i=0 \\ b_i=1}}^{\ell-1} 2^i\right)} = \prod_{\substack{i=0 \\ b_i=1}}^{\ell-1} x^{(2^i)} \tag{2.8}$$

Algorithm 1: Square-and-Multiply (Left-to-Right Exponentiation)

Input: x; $e = (b_{\ell-1}b_{\ell-2} \cdots b_1 b_0)_2$
Returns: x^e

begin
  $R \leftarrow 1$;
  for each bit b_i (i from $\ell - 1$ down to 0) **do**
    $R \leftarrow R^2$;
    if $b_i = 1$ **then**
      $R \leftarrow R \times x$;
    end
  end
  return R;
end

Algorithm 2: Right-to-Left Exponentiation

Input: x; $e = (b_{\ell-1}b_{\ell-2} \cdots b_1 b_0)_2$
Returns: x^e

begin
  $S \leftarrow x$; $R \leftarrow 1$;
  for each bit b_i (i from 0 up to $\ell - 1$) **do**
    if $b_i = 1$ **then**
      $R \leftarrow R \times S$;
    end
    $S \leftarrow S^2$;
  end
  return R;
end

Although there is a potential advantage for the left-to-right version in the context of ECC (through the use of mixed additions), the right-to-left version has some potential advantages as well; Fouque and Valette [27] point out that this version is resistant to their proposed *doubling attack*, a technique that is successful against the left-to-right version. Also, as we will discuss in §4.4, the right-to-left can be easily implemented as a two-thread parallel algorithm.

2.3.2 Signed-Digit and NAF Representation of the Exponent

An important extension for both algorithms above comes from the use of signed-digit representation of the exponent [5]. Of interest to us is the case of expressing an exponent using signed digits representation as follows:

$$e = \sum_{i=0}^{\ell} d_i 2^i \qquad d_i \in \{\overline{1}, 0, 1\} \tag{2.9}$$

where, for convenience $\overline{1} \triangleq -1$ when used to denote the value of a digit.

Signed-digit representation is redundant, and thus, multiple representations for the same value can be found (for example, 111 and $100\overline{1}$ are both valid representations of the value 7). If we introduce the constraint that no two contiguous digits can be nonzero, we obtain the Non-Adjacent Form (NAF) representation, which is unique for every represented value. Furthermore, this representation may require $\ell + 1$ bits to represent an ℓ-bit binary number, but it has lowest Hamming Weight among all signed-digit representations, with one third of the digits being nonzero on average [5] (see also Lemma 5.4 and Appendix B for an alternative analysis). This aspect represents the main advantage of using NAF representation for the exponent, since nonzero digits involve conditional multiplications, and thus, a representation with lowest number of nonzero digits leads to an important reduction in the number of multiplications.

The required modification to Algorithm 2 to work with signed-digit exponent is straightforward, as can be easily seen from the signed-digit expansion:

$$e = \sum_{i=0}^{\ell-1} d_i\, 2^i = \sum_{\substack{i=0 \\ d_i=1}}^{\ell-1} 2^i - \sum_{\substack{i=0 \\ d_i=\overline{1}}}^{\ell-1} 2^i \quad \Longrightarrow \quad x^e = \left(\prod_{d_i=1} x^{(2^i)}\right) \cdot \left(\prod_{d_i=\overline{1}} x^{(2^i)}\right)^{-1} \tag{2.10}$$

Algorithm 3 shows the right-to-left algorithm with exponent in NAF representation.

For cases where the cost of computing inverses is negligible, such as the case of ECC [39], the algorithm could avoid the extra accumulator by proceeding as Algorithm 2, but multiplying by the inverse of S at each iteration where $d_i = \overline{1}$.

Algorithm 3: Right-to-Left Exponentiation with NAF Exponent

Input: x; $e = (d_{\ell-1}d_{\ell-2} \cdots d_1 d_0)_{\text{NAF}}$
Returns: x^e

begin
  $S \leftarrow x$; $R_1 \leftarrow 1$; $R_{\overline{1}} \leftarrow 1$;
  for each digit d_i (i from 0 up to $\ell - 1$) **do**
    if $d_i \neq 0$ **then**
      $R_{d_i} \leftarrow R_{d_i} \times S$;
    end
    $S \leftarrow S^2$;
  end
  return $R_1 \times (R_{\overline{1}})^{-1}$;
end

2.4 Side-Channel Analysis

The area of side-channel analysis or side-channel attacks was pioneered by Paul Kocher during the mid and late 1990s [47, 48]. Side-channel attacks aim to entirely bypass the intrinsic mathematical security of a cryptographic system, algorithm, or protocol, by observing side-effects of the cryptosystem's implementation that may expose some correlation with internal secret parameters. Cryptographic systems are almost always implemented through electronic circuits (often on general-purpose computers), and thus, observable physical phenomena or observable parameters of the process occur as a side-effect of executing the cryptographic algorithm or protocol. If these observable phenomena or parameters have some correlation with the data — which is to be expected from a normal implementation on a normal electronic circuit — then such correlation constitutes leaked information that could be used to recover secret data and thus break the system.

Side-channel attacks are particularly suitable for (though not necessarily restricted to) scenarios where attackers or potential attackers have physical access to the devices implementing the cryptographic operations. This includes mobile devices, smart cards, and in general any embedded devices that include cryptographic functionality.

Examples of these attacks (and indeed, the instances initially presented by Kocher) include timing analysis and power analysis. Timing analysis or timing attacks rely on the fact that the data with which the algorithms work affect the execution time in a way that can be exploited to recover the secret parameters from measurements of execution time. With power analysis, the relationship between the instructions being executed by a processor and the power being consumed is exploited, as well as the relationship between the data with which the processor is working and the power consumption. By exploiting this relationship, the secret parameters can be recovered from one or multiple

power traces — plots of instantaneous power consumption as a function of time. In the next sections, we look into some of the aspects for each of these types of attacks.

2.5 Timing Analysis

The main idea behind timing analysis is exploiting the relationship between the data involved in the operations and the variations in execution time. This data includes both the cryptosystem's parameters — in particular, the secret parameters — and the ciphertext being decrypted.

For simplicity, we will focus on RSA decryption operations, for two reasons: (i) in most cases, the secret parameter is the value of an exponent, so the idea is applicable to other cryptosystems as well; and (ii) RSA involves details that introduce additional vulnerabilities to timing attacks, so it is worth looking at these aspects. Depending on the particular RSA implementation, the secret parameters may include the decryption exponent d, or the modulus prime factorization, p and q. This depends mainly on the types of optimizations used — e.g., straightforward square-and-multiply vs. Montgomery modular multiplication and the CRT to execute two exponentiations modulo each of the factors [56]. All of these optimizations have been used in actual RSA implementations, and different attack techniques have been proposed for these cases [8].

In most cases, timing attacks use statistical processing on a large number of decryption operations with data controlled by the attacker; this is the case due to two main reasons: the attacker usually has no access to timing measurements of the intermediate operations, and the total amount of time is approximately the same for every decryption. By using statistical processing, the attacker can accurately measure the small variations caused by the data (the ciphertexts). A second reason is that the use of statistical techniques makes it possible for the attack to get around measurement errors and various random delays contributing to the timing of the operation that are beyond the attacker's control.

Specific attacks are designed for different implementations of the decryption operation; for the general case where modular exponentiations are implemented as a straightforward square-and-multiply, Paul Kocher describes an approach based on guessing one bit at a time, and measuring the variance in the difference between the times measured for the actual decryption operation and the operation "mimicked" by the attacker; when the bit is guessed correctly, we have one additional iteration where both systems are working with the same data, and thus, the measured variance is lower, allowing the attacker to validate the guess for each bit of the decryption exponent [47].

Schindler [75] presented a more specific timing attack, applicable to implementations that use Montgomery exponentiations combined with the CRT optimization for RSA decryptions. The attack exploits a measurable difference in the decryption time when the

ciphertext is close to a multiple of either p or q. This attack was later refined and demonstrated by Brumley and Boneh [8], presenting a successful timing attack on OpenSSL [23], a real-world cryptographic system that is widely used for Internet applications — they showed that these attacks can be effective even when applied to a remote system over the Internet.

An important common theme in these attacks is the ability on the part of the attacker to measure statistical parameters from the collected measurements of the decryption time over a somewhat long period of time, with data controlled by the attacker.

2.6 Power Analysis

With power analysis, one exploits the relationship between power consumption and the instructions that a processor executes, as well as the data on which those instructions operate. To this effect, an attacker measures current consumption (directly proportional to power consumption, since the voltage is approximately constant) as a function of time on the device during the execution of the cryptographic operation. This "plot" of power consumption as a function of time is referred to as a *power trace*. Different techniques exist, requiring either single or multiple power traces. We describe the main two types of power analysis attacks in the next sections.

2.6.1 Simple Power Analysis

Simple power analysis (SPA) relies on data-dependent optimizations at a coarse-scale level that introduce prominent features in the power traces that make a successful attack possible with a single power trace.

Perhaps among the most dramatic examples are straightforward implementations of binary exponentiation. Both exponentiation algorithms discussed in §2.3.1, in either standard binary or NAF forms, exhibit the same vulnerability to SPA: the multiplication, with a distinct and easily identifiable power consumption profile, is executed conditionally on bits of the exponent, making it possible for an attacker to recover the exponent by observing a single power trace of the device while it executes the exponentiation [48]. We recall from the previous sections that in public-key cryptosystems, the exponent is a parameter that must be kept secret to ensure the security of the system.

We observe that the use of NAF does not eliminate the vulnerability to SPA; though a power trace only allows the attacker to distinguish nonzero digits, without knowing whether they are $\overline{1}$ or 1, the fact that only one third of the exponent bits are nonzero on average means that the attack is slowed down, but the vulnerability is not necessarily eliminated.

2.6.2 Differential Power Analysis

Though the contributions presented in this thesis relate mostly to SPA, one of our proposed techniques, an SPA-resistant algorithm (presented in Chapter 4), exhibits several aspects that could make it suitable as a countermeasure against differential power analysis (DPA). For this reason, we present a brief overview of this technique.

DPA aims at exploiting the relationship between the data that an instruction operates with and the power being consumed by the processor. The additional difficulty is given by the fact that differences in data produce very small variations in the power consumption, almost inevitably below the level of measurement noise, and thus impossible to detect from a single trace.

In DPA, multiple power traces of a cryptographic operation with the same key are combined through digital signal processing and statistical processing to get around the measurement noise — in a sense, we "average out" the effect of all other factors contributing to power consumption and the measurement noise.

To this effect, in the context of binary exponentiation where an attacker's goal is to recover the secret exponent, the attacker guesses one bit of the exponent at a time, and executes the exponentiation based on that guess; the traces are partitioned based on some binary feature of this "predicted" result, and all the traces from each partition are aligned and added; if the guess is correct, there will be a component of the traces that will exhibit correlation from trace to trace, and thus, a peak will show in the superposition of the traces; otherwise, the traces will be entirely uncorrelated, since an incorrect guess means that the data with which the two devices operate will be uncorrelated. This directly allows the attacker to validate each guess, and ultimately recover the exponent.

Countermeasures typically involve some form of randomization, such that any correlation between different traces is eliminated. This randomization could be in the data (done in a way that could be undone after decryption is complete) or in the execution, or both.

2.7 Statistical Pattern Recognition

One of the contributions presented in this thesis relies on statistical pattern recognition techniques [82], with the goal of identifying, or rather, *classifying* power traces according to a database of possible candidates, which allows us to recover the execution trace through non-intrusive measurement.

For the cases where we do not count on analytic models for the probability distribution (which is the case in the work presented in Chapter 6), we can resort to techniques based on databases of *training samples*, for which the classification is known with certainty. These training samples are in principle a set of values drawn from the probability

distribution for the process in question. Thus, they should be representative of the probability distribution of the process. The task of the classification system is described as follows: Let $\mathbf{X}$ be a random variable corresponding to a *feature vector* with features from a given sample associated with an unknown class C from a set of Q possible classes $\mathcal{C} = \{C_1, C_2, \cdots, C_Q\}$. The task of the pattern recognition system is that of obtaining an estimate of C, denoted $\hat{C}$, to which the feature vector $\mathbf{X}$ corresponds with highest a posteriori probability:

$$\hat{C} = \arg \max_{C_k \in \mathcal{C}} \{\Pr\{C_k \mid \mathbf{X}\}\} \tag{2.11}$$

Among the common techniques used to achieve this goal are Linear Discriminant Functions and Nearest Neighbors. We discuss these in the next sections.

2.7.1 Linear Discriminant Functions

With Linear Discriminant Functions (LDF), the training phase of the system collects a database of S *labelled* samples $\{\mathbf{X}_1, \mathbf{X}_2, \cdots, \mathbf{X}_S\}$ of feature vectors (the label being the class C to which the sample is *known* to correspond). For each class C_k, we compute the sample average or centroid $\overline{\mathbf{C}}_k$ as

$$\overline{\mathbf{C}}_k \;=\; \frac{1}{S_k} \sum_{\mathbf{X}_i \in C_k} \mathbf{X}_i \tag{2.12}$$

where S_k is the number of training samples labelled as C_k.

In the detection or classification phase, a given feature vector $\mathbf{X}$ is associated to the class $\hat{C}$ that corresponds to the nearest centroid (usually Euclidean distance in the multi-dimensional feature space is used):

$$\hat{C} = \arg \min_{C_k \in \mathcal{C}} \{ \left\| \mathbf{X} - \overline{\mathbf{C}}_k \right\| \} \tag{2.13}$$

To avoid expensive distance computations, each two centroids define an LDF that, when evaluated at a given point, determines the centroid to which the point is closer. The LDF corresponds to a hyperplane orthogonal to the line between the two centroids and intersecting that line at the point equidistant from the centroids, providing an efficient implementation mechanism.

2.7.2 Nearest Neighbors Rules

For the Nearest Neighbor (NN) rule, the classification phase associates a given feature vector $\mathbf{X}$ to the class of its nearest neighbor among all training samples:

$$\hat{C} = C_I \;\text{ with }\; I = \arg \min_{1 \leqslant i \leqslant S} \{ \|\mathbf{X} - \mathbf{X}_i\| \} \tag{2.14}$$

The k-Nearest Neighbors (k-NN) rule [82] provides a higher level of robustness with respect to noise in the measured features. Given a feature vector $\mathbf{X}$, we obtain the k nearest neighbors among all training samples, and the classification is done by majority vote among the k labels of these nearest neighbors. That is, if the k nearest training samples have labels $\{C_{n_1}, C_{n_2}, \cdots, C_{n_k}\}$, then feature vector $\mathbf{X}$ is associated to class C given by

$$\hat{C} = C_I \quad \text{with} \quad I = \arg\max_n \left\{ \sum_{\substack{i=1 \\ n_i=n}}^{k} 1 \right\} \tag{2.15}$$

A typical way to efficiently implement this rule is to compare *square* distances, instead of the distance itself, to avoid square root computations. Since the function $f(\cdot) = \sqrt{\cdot}$ is a strictly increasing function, comparing square distances necessarily yields the same result as comparing distances.

2.8 Spectral Analysis of Digital Signals

The contribution presented in Chapter 6 also uses digital signal processing (DSP) techniques; specifically, spectral analysis. We present a brief overview of these techniques.

One of the fundamental concepts when applying spectral analysis to digital signals is that of the Discrete Fourier Transform (DFT). Given a discrete-time signal $\mathbf{x}$ of finite duration, represented by a sequence of N real values $\mathbf{x} = \{x_0, x_1, \cdots, x_{N-1}\}$, its DFT $\boldsymbol{\mathcal{X}}$ [70] is given by the sequence of N complex values $\boldsymbol{\mathcal{X}} = \{\mathcal{X}_0, \mathcal{X}_1, \cdots, \mathcal{X}_{N-1}\}$, where each $\mathcal{X}_k$ is given by

$$\mathcal{X}_k = \sum_{n=0}^{N-1} x_n \, \mathrm{e}^{-\mathrm{j}\frac{2\pi k n}{N}} \tag{2.16}$$

where j denotes the imaginary unit [3] (i.e., $\mathrm{j}^2 = -1$)

A straightforward implementation clearly takes $\mathrm{O}(N^2)$ time to compute the DFT of a sequence of N values. In practice, Fast Fourier Transform (FFT) is normally used, being an efficient algorithm to compute the DFT. FFT exploits the symmetry in the DFT to implement an in-place divide-and-conquer [14] algorithm and obtain the DFT in $\mathrm{O}(N \log N)$ time.

When processing a long (possibly continuous/endless) sequence that is split into segments for spectral analysis, a technique known as *windowing* [70] is often used to avoid the effect of the discontinuities due to the endpoints of the sequence. This is done by

[3] We use the standard "electrical engineering" notation j for the imaginary unit, to avoid the confusion of i with the standard notation for electrical current or intensity.

multiplying each of the N values by a *window function* that smoothes the signal at the endpoints.

Many window functions are available, each with their trade-offs between benefits and drawbacks. One of the commonly used window functions is the *Hamming Window* [70], a raised cosine shape of N elements, $\boldsymbol{\mathcal{H}} = \{W_0, W_1, \ \cdots \ , W_{N-1}\}$, where each W_k is given by

$$W_k = 0.54 - 0.46 \cos\left(\frac{2\pi k}{N-1}\right) \tag{2.17}$$

With this, the processing involves computing the windowed version of the signal, $\mathbf{x}_w = \{x_0 W_0, x_1 W_1, \ \cdots \ , x_{_{N-1}} W_{N-1}\}$, and then computing the DFT of this resulting $\mathbf{x}_w$ signal.

Chapter 3

An Adaptive Idle-Wait Countermeasure Against Timing Attacks on Public-Key Cryptosystems

In this chapter, we present our first contribution, a countermeasure against timing attacks on public-key cryptosystems. This work appeared as a Technical Report in the Centre for Applied Cryptographic Research (CACR) as document CACR 2010-16 [61]. The text and contents appearing in this chapter are based on this technical report.

3.1 Motivation

Timing attacks represent an important threat against public-key cryptosystems, not only in cases where the attacker has physical access to the device holding the secret data, but also for the general case of systems with functionality accessible remotely (e.g., accessible through the Internet). This was demonstrated by the successful remote timing attack by Brumley and Boneh [8].

The usually recommended countermeasure is a technique called *blinding*, in which the ciphertext is "randomized" before being decrypted. In the case of RSA, with ciphertext y (corresponding to plaintext x), this operation is done by choosing a random number r_b and obtaining a randomized ciphertext $y' = y \cdot r_b^e$. This y' is then decrypted — we observe that any timing characteristics are now a function of y', over which the attacker has no control or even knowledge.

When decrypting y', we obtain $x' = (y')^d = (y \cdot r_b^e)^d = y^d \cdot r_b^{e \cdot d} = x \cdot r_b$. Since we chose

r_b, we compute its inverse mod m so that we can obtain x (the actual, correct result of the decryption for the supplied ciphertext).

As a slight optimization, it is suggested that the "random" values be obtained by squaring the previous ones modulo m — from the point of view of the attacker, this can not be distinguished from using true random values. Thus, r_b, r_b^e, and r_b^{-1} are precomputed, and with every decryption operation, simply square r_b^e and r_b^{-1} — the resulting squares still have the property that when r_b^e is decrypted, it produces the inverse of the other value. Even with this optimization, this countermeasure has a non-negligible performance penalty.

An alternative countermeasure consists of making the time of decryption operations independent of the data through a delay in the form of an *idle wait* after the decryption operation has been completed ([8] suggests this as a possibility, though not the preferred defense). The idea is that after completing the decryption operation, the system would not hand out the result immediately (which would reveal the duration of the decryption); instead, the system would wait an additional amount of time, so that the *total* observed decryption time is fixed, regardless of the ciphertext being decrypted. Though this technique has been suggested in the existing literature, it has not been as well studied as other countermeasures. From the point of view of performance, this idle-wait technique has a subtle additional advantage, overlooked in previous studies: being *idle* wait, as soon as the actual processing part of a decryption is completed, the processor is now free, and thus it is available for other tasks to be executed, since the idle wait allows for concurrent tasks to proceed [9]. This is clearly not the case when using blinding, in which the performance penalty comes in terms of additional actual processing that prevents other tasks from proceeding. Even in cases where multitasking is not an important factor, the fact that idle-wait countermeasures involve a lower amount of actual processing still represent an advantage in terms of power consumption, which could be an important factor on battery-powered devices.

In cases where the cryptographic subsystem is part of a hard real-time system [51], idle-wait countermeasures could be assisted by predictable timing architectures such as PRET [68], where the software can specify parameters for hardware-assisted idle-wait with high precision.

3.2 Our Contributions

In this work, we proposed and implemented an optimized form of this idle-wait countermeasure, making the duration of the idle-wait *adaptive*, with the goal of minimizing the performance penalty. As part of our study, we experimentally evaluated the effectiveness of our proposed technique in terms of performance penalty, verifying that our solution exhibits a smaller performance penalty than the usual blinding countermeasure.

Though the countermeasure is in principle valid for most public-key cryptosystems, our experimental setup is focused on RSA decryption operations.

As secondary contributions, the study includes the following:

- Straightforward implementations of idle-wait countermeasures, in any form, exhibit a subtle vulnerability that could allow attackers to entirely bypass the idle-wait by requesting concurrent operations and focus the attack on the *throughput* of decryption operations. This aspect has not been mentioned in any previous studies. As part of our work, we identify this vulnerability, describe the details of an attack that exploits it, and describe the correct way to implement idle-wait countermeasures to avoid this vulnerability.
- We present an analytical derivation for the mutual information between the data producing the leakage and the decryption time in the presence of our countermeasure, relative to the mutual information in the absence of countermeasures under the assumption of Gaussian distribution for the decryption time. This in turn shows and quantifies the decrease in mutual information introduced by our countermeasure, highlighting an important slowdown factor for attacks based on mutual information.

3.3 Adaptive Idle-Wait Countermeasure

In principle, for an idle-wait countermeasure to be effective, every decryption time must be extended (through the idle-wait) so that the total time exceeds (or at least matches) the decryption time for any other ciphertext. That is, if T_{D_i} denotes the decryption time for ciphertext $i \in \mathcal{C}_\mathcal{T}$, and T_{W_i} denotes the amount of idle-wait applied after decryption of ciphertext i, then it should hold that

$$T_{D_i} + T_{W_i} \geqslant T_{D_j} \quad \forall\, i, j \in \mathcal{C}_\mathcal{T}$$

Clearly, the use of idle-wait introduces a performance penalty (putting aside the multitasking aspect, as mentioned in §3.1). However, as this study shows, the performance penalty of idle-wait countermeasures may be comparable and even smaller than that of blinding; not necessarily for all implementations, but we show that this is the case for OpenSSL's RSA implementation.

Additional performance gain can be obtained through an optimized version of this countermeasure. The basic idea for the optimized countermeasure is to try to keep the extra delay (the idle wait) to a minimum, while still hiding any variance or useful patterns in the execution time.

Let T denote the target execution time (i.e., the total observed time, including the decryption time and the idle-wait time). The intuition is that it suffices that the target execution time T be greater than the execution time for *most* ciphertexts. Thus, we can make the parameter T *adaptive* with respect to the observed execution times of the ciphertexts, making it converge to a target value, as a function of the collected statistics of the measured decryption times. After the system has been operating for a while, this parameter will remain virtually constant. This value is specified as a target percentile, which can be an adjustable configuration parameter. This approach is described in the next section.

3.3.1 Decryption Time Controlled by Target Percentile

We now explore the alternative of making the parameter T converge to a particular percentile of the decryption time. For example, we could require that T corresponds to percentile 99, such that only 1% of ciphertexts produce a decryption time that exceeds the value of T. In other words, we set the value of T such that $F(T) = 0.99$, where $F(\cdot)$ is the Cumulative Distribution Function (CDF) of the random variable representing the actual decryption time.

Since we do not have a priori knowledge or an analytical description of the CDF of the decryption time, we use the decryption times of the requested operations to make the value of T converge to the specified target. Many approaches can be used for this, including the "brute force" solution of storing all decryption times in an ordered sequence, to choose the value T corresponding to the given percentile. This, of course, would be unacceptably inefficient in most cases.

We now present our proposed approach — a somewhat heuristic method which we believe is appropriate for this scenario, given that it is simple to implement and efficient; our results show that the method converges rather rapidly to the target percentile.

The idea is loosely based on the Newton-Raphson method for solving single-variable equations [69]; the equation in our case is $F(T) = P$, where we are trying to solve for the unknown T given the target percentile P.[1] Unlike in the Newton-Raphson method, we can not compute the value of the function or its derivative; instead, we use the numerical approximations given by the statistics collected from the decryption times. In particular, we count the total number of decryptions, and count the number of instances in which the decryption time was below our current approximation of T, denoted T_k (value of T at iteration k). This gives us an approximation for the value of $F(T_k)$. To approximate the derivative, we use two additional thresholds, closely surrounding the value of T (for example, if we have a target percentile of 99, then we could use thresholds corresponding

[1] For simplicity, whenever a percentile is needed for formulas or derivations, we use a CDF value between 0 and 1 as a percentile — as opposed to a value between 0 and 100

to percentiles 98.5 and 99.5). These thresholds are denoted H_k (*High* threshold value at iteration k) and L_k (*Low* threshold value at iteration k), and we count the number of instances in which the decryption time was below each of these thresholds, to obtain approximations for $F(L_k)$ and $F(H_k)$. The approximation for the derivative at T_k is given by the slope of the straight line between these two surrounding thresholds, with which we obtain our iterative update formula for T:

$$T_{k+1} \;=\; T_k + \frac{(F_T - \hat{F}(T_k))(H_k - L_k)}{\hat{F}(H_k) - \hat{F}(L_k)} \tag{3.1}$$

where F_T denotes the target percentile (e.g., 0.995), and $\hat{F}$ denotes the approximation for F (given the count of instances for which the decryption time has been below the given argument).

Since the thresholds H and L are also the values corresponding to given percentiles, we also work with approximations for those, and thus, we need to update them as well. The approximation for the derivative at those values is given by the straight line going from each of those points and the point at T_k, with which we obtain the iterative update formulas for these thresholds:

$$L_{k+1} \;=\; L_k + \frac{(F_{T_L} - \hat{F}(L_k))(T_k - L_k)}{\hat{F}(T_k) - \hat{F}(L_k)} \tag{3.2}$$

$$H_{k+1} \;=\; H_k + \frac{(F_{T_H} - \hat{F}(H_k))(H_k - T_k)}{\hat{F}(H_k) - \hat{F}(T_k)} \tag{3.3}$$

Like in the Newton-Raphson method, we require an initial estimate not far from the real solution to ensure convergence [69]. In our case, this estimate can be easily obtained by storing all decryption times for an initial sequence of operations and sorting them to determine the required percentile approximations. Alternatively, to avoid any storage overhead, one could assume a normal distribution and estimate the mean and variance from an initial sequence of values, simply by accumulating and counting, to avoid using any storage. By keeping the sum of the first N values t_k ($1 \leqslant k \leqslant N$) and the sum of the squares of these values we can obtain approximations for the mean $\hat{\mu}_t$ and the variance ${\hat{\sigma}_t}^2$, as can be easily shown:

$$\begin{aligned}
\hat{\mu}_t \;&=\; \frac{1}{N}\sum_{k=1}^{N} t_k \\
{\hat{\sigma}_t}^2 \;&=\; \frac{1}{N}\sum_{k=1}^{N}(t_k - \hat{\mu}_t)^2 \;=\; \frac{1}{N}\left(\sum_{k=1}^{N} t_k^2 \;+\; \sum_{k=1}^{N}{\hat{\mu}_t}^2 \;-\; 2\sum_{k=1}^{N} t_k\hat{\mu}_t\right) \\
&=\; \frac{1}{N}\left(\sum_{k=1}^{N} t_k^2 \;+\; N{\hat{\mu}_t}^2 \;-\; 2\hat{\mu}_t\sum_{k=1}^{N} t_k\right) \;=\; \frac{1}{N}\left(\sum_{k=1}^{N} t_k^2 \;-\; N{\hat{\mu}_t}^2\right)
\end{aligned}$$

From these two parameters we obtain an approximation for any required target percentile [67].

3.3.2 Resistance to Any Possible Timing Attacks

Clearly, if the target percentile is set to 100, then the countermeasure will defeat *any* timing attacks — known or otherwise. This follows directly from the fact that timing measurements in such cases only reveal a constant value added to random measurement noise, both of which are independent of the ciphertext and the decryption key. As our results show, a setup using percentile 100 is feasible at a performance penalty comparable to that incurred by blinding in OpenSSL. This result validates the idea of idle-wait as countermeasure to timing attacks, and also provides a conservative setup for our proposed method.

Better performance can be obtained if we want to defend against specific timing attacks that are known, and for which a given percentile could be sufficient to hide all the timing information that the specific attack requires. We will show an example of this scenario (§3.4.3), in which we experimentally verify that our method, with a percentile below 100, defeats the attack presented in [8].

3.3.3 Effect on Attacks Based on Mutual Information Analysis

In this section, we discuss the effect of our countermeasure on a broad class of attacks, based on Mutual Information Analysis [33, 80]. We will evaluate the decrease of mutual information between the data producing the leakage and the measurement taken through the side channel (decryption time, in this case) in the presence of our countermeasure.

Let X denote the measurement taken by the attacker with no countermeasure present, and let Y denote the data producing the leakage to the side-channel. The mutual information is given by

$$\mathrm{I}(X;Y) = \mathrm{H}(X) - \mathrm{H}(X \mid Y)$$

where $\mathrm{H}(\cdot)$ denotes the entropy of the given variable [18].

If we assume both X and $X \mid Y$ to follow normal distributions, we can obtain the associated entropies. Let $X \sim \mathcal{N}(\mu, \sigma^2)$, with probability density function denoted $\Phi(x)$.

Then,

$$\begin{aligned}
\mathrm{H}(X) &= -\int_{-\infty}^{\infty} \Phi(x) \ln \Phi(x)\, dx \\
&= -\int_{-\infty}^{\infty} \frac{1}{\sqrt{2\pi\sigma^2}}\, \mathrm{e}^{-\frac{(x-\mu)^2}{2\sigma^2}} \ln\left(\frac{1}{\sqrt{2\pi\sigma^2}}\, \mathrm{e}^{-\frac{(x-\mu)^2}{2\sigma^2}}\right) dx \\
&= \int_{-\infty}^{\infty} \frac{1}{\sqrt{2\pi\sigma^2}}\, \mathrm{e}^{-\frac{(x-\mu)^2}{2\sigma^2}} \left(\frac{1}{2}\ln\left(2\pi\sigma^2\right) + \frac{(x-\mu)^2}{2\sigma^2}\right) dx \\
&= \frac{\ln\left(2\pi\sigma^2\right)+1}{2}
\end{aligned} \tag{3.4}$$

Let X_p denote the measurement with the countermeasure present at percentile p. We evaluate its entropy; in this case, the integral is evaluated from T to ∞, where T is the threshold corresponding to percentile p, and thus $\int_T^{\infty} \Phi(x)\, dx = 1-p$

$$\begin{aligned}
\mathrm{H}(X_p) &= -\int_T^{\infty} \Phi(x) \ln \Phi(x)\, dx = \int_T^{\infty} \Phi(x) \left(\frac{1}{2}\ln\left(2\pi\sigma^2\right) + \frac{(x-\mu)^2}{2\sigma^2}\right) dx \\
&= \frac{\ln\left(2\pi\sigma^2\right)}{2} \int_T^{\infty} \Phi(x)\, dx + \frac{1}{2\sigma^2}\int_T^{\infty} (x-\mu)^2 \Phi(x)\, dx \\
&= \frac{\ln\left(2\pi\sigma^2\right)}{2}(1-p) + \frac{1}{2\sigma^2}\int_T^{\infty} (x-\mu)^2 \Phi(x)\, dx
\end{aligned} \tag{3.5}$$

We apply integration by parts to the integral in Equation (3.5), with $f(x) = (x-\mu)$ and $g'(x) = (x-\mu)\Phi(x) \;\Rightarrow\; g(x) = -\sigma^2\Phi(x)$, to obtain

$$\begin{aligned}
\int_T^{\infty} (x-\mu)^2 \Phi(x)\, dx &= -\sigma^2(x-\mu)\Phi(x)\Big|_T^{\infty} + \sigma^2 \int_T^{\infty} \Phi(x) dx \\
&= \sigma^2(T-\mu)\Phi(T) + \sigma^2(1-p)
\end{aligned} \tag{3.6}$$

Combining equations (3.4), (3.5), and (3.6), we finally obtain

$$\begin{aligned}
\mathrm{H}(X_p) &= \frac{\ln\left(2\pi\sigma^2\right)}{2}(1-p) + \frac{1}{2}(1-p) + \frac{1}{2}(T-\mu)\Phi(T) \\
&= (1-p)\mathrm{H}(X) + \underbrace{\frac{1}{2}(T-\mu)\Phi(T)}_{=\,\mathrm{O}\left(T\mathrm{e}^{-T^2}\right)} \approx (1-p)\mathrm{H}(X)
\end{aligned} \tag{3.7}$$

Following a similar analysis, we obtain $\mathrm{H}(X_p \mid Y) \approx (1-p)\mathrm{H}(X \mid Y)$; even though the threshold for the given percentile is not necessarily the same, we expect it to be

approximately the same in the general case.[2] This means that

$$\mathrm{I}(X_p; Y) \;\approx\; (1-p)\mathrm{H}(X) - (1-p)\mathrm{H}(X \mid Y) \;=\; (1-p)\mathrm{I}(X;Y) \tag{3.8}$$

That is, in the general case, the countermeasure with target percentile p leads to a reduction of the mutual information by a factor close to $1-p$. We recall that p is in principle chosen to be a value close to 1, and thus $1-p$ is a small value. This means that we can expect any attacks based on Mutual Information Analysis to be slowed down by a reasonably large factor for a typical value of p.

3.3.4 A Note on Implementing Idle-Wait Countermeasures

A naively implemented idle-wait countermeasure could be vulnerable to attacks that could entirely bypass the idle-wait. Indeed, as mentioned in §3.1, as soon as a decryption is completed, the processor is now free, and thus available for other (concurrent) tasks to proceed; if these tasks involve decryption operations requested concurrently by the attacker, then it would be possible to measure the time from the beginning of one operation to the beginning of the next operation, revealing the *actual processing* time, and thus entirely bypassing the idle-wait.

Figure 3.1 demonstrates this aspect — solid lines denote an active decryption operation (i.e., processing is taking place); dashed lines indicate an idle wait, and crosses indicate that the decryption operation is ready to begin, but can not yet, since the processor is being used by another concurrent operation.

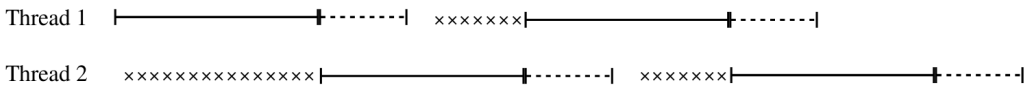


Figure 3.1: Throughput of Concurrent Decryption Operations.

We clearly see that, as long as the system is flooded with requests, such that at all times there is one decryption operation taking place (assuming a single-processor), then the sum of the frequencies of operations measured at each thread of execution is the frequency of execution of the operations, or throughput. Given N_T threads of concurrent execution, let f be the total frequency or throughput, N the total number of decryptions executed, f_i the frequency of decryption executions at thread i, N_i the

[2] Otherwise, we would be talking about a much more specific attack, and thus this analysis does not apply; for such cases, the countermeasure can still be used, adjusting the percentile to the appropriate level (see §3.4.3 for an example).

number of decryptions completed by thread i, and T_D the time it takes for all threads to complete. Then

$$f = \frac{N}{T_D} = \frac{1}{T_D}\sum_{i=1}^{N_T} N_i = \sum_{i=1}^{N_T}\frac{N_i}{T_D} = \sum_{i=1}^{N_T} f_i \tag{3.9}$$

This allows us to determine the decryption time t (the actual time of execution, not including the idle-wait stage) given the measurements t_i for each thread i (notice that t_i does include the decryption time and the idle-wait):

$$t = \frac{1}{f} = \frac{1}{\sum_{i=1}^{N_T} f_i} = \frac{1}{\sum_{i=1}^{N_T}\frac{1}{t_i}} \tag{3.10}$$

We also see that if the idle wait is long, more than two threads would be needed; specifically, the minimum number of threads is given by

$$\#Threads = \left\lceil \frac{T_P(x) + T_I(x)}{T_P(x)} \right\rceil \tag{3.11}$$

where $T_P(x)$ is the processing time (the time that it takes to complete the actual operation), and $T_I(x)$ is the idle wait time. It is straightforward for the attacker to determine this figure: an estimate of the average processing time is done locally; and $T(x) = T_P(x) + T_I(x)$ is measured by requesting decryption operations using a single thread of execution.

All of the above applies to multi-core processors as well — the minimum number of threads is now the minimum number *per processor core*. It is also straightforward to see that if more threads are used, the frequency of operations observed on each thread will be lower, but the sum of the frequencies will still be equal to the frequency of execution of operations, or throughput.

Naturally, the attack in the presence of a poorly implemented idle-wait countermeasure is slower than the original attack, since the threads scheduling introduces additional randomness in the timing of the operations, and thus, additional measurement noise. Still, it is clear that the countermeasure should be considered ineffective if it is vulnerable to this attack on the throughput.

Correct Implementation of Idle-Wait Countermeasures

The vulnerability described in the previous section can be easily avoided; a correct implementation of an idle-wait countermeasure (adaptive or otherwise) should prevent

any decryption from starting as long as any other decryption — *including its idle-wait phase* — is still in progress. This means that the throughput of (concurrent) operations is being forced to be equal to the decryption time as measured directly (i.e., including the idle-wait), which defeats the timing attack if properly adjusted.

To this end, a *mutex* or some suitable synchronization mechanism could be used [9]. An interesting aspect is that this can be done while still maintaining the advantage of leaving the processor free for other concurrent tasks to proceed [9].

3.4 Simulation Setup and Experimental Results

In this section, we present and discuss the experimental part of our study. The results are based exclusively on simulations. The reason for this is that the idle-wait timer resolution required for our proposed method is higher than currently available for typical software implementations; and a hardware implementation is somewhat overkill for the purpose of evaluating the effectiveness of our method.

3.4.1 Setup

The basic idea for the simulations is to take timing measurements by running the actual decryption operation and then updating the value of T, using the technique described in §3.3.1. We simulate the decryption followed by idle-wait, and measure the performance penalty of our proposed method. To this end, we produce a sequence of randomly generated values for the ciphertext, and measure the actual processing time for those.

The decryption operation is done through invocation of the appropriate OpenSSL API function call, with blinding disabled [23], using `/dev/urandom` as the source of randomness for the ciphertexts [81]. With this actual processing time, the simulation can now determine the amount of idle-wait necessary, and with this, the average overhead is obtained.

To optimize the simulations, the processing was split into two independent programs; one program "profiles" the execution speed — generate random ciphertexts, measure the execution time of the decryption, and store these execution times in data files. A second program now uses the stored data, thus avoiding redundant invocations of decryptions, and optimizing the process.

3.4.2 Results

The test platform was an AMD Quad-Core Phenom Processor, 64-bit at 2.5GHz, running Ubuntu Linux 8.04, with gcc/g++ 4.2.4, and OpenSSL 0.9.8g. For the profiling, all

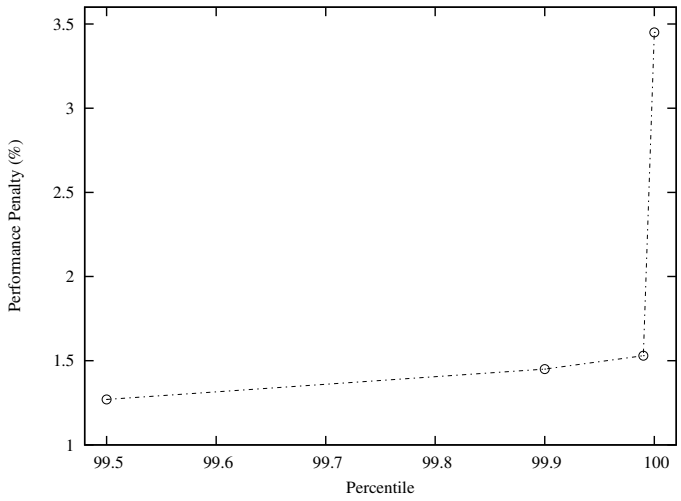


Figure 3.2: Performance Penalty (in %) of Our Proposed Method.

graphical interface and networking was shut down, to avoid disrupting the measurements. For the simulation part, this was not critical, as the processing time had been already measured, and the program simply reads the values from a data file. The simulations were done with a set of 1,000,000 (one million) values of ciphertext, with decryptions done with a 1024 bits key, and the measurements done using the CPU clock cycle counter (sub-nanosecond resolution).

Performance Penalty

The measured blinding overhead for OpenSSL, using 1024-bit keys, is 3.1%. This is consistent with the reported range of 2 to 10% [8]. For the performance penalty of our proposed method, we adjusted the target percentile between 99.5 and 100.[3] Figure 3.2 shows the results for all the simulations.

3.4.3 Resistance to Timing Attacks

Figure 3.2 shows a crucial piece of evidence in favor of our proposed method regarding resistance to timing attacks, as discussed in §3.3.2: for percentile 100, the performance penalty is only 3.45%; as already discussed, setting the target percentile at 100 defeats any possible attack based on measurement of time, since the observed decryption time would be independent of the ciphertext and the decryption key. The corresponding

[3] For percentile 100, a different method to update T was used, since the general iterative method would fail; also, obtaining the value for percentile 100 is much simpler than for the general case.

performance penalty is within the lower half of the reported range, and barely above the figure that we measured for our test platform.

It is important, however, to notice that this is the case for RSA, where the decryption exponent is fixed; since the performance penalty is clearly related to the variance of the decryption time, it is to be expected that this figure will be higher for techniques where the decryption exponent is variable, such as Diffie-Hellman. We did not include these measurements in our study. Also, timing-based cache attacks could incur a considerably large variance, requiring a high performance penalty for any idle-wait countermeasures to be effective. We did not consider this class of attack in this study.

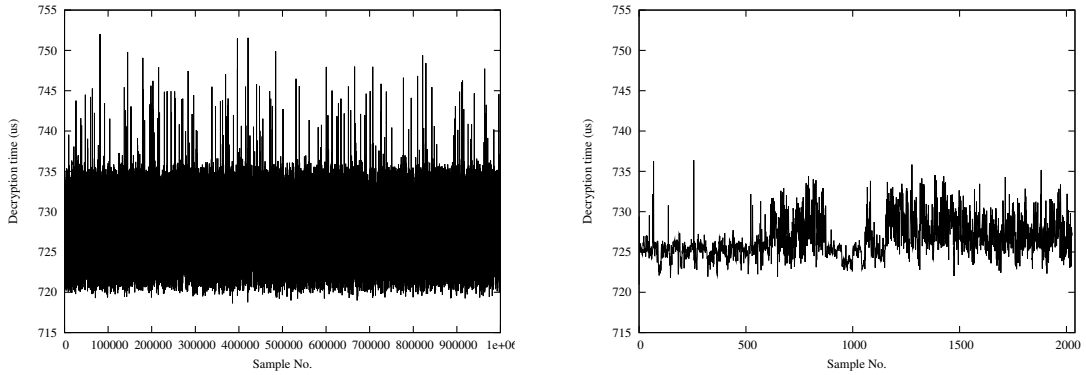
Case-Study: Resistance Against a Concrete Attack

As additional evidence in favor of our method, we evaluated the resistance against a known and effective timing attack, with the goal of demonstrating that additional efficiency can be obtained if we need to defend against known attacks only — not an unreasonable design criterion, given that the method can be easily readjusted, should new, more effective attacks were discovered after it was deployed.

The attack is that presented by Brumley and Boneh in [8]; we simulated the attack by generating all the ciphertexts that would be required (see [8] for details); since we only want to verify that all the decryption times fall within a certain percentile, we made use of the actual values of p and q, instead of executing the attack to guess the bits — clearly, a successful attack would need to use precisely these ciphertexts, as any incorrectly guessed bit would cause the attack to be ineffective for all the following bits. Thus, it makes sense and it is fair to make use of the known factorization $\{p, q\}$, even if a real attack would not have such information.

We verified that the attack is successfully defeated with a target percentile of 99.99 (with a corresponding performance penalty of 1.53%); with these settings, we get a value of $T = 736.4\mu s$, which covers the decryption times for the attack ciphertexts. Figure 3.3 shows the decryption times for the profiling data (randomly selected ciphertexts) and for the attack ciphertexts.

Even though strictly speaking, we need to cover *all* ciphertexts to guarantee that we defeat the attack, in a practical sense, the attack would be defeated with a percentile of 99.9, with a corresponding performance penalty of 1.45%; it is not clearly visible in Figure 3.3, but only two ciphertexts exceed percentile 99.9. It is reasonable to assume that an attack like this one, statistical in nature, would be defeated if only a negligible fraction of the required measurements survives the barrier imposed by the countermeasure.



(a) Random Ciphertexts

(b) Ciphertexts for Attack on OpenSSL

Figure 3.3: Decryption Times – OpenSSL API without Blinding.

3.5 Discussion and Concluding Remarks

The results from our experimental setup confirm the validity and show important aspects of the proposed technique and its implementation. We verified the technique's resistance against known attacks, and we see that it does offer good performance, in that it exhibits a performance penalty below that of the blinding countermeasure for a reasonably high security level. In the highest security setting, where we guarantee that any possible timing attack would be defeated, the performance penalty is comparable to that of blinding.

We emphasize again that even for comparable performance penalties, our method has an additional advantage, at least for software implementations: during the idle-wait phase, the processor is available for other tasks to proceed; this is not the case for blinding, in which the performance penalty is given in terms of actual processing that prevents other concurrent tasks from proceeding. This could be an important advantage on systems where multi-tasking is a requirement for the server where the cryptosystem is operating. We also highlighted the additional advantage that with less actual processing required, the system is more efficient in terms of power consumption — a potentially important issue for, e.g., battery-powered devices. Though blinding could have an advantage in cases where other side-channel attacks such as Power Analysis are a threat, for many situations, remote timing attacks may be the only plausible side-channel attack, and thus our countermeasure would be perfectly suitable.

The method was also shown to be effective in terms of security, in that it was verified that the countermeasure could defeat any possible timing attack, with a performance penalty comparable to that of blinding; it was also shown that the countermeasure defeats a known timing attack with a performance penalty considerably below that of blinding.

On a more theoretical side, we presented analytical evidence that the countermeasure

should be effective against attacks based on Mutual Information Analysis; we showed that any such attacks would be slowed down by a reasonably high factor at a reasonable performance penalty. These types of attacks would of course be completely defeated if we use percentile 100, in which case the mutual information would be reduced to *zero*.

Perhaps one important aspect that could require improvement relates to the practical issue of availability of idle-wait facilities with the required accuracy for software implementations; we observe that there is no fundamental reason why these can not be available; it is simply a matter that at the present time, they are not for the common software platforms. It is perhaps worth noting that a busy-wait countermeasure would not be unreasonable for a software implementation, which would address this issue; we could use the CPU cycle counter on modern processors — with sub-nanosecond resolution — as the basis for a busy-wait loop after the decryption operation; we sacrifice the benefit of the idle-wait in terms of multitasking performance, but we do keep in mind that the performance penalty is still below the performance penalty with blinding (or comparable, for the highest security setting). However, for software implementations with currently available idle-wait facilities — which have lower resolution than required for our method — we notice that the security level would be unaffected, since the measured time is now the required time plus a random value that is uncorrelated to the amount of idle-wait that we introduce. Performance could be negatively affected, of course; but again, this would be compensated by the multi-tasking aspect of idle-wait countermeasures.

Chapter 4

Square and Buffered Multiplications

In this chapter, we present our second contribution, an SPA-resistant exponentiation method with optimal execution time, at the cost of a small amount of storage — $O(\sqrt{\ell}\,)$, where ℓ is the bit length of the exponent. The method is optimal in the sense that it adds SPA-resistance to an underlying exponentiation algorithm with zero computational overhead. This work appeared as a printed article in the Journal of Cryptographic Engineering [63] as well as on the CACR web site, as technical report CACR 2011-03 [64]. The text and contents appearing in this chapter are based on these works.

4.1 Motivation

Many SPA-resistant algorithms have been proposed. However, they all exhibit considerable performance penalties since SPA relies on data-dependent optimizations, usually at coarse-scale level, and existing solutions remove some of these natural optimizations from the algorithm.

For example, the simplest solution to the vulnerability described in §2.6.1 for the case of binary exponentiation is to execute the multiplication *unconditionally*, and discard the result when it is not needed [15]. This way, we achieve a *constant execution path* — that is, a sequence of executed instructions that is independent of the secret data. Algorithm 4 shows an example of this technique, applied to the left-to-right algorithm, to obtain an algorithm known as *square-and-always-multiply*.

The disadvantage is obvious: a strong performance penalty is imposed on the algorithm, as a considerable number of unnecessary multiplications [1] are executed — in the case of a standard binary representation of the exponent, $\ell/2$ extra multiplications are

[1] Unnecessary from the point of view of performance, in that these multiplications are not required to obtain the correct result.

Algorithm 4: SPA-Resistant Exponentiation Algorithm.

Input: x; $e = (b_{\ell-1}b_{\ell-2} \cdots b_1 b_0)_2$
Returns: x^e

begin
  $R \leftarrow 1$;
  for each bit b_i (i from $\ell - 1$ down to 0) **do**
    $TMP[0] \leftarrow R^2$;
    $TMP[1] \leftarrow TMP[0] \times x$;
    $R \leftarrow TMP[b_i]$;
  end
  return R;
end

executed on average; if using NAF representation for the exponent, $2\ell/3$ extra multiplications are executed on average.

Marc Joye proposed a scheme that reduces this penalty [45] (later generalized in [12]), but it still exhibits a potentially high performance penalty, depending on the implementation of the underlying multiplication and squaring procedures. Indeed, the scheme proposed in [45] is based on a rearrangement of the loops that avoids operations where the result is discarded — however, this is achieved by implementing the squaring operations as a multiplication where the two operands are the same value. Depending on the implementation, there is a potential for a considerable speedup in squarings with respect to multiplication; with integer arithmetic, a factor of up to 2 (typically in the order of 1.5 in actual implementations), given that redundancy in the operands can be exploited. This potential speedup is completely unutilized in [45] and [12].

An additional problem with the scheme presented in [45] is that in the context of ECC, adding two different points and adding two points that are the same (i.e., doubling) are two different operations in practice [39], and thus, a difference is expected to be observed on power traces during the execution of scalar multiplications, making the technique less effective in the context of ECC operations. The use of Edwards curves [20] would represent an exception to this issue. However, Edwards curves are not part of any standards, and thus they are probably uncommon in actual practical applications.

Ha and Moon [37] presented a scheme resistant to DPA, and combined it with a simple SPA-resistant approach. The SPA-resistant algorithm is essentially equivalent to the square-and-always-multiply technique, and the benefit that they obtain is only observed when combined with the DPA-resistance component. Still, the efficiency that they report for an exponent in NAF representation is comparable with the efficiency when using standard binary representation for the exponent, clearly indicating that their SPA-resistant component is sub-optimal.

Sun et al. [79] proposed a novel and very ingenious scheme resistant to SPA, in which the exponent is split in two halves and blocks of two bits — one bit from each half — are combined together for processing; however, the algorithm does involve operations where the result is discarded (albeit, a smaller fraction than in the case of the square-and-always-multiply technique), which necessarily means that is not optimal. Furthermore, their method is specific to standard binary representation of the exponent, which is considerably less efficient than a solution using NAF representation; though it can be adapted to NAF, the complexity of the implementation would increase; more importantly, the fraction of operations where the result is discarded is considerably higher when using NAF representation for the exponent, making their method fundamentally incompatible with NAF. Incidentally, our proposed method can be combined with the method proposed in [79], which would increase efficiency even further; indeed, a non-SPA-resistant version of their method (one in which operations with no effect are not executed, instead of executed to then discard the result) is slightly more efficient (in terms of execution time) than the standard binary exponentiation (with exponent in either binary or NAF representation). Combining this method with our proposed method would provide resistance to SPA without any single operation where the result is discarded. We will discuss these aspects in §4.3.2. Moreover, when combining these two techniques, we can take advantage of signed-digit representations to further increase efficiency — something that can not be done with the technique by Sun et al. in its original form, as it is fundamentally incompatible with the use of signed-digit representations. We will discuss this in Chapter 5.

4.2 Our Contributions

In this work, we presented an SPA-resistant algorithm with optimal execution time, at the cost of a small amount of storage — $O(\sqrt{\ell})$, where ℓ is the bit length of the exponent. The method is optimal in the sense that it adds SPA-resistance to an underlying exponentiation algorithm with zero computational overhead. We present the method in its basic form, using right-to-left binary exponentiation as the underlying exponentiation algorithm, and then we combine it with the algorithm proposed by Sun et al., showing that we obtain better performance with respect to their algorithm in its original form, while maintaining resistance to SPA.

The scope of our contribution does not cover resistance against template attacks [10]. A simple modification to the algorithm could suffice to guarantee such resistance, but this would require further investigation, and this aspect was not covered in the study presented here.

The method is suitable for embedded devices with moderately constrained resources; since it requires a small amount of storage, it may not be suitable for highly-constrained devices such as RFID devices, and possibly for smart cards as well. For most hand-held

devices, including mobile devices, the cost of the method in terms of storage should be well within the capabilities of the device, making the method suitable for this class of device.

As secondary contributions, this work includes the following aspects:

- A two-thread parallel implementation of the algorithm is proposed; this is an interesting optimization in that it is suitable for many modern embedded processors that are multi-core, as well as desktop and server processors, which are virtually with no exception multi-core. Two threads is not too demanding, so the technique should be suitable for a wide range of available processors.
- We present a *correct* analytical derivation of the space required for a buffer to adapt an input with events occurring at random times and an output with either constant rate (the average rate) or at random, uncorrelated times with the same average rate. To the best of our knowledge, this derivation has not been done in the literature, and an *incorrect* proof is presented in [83] — they seek a result in asymptotic, big-Oh notation as a function of n, and to obtain the sought result, they assign a fixed value to n, completely invalidating the argument.
- A potential additional contribution, though not properly studied and evaluated in this work, adds DPA-resistance to the exponentiation algorithm with zero computational overhead. This would represent an important advantage with respect to most existing DPA-resistant algorithms, in which a performance penalty is involved due to randomization of the data. The contribution, thus, consists of a preliminary idea that needs to be studied. Still, the idea is original and the claim is reasonable, even if it does require further study.

4.3 Our Proposed Method: SABM

We now present our proposed approach and discuss some of its aspects, as well as a comparison to previous solutions.

4.3.1 Square and Buffered Multiplications

In the right-to-left algorithm (Algorithm 2) described in §2.3.1, a set of values are multiplied together to obtain the result. The key observation that allows us to achieve a constant execution path at coarse-level while maintaining optimality in execution time is the fact that these values need not be multiplied at the time that they are obtained. This allows us to *buffer* the execution of these multiplications, to hide any temporal patterns that would be visible through a power trace of the execution.

In its basic form, our proposed algorithm is an extension of the right-to-left algorithm; the difference being a buffer placed between the source of values to be multiplied together and the component that actually performs the multiplications. Whenever the exponent bit is 1, we send the value of S through a buffer; values enter at the times the corresponding bit is processed, but they exit at a constant rate, making power traces for different exponents identical. We thus obtain a *square-and-buffered-multiplications* (SABM) algorithm.

In an extreme (and simplified) scenario, one could store all the values that need to be multiplied together, and then, after all the squarings are done, one executes all the multiplications; this, of course, would require an unnecessarily high amount of storage. Instead, as we will discuss in §4.3.4, a small amount of storage, $O(\sqrt{\ell}\,)$, is necessary to implement a buffer so that the multiplications are interleaved with the squarings and executed at a constant rate — the average rate of ones in the exponent.

Algorithm 5 shows a sketch of our proposed SABM algorithm for exponent in standard binary representation. It is shown in a simplified form (in particular, the conditional on the exponent bit) for the purpose of illustrating the idea. See next section for a discussion on this aspect. For NAF, the average rate of nonzeros would be one third; thus, the condition for executing the multiplication or division would be `i` being divisible by 3.

Algorithm 5: Square-and-Buffered-Multiplications (SABM) Algorithm.

Input: x; $e = (b_{\ell-1}b_{\ell-2}\ \cdots\ b_1b_0)_2$
Returns: x^e

begin
  $S \leftarrow x$;
  $R \leftarrow 1$;
  for each bit b_i (i from 0 up to $\ell - 1$) **do**
    if $b_i = 1$ **then**
      $S \rightarrow Buff_S$;
    end
    $S \leftarrow S^2$;
    if i *is even* **then**
      $Tmp \leftarrow Buff_S$;
      $R \leftarrow R \times Tmp$;
    end
  end
  return R;
end

The line $Tmp \leftarrow Buff_S$ denotes assignment of a *single* element, extracted from the buffer, into Tmp.

It is assumed that the cost, in terms of execution time, to perform buffer operations is negligible compared to the operations involved in the exponentiation; this assumption is reasonable, since squarings and multiplications can have execution cost proportional to the square of the bit length of the elements (which is in general large), whereas operations on buffers typically can be implemented in constant time, with very short actual execution time. This has two important implications; one of them relates to the issue of preventing information leakage to the power trace by the buffer operations, which will be discussed in the next section. And also, for the purpose of evaluating the computational cost of our method, only the number of squarings and the number of multiplications are considered — a standard assumption in the context of cryptographic computations.

We notice that the algorithm, as shown above, does not address the possibility of buffer underflow; to avoid this issue, the buffer is pre-filled to half its capacity before starting the multiplications (this will be discussed more in detail in §4.3.4, and was omitted in Algorithm 5 for simplicity).

Avoiding Power Analysis on Buffer Operations

We now discuss the strategy and the assumptions that guarantee that no information is leaked to power traces for SPA to exploit the buffer operations. We notice that DPA would be successful at exploiting this aspect, since different operations, with some correlation between multiple power traces, is involved for the cases where exponent bits are 0 vs. cases of exponent bits being 1. However, we observe that standard DPA countermeasures (which would be needed in most cases anyway) would suffice to avoid this problem.

The key observation is that not only buffer operations are very inexpensive, in that they typically execute in constant time, with a very low number of operations; additionally, we notice that the actual data (the value being inserted in the buffer, e.g., a large integer, or a point on an elliptic curve) does not have to be copied or moved; only a reference to it (implemented for instance as a pointer in languages such as C or C++, or as a reference in languages such as Java) needs to be inserted in the buffer, making the operation truly negligible in terms of computational cost and power consumption.

If we statically allocate the storage space for the buffer, and set it up as a structure similar to a circular linked list (details are discussed below), then only a handful of pointer assignments is required — typically word or double-word data that processors natively handle (i.e., typically fit within the processor's bus).

This allows us to assert the guarantee that no leakage to power traces occur due to the buffer operations, based on either one of the following premises:

- If we can make the assumption that the cost of a few *pointer assignments* is truly

negligible and infeasible to observe through Analysis on a single power trace, then this directly translates into the guarantee that no information is leaked.

- If the above assumption does not hold for the particular architecture or attack model (for example, if the memory is external to the CPU and draws power from a separate line to which the attacker has access), then the fact that the computational cost of the pointer operations for buffer insertions is negligible makes it reasonable to set up an additional buffer for "fake" insertions — when the exponent bit is 0, we insert into this "fake" buffer. Since these values are discarded, this additional buffer does not have to be large (capacity for just one or two elements could suffice). Alternatively, we could simply set up a small set of pointer variables so that the exact same sequence of pointer assignments can be executed when the exponent bit is 0. Either way, we make each iteration *strictly identical* regardless of the value of the exponent bit. We could set up the two buffers (or the two sets of pointers) as an array of two elements, so that we use the exponent bit for subscripted (indexed) access, thus guaranteeing that the sequence of executed instructions is identical, simply operating with different data, as opposed to conditionally executing one path or the other. For simplicity, we omit this detail in all remaining algorithms and figures where buffering is involved, and simply show it as a conditional insertion on the buffer.

The data structure used to this effect operates as follows: the buffer storage space may be represented as a linked list (though all the elements could be allocated as a single block, as opposed to each element being independently and dynamically allocated). Figure 4.1 illustrates this; in the diagram, L_i denotes the storage location for the (physical) i^{th} element of the buffer, and $S^{(k)}$ denotes the storage location for S at iteration k (notice that as the algorithm progresses, the correspondence changes as elements are assigned to the buffer; so, in the diagram, it is just a coincidence that each of the $S^{(k)}$ items shown corresponds to L_k).

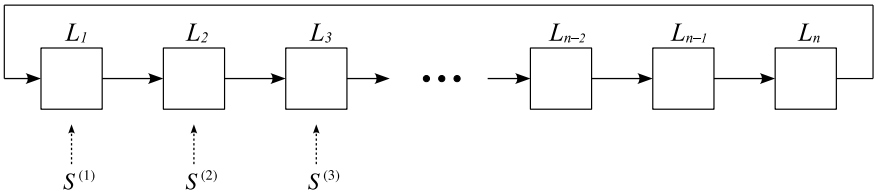


Figure 4.1: Data Structure for Buffer Storage Space.

Figure 4.2 illustrates the usage of the buffer — also a linked-list structure; in the example shown, storage locations L_2, L_{n-1}, and L_n correspond to the three elements that have been inserted in the buffer (to be multiplied together).

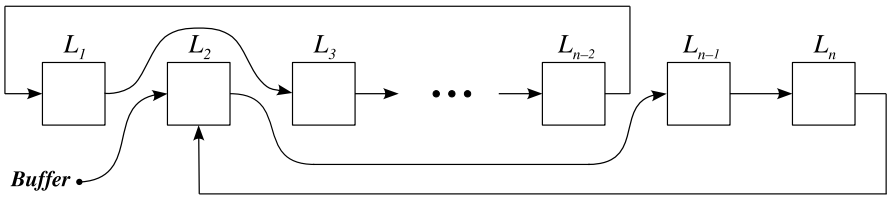


Figure 4.2: Data Structure for Buffer Usage.

Insertion in the buffer is done by "disconnecting" the pointers from the storage space linked-list and inserting it (through reassignment of the appropriate pointers) to the linked-list corresponding to the buffer. Removal from the buffer (which requires releasing the storage location being used) is done by disconnecting the element from the buffer and inserting at any logical position in the storage space linked-list (e.g., it could be after the current element, or after L_1, etc.).

We notice that the values stored will be overwritten several times, since the buffer size is smaller than the exponent length; this is not a problem, since we only need the value of S at the current iteration if the exponent bit (or combination of bits) is nonzero, in which case we "disconnect" the element from the storage space linked-list to insert it in the buffer; hence, it will not be overwritten until it has been processed and removed from the buffer. We also observe that there shall always be enough elements available in the storage space linked-list, since the buffer size is always specified so that buffer overflow does not occur (or occurs with a probability arbitrarily low).

Lastly, two aspects need to be taken into account when actually implementing this technique, to make the execution path of every iteration identical, regardless of whether the value of S is inserted in the buffer or not: since insertion in the buffer requires a few pointer operations, we should set up an additional set of pointers for "fake" insertions in the buffer; that is, when the exponent bit is 0, the same sequence of pointer assignments can be executed on these additional pointers. And also, we observe that at every iteration, the new value of S must be *always* stored in the "next" location in the linked list (following the pointer to next element), even though it is only necessary when the element needs to be inserted in the buffer.

A Side-Effect of Optimal Execution Time

We observe that in the presence of SPA, our method reveals the number of nonzero bits in the exponent. This is a natural side-effect of the method being optimal while processing individual exponent bits, and should not be a reason for concern — the amount of leaked

information is small enough that it should not compromise the security of the system.[2] Consider, in the binary case, what happens if the attacker learns that k bits are nonzero; there are $\binom{\ell}{k}$ possible exponent values, instead of 2^ℓ, reducing the entropy of the exponent by the log of the fraction (negative log, if we talk about a reduction by a positive amount).

$$\Delta H_{e;k} \;=\; -\log_2\left(\binom{\ell}{k}\,2^{-\ell}\right) \tag{4.1}$$

We consider the weighted average of this reduction in entropy (weighted by the probability of each number of nonzero bits) to determine the reduction in the exponent entropy ΔH_e as:

$$\Delta H_e \;=\; -\sum_{k=0}^{\ell} \mathrm{P}\{k\}\;\log_2\left(\binom{\ell}{k}\,2^{-\ell}\right) \tag{4.2}$$

We observe that for the binary case, with $p = \frac{1}{2}$, we have

$$\Pr\{k\} \;=\; \binom{\ell}{k}\left(\tfrac{1}{2}\right)^k\left(1-\tfrac{1}{2}\right)^{\ell-k} \;=\; \binom{\ell}{k}\,2^{-\ell} \tag{4.3}$$

From Equation (4.3), we see that the argument of $\log_2$ in Equation (4.2) is precisely the probability of k nonzero bits, and we easily recognize the sum in Equation (4.2) as the entropy of the probability mass function [18]. Since the values of ℓ that we consider are large, the distribution is closely approximated by a normal distribution $\mathcal{N}(\ell p, \ell p(1-p))$ [67]. In the case of standard binary exponent, $\mathcal{N}(\ell/2, \ell/4)$, with entropy H_Φ in bits given by

$$\begin{aligned} H_\Phi \;&=\; -\sum_k \tfrac{1}{\sqrt{2\pi\ell/4}}\,\mathrm{e}^{-\frac{(k-\ell/2)^2}{2\ell/4}}\;\log_2\left(\tfrac{1}{\sqrt{2\pi\ell/4}}\,\mathrm{e}^{-\frac{(k-\ell/2)^2}{2\ell/4}}\right) \\ &=\; \sum_k \tfrac{1}{\sqrt{2\pi\ell/4}}\,\mathrm{e}^{-\frac{(k-\ell/2)^2}{2\ell/4}}\left(\tfrac{1}{2}\log_2(2\pi\ell/4) + \tfrac{(k-\ell/2)^2}{2\ell/4}\cdot\log_2\mathrm{e}\right) \\ &=\; \frac{1}{2}\log_2\left(\frac{\pi\mathrm{e}\,\ell}{2}\right) \end{aligned} \tag{4.4}$$

As an example, for a 1024-bit exponent in standard binary representation, we see that leaking the number of nonzero bits in the exponent reduces its entropy by just 6 bits. The amount of information leaked when using NAF is even smaller, since the variance for the case of NAF is lower. We omit the details here, but §4.3.4 deals with this aspect — intuitively, this has to be the case, since for large values of ℓ, approximation by a normal distribution is valid, but the range of values for NAF is limited to a maximum of $\ell/2$ nonzero bits, unlike for the binary case; this suggests that the variance has to be lower for the NAF case.

[2] This assumes that for an attacker with a given level of computing power, the key sizes are high enough that after reducing the search space by this small amount, exhaustive search is still out of reach.

4.3.2 Simultaneous Processing of Half-Exponents

As mentioned in §4.1, our SABM algorithm can be combined with the method proposed by Sun et al. [79]. In their method, the exponent is split into two halves, and these are processed simultaneously. There are two key details in their method: (a) Each two bits involve a single multiplication, since they use four "accumulators" (one for each combination of the two bits); and (b) the results of multiplications where the two exponent bits are zero are discarded; but the fraction of two-bit combinations that are both zeros is $\frac{1}{4}$ of the length of the exponent on average, reducing the impact of this inefficiency on the overall performance. Furthermore, we observe that a non-SPA-resistant version of their method (one in which operations that have no effect on the final result are not executed) is more efficient (in terms of execution time) than the standard binary exponentiation. Thus, combining their algorithm with our proposed technique would provide resistance to SPA without any single operation where the result is discarded, providing increased efficiency with respect to the basic form of the SABM algorithm.

If we implement our proposed method using the algorithm by Sun at al. as a replacement for the right-to-left binary exponentiation, we avoid the multiplications corresponding to the two bits being zero; instead, we buffer *all* multiplications, thus making the entire execution independent of the exponent bits, yet optimizing away the unnecessary multiplications. An additional advantage when combining Sun's method with our proposed technique is that with this method, the exponent length is reduced to half, which means that our storage requirement is reduced by a factor of $\sqrt{2}$. Algorithm 6 shows the details of this alternative implementation of our method; notice that the average rate of multiplications is $\frac{3}{4}$, since a multiplication occurs for every bit pair with at least one nonzero.

4.3.3 Comparison to Existing Solutions

Algorithm SABM executes in optimal time (optimal number of operations) in the sense that no unnecessary operations are executed; indeed, no result of any operation is discarded, meaning that the minimum number of operations required to obtain the correct result is performed. Also importantly, every operation is executed in its optimal form; that is, every square operation is done with an optimized procedure for squaring, and not as a multiplication routine where the two operands are equal. This constitutes an important advantage with respect to existing solutions, where a performance overhead exists with respect to the optimized (and vulnerable to SPA) forms of the algorithm.

The optimality of our method is of course relative to the underlying exponentiation technique — when used in combination with the standard binary exponentiation, our algorithm is optimal in the sense that it adds resistance to SPA with zero computational overhead; that is, it executes the exact same number of squarings and the exact same

Algorithm 6: SABM with Simultaneous Processing of Half-Exponents.

Input: x; $e = (b_{\ell-1}b_{\ell-2}\ \cdots\ b_1b_0)_2$
with ℓ divisible by 2
Returns: x^e

begin
  $S \leftarrow x$;
  $R_{01} \leftarrow 1$;
  $R_{10} \leftarrow 1$;
  $R_{11} \leftarrow 1$;
  $\ell' \leftarrow \ell/2$;

  for each bit pair $b_{\ell'+i}b_i$ (i from 0 up to $\ell' - 1$) **do**
    if $b_{\ell'+i}b_i \neq 00$ **then**
      $\langle S, b_{\ell'+i}b_i\rangle \rightarrow Buff_{\mathrm{S}}$;
    end
    $S \leftarrow S^2$;
    if *i is not divisible by 4* **then**
      $\langle Tmp, b_{\mathrm{H}}b_{\mathrm{L}}\rangle \leftarrow Buff_{\mathrm{S}}$;
      $R_{b_{\mathrm{H}}b_{\mathrm{L}}} \leftarrow R_{b_{\mathrm{H}}b_{\mathrm{L}}} \times Tmp$;
    end
  end

  $R_{01} \leftarrow R_{01} \times R_{11}$;
  $R_{10} \leftarrow R_{10} \times R_{11}$;

  repeat ℓ' times:
    $R_{10} \leftarrow (R_{10})^2$;
  ;
  return $R_{01} \times R_{10}$;
end

number of multiplications as either the left-to-right or right-to-left optimized versions that are vulnerable to SPA attacks. Similarly, when used in combination with the technique proposed in [79], it maintains resistance to SPA while eliminating every unnecessary operation in their algorithm.

Table 4.1 summarizes the differences in performance with respect to existing solutions, showing the Square-and-Multiply (SAM) performance as a baseline. By convention, squaring routines execute in 1 unit of time; results are shown for the assumption that multiplication routines execute in 1.2 units of time, a reasonable figure as an average ratio for practical implementations with prime field arithmetic [39]. Values in Table 4.1 are *average* amount of units of time to execute an exponentiation with an ℓ-bit exponent.

	Binary	NAF
S-A-M	1.6ℓ	1.4ℓ
S-A-A-M	2.2ℓ	2.2ℓ
Joye	1.8ℓ	1.6ℓ
Sun et al.	$1.6\ell + \mathrm{O}(1)$	$--$
SABM	1.6ℓ	$\mathbf{1.4\ell}$
SABM + Sun	$1.45\ell + \mathrm{O}(1)$	$--$

Table 4.1: Performance Comparison of Exponentiation Algorithms.

4.3.4 Storage Requirements

We now discuss the storage requirements for our method to work without producing buffer overflows or underflows. Both conditions are critical for the security of the system; an instance of buffer overflow would require that a multiplication takes place immediately, when in principle one should not have taken place. Buffer underflow would make the algorithm skip a multiplication when one should have taken place. In both cases, we would leak partial information about the exponent to the power trace, since the attacker can determine the number of nonzero bits in the exponent portion that has been processed.

Let the exponent be an ℓ-bit random variable (either in binary or NAF representation), with probability p that a bit is nonzero — in the case of binary representation, p is $\frac{1}{2}$; with NAF, p is $\frac{1}{3}$. Let $\mathbf{k}$ be the number of nonzero bits in the exponent; clearly, for the binary case this random variable $\mathbf{k}$ follows the binomial distribution $\mathcal{B}(\ell, p)$, which, for sufficiently large values of ℓ can be approximated by $\mathcal{N}(\ell p, \ell p(1-p))$ [67].

In the case of NAF representation, the distribution is not binomial, since constraints between contiguous bits exist. Nonetheless, for large values of ℓ, it can also be approxi-

mated by a normal distribution. A closed-form description of this distribution is given in [5], from which it can be observed that for large values of ℓ, the variance is $\frac{2\ell}{27}$. For convenience, we use $\mathcal{N}(\ell p, \ell\zeta_p)$ as the normal approximation covering both cases; for standard binary, $\zeta_p = \frac{1}{4}$; for NAF, $\zeta_p = \frac{2}{27}$:

$$\Pr\{\mathbf{k} = k\} \approx \frac{1}{\sqrt{2\pi\ell\zeta_p}}\, \mathrm{e}^{-\frac{(k-\ell p)^2}{2\ell\zeta_p}} \tag{4.5}$$

In Equation (4.5), the term $(k-\ell p)$ represents deviation from the mean — multiplications are done at the average rate of nonzero bits, for a total of ℓp multiplications on average.

Let $\boldsymbol{\delta}$ denote the random variable corresponding to this deviation, and let us consider the probability of a deviation $\delta = c\sqrt{\ell}$, where c is a positive real number:

$$\begin{aligned} \Pr\left\{\boldsymbol{\delta} = c\sqrt{\ell}\right\} &\approx \frac{1}{\sqrt{2\pi\ell\zeta_p}}\, \mathrm{e}^{-\frac{(c\sqrt{\ell})^2}{2\ell\zeta_p}} \\ &= \frac{1}{\sqrt{2\pi\ell\zeta_p}}\, \mathrm{e}^{-\frac{c^2}{2\zeta_p}} \end{aligned} \tag{4.6}$$

If we set up a buffer of size $c\sqrt{\ell}$, then the above probability corresponds to the probability that the processing ends with the buffer at its full capacity. Thus, the probability of buffer overflow P_{BOF} is given by all deviations above this value:

$$\begin{aligned} P_{\text{BOF}} &\approx \sum^{\delta > c\sqrt{\ell}} \frac{1}{\sqrt{2\pi\ell\zeta_p}}\, \mathrm{e}^{-\frac{\delta^2}{2\ell\zeta_p}} \\ &\approx \int_{c\sqrt{\ell}}^{\infty} \frac{1}{\sqrt{2\pi\ell\zeta_p}}\, \mathrm{e}^{-\frac{\delta^2}{2\ell\zeta_p}}\, d\delta \end{aligned}$$

Making the substitution $t = \frac{\delta}{\sqrt{2\ell\zeta_p}}$, we obtain:

$$\begin{aligned} P_{\text{BOF}} &\approx \frac{1}{\sqrt{\pi}} \int_{\frac{c}{\sqrt{2\zeta_p}}}^{\infty} \mathrm{e}^{-t^2}\, dt \\ &= \frac{1}{2}\, \mathrm{erfc}\left(\frac{c}{\sqrt{2\zeta_p}}\right) \end{aligned} \tag{4.7}$$

where erfc is the complementary error function [1], defined as

$$\mathrm{erfc}\,(x) = \frac{2}{\sqrt{\pi}} \int_x^{\infty} \mathrm{e}^{-t^2}\, dt$$

Equation (4.7) considers buffer overflows that occur at the end of the processing of the exponent bits; buffer overflow could occur earlier in the processing. However, we

notice that some cases overlap — for example, if the buffer size is exceeded by two units at the end of the processing, then overflow must have occurred one bit before completing the processing, and so we should not add the probability corresponding to this "early overflow", since it has been already accounted for. There are, however, instances where overflow occurs early but it is then "corrected" by the time the ℓ bits are processed (i.e., deviation exceeds the buffer size, but the bits that follow make the deviation fall back within the buffer size). We will refer to these cases as *legitimate early overflows*. The probabilities corresponding to these legitimate early overflows should be added to our estimate of the probability of buffer overflow. To this end, we make use of the following lemma:

Lemma 4.1 *Legitimate early overflows occur with lower probability than non-legitimate early overflows for* $p = \frac{1}{2}$ *and* $p = \frac{1}{3}$.

Proof included in Appendix A.

From Lemma 4.1, it follows that a conservative estimate is obtained if we assume that these cases of legitimate early overflows occur with the same probability as non-legitimate early overflows. This means that the estimate of the probability of buffer overflow *within* ℓ bits should be twice the probability of overflow at exactly ℓ bits, which is what Equation (4.7) represents.

The above deals with buffer overflow; to consider buffer underflow, we recall that the distribution (for the deviation δ) is assumed to be symmetric (we are using the normal distribution as an acceptable approximation for large values of ℓ). This means that buffer underflow occurs with the same probability as buffer overflow, provided that we set up a buffer of size $2c\sqrt{\ell}$ and fill it to half its size before we start removing items to perform the multiplications. Of course, we have to do this step without leaking information about exponent bits to the power traces; that is, we process the first $p^{-1}c\sqrt{\ell}$ bits before starting to remove items — this way, we fill the buffer to half its size *on average*, and the procedure is independent of the exponent bits, which ensures that no information about the exponent is leaked.

Thus, we finally obtain the probability of buffer failure P_{BF} (including overflow and underflow):

$$P_{\text{BF}} \approx 2 \operatorname{erfc}\left(\frac{c}{\sqrt{2\zeta_p}}\right) \tag{4.8}$$

From Equation (4.8), we see that with a buffer of size $\mathrm{O}(\sqrt{\ell})$ we can make the probability of success (i.e., probability of no buffer failure) arbitrarily close to 1, as a function of the security parameter c, corresponding to the multiplicative constant in the $\mathrm{O}(\sqrt{\ell})$ measure.

As an example, for ECC with a 256-bit exponent (i.e., $\ell = 256$) in NAF representation ($\zeta_p = \frac{2}{27}$), setting the buffer size to $4\sqrt{256} = 64$, corresponding to $c = 2$, gives us:

$$P_{\text{BF}} \approx 2\,\text{erfc}\left(\frac{2}{\sqrt{2\cdot\frac{2}{27}}}\right) \approx 2\,\text{erfc}\left(\sqrt{27}\right) \approx 4\cdot 10^{-13}$$

Table 4.2 shows the resulting probabilities of buffer failure for several additional choices of c and typical values of exponent lengths (256 bits for ECC and 1024 bits for RSA, assuming $2048 - bit$ modulus and CRT optimization) assuming NAF representation of the exponent. The N/A cases are where the resulting buffer size exceeds one third of the exponent length, in which case the approximation is no longer valid, as that buffer size would allow us to buffer *all* multiplications.

Notice, however, that these are only examples to illustrate the analytic result; in §4.6 we discuss some practical aspects that can lead to much smaller buffer sizes with guaranteed execution without buffer failure.

	$\ell = 256$	$\ell = 1024$
$c = 1$	$32\ /\ 4.77\cdot 10^{-4}$	$64\ /\ 4.77\cdot 10^{-4}$
$c = 1.5$	$48\ /\ 7.1\cdot 10^{-8}$	$96\ /\ 7.1\cdot 10^{-8}$
$c = 2$	$64\ /\ 4\cdot 10^{-13}$	$128\ /\ 4\cdot 10^{-13}$
$c = 3$	N/A	$192\ /\ 5.9\cdot 10^{-28}$
$c = 4$	N/A	$256\ /\ 1.35\cdot 10^{-48}$

Table 4.2: Examples of Buffer Size / Probability of Buffer Failure

Buffer Structure to Prevent Buffer Underflow

As mentioned in the previous section, we need to start by filling the buffer to half its capacity before starting the multiplications, to prevent buffer underflow. However, doing this without leaking information about exponent bits requires that we process the first $\frac{1}{2}\,p^{-1}|\,\mathrm{B}\,|$ bits (where $|\,\mathrm{B}\,|$ denotes the size of the buffer) so that we fill the buffer to half its capacity *on average*, and the procedure is independent of the exponent bits.

This of course means that the algorithm will finish the processing of the exponent bits with $\frac{1}{2}\,p^{-1}|\,\mathrm{B}\,|$ elements remaining in the buffer on average, pending processing. These remaining elements would then be multiplied together before producing the result. This would also apply, though not as a requirement, to the parallelized implementation (discussed in §4.4); in this case, since the multiplications are being done in parallel (i.e., simultaneously) with the squarings, other adjustments could be made to avoid this issue, or at least reduce its impact.

Thus, Algorithms 5, 6, and possibly the parallel version, shown in Figure 4.3, in either binary or NAF versions, would be adjusted as shown in Algorithm 7 — the algorithm

shown corresponds to algorithm SABM (Algorithm 5); however, the modification is directly applicable to all other forms of the algorithm. Notice also that Algorithm 7 is written in its general form, using p to determine the point at which the buffer should be filled to half its capacity.

Algorithm 7: Buffer Pre-Filling and Post-Processing for SABM.

Input: x; $e = (b_{\ell-1}b_{\ell-2} \cdots b_1 b_0)_2$
Returns: x^e

begin
  $S \leftarrow x$;
  $R \leftarrow 1$;
  for each bit b_i (i from 0 up to $\ell - 1$) **do**
    if $b_i = 1$ **then**
      $S \rightarrow Buff_S$;
    end
    $S \leftarrow S^2$;
    if $i > |Buff_S|/(2p)$ *and* i *is even* **then**
      $Tmp \leftarrow Buff_S$;
      $R \leftarrow R \times Tmp$;
    end
  end
  while $Buff_S$ *is not empty* **do**
    $Tmp \leftarrow Buff_S$;
    $R \leftarrow R \times Tmp$;
  end
  return R;
end

4.4 Parallelized Version of Algorithm SABM

Our proposed SABM algorithm has the additional benefit of being easily parallelizable while maintaining resistance to SPA attacks. This comes as a simple extension of the idea that the right-to-left exponentiation algorithm is naturally parallelizable up to two threads — as exponent bits are tested and operands for the multiplications are produced, these multiplications can be done in parallel, simultaneously to the squarings.

Incidentally, a two-thread parallel version of the right-to-left exponentiation algorithm requires a buffer for the multiplication operands. This is due to the fact that multiplication takes longer than squaring, and thus, if several contiguous nonzero bits are found,

the new multiplication operands will be ready before the previous multiplications have completed.

Thus, the parallelized version of our SABM algorithm is similar to a straightforward parallelized version of the right-to-left exponentiation algorithm — the latter being vulnerable to SPA attacks. Thus, the main difference relates to the rate of execution of the multiplications; the second thread, responsible for the multiplications, should extract operands from the buffer *at a fixed rate*, taking advantage of the buffer that accommodates for the difference. The first thread adds elements to the buffer at the times that they are produced, and the second thread extracts elements from the buffer at the average rate of nonzero bits (one multiplication every two squarings for standard binary representation, or one every three squarings for NAF representation). Figure 4.3 shows the details of the parallel version of the SABM algorithm.

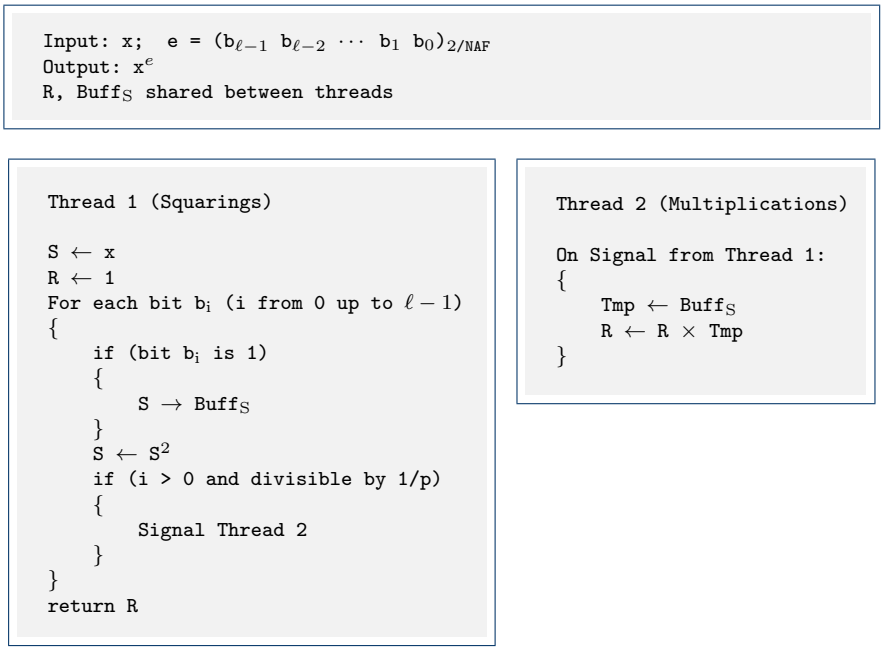


Figure 4.3: Two-Thread Parallel version of Algorithm SABM.

The thread synchronization details are omitted in Figure 4.3; roughly speaking, access to the buffer $\texttt{Buff}_S$ should be synchronized, depending on how it is implemented — since insertion and removal operations are done at opposite ends of the sequence, it might be possible to implement them in such a way that no common data is accessed. In that case, simultaneous access to $\texttt{Buff}_S$ by both threads may not be a problem. The result `R`, however, needs synchronization, since the last operation of thread 1 (to return the result of the exponentiation) and operations from thread 2 might be subject to a race condition, since no guarantee should be assumed about relative order of execution between threads [9]. In particular, an additional shared (and synchronized) flag might be

needed, so that thread 2 can communicate to thread 1 the fact that the last multiplication has completed, and thus the result is now available.

4.5 Randomized Execution of Multiplications

Given an SPA-resistant cryptosystem, [48] showed that correlation between power consumption and the data that the device is working with at a particular time can be exploited through the use of statistical and digital signal processing techniques on multiple power traces. As countermeasures, several techniques have been suggested that introduce randomization to eliminate this exploitable correlation. For example, [48] suggests blinding to randomize the data and eliminate any correlation between traces for different executions; [15] suggests several randomization techniques, some of them specific to ECC, as well as [40], which proposes techniques specific to Koblitz curve cryptosystems. These proposed techniques involve a small performance penalty, since they transform the data, so that the operations in different traces correspond to different actual data.

An interesting aspect of our proposed method — in both its parallel and non-parallel forms — is that it introduces the opportunity for randomization in the execution, with virtually no performance penalty; in particular, two different aspects can be randomized: timing and order of execution of the multiplications. Indeed, with respect to the timing of the multiplications, the detail that makes the technique work as countermeasure against SPA is not that the multiplications be executed at a fixed rate or with some fixed timing pattern: the important detail is to execute them at times independent or uncorrelated from the times at which the operands are produced. As much as we can extract elements from the buffer at a fixed rate corresponding to the average rate of nonzero bits, we can also extract them at random times, as long as the average rate at which they are extracted is the same average rate at which they are inserted into the buffer.

We can randomize this timing in several possible ways. For example, we could use a source of randomness to decide whether or not to execute a multiplication after each squaring operation: with probability p, we execute the multiplication ($p = \frac{1}{2}$ for standard binary exponent, $\frac{1}{3}$ for NAF). We could also do a random permutation of the bits of the exponent, such that we have the exact same number of nonzero bits as in the exponent, but at random positions, uncorrelated from the positions of the nonzero bits in the exponent; thus, we execute the multiplications conditioned on the bit of this permuted value. In the parallel version of the method, we could introduce randomness simply by adding small delays (idle waits) of random length after receiving the signal from the other thread indicating that a multiplication should take place.

We have to be careful with the way that this may affect the requirements on the buffer size; in particular, if we extract elements from the buffer with a random pattern, the buffer size has to be increased by a factor of $\sqrt{2}$. This is the case since now the condition

for buffer failure is given by deviation from one random value to another random value. Let $\boldsymbol{\delta_e}$ be the deviation from the mean for the times of extractions from the buffer, and $\boldsymbol{\delta}$ for insertions, as defined in §4.3.4. Since by assumption these two random variables are uncorrelated, then

$$\mathrm{Var}(\boldsymbol{\delta} - \boldsymbol{\delta_e}) = \mathrm{Var}(\boldsymbol{\delta}) + \mathrm{Var}(\boldsymbol{\delta_e}) \;=\; 2\,\mathrm{Var}(\boldsymbol{\delta})$$

(since both variables have the same distribution).

From equations (4.8) and (4.5), and given that the value of ℓ does not change, we conclude that the buffer size has to be increased by a factor of $\sqrt{2}$ to maintain a given probability of buffer failure.

Notice that this is not the case if we randomly permute the bits of the exponent — on the contrary, the buffer size could be reduced and still maintain a given probability of failure. This is the case since with the bits of the exponent permuted, we guarantee that at the end of the processing — which is where buffer failure occurs with higher probability — deviation from the mean is zero.

As for randomization in the order of execution, it is clear that the order in which the multiplications are executed need not be the same as the order in which the operands are generated, since the final result is simply the product of the set of values; thus, we can permute the values in the buffer at random, so that they are extracted in a random order. This, however, provides a limited level of randomization, in that only $\mathrm{O}(\sqrt{\ell}\,)$ elements are in the buffer at a given time, so only that many at a time can be randomized in the average case.

Furthermore, we do not even need to multiply the partial result times each value and reassign the partial result: we can, as long as there are enough values in the buffer, take any two random elements from the buffer, multiply them together, remove them from the buffer and insert the result of that multiplication back into the buffer (or remove one of the values and replace the other one with the result). Clearly, the final result will be the same. Also, the number of multiplications is necessarily the same, since each multiplication, whichever way is done, reduces the number of elements in the buffer, and thus the number of multiplications remaining, by exactly one. We notice that this compensates for the aspect mentioned in the previous paragraph about the limited level of randomization; indeed, the number of possible combinations for the sequence of operations greatly increases if we randomly choose pairs of elements from the buffer.

All of these aspects apply equally to the parallel and non-parallel versions of the method.

4.6 Practical Considerations

In §4.3.4, we discussed the storage requirements for our solution to work with probability arbitrarily close to 1. Several aspects can contribute to a substantial reduction in the amount of storage required in practice when implementing our proposed solution. In terms of asymptotic (big-Oh) notation, the requirement remains the same: we still require $O(\sqrt{\ell}\,)$ storage; the improvements relate to a substantial reduction in the multiplicative constant that asymptotic notation hides, but that plays an important role when it comes to an actual, practical implementation.

4.6.1 Buffer Underflow

The impact of buffer underflows could be entirely avoided; if the multiplications buffer is empty at the time that a multiplication must take place, a "dummy" multiplication can be performed, such as a multiplication by 1, or in the case of integer modular multiplications, by $m + 1$, where m is the modulus; if a multiplication by 1 is suspected to have an identifiable power consumption profile, or in cases where multiplication by the identity is by nature a null procedure, we could multiply by a random value, as long as each of these dummy multiplications by a random value is matched by another dummy multiplication by its inverse. This can be done efficiently, since pseudo-random values suffice, and a stock of these pseudo-random values can be pre-computed. By eliminating the impact of buffer underflows, we can reduce the storage requirement to half the original amount. However, we must keep in mind that if we introduce "dummy" operations, execution time is no longer optimal; still, the fraction of "dummy" operations may be reasonably low.

4.6.2 Avoiding "Bad" Exponents

Perhaps a more important practical consideration is the fact that we can restrict the use of exponents to those that do not lead to buffer overflows or underflows for a given amount of storage, without noticeably sacrificing the security of the system, or any other aspect of the system's performance. We recall that for commonly used exponentiation based cryptosystems such as RSA and Diffie-Hellman/ElGamal, the secret exponent is randomly chosen when generating the key pair, or randomly chosen for each session. In both cases, the choice can be constrained to avoid buffer overflows or underflows for a given amount of storage. We observe that this can be done with a negligible computational cost, as it only requires scanning the exponent bits and counting, without any group operations required. Another crucial aspect is the fact that this constraint can be introduced without causing a noticeable reduction in the entropy of the secret exponent, as explained below.

If a buffer overflow or underflow occurs with probability P_{F}, then restricting the random choice of exponent to the subset of values that do not produce overflow or underflow reduces the entropy H_{e} of the secret exponent by $\Delta H_{\text{e}} = \log_2(1 - P_{\text{F}})$. This can be easily seen, since a value of entropy of N bits is in this case associated to a uniformly distributed random choice in a space of size 2^N [18]; if invalid values occur with probability P_{F}, then the size is reduced to $2^N(1 - P_{\text{F}})$, and the random selection is still uniformly distributed, which means that the entropy is $H'_{\text{e}} = \log_2\left(2^N(1 - P_{\text{F}})\right) = N + \log_2(1 - P_{\text{F}})$.

As an example to illustrate the validity of this argument; a probability of buffer overflow or underflow of 0.1 corresponds with a reasonably small buffer size (a value of $c \approx 0.5$ for NAF, as per Equation (4.8)); yet, restricting the choice of exponents produces a change in the entropy of only $\log_2 0.9 = -0.152$ bits — a negligible reduction, as we usually consider exponents in the order of at least hundreds of bits. Even a more aggressive setting with a probability of failure close to 0.3 — which at first glance may seem alarmingly high — corresponds to a reduction in the entropy of only $\log_2 0.7 \approx -0.5$ bits, which could still be considered a negligible reduction, depending on the application. For these cases, however, it would be important to take into consideration the cost of generating random values — if the cost is too high, we may rather want to avoid low buffer sizes that would lead to discarding a high fraction of random bits.

4.6.3 Secure Validation of Exponents

Another crucial aspect to notice is the fact that the above constraint in the choice of exponents can be implemented without introducing any new vulnerabilities to side-channel analysis (or any other type of cryptanalysis). The validation of the exponent can be easily done with a constant execution path procedure, to avoid exposing the exponent to any side-channels; we notice that we only need to have a constant execution path until the point where it is determined that the exponent is invalid (that is, the procedure can use a conditional "early exit" as an optimization). This means that an attacker could still observe the fact that exponent values are being tested and discarded, but this constitutes useless information, since an invalid exponent is a random event that is independent from the next choice. The "early exit" optimization would leak information about the exponent, but only about exponents *being discarded*. If the exponent is valid, then the procedure will exhibit constant execution path, so no information will be leaked related to the exponents that pass the test.

4.7 Discussion and Concluding Remarks

One important aspect to take into account when considering the use of our proposed SABM algorithm is the issue of trading storage in exchange for optimal execution time.

Though computing power in mainstream devices has greatly increased, and thus one could be inclined to somewhat dismiss the importance of a good execution time, the fact remains that public-key cryptographic computations — in particular exponentiation with large exponents — is usually the performance bottleneck in the security-related aspects of a device. Thus, the impact of improvement in this area on the overall performance is greater than that of improvement in any other areas. On the other hand, for storage, the cost and requirements in terms of area have decreased dramatically over the recent past, and one could reasonably expect them to continue to decrease. Even for embedded applications, where resistance to power analysis is usually a critical requirement, sufficient storage to implement our proposed method is easily available. This is certainly the case for hand-held mobile devices; maybe a little less for smart cards and almost certainly not the case for RFID devices.

Incidentally, for embedded devices — often relying on battery power — there is an additional subtle advantage with our proposed method: the savings in computations translate not only into better execution time, but also into decreased power consumption, which is often another critical requirement for these types of devices.

For most applications, the use of our technique may not even introduce the need for additional memory in the device, but simply make use of available memory that is present anyway for other reasons. Indeed, one could reasonably claim that systems for which our proposed method is suitable (including hand-held mobile devices) usually are sophisticated enough that multitasking is certainly an included feature, with other tasks that will require storage.

For situations where this is not the case, there is certainly the disadvantage that the use of additional storage introduces manufacturing cost *per unit*, unlike with techniques based on algorithmic improvement without extra storage; however, the fact remains that our proposed technique exhibits better execution performance than any of the existing techniques so far, possibly making the additional cost justifiable, depending on the situation.

Another important aspect to consider is the fact that the main idea of our technique — namely, buffering to execute multiplications at a constant rate — can be combined with other exponentiation techniques, providing optimal execution time with respect to the underlying exponentiation technique. This was shown to be the case with the technique proposed by Sun et al. [79], in which combination with our technique avoids any unnecessary operations, yielding an even more efficient method than when combining with basic right-to-left exponentiation. It was noted that the method by Sun et al. is specific to binary exponent representation, as both the complexity and the fraction of unnecessary operations increase if their method is adapted to NAF representation. However, additional work might prove useful in adapting it to NAF in combination with our technique, further improving execution time. We also emphasize the perhaps subtle detail that with their method, exponent size is reduced to half, which means that when

combined with our technique, the amount of buffer storage required decreases by a factor of $\sqrt{2}$.

Additional work is also suggested to further investigate the prospect of resistance to DPA through randomization of timing pattern and order of multiplications. Though intuition suggests that it may be the case that DPA attacks could be defeated, or at least slowed down to a point where they are impractical, further research is necessary to determine whether such technique can be implemented in a way that attacks can not bypass or compensate for such randomization.

Chapter 5

Simultaneous Processing of Half-Exponents

This chapter presents a follow-up work to that presented in Chapter 4. We extended the work by adapting the half-exponents technique to make use of signed-digits for the representation of the exponent, which in turn leads to several new SPA-resistant algorithms with improved performance with respect to the work presented in Chapter 4, or any other SPA-resistant techniques. This is only possible when combining that technique with our SABM method, as the technique, in its original form, is fundamentally incompatible with signed-digit or NAF representation.

This work appeared as CACR technical report CACR 2011-13 [62]. The text and contents in this chapter are based on this work.

5.1 Motivation

In Chapter 4, we presented our SPA-resistant technique based on buffering multiplications to execute them at a constant rate, or in any case, at times unrelated to the positions of the exponent bits. We also showed that the method can be combined with alternative algorithms as the underlying binary exponentiation. In particular, this aspect was demonstrated by combining the SABM technique with that proposed by Sun et al. [79]. However, Sun's method of simultaneously processing half-exponents, as originally proposed, is fundamentally incompatible with the use of signed-digit representation of the exponent, in particular the Non-Adjacent Form (NAF), which limits its computational efficiency.

5.2 Our Contributions

In this work, we extended our previous results and presented several new SPA-resistant algorithms that result from modifying the method by Sun et al. to make use of signed-digit representation of the exponent, which becomes feasible only when combining their method with our SABM technique. We demonstrate the technique with several representations of the exponent, showing further improvements in performance with respect to the technique in its basic form. Specifically, we first employ NAF for the representation of the exponent and Joint Sparse Form (JSF) for the representation of the two exponent halves. Then we adapt the technique to process blocks of multiple digits of the exponent. We present this kind of processing for two scenarios: i) the digit block being a signed-digit base-4 representation derived from the NAF representation of the exponent, and ii) the block being a signed-digit base-8 representation derived from JSF. Unlike existing multi-digit exponentiation techniques, which focus on performance, these approaches are suitable to be combined with the SABM technique, providing increased efficiency while maintaining resistance to SPA.

As an important secondary contribution, we provide an alternative analysis of some of the properties of NAF, based on an alternative algorithm to convert binary to NAF representation (to the best of our knowledge, also our own original contribution). We claim that the analysis is much simpler than existing mathematical models for the properties of NAF, and also reveals certain properties that, to the best of our knowledge, have not been studied in the existing literature (or at the very least are extremely hard to come by).

5.3 Exponent in Signed-Digit Representations

The central idea of simultaneous processing of half exponents is the use of multiple accumulators, to hold the product of subsets of the set of values that produce the correct result; these are then multiplied together to obtain the correct result with the product of the entire set of values. In particular, it uses one accumulator for each combination of bits (one bit from each half-exponent in corresponding positions) where at least one of the bits is nonzero. Thus, for the binary case we have R_{01}, R_{10}, and R_{11}. Similarly, when extending this algorithm to work with signed-digit representations, we still have one accumulator for each combination of digits from each exponent half; thus, in the case of signed-digit exponent, we have accumulators $R_{\bar{1}\bar{1}}$, $R_{\bar{1}0}$, $R_{\bar{1}1}$, $R_{0\bar{1}}$, R_{01}, $R_{1\bar{1}}$, R_{10}, and R_{11}.

With this setup, the result of the exponentiation is computed as follows: Let e be the ℓ-digit exponent, with signed-digit representation $e = d_{\ell-1}d_{\ell-2}\cdots d_2d_1d_0$, $2 \,|\, \ell$, let $\ell' = \ell/2$, let $e_{\text{L}} = d_{\ell'-1}d_{\ell'-2}\cdots d_2d_1d_0$ and $e_{\text{H}} = d_{\ell-1}d_{\ell-2}\cdots d_{\ell'+1}d_{\ell'}$.

Then, x^e is obtained as $x^e = x^{e_{\rm L}} \cdot (x^{e_{\rm H}})^{\left(2^{\ell'}\right)}$, with $x^{e_{\rm L}}$ and $x^{e_{\rm H}}$ computed as follows:

$$x^{e_{\rm L}} = R_{01} \cdot R_{\bar{1}1} \cdot R_{11} \cdot (R_{\bar{1}\bar{1}} \cdot R_{0\bar{1}} \cdot R_{1\bar{1}})^{-1} \quad (5.1)$$

$$x^{e_{\rm H}} = R_{10} \cdot R_{1\bar{1}} \cdot R_{11} \cdot (R_{\bar{1}\bar{1}} \cdot R_{\bar{1}0} \cdot R_{\bar{1}1})^{-1} \quad (5.2)$$

where

$$R_{\mathrm{d_H d_L}} = \prod_{\substack{i=0 \\ d_i = \mathrm{d_L} \\ d_{\ell'+i} = \mathrm{d_H}}}^{\ell'-1} x^{\left(2^i\right)}$$

That is, the required product of values is spread across eight different accumulators, since for each possible value of one digit, the other digit could have three different values (we recall that the combination 00 does not have a corresponding accumulator). We also notice that the products of values corresponding to positions where the digit is $\bar{1}$ need to be inverted. Algorithm Exp-HE shows these details (shown below as Algorithm 8). Correctness of Algorithm Exp-HE is asserted by Theorem 5.1.

Theorem 5.1 *Given inputs x and ℓ-digit exponent e, with e in signed-digit representation, Algorithm Exp-HE correctly computes the value of x^e.*

Proof:

Without loss of generality, we assume that $2 \mid \ell$ (we can pad with a leading zero digit as needed). Let $\ell' = \frac{\ell}{2}$, and consider the two ℓ'-digit exponent halves:

$$e_{\rm H} = d_{\ell-1} d_{\ell-2} \cdots d_{\ell'+1} d_{\ell'}$$

$$e_{\rm L} = d_{\ell'-1} d_{\ell'-2} \cdots d_2 d_1 d_0$$

The required value x^e can be obtained in terms of $x^{e_{\rm H}}$ and $x^{e_{\rm L}}$ as follows:

$$e = e_{\rm L} + e_{\rm H} 2^{\ell'} \;\Rightarrow\; x^e = x^{e_{\rm L}} \cdot x^{e_{\rm H} 2^{\ell'}} = x^{e_{\rm L}} \cdot (x^{e_{\rm H}})^{\left(2^{\ell'}\right)} \quad (5.3)$$

Since each of the half exponents are themselves numbers in signed-digit representation, the values $x^{e_{\rm L}}$ and $x^{e_{\rm H}}$ are given by:

$$x^{e_{\rm L}} = \left(\prod_{i \in \mathcal{D}_{\rm L}^+} x^{\left(2^i\right)} \right) \cdot \left(\prod_{i \in \mathcal{D}_{\rm L}^-} x^{\left(2^i\right)} \right)^{-1} \quad (5.4)$$

$$x^{e_{\rm H}} = \left(\prod_{i \in \mathcal{D}_{\rm H}^+} x^{\left(2^i\right)} \right) \cdot \left(\prod_{i \in \mathcal{D}_{\rm H}^-} x^{\left(2^i\right)} \right)^{-1} \quad (5.5)$$

Algorithm 8: Exp-HE – Simultaneous processing of half-exponents

Input: x; $e = (d_{\ell-1}d_{\ell-2}\ \cdots\ d_1 d_0)_{\text{S.D.}}$ with ℓ divisible by 2
Returns: x^e

begin
$S \leftarrow x$;
$R_{\bar{1}\bar{1}} \leftarrow 1$, $R_{\bar{1}0} \leftarrow 1$, $R_{\bar{1}1} \leftarrow 1$;
$R_{0\bar{1}} \leftarrow 1$, $R_{01} \leftarrow 1$;
$R_{1\bar{1}} \leftarrow 1$, $R_{10} \leftarrow 1$, $R_{11} \leftarrow 1$;
$\ell' \leftarrow \ell/2$;

for each digit pair $d_{\ell'+i}d_i$ (i from 0 up to $\ell' - 1$) **do**
if $d_{\ell'+i}d_i \neq 00$ **then**
$R_{d_{\ell'+i}d_i} \leftarrow R_{d_{\ell'+i}d_i} \times S$;
end
$S \leftarrow S^2$;
end

$R_{01} \leftarrow R_{01} \times R_{\bar{1}1} \times R_{11} \times (R_{\bar{1}\bar{1}} \times R_{0\bar{1}} \times R_{1\bar{1}})^{-1}$;
$R_{10} \leftarrow R_{10} \times R_{1\bar{1}} \times R_{11} \times (R_{\bar{1}\bar{1}} \times R_{\bar{1}0} \times R_{\bar{1}1})^{-1}$;

repeat ℓ' times:
$R_{10} \leftarrow (R_{10})^2$;
;
return $R_{01} \times R_{10}$;
end

where $\mathcal{D}_{\mathrm{L}}^{+}$ denotes the set $\{\, i \;:\; 0 \leqslant i < \ell',\ d_i = 1 \,\}$, $\mathcal{D}_{\mathrm{L}}^{-}$ the set $\{\, i \;:\; 0 \leqslant i < \ell',\ d_i = \overline{1} \,\}$, $\mathcal{D}_{\mathrm{H}}^{+}$ the set $\{\, i \;:\; 0 \leqslant i < \ell',\ d_{i+\ell'} = 1 \,\}$, and $\mathcal{D}_{\mathrm{H}}^{-}$ the set $\{\, i \;:\; 0 \leqslant i < \ell',\ d_{i+\ell'} = \overline{1} \,\}$.

Consider now the sets $\mathcal{R}_{\overline{1}\overline{1}}$, $\mathcal{R}_{\overline{1}0}$, $\mathcal{R}_{\overline{1}1}$, $\mathcal{R}_{0\overline{1}}$, $\mathcal{R}_{01}$, $\mathcal{R}_{1\overline{1}}$, $\mathcal{R}_{10}$, and $\mathcal{R}_{11}$, where $\mathcal{R}_{\mathrm{d_H d_L}}$ denotes the set $\{\, i \;:\; 0 \leqslant i < \ell',\ d_i = \mathrm{d_L},\ d_{i+\ell'} = \mathrm{d_H} \,\}$.

Clearly, $\mathcal{R}_{\overline{1}1} \subset \mathcal{D}_{\mathrm{L}}^{+}$, $\mathcal{R}_{01} \subset \mathcal{D}_{\mathrm{L}}^{+}$, and $\mathcal{R}_{11} \subset \mathcal{D}_{\mathrm{L}}^{+}$. Furthermore, $\mathcal{R}_{\overline{1}1} \cup \mathcal{R}_{01} \cup \mathcal{R}_{11} = \mathcal{D}_{\mathrm{L}}^{+}$, since $\overline{1}$, 0 and 1 are the only possible values for $\mathrm{d_H}$. Similarly, we have $\mathcal{R}_{\overline{1}\overline{1}} \cup \mathcal{R}_{0\overline{1}} \cup \mathcal{R}_{1\overline{1}} = \mathcal{D}_{\mathrm{L}}^{-}$.

In Algorithm Exp-HE, S is initialized with the value of x, and at the end of each iteration it is squared; this means that at the beginning of iteration i, the value in S is $x^{(2^i)}$. This value of S will be included in the product of values stored in one of the variables $R_{\mathrm{d_H d_L}}$, since all of the $R_{\mathrm{d_H d_L}}$ defined are such that $\mathrm{d_H d_L}$ is not 00; this variable $R_{\mathrm{d_H d_L}}$ is precisely the one corresponding to the digit pair $\mathrm{d_H d_L}$. Thus, the values stored in each variable $R_{\mathrm{d_H d_L}}$ are:

$$R_{\mathrm{d_H d_L}} = \prod_{i \,\in\, \mathcal{R}_{\mathrm{d_H d_L}}} x^{(2^i)} \tag{5.6}$$

Therefore, we have

$$R_{01} \times R_{\overline{1}1} \times R_{11} = \prod_{i \,\in\, \mathcal{R}_{01} \cup \mathcal{R}_{\overline{1}1} \cup \mathcal{R}_{11}} x^{(2^i)} \tag{5.7}$$

But $\mathcal{R}_{01} \cup \mathcal{R}_{\overline{1}1} \cup \mathcal{R}_{11} = \mathcal{D}_{\mathrm{L}}^{+}$, and thus

$$R_{01} \times R_{\overline{1}1} \times R_{11} = \prod_{i \,\in\, \mathcal{D}_{\mathrm{L}}^{+}} x^{(2^i)} \tag{5.8}$$

We also have

$$R_{\overline{1}\overline{1}} \times R_{0\overline{1}} \times R_{1\overline{1}} = \prod_{i \,\in\, \mathcal{R}_{\overline{1}\overline{1}} \cup \mathcal{R}_{0\overline{1}} \cup \mathcal{R}_{1\overline{1}}} x^{(2^i)} \tag{5.9}$$

With $\mathcal{R}_{\overline{1}\overline{1}} \cup \mathcal{R}_{0\overline{1}} \cup \mathcal{R}_{1\overline{1}} = \mathcal{D}_{\mathrm{L}}^{-}$, and thus

$$R_{\overline{1}\overline{1}} \times R_{0\overline{1}} \times R_{1\overline{1}} = \prod_{i \,\in\, \mathcal{D}_{\mathrm{L}}^{-}} x^{(2^i)} \tag{5.10}$$

Combining equations (5.8) and (5.10) with Equation (5.4), we see that the final value assigned to R_{01} is $x^{e_{\mathrm{L}}}$.

By an identical argument, we have that the value assigned to R_{10} by the end of the ℓ' iterations of the loop is $x^{e_{\mathrm{H}}}$. This value is then repeatedly squared ℓ' times, meaning that the final value stored in R_{01} is $(x^{e_{\mathrm{H}}})^{(2^{\ell'})}$. Since the output of the algorithm is the product of R_{01} and R_{10}, Equation (5.3) shows that the output is x^e, completing the proof. □

We observe that for the same inputs, with e in NAF representation, Theorem 5.1 still applies, since NAF is a particular case of signed-digit representation.

5.3.1 Exponent in NAF Representation

The use of NAF for exponent representation exhibits the same advantage as for the case of straightforward binary exponentiation: with lowest Hamming Weight among all possible signed-digit representations for a given value, we reduce the number of multiplications for the straightforward binary exponentiation to one third of the bit length of the exponent on average.

Proposition 5.2 *Given inputs x and ℓ-digit exponent e, with e in NAF representation, algorithm Exp-HE executes $\frac{5}{18}\ell$ multiplications on average.*

Proof:

For each i, $0 \leqslant i < \ell'$, where $\ell' = \frac{\ell}{2}$, a multiplication takes place if $d_{i+\ell'}d_i \neq 0$

For sufficiently large exponents, we can reasonably assume that $\Pr\{d_k = 0\} = \frac{2}{3}$ [5]. Under the assumption that digits d_i and $d_{i+\ell'}$ are independent, we have that

$$\Pr\{d_{i+\ell'}d_i = 00\} = \frac{4}{9} \Rightarrow \Pr\{d_{i+\ell'}d_i \neq 00\} = \frac{5}{9}$$

Thus, the average number of multiplications is $\frac{5}{9}\ell' = \frac{5}{18}\ell$ □

5.3.2 Exponent Halves in Joint Sparse Form Representation

Further improvement is obtained by observing that this technique of simultaneously processing half exponents is similar to, or at least shares certain aspects with, multi-exponentiation; Joint Sparse Form (JSF) representation of exponent pairs has been suggested as a means to optimize the signed-digit representation so that as many digit pairs (columns) as possible are 00. It has been shown that with JSF, we can obtain representations for each of the exponents (in our case, each of the exponent halves) with one half of the columns being 00 on average [77]. This constitutes a non-negligible improvement over the use of NAF, since the number of multiplications with JSF is half the bit length of the exponents, compared to five ninths for NAF.

Proposition 5.3 *Given inputs x and ℓ-digit exponent e, with exponent halves e_{H} and e_{L} jointly represented in JSF, algorithm Exp-HE executes $\frac{1}{4}\ell$ multiplications on average.*

Proof:

The statement follows directly from the fact that JSF achieves a representation of the exponent halves with one half of the digit pairs (columns) being 00 on average. □

Notice that with JSF, the representation for each half exponent remains a legitimate signed-digit representation. Thus, Theorem 3.1 holds, and algorithm Exp-HE remains valid when using JSF representation for the exponent halves.

5.4 Processing Multi-digit Blocks

We now discuss two extensions to the methods presented in the previous sections, where several columns are combined for simultaneous processing. As we will see, these methods lead to better asymptotic performance, but with a larger constant factor, making them suitable for large exponents (approx. above 1000 bits). In light of recent NIST recommendations, including the use of RSA with exponents of 2048 bits or above [6], it becomes important to consider techniques that target this range of exponent lengths.

As will become apparent, there are fundamental differences between these and existing techniques that use multi-digit processing for increased performance, such as those described in [56]; in our case, the characteristics of the algorithms are constrained by the fact that they have to be suitable to be combined with the SABM buffering technique. Also, our proposed methods have an advantage over the method presented in [65] in that they do not require Look-Up Tables or precomputations, making them equally well suited for the cases of fixed and non-fixed base.

5.4.1 Two-digit Blocks Derived from NAF Representation

We first present an extension of the use of NAF for the half exponents that allows us to obtain an even better performance than that obtained through the use of JSF.

If we split the exponent into blocks of two digits (two columns, since we do this for the two exponent halves), the basic property of NAF guarantees that for each block of each half-exponent, at least one of the two digits must be 0. Thus, the only possible values for the digit pair are $0\overline{1}$, 00, 01, 10, and $\overline{1}0$, corresponding to numeric values -1, 0, 1, 2, and -2. Thus, if we take two-digit blocks and think of these blocks as digits in base-4 signed-digit representation, we can adapt the algorithm to work with these parameters.

The exponentiation algorithm is easily modified to work with this non-binary representation of ℓ_2-digit exponent $e = \sum_{i=0}^{\ell_2-1} d_i 4^i$ with $d_i \in \{\overline{2}, \overline{1}, 0, 1, 2\}$, as shown below:

$$\begin{aligned} x^e &= x^{\left(\sum_{i=0}^{\ell_2-1} d_i 4^i\right)} = x^{\left(\sum_{\substack{i=0\\d_i=1}}^{\ell_2-1} 4^i \,+\, 2\sum_{\substack{i=0\\d_i=2}}^{\ell_2-1} 4^i \,-\, \sum_{\substack{i=0\\d_i=\overline{1}}}^{\ell_2-1} 4^i \,-\, 2\sum_{\substack{i=0\\d_i=\overline{2}}}^{\ell_2-1} 4^i\right)} \\ &= \left(\prod_{d_i=1} x^{(4^i)}\right) \times \left(\prod_{d_i=2} x^{(4^i)}\right)^2 \times \left(\prod_{d_i=\overline{1}} x^{(4^i)}\right)^{-1} \times \left(\prod_{d_i=\overline{2}} x^{(4^i)}\right)^{-2} \\ &= (\mathbf{P}_1)\,(\mathbf{P}_2)^2\,(\mathbf{P}_{\overline{1}})^{-1}\,(\mathbf{P}_{\overline{2}})^{-2} \end{aligned} \tag{5.11}$$

where $\mathbf{P}_{\mathrm{B}}$ denotes the product corresponding to $d_i = \mathrm{B}$.

From Equation (5.11), it is clear that algorithm Exp-HE can be modified by adding additional accumulators R_{xy} for the additional digit values 2 and $\overline{2}$. That is, we need accumulators $R_{\overline{2}\overline{2}}$, $R_{\overline{2}\overline{1}}$, $R_{\overline{2}0}$, $R_{\overline{2}1}$, $R_{\overline{2}2}$, $R_{\overline{1}\overline{2}}$, etc.

Equation (5.11) can be rearranged for use in the context of ECC: since inversion is a virtually free operation in ECC, we can group the two squarings (doublings, in the context of ECC) into a single operation, at the cost of one additional inversion.

We observe that asymptotic performance for this method is better than with the use of JSF. To show this, we first present the following lemma, addressing adjacent digits in NAF representations:

Lemma 5.4 *Let x be a randomly chosen ℓ-bit non-negative value (that is, with uniform distribution in the interval $[0, 2^\ell - 1]$), and let $d_\ell d_{\ell-1} \cdots d_1 d_0$ be its NAF representation. For sufficiently large values of ℓ, the probability of contiguous zeros, $\Pr\{d_{k+1} d_k = 00\}$ for k even approaches $\frac{1}{3}$ as k becomes large.*

Proof:

We will prove the statement based on a procedure to construct the NAF representation from the standard binary representation by processing individual bits from right to left (LSB to MSB) to generate the digits d_i. Since the procedure is different from the standard, commonly known algorithm (shown for example as Algorithm 3.30 in [56]), we include a description in Appendix B.

The procedure works in an "online" manner, processing each input bit to generate new output digits, with the key detail that each bit in the standard binary representation is independent of every other bit (even for adjacent bits) and can take values 0 and 1 with equal probability.

This allows us to obtain the following recurrence relations for the possible outputs of the algorithm at each step (i.e., upon processing each input bit), taking into account that $\Pr\{b_n = 0\} = \Pr\{b_n = 1\} = \frac{1}{2}$; $P_0(k)$ denotes the probability of producing a digit 0 with no carry at iteration k (that is, $\Pr\{d_k = 0, \mathrm{C} = 0\}$), $P_1(k) = \Pr\{d_k = 1\}$, and $P_\mathrm{C}(k)$ denotes the probability of producing a carry at iteration k:

$$\begin{aligned} P_0(n) &= \Pr\{b_n = 0\} P_0(n-1) + \Pr\{b_n = 0\} P_1(n-1) \\ &= \tfrac{1}{2} P_0(n-1) + \tfrac{1}{2} P_1(n-1) \qquad (5.12) \\ P_\mathrm{C}(n) &= \Pr\{b_n = 1\} P_1(n-1) + \Pr\{b_n = 1\} P_\mathrm{C}(n-1) \\ &= \tfrac{1}{2} P_1(n-1) + \tfrac{1}{2} P_\mathrm{C}(n-1) \qquad (5.13) \\ P_1(n) &= 1 - P_0(n) - P_\mathrm{C}(n) \qquad (5.14) \end{aligned}$$

Equation (5.14) corresponds to the fact that these are the only three possibilities at each iteration, so the corresponding probabilities must add to 1.

Since we are interested in the probability of adjacent zeros, we only need to solve for $P_0(n)$ and $P_\mathrm{C}(n)$, so we rewrite Equation (5.14) at $n-1$ and substitute in the other two

equations, obtaining

$$\begin{aligned} P_0(n) &= \tfrac{1}{2}(1 - P_{\mathrm{C}}(n-1)) \\ P_{\mathrm{C}}(n) &= \tfrac{1}{2}(1 - P_0(n-1)) \end{aligned}$$

Rewriting again for the left-hand sides at $n-1$ and substituting in each other, we finally obtain the following recurrence relations:

$$\begin{aligned} P_0(n) &= \tfrac{1}{4} + \tfrac{1}{4}P_0(n-2) \\ P_{\mathrm{C}}(n) &= \tfrac{1}{4} + \tfrac{1}{4}P_{\mathrm{C}}(n-2) \end{aligned}$$

With initial conditions being $P_0(0) = P_0(1) = \frac{1}{2}$, $P_{\mathrm{C}}(0) = 0$, and $P_{\mathrm{C}}(1) = \frac{1}{4}$ (these are trivially obtained by counting occurrences in the four possible two-bit combinations).

The above recurrence relations are easily solved by repeated substitution, obtaining identical solutions for both (the difference given by the different initial conditions):

$$P(n) = \tfrac{1}{3} + \left(\tfrac{1}{2}\right)^n (P(0) - \tfrac{1}{3}) \quad \text{(for } n \text{ even)}$$

Since we are interested in processing pairs of bits, we want the probability of digits $d_{2k}d_{2k+1}$ being 00, and thus, we only need to obtain the above solution for n even.

$$\begin{aligned} P_0(n) &= \tfrac{1}{3} + \tfrac{1}{6}\left(\tfrac{1}{2}\right)^n \\ P_{\mathrm{C}}(n) &= \tfrac{1}{3}\left(1 - \left(\tfrac{1}{2}\right)^n\right) \end{aligned}$$

From this, we obtain, for n even:

$$\begin{aligned} \Pr\{d_{n+1} = 0, d_n = 0\} &= \Pr\{b_{n+1} = 0, d_n = 0\} + \Pr\{b_{n+1} = 1, C = 0\} \\ &= \Pr\{b_{n+1} = 0\} \cdot P_0(n) + \Pr\{b_{n+1} = 1\} \cdot P_{\mathrm{C}}(n) \\ &= \tfrac{1}{2}(P_0(n) + P_{\mathrm{C}}(n)) \\ &= \tfrac{1}{3}\left(1 - \left(\tfrac{1}{2}\right)^{n+2}\right) \end{aligned} \tag{5.15}$$

From Equation (5.15), we clearly observe exponential convergence towards $\frac{1}{3}$.

To complete the proof, we should mention the fact that these probabilities correspond to the probabilities of 0 at the given positions for the NAF representations of large numbers. Indeed, the construction procedure (see Appendix) is such that once the most-significant-digit at some iteration is 0, the next iteration can not make this change (and as a consequence, the same holds for the two most recent digits being 00), and this regardless of whether there is a carry or not at that iteration. Conversely, if we have a 1 at some iteration (the only possible nonzero as the most-significant digit), the next iteration can only make it change to $\overline{1}$, and not to 0. □

Proposition 5.5 *Given inputs x and ℓ-digit exponent e in NAF representation, algorithm Exp-HE-Base4 executes $\frac{2}{9}\ell$ multiplications on average, and never executes more than $\frac{1}{4}\ell$ multiplications.*

Proof:

The upper-bound of $\frac{1}{4}\ell$ multiplications follows directly from the fact that when processing two-digit blocks as a base-4 signed-digit representation, we never have more than one multiplication per two-digit block, meaning that we guarantee the number of multiplications to be half the bit length of the (half-)exponent in the worst case.

For the average case, we have that a fraction of the two-digit blocks have four zeros, meaning that no multiplication is required for those blocks. From Lemma 5.4, given a sufficiently large exponent, we have that $\Pr\{d_{i+1}d_i = 00\} = \frac{1}{3}$.[1]

Under the assumption of independence of the digits from the lower and upper halves of the exponent, we have that

$$\Pr\{d_{i+1}d_i = 00,\ d_{\ell'+i+1}d_{\ell'+i} = 00\} = \frac{1}{9}$$

where $\ell' = \frac{\ell}{2}$. The stated result follows immediately. □

We notice that we only have extra digits 2 and -2, as a consequence of the exponent being in NAF representation; thus, the additional power requires a single squaring, as opposed to having powers 3 and -3, requiring one squaring and one multiplication for the exponentiation corresponding to those accumulators.

An additional advantage of this approach is that when grouping two-bit blocks for processing, instead of executing two squarings, we can obtain the fourth power directly, without having to explicitly compute an intermediate result, potentially obtaining better performance than with two successive squarings. This has been shown to be the case for ECC, where, under certain conditions, computing $4P$ directly can be faster than doubling twice [35].

Algorithm Exp-HE-Base4 is shown below (as Algorithm 9) for the case of inversion being less expensive than squaring.

5.4.2 Three-digit Blocks Derived from JSF Representation

Algorithm Exp-HE-Base4 takes advantage of the constraints that NAF representation imposes on adjacent digits, and therefore on any two-digit blocks; following the same

[1] The approximation is reasonable, since the probability converges to 1/3 quite rapidly, and the exponent lengths are always in the order of at least hundreds of bits.

Algorithm 9: Exp-HE-Base4 – Simultaneous Processing of Half-Exponents, Base-4 Mode

Input: x; $e = (d_{\ell-1}d_{\ell-2}\ \cdots\ d_1 d_0)_{\text{NAF}}$ with ℓ divisible by 4
Returns: x^e

begin
$R_{ij} \leftarrow 1\ \ (\forall\, i,j \in \{\overline{2},\overline{1},0,1,2\},\ \ ij \neq 00)$;
$S \leftarrow x$;
$\ell' \leftarrow \ell/2$;

for i **from** *0* **up to** $\ell' - 2$ **in steps of** 2 **do**
$D_{\text{H}} \leftarrow d_{\ell'+i} + 2 \cdot d_{\ell'+i+1}$; $D_{\text{L}} \leftarrow d_i + 2 \cdot d_{i+1}$;
if $D_{\text{H}}D_{\text{L}} \neq 00$ **then**
$R_{D_{\text{H}}D_{\text{L}}} \leftarrow R_{D_{\text{H}}D_{\text{L}}} \times S$;
end
$S \leftarrow S^4$;
end

$R_{02} \leftarrow R_{02} \times R_{\overline{2}2} \times R_{\overline{1}2} \times R_{12} \times R_{22} \times (R_{\overline{2}\overline{2}} \times R_{\overline{1}\overline{2}} \times R_{0\overline{2}} \times R_{1\overline{2}} \times R_{2\overline{2}})^{-1}$;
$R_{01} \leftarrow R_{01} \times R_{\overline{2}1} \times R_{\overline{1}1} \times R_{11} \times R_{21} \times (R_{\overline{2}\overline{1}} \times R_{\overline{1}\overline{1}} \times R_{0\overline{1}} \times R_{1\overline{1}} \times R_{2\overline{1}})^{-1} \times (R_{02})^2$;

$R_{20} \leftarrow R_{20} \times R_{2\overline{2}} \times R_{2\overline{1}} \times R_{21} \times R_{22} \times (R_{\overline{2}\overline{2}} \times R_{\overline{2}\overline{1}} \times R_{\overline{2}0} \times R_{\overline{2}1} \times R_{\overline{2}\overline{2}})^{-1}$;
$R_{10} \leftarrow R_{10} \times R_{1\overline{2}} \times R_{1\overline{1}} \times R_{11} \times R_{12} \times (R_{\overline{1}\overline{2}} \times R_{\overline{1}\overline{1}} \times R_{\overline{1}0} \times R_{\overline{1}1} \times R_{\overline{1}\overline{2}})^{-1} \times (R_{20})^2$;

repeat ℓ' Times:
$R_{10} \leftarrow (R_{10})^2$;
;
return $R_{01} \times R_{10}$;
end

idea, we observe that JSF representation does impose constraints for three-digit blocks, which allows us to extend the method to a base-8 processing while taking advantage of the reduced number of base-8 digit combinations due to the constraints present in a three-digit block.

Following the same idea and using the same notation as for the base-4 case, we have a representation of an ℓ_3-digit exponent $e = \sum_{i=0}^{\ell_3-1} d_i 8^i,\ \ d_i \in \{0, \pm1, \pm2, \pm3, \pm4, \pm5, \pm6\}$. For this base-8 representation, we obtain:

$$\begin{aligned} x^e &= x^{\left(\sum_{i=0}^{\ell_3-1} d_i 8^i\right)} \\ &= (\mathbf{P}_1)\ (\mathbf{P}_2)^2\,(\mathbf{P}_3)^3\,(\mathbf{P}_4)^4\,(\mathbf{P}_5)^5\,(\mathbf{P}_6)^6 \\ &\quad (\mathbf{P}_{\overline{1}})^{-1}\,(\mathbf{P}_{\overline{2}})^{-2}\,(\mathbf{P}_{\overline{3}})^{-3}\,(\mathbf{P}_{\overline{4}})^{-4}\,(\mathbf{P}_{\overline{5}})^{-5}\,(\mathbf{P}_{\overline{6}})^{-6} \end{aligned} \tag{5.16}$$

From Equation (5.16), we see that the exponentiation procedure remains essentially the same, with a higher cost in terms of storage and post-processing (the step where the various accumulators are combined into the exponentiation results). However, the JSF representation introduces constraints in triplets of contiguous digits [77], reducing the number of combinations of values, and thus reducing the number of accumulators. Specifically, we have the following properties for JSF [77] that affect our method:

- At least one column is zero in any three contiguous columns — this means that a value 7 can not occur for any of the two digits, and also, combinations such as 5-2 or 1-6 can not occur.
- If $e_{\mathrm{H}i+1}e_{\mathrm{H}i} \neq 0$, then $e_{\mathrm{L}i+1} \neq 0$ and $e_{\mathrm{L}i} = 0$, and if $e_{\mathrm{L}i+1}e_{\mathrm{L}i} \neq 0$, then $e_{\mathrm{H}i+1} \neq 0$ and $e_{\mathrm{H}i} = 0$ — this means that combinations such as 6-6, or 6-2 can not occur.

The number of accumulators is reduced from 224 (15 possible values for each digit, minus the combination 00) to 120 — an important reduction, but still leaving a considerably large number of accumulators, given that they incur both additional storage, and computational cost due to increased post-processing.

The interesting aspect of this method is its (asymptotic) computational efficiency; three digits of each half-exponent are processed with a single multiplication, bringing the average number of multiplications down to one sixth of the length of the exponent. We need to additionally factor in the fraction of blocks that are all-zeros (i.e., the base-8 digit pair is 00), such that no multiplication is required for those digit pairs. This fraction is of course lower than for the previous cases, since we're dealing with triplets of signed binary digits, with the constraints given by the JSF. We experimentally obtained this fraction to be approximately $\frac{1}{96}$, with which we obtain an average number of multiplications of 0.165ℓ.

We omit any additional details or a step-by-step diagram for the algorithm, since the details follow the same idea as algorithm Exp-HE-Base4.

5.5 SABM with Simultaneous Processing of Half-Exponents

In the previous sections, we discussed various algorithms that are the subject of this work, but we presented them in their SPA-vulnerable version, for the purpose of discussing the computational efficiency of each of the variants. All of the algorithms presented in the previous sections can be modified to incorporate the buffering aspect of the SABM technique, thus adding resistance to SPA while introducing zero computational overhead. The idea is similar for all the variants of the algorithm — instead of conditionally executing the multiplication, we buffer the term to be multiplied, and then execute the multiplications at a fixed rate.

When simultaneously processing half-exponents, the algorithm requires a selection of the accumulator where the multiplication should be performed; thus, in this case, we need to buffer the combination of the value to be multiplied and the digit pair for the selection of the accumulator (or equivalently, a direct reference, e.g., in the form of a pointer, to the selected accumulator could be used).

Algorithm 10 shows the details for the case of exponent in NAF representation. Notice the pre-filling of the buffer to half its capacity on average: the probability of insertion on the buffer is $p = \frac{5}{9}$, so we process $p^{-1}|\texttt{Buff}_\text{S}|/2$ exponent digits before starting to extract elements from the buffer; at the end of the first loop, the buffer will be at half capacity on average, so we need to process any remaining elements.

The idea is almost identical and directly applicable to the other forms discussed in the previous section. In particular, for JSF representation of the exponent halves, the algorithm remains the same (except for a pre-processing stage to convert the exponent representation to JSF, or the precondition that the input be in this form).

We omit proofs of correctness for these algorithms, since it is straightforward to observe that the buffering aspect only changes the time at which the operations take place, without affecting the values being multiplied together.

5.6 Performance Comparison

We now focus on the performance of the various methods proposed in this work, and present experimental results that confirm our analysis.

5.6.1 Analytic Comparison

Table 5.1 summarizes the differences in performance of the various techniques described in this section, and compares against existing solutions, showing the right-to-left exponen-

Algorithm 10: SABM-HE – SABM with Simultaneous Processing of Half-Exponents, NAF

Input: x; $e = (d_{\ell-1}d_{\ell-2}\ \cdots\ d_1d_0)_{\text{NAF}}$ with ℓ divisible by 2
Returns: x^e

begin
$S \leftarrow x$;
$R_{\bar{1}\bar{1}} \leftarrow 1,\ R_{\bar{1}0} \leftarrow 1,\ R_{\bar{1}1} \leftarrow 1$;
$R_{0\bar{1}} \leftarrow 1,\ R_{01} \leftarrow 1$;
$R_{1\bar{1}} \leftarrow 1,\ R_{10} \leftarrow 1,\ R_{11} \leftarrow 1$;
$\ell' \leftarrow \ell/2$;

for each digit pair $d_{\ell'+i}d_i$ (i from 0 up to $\ell'-1$) **do**
if $d_{\ell'+i}d_i \neq 00$ **then**
$< S, d_{\ell'+i}d_i > \ \rightarrow\ \text{Buff}_\text{S}$;
end
$S \leftarrow S^2$;
if $i > 9 \cdot |\text{Buff}_\text{S}|/10$ **and** i mod 9 is even **then**
$< Tmp, d_\text{H}d_\text{L} > \ \leftarrow\ \text{Buff}_\text{S}$;
$R_{d_\text{H}d_\text{L}} \leftarrow R_{d_\text{H}d_\text{L}} \times Tmp$;
end
end

while Buff_S is not empty **do**
$< Tmp, d_\text{H}d_\text{L} > \ \leftarrow\ \text{Buff}_\text{S}$;
$R_{d_\text{H}d_\text{L}} \leftarrow R_{d_\text{H}d_\text{L}} \times Tmp$;
end

$R_{01} \leftarrow R_{01} \times R_{\bar{1}1} \times R_{11} \times (R_{\bar{1}\bar{1}} \times R_{0\bar{1}} \times R_{1\bar{1}})^{-1}$;
$R_{10} \leftarrow R_{10} \times R_{1\bar{1}} \times R_{11} \times (R_{\bar{1}\bar{1}} \times R_{\bar{1}0} \times R_{\bar{1}1})^{-1}$;

repeat ℓ' Times:
$R_{10} \leftarrow (R_{10})^2$;
;
return $R_{01} \times R_{10}$;
end

tiation performance as a baseline. Values shown are *average* number of multiplications required to execute an exponentiation with an ℓ-bit exponent (we recall that all the methods listed, except for Joye's method, require exactly ℓ squarings in addition to the number of multiplications shown), and *average* amount of units of time to execute, where by convention, squaring routines execute in 1 unit of time and multiplication routines execute in 1.2 units of time (a typical ratio for practical implementations [39]). We observe that even for different ratios of multiplication to squaring times, the number of squarings is the same for all the methods; thus, we unconditionally benefit from a reduced number of multiplications.

	Multiplications		Execution Time	
	Binary	NAF/S.D.	Binary	NAF/S.D.
R-T-L	0.5ℓ	0.33ℓ	1.6ℓ	1.4ℓ
S-A-A-M	ℓ	ℓ	2.2ℓ	2.2ℓ
Joye	$--$	$--$	1.8ℓ	1.6ℓ
Sun et al.	$0.5\ell + O(1)$	$--$	$1.6\ell + O(1)$	$--$
SABM	0.5ℓ	0.33ℓ	1.6ℓ	1.4ℓ
SABM-HE (*)	$0.375\ell + O(1)$	$0.275\ell + O(1)$	$1.45\ell + O(1)$	$1.33\ell + O(1)$
SABM-HE-JSF (*)	$--$	$0.25\ell + O(1)$	$--$	$1.3\ell + O(1)$
SABM-HE-Base4 (*)	$--$	$0.22\ell + O(1)$	$--$	$1.264\ell + O(1)$
SABM-HE-Base8 (*)	$--$	$\mathbf{0.165\ell + O(1)}$	$--$	$\mathbf{1.198\ell + O(1)}$

(*) This work

Table 5.1: Performance Comparison of S.D. Exponentiation Algorithms.

5.6.2 Experimental Results

As part of this study, we implemented several of the methods for the purpose of experimentally verifying their efficiency; in particular, the implementations did not include the buffering aspect, since in our view, it seems rather clear that this aspect does not affect the computational performance of the methods. In addition to the fully optimized (and thus, SPA vulnerable) RTL exponentiation with NAF exponent, used as a baseline for comparison, we implemented the methods Exp-HE (using NAF exponent), Exp-HE-JSF, and Exp-HE-Base4. All the implementations are based on the GMP library (version 5.0.2, the latest version at the time of this work) for the underlying arithmetic operations [24]. We tested the methods with exponent lengths of 256 and 512 bits, in the range of typical ECC applications,[2] and also 1024, 2048 and 4096; 1024 and 2048 are in the typical range of RSA applications [6], assuming a CRT-based implementation. Even though

[2] Though all the implementations use exponentiation based on modular integer arithmetic, performance comparisons should still be meaningful when combining with exponent lengths typical for ECC,

4096 bits is not widely used in practical applications, we include it since it confirms the asymptotic behaviour expected as exponent lengths become larger.

Table 5.2 shows the results; measurements are actual execution time of the exponentiation routines (excluding startup and initialization time for the library facilities). Multiple measurements (1000) with randomly chosen exponents were performed, and Table 5.2 shows the average value. The implementations were compiled and executed on a low-power Intel Atom processor N270 [17] system running Ubuntu Linux 10.04LTS. CPU frequency scaling was disabled, as well as the graphical interface, networking, and all other applications, to avoid any disruption on the measurements.

	256 bits	512 bits	1024 bits	2048 bits	4096 bits
R-T-L (SPA vulnerable)	282μs	1.42ms	8.19ms	49ms	314.3ms
Exp-HE (NAF)	301μs	1.43ms	7.93ms	46.8ms	298ms
Exp-HE-JSF	290μs	1.39ms	7.72ms	45.5ms	289.7ms
Exp-HE-Base4	302μs	1.395ms	7.61ms	44.5ms	282.6ms

Table 5.2: Execution time of exponentiation algorithms.

The results are consistent with the expected execution times of the various methods; the measurements confirm that the Base-4 method is actually at disadvantage for short keys, such as those typically used in ECC, but it is asymptotically more efficient, as shown by the results for larger exponent sizes.

5.7 Discussion and Concluding Remarks

Several extensions to our previous results were presented, with various degrees of improvement with respect to previous works and various trade-offs. The additional storage required for the extra accumulators with respect to the method in our previous work (eight accumulators for signed-digit/NAF exponent vs. three accumulators for standard binary exponent in our previous work) should be offset by the fact that the variance for the distribution of nonzeros is smaller when using signed-digit representations, thus leading to a reduced buffer size requirement for a given probability of buffer failure.

More importantly, the improvement in performance is substantial when using signed-digit representations for the exponent, presenting an interesting situation where we improve (or perhaps just maintain) the storage requirements and still observe a considerable improvement in terms of computational cost. We recall that this reduction in computational cost also leads to a reduction in power consumption, which could be a critical aspect for systems relying on battery power, such as hand-held mobile devices.

illustrating the usefulness of the various methods for ECC applications.

In the various methods proposed in this work, performance in terms of asymptotic computational cost has distinct values and the various methods are clearly ranked by their efficiency; however, the cost hidden in the O(1) expression (the constant number of operations in the post-processing) is different for each of the various methods, with the optimal trade-off — at least for the typical exponent bit lengths used in ECC — possibly being the method using JSF; the improvement derived from using JSF for the exponent halves comes at no cost whatsoever, since the algorithm is identical with respect to the version that uses NAF. However, the method processing two-digit blocks incurs additional computational cost in terms of post-processing: multiplications, squarings, and inversions to combine the various accumulators into the required results. Given the higher number of multiplications in the post-processing stage, for exponents below 1000 bits, the NAF or JSF methods actually require fewer multiplications, and thus this Base-4 method is not particularly attractive for ECC protocols, where the typical exponent lengths are in the hundreds of bits. However, the Base-4 exhibits superior asymptotic performance; with the threshold being around 1000 bits, the method is suitable for protocols involving RSA with the currently NIST-recommended key sizes [6]. The same holds for the Base-8 method, where the number of operations in the post-processing stage makes it unattractive for the typical exponent bit lengths used in ECC, even though its asymptotic performance is considerably better than any of the other methods.

Chapter 6

Non-Intrusive Program Tracing Through Side-Channel Analysis

This chapter presents our contribution merging, or rather combining, the areas of side-channel analysis, specifically SPA, with the area of embedded systems. Specifically, we propose a technique that draws upon some of the basic ideas from the field of side-channel analysis and applies them to the debugging of embedded systems at their deployment or production stages.

This work has been accepted as a Full Paper in Languages, Compilers and Tools for Embedded Systems (LCTES-2013) [60]. The text and contents in this chapter are based on this work.

6.1 Motivation

Debugging is one of the hardest aspects of embedded software development. The task is especially hard when the faulty behavior is observed at the production or deployment stage, when the software no longer has any auxiliary components dedicated to assist in the debugging task [13]. For systems at this stage of the development cycle, non-intrusive observation of the system's behavior is likely the only available technique — developers are no longer allowed to modify the source code, or even re-compile to include or activate the debugging tools. Furthermore, if we need to restart the device to enable any available debugging techniques, we may not be able to reproduce the faulty behavior that the device was exhibiting. Without these debugging tools usually available in earlier phases of development, developers may be limited to non-intrusive observation, which often provides insufficient information to infer the cause of the problem and identify and fix the bug.

6.2 Our Contributions

In this work, we present a novel approach for non-intrusive debugging of deployed embedded systems. A device can be observed and an output indicating the sequence of executed code is produced, without having to modify anything in the device or even restart it. The approach is non-intrusive both from the hardware and software perspectives in that it does not require any modifications or instrumentation to the software or any hardware modifications or extra connections except for signals external to the device.

The technique is rooted in cryptography, in particular the area of side-channel analysis, focusing on power consumption (though the underlying techniques are in principle applicable to EM emissions), where the relationship between the CPU operations and power consumption is exploited. To determine power consumption, a current sensing shunt resistor is placed in series with the Power-In signal going to the Microcontroller Unit (MCU),[1] producing a voltage proportional to the current being consumed. The resistor is selected to produce a voltage in the range of a few millivolts, thus not affecting the operation of the device. Our technique expands the scope of the cryptographic techniques so that we recover the sequence of operations executed by a processor, as opposed to simply one piece of data accessed during a particular operation of the device. To this end, we use digital signal processing techniques (in particular, spectral analysis) to extract *features* of the signal (the power trace) that allow us to match sections of the power trace against fragments of the source code through the use of statistical pattern recognition techniques [82].

Our approach exhibits some fundamental differences with respect to side-channel attacks, requiring additional, novel approaches to process the power trace — clearly, as we can see from the descriptions and discussions in the previous chapters, these cryptanalytic techniques, as they exist, are not directly applicable to the debugging of embedded software, since they focus on obtaining specific pieces of secret data embedded in the device (and inaccessible through "legitimate" means), and they typically require interaction and direct control over what the target device is executing. On the other hand, the goal when tracing and debugging a deployed embedded system is to analyze an operating device for which we have observed a faulty behavior, and obtain information allowing us to identify and fix the bug. It may be essential that we allow the device to continue its operation without restarting it or in any way exerting control over what the device is doing; otherwise, we could lead the device to a state where we may not be able to reproduce the faulty behavior.

As secondary contributions or highlights of this work, we have the following aspects:

- One of the important highlights of this work is the fact that the system works on a standard personal computer (PC), capturing the power traces through the

[1] Though the technique is applicable to both CPUs and MCUs, we use MCU throughout the chapter, to simplify the text, and also since it is the more likely target for our technique.

recording input of the sound card, avoiding the need of expensive and bulky pieces of equipment. A standard, reasonably high-quality sound card (24-bits, 192kHz sampling rate, nowadays available at prices below $200) suffices to make the system work on a wide range of microprocessors and microcontroller units. To the best of our knowledge, this approach has not been presented in the existing literature. Notice that we are referring to the use of a sound card to capture an electrical signal, and not to capture sound, like in the case of *acoustic* cryptanalysis [76].

Among the advantages of using a sound card, perhaps the most obvious is the practical aspect of using off-the-shelf inexpensive equipment that is widely available and mainstream. The main disadvantage is the rather low limit for the sampling frequency, which certainly limits the range of target devices for which our approach is applicable. A potential secondary disadvantage is the fact that sound cards typically block DC, which for some cases it could be a valuable feature in terms of increasing the performance of our proposed technique.

- An actual practical implementation of our tracing system could be used for monitoring as an *intrusion detection system* (IDS) [7] for embedded systems. In the wake of threats like Stuxnet [50], the field of embedded systems security gets increased attention and one should definitely consider adapting tools like IDSs, classically viewed as applicable to servers and networks, to embedded systems as well. Unlike a software-based IDS embedded in the device, our approach could lead to a tamper-proof IDS, given that the monitoring system is physically independent of the device being monitored and the software running in it; thus, any malware that tampers with the functionality of the device will not be able to tamper with the IDS and as a consequence, any anomaly in the device's behavior will most likely be detected.

  Additionally, this IDS functionality could find interesting applications in areas such as device fingerprinting for Intellectual Property (IP) protection or other forms of Digital Rights Management, and many other areas where monitoring of secure co-processors for tamper detection purposes could be useful (for example, see [83]).

6.3 Our Proposed Technique

As briefly described in the previous section, our proposed technique is centered around the idea of non-intrusively measuring power consumption as a function of time (i.e., capturing power traces), and use the relationship between what the processor is executing and the power consumption, combined with statistical processing to determine the sequence of instructions that were executed, thus assisting in the debugging process. It is reasonably likely that this information would be valuable for the purpose of identifying and fixing the bug when faulty behavior is exhibited by a device. We observe that in

the context of embedded systems, this relationship between operations being executed and power consumption has been used for the purpose of estimating or minimizing power consumption, obtaining power consumption as a function of the executed instructions. Going in the other direction may be seen as a far bigger challenge, for at least two reasons: (1) the operation being executed is not uniquely determined as a function of the power consumption; thus, information about the progression of power consumption through an interval of time may be needed, combined with statistical processing; and (2) we need to get around the "polluting" effect of the data the processor is working with (i.e., the same operation with different data produces a different amount of power consumption) and the measurement noise.

One of our important assumptions derives from the use of statistical pattern recognition techniques: our goal is to *classify* a given segment of execution as an instance of one of the possible fragments of source code according to a database, given noisy observations that in principle provide enough statistical information to determine the most likely fragment of code that produced such observation [82]. Since we use this technique for tracing and debugging, we know that the source code will be accessible, and this leads us to the assumption that the set of all possible fragments of code being executed under normal conditions is known with certainty. This assumption is reasonable as part of our initial phase of this project, where the main goal is to determine whether our approach is viable. Clearly, for an actual practical implementation of such a system, the assumption is not reasonable, since it dismisses the aspect of stack corruption, invalid pointer operations or other situations leading to execution of "random" code. In §6.6, we briefly discuss possible measures to address this aspect.

An additional assumption in this initial phase of the project is that the target device does not use hardware interrupts. A more detailed discussion and rationale will be presented in §6.5.3, but the assumption is also related to the aspect mentioned in the previous paragraph: the asynchronous and short-lived nature of interrupts do not fit well within the scheme of pattern classification. We claim that hardware interrupts are easy to detect through other means, and thus they will be relevant for future phases of this project, rather than this initial phase where we are evaluating the feasibility of our proposed approach.

As briefly mentioned earlier, one of the important highlights of our contribution is the fact that the system works on a standard personal computer (PC), capturing the power traces through the recording input of the sound card — side-channel analysis techniques usually rely on digital oscilloscopes or other expensive and bulky pieces of equipment. A standard, reasonably high-quality sound card (24-bits, 192kHz sampling rate, nowadays available at prices below $200) suffices to make the system work on a wide range of MCUs. To be able to measure the power consumption of an MCU, a current sensing shunt resistor is placed in series with the Power-In line going to the MCU, so that a voltage proportional to the power consumption is produced. This shunt resistor is selected to produce a voltage in the order of a few millivolts, thus not disrupting the functionality

of the MCU. This voltage is then captured through the Line input of a sound card, as shown in Figure 6.1. Given the typical computing power of today's mainstream PCs,

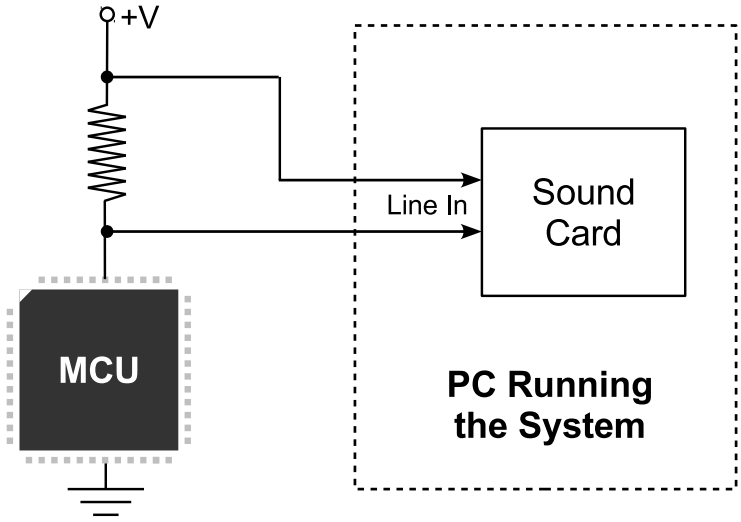


Figure 6.1: Simplified Diagram of our System.

with this setup, we claim that on-the-fly processing is within reach for a wide range of target devices. Of course, the technique is suitable for use with a digital oscilloscope; indeed, for processors with high clock frequencies, higher sampling rates will be required for the system to work, most likely without on-the-fly processing, depending on the clock frequency and architectural aspects of the target device such as pipeline depth, memory management unit (MMU), and cache memory [41].

The technique is centered around the idea of identifying fragments of code, corresponding to segments of the power trace. These fragments need to be sufficiently long so that: (1) there is a large enough amount of actions happening at the circuitry level to create a distinctive profile of power consumption. As an example, assigning a variable is unlikely to be distinguishable from any other operation involving memory, or even from a portion of some other action, such as fetching an instruction, or the additional memory access for instructions with immediate operands; and (2) so that the signal (in our setup, produced by an MCU running at 1MHz) can be sampled by a sound card at lower sampling frequency (in our setup, the sampling rate was 96kHz) and be able to extract meaningful information from it.

In this work, which is the initial phase of a longer project (we discuss future work in §6.6), we decided to use whole functions as the fragments of code to be considered (with only one exception, as will be discussed in §6.4 when describing our experimental setup). We used an Atmel MCU, AVR Atmega2560 [16] as the target device (8-bit MCU running at 1MHz) and as the source code we included a subset of the MiBench suite [36]

as a set of tasks representative of typical embedded software (at least typical for certain applications areas). The use of source code from MiBench offers two advantages: on one hand, it provides a set of *realistic* tasks, so that we evaluate the effectiveness of our technique with tests that are compatible with real functionality of embedded systems; and on the other hand, we chose to use source code not written by ourselves, to avoid the possibility (unintended or otherwise) that the technique may work because the source code was somewhat "custom-written" to make it work.

The pattern recognition system then extracts features from these segments of the power trace, and uses one of the classification techniques described in §2.7. We tried all three techniques, observing a much better classification performance when using the k-NN technique. We believe that the main factor is the fact that a function can do alternative things depending on the input data — and in general, a given fragment of code could do different things depending on the data it is working with. This leads to different execution times for different instances of the same function, and in general, it may lead to feature vectors that tend to be spread in the feature space, making the technique based on centroids less effective. For the k-NN rule, we tried values between 3 and 100 for k, obtaining best results for $k = 5$ for individual classification and $k = 21$ for continuous classification (we discuss this distinction in §6.4 as part of the description of our experimental setup).

One of the difficulties in our scenario is that the processing for the classification needs to be done in a continuous way, and it is the system's responsibility to achieve synchronization with the fragments of code to detect. That is, the system is not given a power trace with the guarantee that this is the power trace for one of the fragments of code. Instead, the system is given a single power trace that extends indefinitely (in any case, as long as the system is running), and it has to apply the pattern recognition technique for variable starting position and length of the sequences to classify. As an example to illustrate the difficulties arising from this constraint, we can not use the length of the power trace as one of the features to extract — if we could, then this would provide a very relevant piece of information that even alone would give a very high probability of correct classification (since we could always select fragments of code that execute with distinct durations).

The starting point of the fragment is mainly a problem when the system starts up and has to synchronize to the execution; after that, once the system has recognized/classified a given section of the power trace, it has information of where that fragment ends, so the starting point for the next item to be classified is known, even though adjustments may be necessary to compensate for "noisy" outcomes from the previous segments (e.g., a segment that was in reality L samples long may have been detected as being L' samples long). The system has to try various lengths and see which one gives the closest match with training samples from the database.

When deciding what parameters to use as features to be extracted from the entities

(in our case, the power traces), there is often a bit of heuristics and intuition involved, especially when there is no analytic or otherwise simple description of a PDF with "nice" characteristics. In our case, we decided to use spectral information — logarithmic magnitude and phase — as the feature vector. The intuition on why spectral information may give useful *and robust* information to identify the power trace as corresponding to one of the given fragments of code is based mainly on the following two aspects:

- Getting around issues of alignment — spectral contents are similar even when the signal or portions of the signal are shifted. Thus, for different instances of the same function, prominent portions of the code may still be common to all other traces, but located at different points in the trace (as the result of conditional execution affected by the input data). This same idea has been exploited in a rather different context, where some side-channel attacks use correlation in the spectral domain, precisely to get around issues of alignment between traces (for example, see [53]).
- Variations due to "disrupting" factors in the system (such as noise or artefacts that occur due to the mechanism of leakage to the side-channel or the measurement) tend to produce higher deviations in the signal than in its spectrum, making the latter a more robust tool to identify a given power trace. In any case, the deviations in the spectrum tend to have simpler patterns, making it easier to extract the identifying features from spectral information than directly from the signal.

One additional difficulty, for which we resorted to a heuristic approach as our adopted solution, comes from the fact that we use the DFT of the trace directly as the feature vector; that is, each of the N elements of the DFT (more precisely, its complex logarithm, which directly provides logarithmic magnitude and phase) corresponds to one of the coordinates (or one of the dimensions) in the N-dimensional feature space. However, since different traces have different lengths, then we do not have a fixed value of N. That is, computing and comparing Euclidean distances in the feature space poses a challenge. This was an additional issue that contributed to our decision to use k-NN instead of the nearest centroid classification technique, for which we had to come up with some additional tricks that proved to be computationally expensive (in addition to exhibiting poor performance compared to the k-NN rule, as already mentioned).

Our heuristic includes two aspects: First, when given a trace and a starting point, we try all of the lengths present in the training database. That is, when looking for the nearest neighbors among the training samples, for each sample from the database, we take its length and consider the segment of the trace that matches that length, so that the distance can be evaluated. This is also consistent with the idea that we need to try different lengths, since we are only given the starting point, but the system needs to determine the length of the fragment as part of the task of identifying it.

The second aspect is that, given the detail mentioned above, it is clear that comparing distances for pairs of traces of one length with distances for pairs of traces of a different

length becomes an issue. To get around this, we used the notion of a *normalized* distance, where we normalize with respect to the number of dimensions. As an example, if we have two tridimensional vectors, say

$$\begin{aligned} \mathbf{u_1} &= (x_1,\, x_2,\, x_3) \\ \mathbf{u_2} &= (x_1+\delta,\, x_2+\delta,\, x_3+\delta) \end{aligned}$$

then we get [2]

$$|\mathbf{u_2} - \mathbf{u_1}|^2 = 3\delta^2$$

Our intuition is that for two, say, 5-dimensional vectors

$$\begin{aligned} \mathbf{v_1} &= (y_1,\, y_2,\, y_3,\, y_4,\, y_5) \\ \mathbf{v_2} &= (y_1+\delta,\, y_2+\delta,\, y_3+\delta,\, y_4+\delta,\, y_5+\delta) \end{aligned}$$

the distance, in the context of comparing which of the two pairs are closer, should be the same, since each of the coordinates, corresponding to one descriptive feature, are equally apart. However, a direct Euclidean distance computation for this case gives us $|\mathbf{v_2} - \mathbf{v_1}|^2 = 5\,\delta^2$

Thus, to avoid the nearest neighbors selection to be biased towards the shorter traces, we need to normalize by computing the *square distance per dimension*.

Also related to this aspect of traces with different lengths is the following issue: a longer trace may be at a disadvantage if sub-sections of it provide a sufficiently good match to other, shorter traces. We observed that this was the case for the set of MiBench functions that we used. Two different approaches were considered: (1) using an adjustment factor to favor longer traces when otherwise approximately equally close matches; and (2) using an adjustment factor to favor matches at the "nominal" position as determined by the classification at the previous iteration. We tried both approaches, and observed that (2) had the severe adverse effect of reducing the ability to maintain synchronization with the trace, especially resynchronizing after a misclassification.

Putting all the pieces together, we define our distance metric as follows: given a trace $\mathbf{x}$ of length N, with DFT $\boldsymbol{\mathcal{X}}$, the associated feature vector is given by

$$\mathbf{X} \;=\; \{\mathrm{Log}\,\mathcal{X}_0, \mathrm{Log}\,\mathcal{X}_1,\; \cdots\; , \mathrm{Log}\,\mathcal{X}_{N-1}\} \tag{6.1}$$

where $\mathrm{Log}(\cdot)$ denotes the complex logarithm function. With this, the distance between N-dimensional feature vectors $\mathbf{X}$ and $\mathbf{Y}$ is given by

$$\|\mathbf{X} - \mathbf{Y}\| \;=\; \frac{1}{N} \sum_{k=0}^{N-1} |\, X_k - Y_k \,|^2 \tag{6.2}$$

where X_k and Y_k are the entries in the feature vectors, corresponding to the complex log of the DFT entries.

[2] We use square distance since this is the common approach used when implementing NN rules.

6.3.1 Speeding Up Spectral Analysis Computations

For simplicity, we used libfftw [28] to compute every DFT that we required, as it is, to the best of our knowledge, a correct and very efficient FFT implementation; however, for an actual practical application, we could speed up some of the spectral analysis computations; in particular, those related to determining or refining the starting position of a segment of a trace, requiring computation of DFTs of segments of the power trace that are one sample apart; instead of using FFT independently for each computation, taking $O(N \log N)$ time, we can compute them incrementally.[3] Let $\mathbf{x} = \{x_0, x_1, \cdots, x_{N-1}\}$ and $\mathbf{x}' = \{x_1, x_2, \cdots, x_N\}$ be two signals, and let $\boldsymbol{\mathcal{X}} = \{\mathcal{X}_0, \mathcal{X}_1, \cdots, \mathcal{X}_{N-1}\}$ and $\boldsymbol{\mathcal{X}'} = \{\mathcal{X}'_0, \mathcal{X}'_1, \cdots, \mathcal{X}'_{N-1}\}$ be their DFTs, respectively. Then, each $\mathcal{X}'_k$ can be obtained by considering the previous signal shifted one position in the time-domain, removing the contribution of the first sample (which is not present in the second signal), and adding the contribution of the last sample (which is not present in the first signal):

$$\begin{aligned}
\mathcal{X}'_k &= \sum_{n=1}^{N} x_n \, \mathrm{e}^{-\mathrm{j}\frac{2\pi k(n-1)}{N}} \\
&= \mathrm{e}^{\mathrm{j}\frac{2\pi k}{N}} \sum_{n=1}^{N} x_n \, \mathrm{e}^{-\mathrm{j}\frac{2\pi kn}{N}} \\
&= \mathrm{e}^{\mathrm{j}\frac{2\pi k}{N}} \left(\mathcal{X}_k - x_0 \mathrm{e}^{-\mathrm{j}\frac{2\pi k \cdot 0}{N}} + x_N \mathrm{e}^{-\mathrm{j}\frac{2\pi kN}{N}} \right) \\
&= \mathrm{e}^{\mathrm{j}\frac{2\pi k}{N}} \left(\mathcal{X}_k - x_0 + x_N \right) \qquad (6.3)
\end{aligned}$$

It is clear from Equation (6.3) that we get a constant-time ($O(1)$) procedure to compute each of the elements of the new DFT, obtaining the entire DFT in $O(N)$. Thus, for any sequence of DFTs of contiguous intervals (one sample apart), we compute the DFT for the first interval using FFT, and the remaining ones incrementally. Notice that the values of $\mathrm{e}^{\mathrm{j}\frac{2\pi k}{N}}$ for $0 \leqslant k < N$ can be pre-computed, to further speed up the computation. This scheme has been proposed in the DSP literature, although with a different implementation focused on a hardware implementation using an Infinite Impulse Response (IIR) digital filter implementation [22].

Our system also requires computation of DFTs of intervals starting at the same sample but with lengths that differ by one sample. In this case as well, we could compute the DFTs of the augmented intervals incrementally, provided that we "quantize" the sizes of the DFTs and use zero-padding for the signals. This is not a problem, since the DFT corresponds to the same spectrum, computed at higher resolution [70]. Let $\mathbf{x} = \{x_0, x_1, \cdots, x_{N-1}\}$ and $\mathbf{x}' = \{x_0, x_1, \cdots, x_N\}$ be two signals, and let $\boldsymbol{\mathcal{X}} = \{\mathcal{X}_0, \mathcal{X}_1, \cdots, \mathcal{X}_{M-1}\}$ and $\boldsymbol{\mathcal{X}'} = \{\mathcal{X}'_0, \mathcal{X}'_1, \cdots, \mathcal{X}'_{M-1}\}$, with $M \geqslant N+1$, be the DFTs of

[3] Assuming no windowing is used.

the zero-padded versions of the signals, respectively. Then, each $\mathcal{X}'_k$ $(0 \leqslant k < M)$ can be obtained as follows:

$$\begin{aligned} \mathcal{X}'_k &= \sum_{n=0}^{N} x_n \, \mathrm{e}^{-\mathrm{j}\frac{2\pi kn}{M}} \\ &= x_{_N} \, \mathrm{e}^{-\mathrm{j}\frac{2\pi kN}{M}} + \sum_{n=0}^{N-1} x_n \, \mathrm{e}^{-\mathrm{j}\frac{2\pi kn}{M}} \\ &= x_{_N} \, \mathrm{e}^{-\mathrm{j}\frac{2\pi kN}{M}} + \mathcal{X}_k \end{aligned} \tag{6.4}$$

Showing the constant-time procedure for each term, for a linear-time procedure for the entire DFT. The value of the quantization steps should be carefully chosen to ensure that the computation of the sequence of sizes, done at the "rounded up" size in linear-time is better than each of the signals computed in $\mathrm{O}(n \log n)$ for the non-rounded-up sizes.

6.4 Experimental Setup

This phase of our work consists of two experiments. In the first experiment we evaluate the effectiveness of the pattern recognition system by classifying power traces of known fragments of code and determining the success rate or *precision* (the fraction of power traces that were classified correctly). That is, we test detection of the various fragments of code in isolation, and evaluate the performance of the classification system. To isolate the trace corresponding to the exact time interval of execution, we use markers which are actions known to have high power consumption and thus produce a prominent pulse in the trace. This is one of many possible approaches, and we chose it for our experiments due to its simplicity. Given our STK600 setup, we used the LEDs for this purpose. Figure 6.2 illustrates these steps.[4]

In the second experiment, we execute a sequence of function calls (each function being one of the fragments of code for the pattern recognition system) and have the system determine the sequence of functions that was executed, with the power trace as its only input. Figure 6.3 shows the details. The first experiment tests the building blocks, the basic operations of the system, while the second experiment aims at modeling the operation of an eventual practical implementation of our proposed technique.

Both experiments rely on the training phase of the classification system — thus, in this initial step we execute each of the functions S_c times, where S_c corresponds to the number of training samples per class (per fragment of code to be detected); we decided to use $S_\mathrm{c} = 1000$ as a reasonably large number to be used as a starting point (we present a more

[4] For some of the functions, the input is a graph or a tree. For these cases, we inserted random values in the data structure.

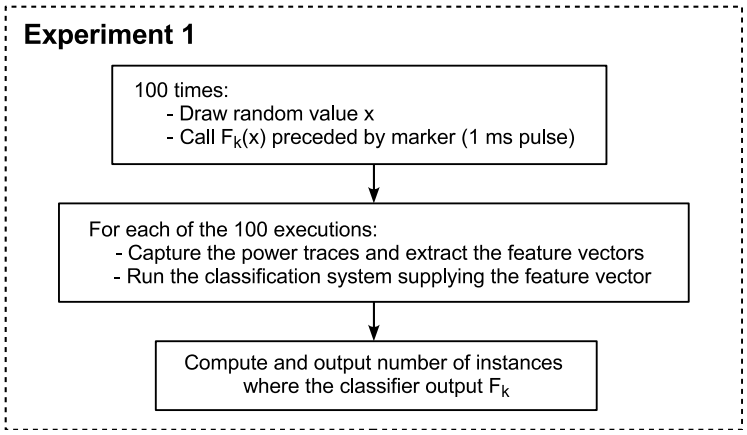


Figure 6.2: Experiment 1 – Classifier Performance.

detailed discussion in the next section). The fragments of code, denoted $\mathrm{F_k}$ ($1 \leqslant k \leqslant |\mathcal{C}|$), are either functions or fragments of a function, and for each call we supply a randomly selected value as input argument for the function, as illustrated in Figure 6.4. For Experiment 2, the training samples were not marked by surrounding them with pulses, but rather, surrounding them with some other (randomly selected) function, since this is how they would appear in the classification phase when operating in continuous mode, and the samples in the training phase have to be consistent with the traces captured during normal operation. We emphasize the detail that, for both experiments, these traces used for the training database are different from those being identified/classified, since "fresh" random values are chosen at every instance — the fragments of code are part of the same set of possible classes, but each instance of a trace being classified is different from every trace in the training database.

In a real-life application, this step should consider, if available, the probability distribution for the arguments to each function. For example, if a given function receives as input parameter the measured temperature, we will draw values from a normal distribution with mean 25 °C and relatively small variance.[5] For our experiments, we used uniformly distributed random variables in a reasonable range (the ranges were consistent between the training phase and the classification phase). This does not take into account the possibility of a function being called with unreasonable parameters due to a defect in the software; however, one can easily compensate for this aspect by including a small fraction of training samples using parameter values outside the reasonable range.

The experiments were run on an Ubuntu Linux system, with avr-gcc 4.3.5 and avrdude 5.10. The target device was an Atmel AVR Atmega2560 MCU on an STK600 board [16], and we assembled a quick prototype card to facilitate the connections — including a snap-in connector to place the shunt resistor so that different values can be easily tried (in

[5] Assuming a system intended to work at room temperature.

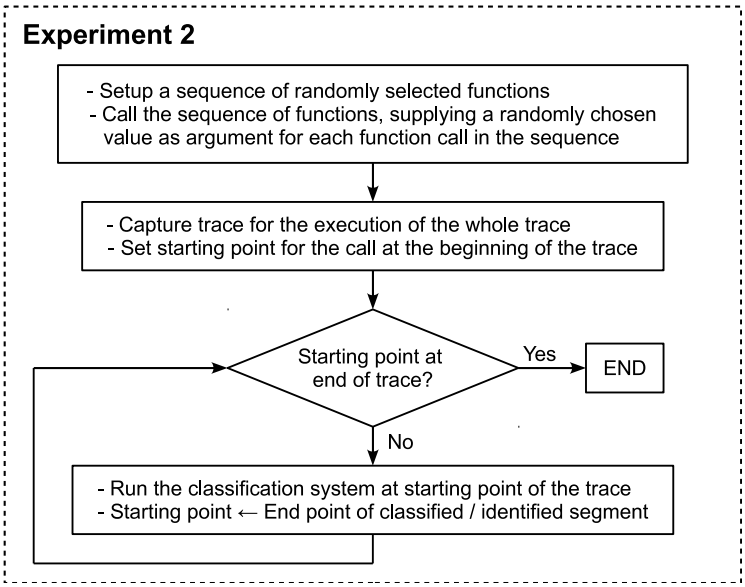


Figure 6.3: Experiment 2 – "Online" operation.

our case, a 10 Ω resistor produced voltage in the correct range). Figure 6.5 shows a photograph of this simple prototype card. The red/black cable on the right, ending in a two-pin header connector goes to the VTARGET connector on the STK600 board (so that current to the MCU passes through), and the green connector on the left is the RCA audio connector to easily connect to the input of the sound card. The card also includes pins to connect oscilloscope probes (for verification purposes, or for future experiments using a digital oscilloscope).

The sound card was an HT Omega Claro+ [43], and we used Audacity [54] to record the power traces. Figure 6.6 shows a screenshot from one of the power traces in the training phase, showing the two surrounding pulses.

We used code from MiBench [36] as the source code of the target device for the experiments. That is, the set of fragments $\mathcal{C}$ includes fragments of code from MiBench; in particular, from the telecommunications, network, and security sections of it (we excluded code that required file access or intensive operations as well as code for which we required many modifications for it to compile with avr-gcc). We also excluded redundant items — for example, from the security section, there are several symmetric encryption algorithms and several hash functions; we used only AES (which is the one generally recommended for practical use) for a sample of symmetric encryption, and SHA as a sample of a cryptographic hash function. For simplicity reasons, our work currently operates at the granularity level of entire functions (that is, the fragments of code to be matched are entire functions), with the exception of the SHA algorithm. This exception is due to the fact that SHA executes a large number of rounds repeating the

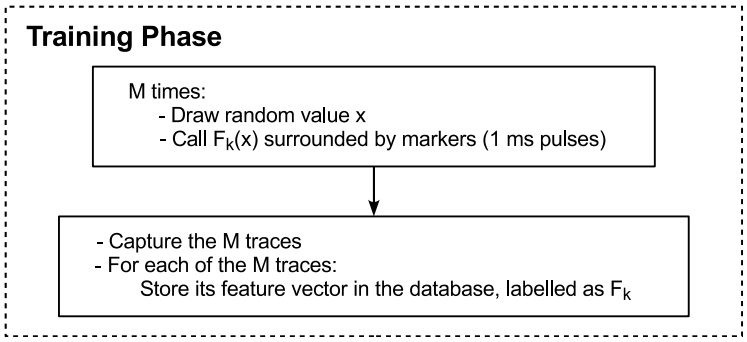


Figure 6.4: Training Phase.

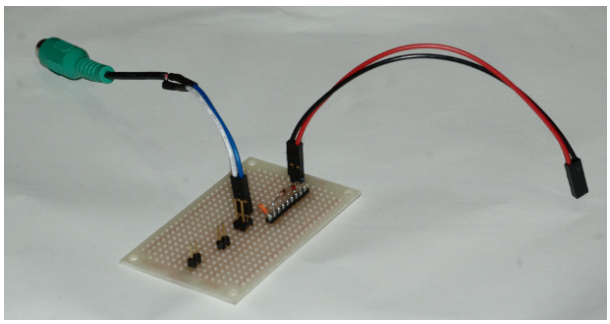


Figure 6.5: Prototype Card to Facilitate Connections.

same procedure, thus taking a very long time to execute, making it more reasonable to choose that procedure as the fragment to consider. The exact set of functions used for our experiments is the following: ADPCM encode, ADPCM decode, CRC-32, FFT, SHA (fragment), AES (Rijndael) symmetric encryption, Dijkstra's shortest path algorithm, Patricia Trie (insertion), and pseudo-random number generation (C's `random()` function).

6.5 Results

We now present and discuss the results for both phases of the experimental setup.

6.5.1 Experiment 1 – Individual Classification

For Experiment 1, we evaluate and report the precision of the classifier. Since the classifier chooses one of the possible classes (one of the functions being considered), there are no false negatives; that is, there is always an output from the classifier, and it is either a

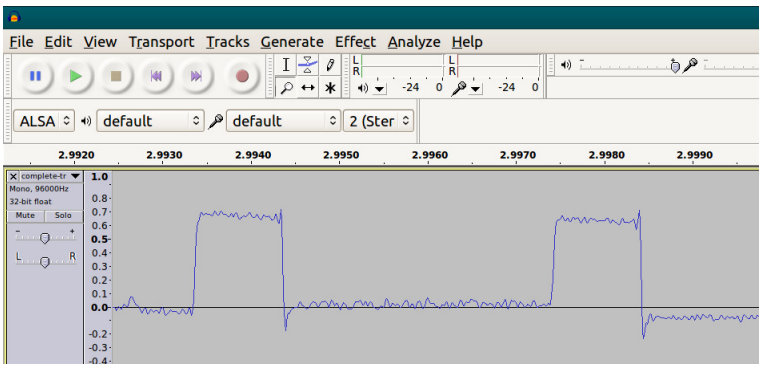


Figure 6.6: Screenshot – Power Trace in Audio Editor.

true positive or a false positive. Thus, the precision fully describes the performance of the classifier. The precision P is given by

$$P \triangleq \frac{T_{\mathrm{P}}}{T_{\mathrm{P}} + F_{\mathrm{P}}}$$

where T_{P} denotes the number of true positives (i.e., correct classifications) and F_{P} denotes the number of false positives (i.e., misclassifications, or incorrect classifications). For example, consider a scenario with ten candidate functions, F_1, F_2, $\cdots$, F_{10}, Experiment 1 is run and it executes 100 times function F_1. The classifier outputs 90 times F_1, 4 times F_2, 3 times F_5 and 3 times F_8. Then, the number of true positives is 90, and the number of false positives is $4+3+3=10$. The resulting precision for this example, P_{ex}, would be

$$P_{\mathrm{ex}} = \frac{90}{90+10} = 0.9 \quad (90\%)$$

We first adjusted the system's parameters to obtain the best performance. For each of the functions, we initially captured 1000 training samples, but then varied the number of samples effectively used, to determine the optimal value (optimal within the range 1 to 1000, which is the maximum number of available samples). We first maintained the same number of samples for every function and varied the value to obtain the optimal. We then used this as the initial estimate in a simple optimization procedure to determine the optimal number of training samples *for each function*. We omit any additional details or figures, since there is nothing particularly relevant that they would show. The final sizes after running this optimization process are shown in Table 6.1. For some of the functions, more than 1000 samples were required for good performance, so we captured additional traces for those. In particular, functions like Dijkstra's shortest path required a larger set, possibly due to the variable nature of the algorithm — depending on the graph contents (weights, connections, etc.) there may be wide variations in execution time, requiring larger numbers of training samples to compensate for the spread nature

Function	Training samples
adpcm_encode	400
adpcm_decode	50
CRC32	12500
FFT	900
SHA (Fragment)	600
AES (Rijndael)	800
Dijkstra's shortest path	9500
Patricia Trie (insertion)	900
`random()`	600

Table 6.1: Number of Training Samples for Each Function.

of its PDF. For CRC32, we were obtaining a low precision when using 1000 training samples, so we increased the number of samples for this one as well.

With these parameters in place, we started measuring the performance for Experiment 1. Table 6.2 shows the results for each of the functions being tested. That is, it shows the precision obtained for the classifier when executing each different function.

Function	Precision
adpcm_encode	100%
adpcm_decode	97%
CRC32	92%
FFT	99%
SHA (Fragment)	100%
AES (Rijndael)	97%
Dijkstra's shortest path	98%
Patricia Trie (insertion)	100%
`random()`	99%
Overall (avg.)	**98.0%**

Table 6.2: Classifier Precision

The results clearly indicate an excellent performance for the classifier, with only one of the functions scoring a 92% precision and no other function scoring below 97% precision. The overall precision is given by the arithmetic mean — every function, being executed the same number of times, has the same overall weight, thus the arithmetic mean is the appropriate averaging mechanism. A figure of 98% for the overall precision is also a solid indication of the excellent performance that our classifier achieves.

6.5.2 Experiment 2 – Continuous Classification

For Experiment 2, we had to overcome several obstacles. For example, due to the limit of program size of the MCU, we were unable to simultaneously include all sections of MiBench and make them execute correctly. In particular, Patricia trie insertion and Dijkstra's algorithm fail to run on the target due to insufficient resources. Excluding these two functions, we can run the experiment.

Description of the Experiment

To evaluate the performance of the classifier in continuous operation, we execute a long sequence of randomly chosen functions, with the only constraint being that we always call ADPCM encoding first, to then decode the data. We disregard the distinction between random and pseudo-random, and will refer to random values through the rest of the discussion. In that sense, we used the cryptographic-quality pseudo-random generator `/dev/urandom`, which is, for most practical purposes, "as close as it gets" to true random values [81]. Notice, however, that this random selection is done offline and the sequence is ultimately "hardcoded" in the source code to be compiled and run on the target. This restriction does not affect the random nature of the experiment, yet it is necessary: on-the-fly generation of random values by the target device itself between function calls would introduce artefacts that could skew or possibly even invalidate the results. The source code for this offline program is included in Appendix C.

We could not include arbitrarily long sequences of functions, since the entire sequence of function calls had to be hardcoded, and the target device imposes a limit on the size of the executable — we observe that the randomly generated data *for each call to a function* had to be stored in a buffer, for the same reason explained above. The longest sequence that we could fit in the target was 500 calls long (close to 100 calls per function on average). Though this number could be considered large enough to claim that the experiment is valid, we repeated the process ten times and collected statistics over a total of 5000 function calls.

The offline program that generates this random sequence of calls, as well as the random data, produces two files to be included in the program to be run on the target. One file, `buffer_sizes.h`, defines (through `#define` directives) the sizes of the buffers that contain the parameter values. Figure 6.7 shows an example of the contents of this file:

The other file that this offline program generates is the file containing the actual function calls. Each function call receives input data obtained from one of the elements of the buffer, and while generating this sequence, the offline program goes over each element in sequence, hardcoding the value in this output program, as shown in Figure 6.8, taken from one of the generated files (some of the lines were manually wrapped to fit the text within the page width).

```
#define ADPCM_COUNT 50
#define FFT_COUNT 97
#define AES_COUNT 111
#define CRC_COUNT 101
```

Figure 6.7: Buffer Sizes for Randomly Generated Sequence.

```
encrypt (plaintext + 0*AESSIZE, ciphertext, &ctx);
adpcm_coder(pcmdata + 0*PCMSIZE, adpcmdata, PCMSIZE, &coder_1_state);
rc = crc32buf (crcdata + 0*CRCSIZE, CRCSIZE);
fft_float (FFTSIZE, 0, real_in + 0*FFTSIZE, imag_in + 0*FFTSIZE,
           real_out, imag_out);
rc = crc32buf (crcdata + 1*CRCSIZE, CRCSIZE);
adpcm_decoder(adpcmdata, pcmdata_2, PCMSIZE, &decoder_state);
nothing = (random() ^ random()) & 0xFFFF;
adpcm_coder(pcmdata + 1*PCMSIZE, adpcmdata, PCMSIZE, &coder_1_state);
fft_float (FFTSIZE, 0, real_in + 1*FFTSIZE, imag_in + 1*FFTSIZE,
           real_out, imag_out);
nothing = (random() ^ random()) & 0xFFFF;
```

Figure 6.8: Fragment of Randomly Generated Sequence.

We can see, for example, for the first two calls to `crc32buf`, the input data coming from the first and second elements of buffer `crcdata` (offsets 0 and 1 hardcoded in the call). Same for `adpcm_coder` and `fft_float`. These hardcoded offsets continue to increase with each subsequent call to each function, until the `XXX_COUNT` value. Since we only show a short fragment of one of the files, the only offsets that we see are 0 and 1.

Figure 6.9 shows the declarations for these buffers (in the program that runs in the target device).

Performance Evaluation

We used a similar metric to that used for Experiment 1. Since we still have a classifier that always outputs one of the possible candidates, we only have true positives or false positives, which means that the precision still provides a complete picture of the classifier's performance.

However, an important distinction arises from the fact that in continuous classification, the sizes of the traces need to be determined and affect the performance of the process, as they affect the necessary resynchronization process in the cases of misclassifications. In that sense, a more sensible formula for the precision of the classifier in

```
#include "buffer_sizes.h"

short volatile pcmdata[ADPCM_COUNT*PCMSIZE];
char volatile  adpcmdata[PCMSIZE/2];  // Encoder output
short volatile pcmdata_2[PCMSIZE];    // Decoder output

volatile char plaintext[AES_COUNT*AESSIZE];
volatile char ciphertext[AESSIZE];

volatile float real_in[FFT_COUNT*FFTSIZE];
volatile float real_out[FFTSIZE];
volatile float imag_in[FFT_COUNT*FFTSIZE];
volatile float imag_out[FFTSIZE];

char crcdata[CRC_COUNT*CRCSIZE];
```

Figure 6.9: Buffer Declarations for Functions Calls Sequence.

continuous mode, P_c, is given by the fraction of the time during which the output of the classifier corresponds to a true positive:

$$P_c \triangleq \frac{\sum |I_{T_P}|}{\sum |I_{T_P}| + \sum |I_{F_P}|}$$

where I_{T_P} denotes intervals during which the output of the classifier is a true positive, I_{F_P} denotes intervals during which the output is a false positive or a misclassification, and $|\cdot|$ denotes the length of its argument (the length of the interval).

As an example, consider the scenario with three candidate functions, F_1, F_2, F_3, which take 10 ms, 20 ms, and 30 ms to execute, respectively. If the sequence F_3 – F_1 – F_2 is executed and the classifier outputs F_3 at time 0 ms, F_2 at time 20 ms, and F_2 at time 20 ms then, with all units implicitly ms, the intervals with true positive are $I_{T_1} = (0, 20)$ and $I_{T_2} = (40, 60)$, and the interval $I_F = (20, 40)$ is a false positive — the output is F_2, and during the sub-interval $(20, 30)$ the correct class is F_3 and during the sub-interval $(30, 40)$ the correct class is F_1. Thus, in this example, the precision P_{ex_2} would be approximately 67%:

$$P_{ex_2} = \frac{|I_{T_1}| + |I_{T_2}|}{|I_{T_1}| + |I_{T_2}| + |I_F|} = \frac{2}{3}$$

Results

As described in §6.5.2, ten sequences of 500 function calls each were executed, and traces were captured for each of them. Figure 6.10 shows a screenshot of the audio editor displaying the first few milliseconds of one of the traces; in particular, this trace fragment

corresponds to the sequence of ten function calls shown in Figure 6.8. The markers were manually added to the image for illustration purposes, indicating the boundaries between functions.

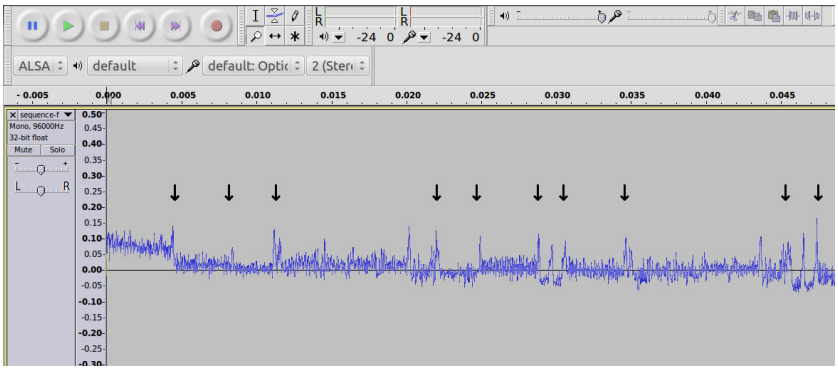


Figure 6.10: Power Trace for Sequence of Function Calls.

The complete traces were fed to the processing program implementing the classifier as described in the previous sections. The source code for this classifier program is included in Appendix D. An example of the output of this program is shown below (only a fragment, since each trace contains 500 function calls). The reported time uses the unit of audio samples, which is about 10.4 μs:

```
Executed aes at time 18
Executed adpcm-encode at time 428
Executed crc32 at time 810
Executed fft at time 1077
Executed crc32 at time 2127
Executed adpcm-encode at time 2394
Executed random at time 2786
Executed adpcm-encode at time 2942
Executed fft at time 3328
Executed random at time 4407
```

This example again corresponds to the same sequence of ten function calls shown in Figure 6.8 and corresponding to the trace fragment shown in Figure 6.10. We observe that in this particular example, there was one misclassification (the sixth function call is reported as `adpcm_encode` when the actual function being executed is `adpcm_decode`).

To measure the performance (the precision) of the classifier in Experiment 2, the output for each of the ten traces was fed, along with the C source code corresponding to each of the ten sequences, to a custom-made program that compares the output and reports statistics allowing us to determine the precision and provide some additional potentially insightful information. The source code of this program is included in Appendix E.

An important reason to use a custom-made program was to allow for user intervention in the process of matching sequences that might be hard to properly identify and match

algorithmically. Even more importantly, the matching includes timing that may need to be verified against the traces, for the less obvious cases (though only on two occasions we needed to resort to the traces in the audio editor to resolve a mismatch). The program does as much as possible in an automated way to minimize user intervention, and of course does as much validation as possible for the user input, to minimize the effect of human errors and oversights. Below is an example of the program finding a misclassification and prompting the user to resolve it. Again we used the example corresponding to the code shown in Figure 6.8 and trace shown in Figure 6.10, where we saw that an instance of ADPCM encoding was mistaken for decoding:

```
Difference (at src_line 5, classif_line 5):
#   Src Code        Classifier output
-2  FFT             Executed fft at time 1077
-1  CRC32           Executed crc32 at time 2127
0   adpcm-decode    Executed adpcm-encode at time 2394
1   random          Executed random at time 2786
2   adpcm-encode    Executed adpcm-encode at time 2942
3   FFT             Executed fft at time 3328
4   random          Executed random at time 4407
5   adpcm-decode    Executed adpcm-encode at time 4553
6   adpcm-encode    Executed adpcm-encode at time 4895
7   random          Executed random at time 5290
8   adpcm-decode    Executed crc32 at time 5446
9   AES             Executed random at time 5703
10  CRC32           Executed aes at time 5803
11  FFT             Executed crc32 at time 6217
12  CRC32           Executed fft at time 6487
13  adpcm-encode    Executed crc32 at time 7531
14  FFT             Executed adpcm-encode at time 7804
15  CRC32           Executed fft at time 8185

Enter number of skip lines (skip src <space> skip classif -- END if at end): _
```

The program displays both sequences, from two items before the mismatch, and going for 15 items after. The user is prompted to enter the number of items to be skipped from the actual source code and the number of items to be skipped from the trace (these are not necessarily equal in all cases: a fragment of a trace may be misclassified for a function of different duration) that would bring the two sequences back in sync. In the above example, we just skip one item on each, since the items immediately following the mismatch are already in sync. The processing software normally handles cases like this one; but to ensure proper functionality, the software automatically makes that decision only when the five following items match. In the case above, the fifth item does not match, so the program requests user intervention.

Estimating the Precision

Measuring the exact value of the precision for the continuous classifier requires us to obtain more information than reasonably feasible given our setup. Computing the exact value of the precision requires that we determine the intervals where the output is correct, and this would require us to have exact timing information for the sequence being executed. However, the actual trace contains calls with random parameter values, so the duration is unknown, and instrumenting the program for the purpose of obtaining that information would affect the measurements, at least for the family of devices that we targeted in our work.

We can, however, obtain a good approximation of the precision if we use the timing that the classifier outputs. Specifically, the skipped elements from the classifier output (the right column in the above example of the processing software) are considered to be false positives and the rest is considered a true positive. As long as the sequences match, we assume that the timing for the matching items is correct and disregard any inaccuracies in the exact positions of the boundaries between functions.

Thus, the difference in the time indexes for the items that have to be skipped to reestablish synchronization reveal the length of the false positive intervals. The estimate is accurate provided that the misclassifications occur with a deviation that is balanced; that is, provided that some errors confuse a function with a longer (in duration) function and some confuse a function with a shorter function, without any imbalance on average. This is a reasonable assumption, and we did not observe any evidence suggesting that there would be any imbalance in the errors for the traces that we processed.

Table 6.3 summarizes the important parameters describing the performance of the classifier when operating in continuous mode. In addition to the precision, we also determined the average number of items that it took the system to recover from a misclassification. This metric provides information about the robustness of the system in terms of ability to recover from a misclassification and reestablish synchronization with the trace. It also provides evidence to the quality of the classifier in general, in that short sequences of missed items are certainly easier to compensate for through auxiliary methods, such as doing additional validation of timings, validating feasible program sequences through static analysis, etc. Though we did not use any of these techniques in this initial phase of the project, it still makes sense to claim that smaller values for this parameter correspond to higher quality for the continuous classifier.

For this second experiment we also obtained results indicating a good performance. This is encouraging, since this is clearly the more important of the two experiments, since it models the way an actual practical system would operate. The precision is not as good as that obtained for Experiment 1; this is expected, as this operating mode involves more parameters, more ambiguities and degrees of freedom, and the additional functionality of maintaining synchronization — with or without misclassifications. Also,

Sequence	Precision	Avg. Recovery
1	88.75%	1.29
2	89.78%	1.30
3	87.65%	1.27
4	88.63%	1.38
5	87.07%	1.34
6	89.03%	1.29
7	89.03%	1.32
8	86.97%	1.48
9	89.59%	1.19
10	87.84%	1.30
Overall (avg.)	**88.74%**	**1.32**

Table 6.3: Continuous Classifier Performance

the fact that traces are now in sequence one after another introduces the possibility that sub-fragments of a trace combined with sub-fragments of another trace could be a good match for some incorrect function. Experiment 1 does not face any of these difficulties. Thus, we believe that a figure of close to 90% precision for continuous classification is a very good result for this initial phase of the project, where the goal is to study the feasibility of our proposed approach.

Another important aspect revealed by these results is that of the robustness of the continuous classifier, in that the system never faced a situation where an error threw it irreversibly out of sync. In all cases, the system reestablished synchronization after a misclassification, and in most cases the misclassification involved just one function replaced with another, and then resynchronization immediately after, as suggested by an average of 1.32 functions skipped before resynchronization.

As suggested before, we can reasonably claim that individual incorrect classifications are essentially irrelevant, as the tool could be extended to make use of the control-flow graph (CFG) and consider possible execution sequences (this aspect is discussed more in detail in the next section on future work). Given this information, a match to an incorrect function may have been rejected with high probability, because the CFG would have given indication that it is impossible to reach that particular function within a short period of time after the preceding sequence.

Additional Insights

As additional observations and insights that we gained from the experimental results in this initial phase of the project, we could mention the following:

For some of the functions, long sequences of consecutive calls to the same function showed a higher likelihood of misclassification. For example, sequences of three consecutive calls to CRC32 showed up very often as a prompt for user assistance in the processing software, and in some cases required three or four skipped items from the trace to reestablish synchronization.

Also interestingly, we observed several instances where a sequence of three or more consecutive calls to random() either caused the function immediately following that sequence to be misclassified, or was misclassified as a smaller number of consecutive calls.

These two aspects suggest that it may be a good idea to restrict the fragments of code to sections of the source code with no control structures (conditionals and loops) such that every instance of a given fragment of code exhibits the same execution time. This not only has the potential to increase the precision, but also could play a role in dramatically increasing the computational performance of the system, in that a smaller training database could work well, and possibly the more efficient nearest centroid technique (using LDFs) could be applicable, since it was precisely this aspect of variable execution time within classes what put that approach at a disadvantage with respect to the k-NN technique.

Another aspect that caught our attention while working with and analyzing the experimental results was the potential effect of the DC level (or we should probably say, although more informally, the "short-term" DC level). We observed that the sound card blocks DC of the signal (although this feature is not clearly documented, it makes sense to expect such aspect, since DC plays no role on audio or audio quality, and if anything, it could have the negative role of potentially producing physical damage to the speakers or other components). This introduced a certain degree of difficulty in handling DC, for which we decided to disregard that parameter; however, we notice for example the first fragment in Figure 6.10, corresponding to AES, having a noticeably higher DC level, which makes us consider the possibility that adding DC could increase the effectiveness of the system. Simple measurements revealed a first-order high-pass filter at approximately 5 Hz in the traces. We could consider compensating for this by applying a filter with a 6 dB/octave slope from 5 Hz downwards, perhaps going for two or three octaves (i.e., down to 1.25 Hz or 0.6125 Hz) and then levelling. This could, however, introduce other adverse effects or unwanted artefacts. An additional option that one could consider would be to obtain relative DC levels from the transitions, since abrupt transitions in the short-term DC level survive intact the effect of a first-order high-pass filter.

6.5.3 Interrupts and Interrupt Service Routines

The current experiments were oblivious to interrupts and interrupt service routines (ISR) for two main reasons: (1) the approach to detect them differs due to their asynchronous

and usually short-lived nature; and (2) from the evidence we collected and observed, we claim that their detection should be really easy.

We should expect an interrupt request (IRQ) to cause a "power-heavy" reaction by the processor, in that a lot of hardware components need to react and work on processing the IRQ correctly [38]. Consequently, we should see a prominent component in the power trace that would identify the exact moment at which the processor responds to an IRQ.

We collected some experimental evidence supporting this claim; Figure 6.11 shows a trace of a simple LED animation program that uses timer interrupts, with the IRQ firing approximately every 6.5 ms. We observe the prominent peaks that IRQs produce in the trace.

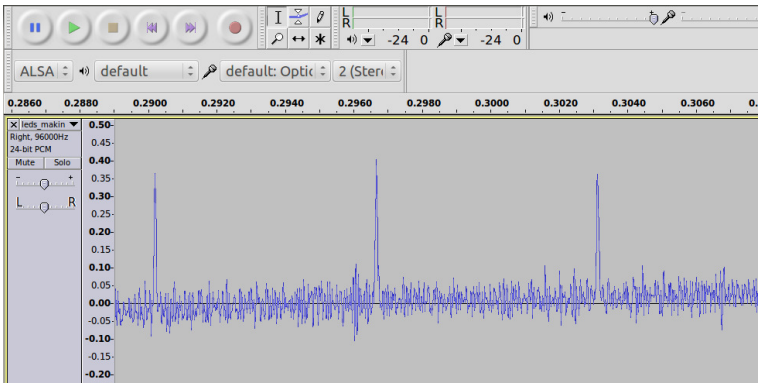


Figure 6.11: Power Trace Showing the Effect of IRQs.

For an actual practical implementation, however, we must consider interrupts for at least two reasons: (1) any fragment of code that the system is attempting to identify can be subject to another (small) fragment corresponding to the interrupt processing and the ISR to be inserted at any arbitrary and unpredictable position of the fragment to identify; and (2) many bugs arise from improper interactions between an ISR and the main/background processing, thus triggering the faulty behavior around the time that an interrupt occurs.

6.6 Discussion and Concluding Remarks

This initial phase of our project presents encouraging results; the effectiveness of the technique was confirmed by the experimental phase, at least in a preliminary way.

Further research is needed in several areas of the project; we focused on one target device, the Atmega2560, running at 1MHz. In principle this is not an important limitation, in that we could extrapolate from the field of side-channel attacks, where SPA

has been successfully applied on a wide variety of architectures and target devices. The important aspect for us to consider in this respect is the approach's ability to work with simple hardware, in particular with an off-the-shelf inexpensive sound card. With our Atmega2560 setup, we verified that a standard PC sound card sufficed for the system to run at the granularity level of function calls, producing traces in the order of 200 to 300 samples in length. However, to address either finer granularity, or higher clock speeds for the target device, we may need to investigate the relationship between these: for a given granularity, what is the maximum ratio between the device's clock speed and the sampling rate at which we capture the power trace? Equivalently, for a given clock speed and a given sampling rate, what is the finest granularity at which we can detect fragments of code?

Ideally, we would like our system to detect every possible code segment with fixed execution time (i.e., contiguous blocks of code without conditionals or loops). For example, instead of detecting execution of a loop (as a whole, with variable execution time), we would prefer to individually detect the evaluation of the condition and the body (assuming no nested loops or conditionals inside the body of the loop). This would be beneficial not only for the higher level of details in the output of our system, but also due to the potential increase of precision and processing speed, as discussed earlier.

We would like to emphasize the fact that these limitations in no way negate or compromise the validity or the value of the reported results. A large class of embedded systems run at low clock frequencies, and for those, the presented approach will be perfectly fine and valuable when assisting in the debugging task during advanced phases of the development cycle. Incidentally, this low-frequency aspect may be correlated with low transistor count MCUs, presumably with simple architectures that may lack any sophisticated debugging tools embedded in the hardware, making our technique particularly valid for this class of target device.

A positive aspect of the results derives from the fact that all of our tests and functions are CPU-bound. Practical systems typically use I/O, which makes a more prominent mark on the power trace and thus helps the classification process. The results of our experiments show a good level of performance even with this disadvantage.

At the present stage, our approach is applicable to background/foreground programming (superloop structure), multitasking with run-to-completion semantics, and possibly also to co-operative multitasking, depending on whether we can easily identify the `yield` calls. Also worth noting, since our experiments produced good results even when using an inexpensive off-the-shelf sound card, we conclude that this technology is perfectly suitable for hobbyists as well as professional developers.

Among the important aspects that we intend to tackle through future research are:

- Introduce the notion of *conditional* classification, possibly manually in an initial phase, but with the goal of using a CFG tool when it comes to a practical imple-

mentation. The idea is that by looking at the source code, we gain information about the possible fragments of code that could be executed at a particular time, given the previous fragment executed, or even better, the sequence of past fragments. Thus, the classifier can count on additional information, and thus its efficacy should improve.

In this sense, the fact that our Experiment 1 used only nine fragments should not be seen at all as a number too low to produce valid results — in a practical setup that makes use of the CFG, for most classifications the system may need to consider no more than two or three possible candidate fragments following the most recent classified fragments.

- Reliable detection of a crash condition where the processor ends up executing random code. Detecting such condition, as well as the precise time at which it started, is clearly a valuable piece of information when assisting the developers in the debugging task. This may be related to the option of *reject* in the classifier [82], and would allow us to eliminate the assumption that the execution is restricted to a set of possible fragments of code — an assumption that is reasonable in the sense that the developers always can count on the source code, but less reasonable from the point of view of considering cases such as stack corruption, invalid pointer operations or other situations leading to "random" execution.

- Considering different architectures; for example, processors with cache memory, deep pipeline or other forms of parallelism, etc. These in principle make our task harder, given that more information is combined together before leaking to the power trace. However, for some of these aspects, the additional complexity in the architecture may go hand-in-hand with additional information being leaked to the power trace, and those could end up making the task easier.

- Considering systems based on discrete components; for example, a system where the CPU is in one chip, memory, peripheral devices, and possibly things like the interrupt controller, all in independent chips. Monitoring power consumption for each of the chips *individually* should provide much more valuable and more accurate information, thus increasing the efficacy of the classifier.

Chapter 7

Discussion, Future Work and Conclusions

This chapter concludes the thesis by presenting a summary and discussion on the contributions presented as well as future work and concluding remarks.

7.1 Discussion

In this thesis, we have presented several related contributions with the common theme of side-channel analysis countermeasures and an application to embedded systems debugging. In particular, all of our proposed countermeasures aim at reducing the computational overhead required to protect cryptographic implementations against this class of attack, by presenting countermeasures that exhibit lower performance penalty and lower computational overhead than previously proposed countermeasures.

We claim that these contributions are potentially very relevant, as they relate to the security of embedded devices, which are in general a suitable target for side-channel attacks. Indeed, given the increase in usage of mobile, hand-held devices that make use of cryptography as one of the main aspects in the security of the involved systems, it becomes more and more important to protect cryptographic systems from attackers with physical access to the device executing cryptographic operations that rely on secret data embedded in the device. At the same time, with increased interest in more sophisticated functionality while operating with the important constraint of power supplied by battery, it is equally important to maintain a good level of computational efficiency in all subsystems of these hand-held devices.

Additional aspects may contribute to enhance this aspect of reducing performance penalty and computational overhead. For example, our adaptive idle-wait countermeasure, in addition to reducing the performance penalty with respect to the standard blind-

ing countermeasure, has the additional advantage that its performance penalty comes in terms of idle wait, and thus the processor is available for other tasks to proceed, in cases of multitasking systems — not an uncommon occurrence given the computing power of hand-held mobile devices at the present time and for the past several years.

Of course, it is important to ensure that the security of the device is not sacrificed by excessively focusing on the computational efficiency of the countermeasures. In that sense, our adaptive idle-wait was also shown to be effective, in that it was verified that the countermeasure could defeat any possible timing attack — known or yet to be discovered — with a performance penalty comparable to that of blinding. Furthermore, it was also shown that the countermeasure defeats a known powerful timing attack with a performance penalty considerably below that of blinding. Again we emphasize that, even at comparable performance penalties, idle-wait solutions have the advantage of allowing concurrent tasks to proceed in the case of multitasking systems, as well as the advantage from the point of view of reduced power consumption as a consequence of reduced computations. Our proposed adaptive idle-wait countermeasure compounds that advantage by further reducing the penalty given by the amount of idle-wait introduced.

For our idle-wait countermeasure, we also considered the issue that for software implementations, standard software platforms do not feature idle-wait or timer facilities with the required accuracy. However, there is nothing that fundamentally prevents our method to be implemented in software — it is simply a matter of the required tools not being currently available in standard software platforms. Timers and idle-wait facilities with high enough accuracy could become mainstream in software platforms in the near future.

The theme of reduced computational overhead is also present in our countermeasures against SPA. The optimality in these methods is defined relative to the underlying exponentiation algorithm being used: SPA-resistance is added while introducing zero computational overhead; this is achieved by avoiding any unnecessary operations through the idea of buffering the conditional multiplications to gain control over the time at which those operations are executed. Also, operations are executed always in their optimal form: squarings are always executed as an optimized squaring procedure, as opposed to implementing them as multiplications where the two operands have the same value, as in some existing countermeasures where the potential speedup from the more efficient squaring procedure is unutilized.

In the case of our Square-and-Buffered-Multiplications (SABM) countermeasure and its variants, though optimal in terms of performance penalty and computational overhead, the extra security does come at a price in terms of a small amount of storage — $O(\sqrt{\ell}\,)$, where ℓ is the bit length of the exponent. Storage is a commodity that most systems today, including embedded systems such as hand-held mobile devices, can afford. We claim that the reduction on computational overhead plays a more important role than any overhead in terms of storage required, especially for devices relying on battery power,

where any reduction in computations translates into prolonged battery life. We also noted that even for large amounts of storage (which are necessary in some of the variants or extensions of the method), power consumption is not affected, since the amount of computations and write operations are the same, spread across a larger number of storage locations.

The SABM method, in its basic form, is an extension of the right-to-left binary exponentiation algorithm, with exponent in either binary or NAF representation. We also demonstrated the aspect of combining our SABM technique with alternative exponentiation algorithms (instead of the basic right-to-left exponentiation), which was shown to lead to further improvements in execution time. This was done at several levels — as part of the first study, we showed this to be the case with the exponentiation technique proposed by Sun et al. [79], a very ingenious method where the exponent is split in two halves and pairs of bits (one from each exponent half) are combined for simultaneous processing. However, their method does involve operations where the result is discarded, which means that the method is necessarily sub-optimal. By combining their method with our SABM technique, we eliminate the need for any unnecessary operations while maintaining resistance to SPA, leading to a method that is more efficient than either their method in its original form or our SABM method in its basic form. In the second study related to our SABM technique, we further extend this aspect by adapting Sun's algorithm to make use of signed-digit exponent representation, an aspect that is fundamentally incompatible with their algorithm in its original form, limiting its efficiency. We showed that by doing this, several important opportunities arise that allow us to further increase the efficiency of the exponentiation while maintaining resistance to SPA. We emphasize the detail that adapting Sun's algorithm to the use of signed-digits exponent representation to increase its efficiency is only possible when combining it with our buffering technique.

This aspect of SPA resistance with low computational overhead also applies to all of the extensions of the SABM method resulting from the use of signed-digits exponent for the representation of half-exponents; namely, the use of NAF for each of the half-exponents, the use of JSF, and the multi-bit processing, including two-bit blocks derived from the NAF representation and three-bit blocks derived from the JSF representation. From these various methods that take advantage of signed-digits exponent representation, the method using JSF representation for the exponent halves is perhaps the better suited for typical cryptosystems based on ECC, since the exponents are relatively small (in the order of hundreds of bits), and the lower post-processing cost of this method means that the total amount of operations is lower than the amounts required by the other methods, even if some of these other methods are superior in terms of *asymptotic* performance. Furthermore, we argued that the method using JSF possibly represents the optimal trade-off between simplicity and reduced storage space, and computational performance: the improvement derived from using JSF for the exponent halves comes at no cost whatsoever, since the algorithm is identical with respect to the version that uses NAF. The multi-

bit methods, however, incur additional computational cost in terms of post-processing: multiplications, squarings, and inversions to combine the various accumulators into the required results. This aspect possibly makes these multi-digit methods unattractive for typical exponent lengths in ECC cryptosystems, and instead may be suitable for systems with exponent lengths in the thousands of bits such as RSA, since their asymptotic performance is superior by a comfortable margin.

In our most recent contribution, we extended and connected the ideas behind SPA to propose an application making constructive use of those techniques. We proposed a novel approach for non-intrusive debugging of embedded systems, especially useful for debugging faulty behavior observed at advanced phases of the development cycle, such as during production or even after deployment. The idea is based on exploiting the relationship between what a processor is executing and its power consumption to determine the sequence of code executed from observations of power consumption as a function of time (power traces). At the present stage, our approach is applicable to background/foreground programming (superloop structure), multitasking with run-to-completion semantics, and possibly also to co-operative multitasking, depending on whether we can easily identify the `yield` calls. We also discussed the possibility of using such a system for monitoring as an intrusion detection system (IDS) for embedded devices. In the wake of threats like Stuxnet [50], the area of embedded systems security gains increased attention, and it seems reasonable to consider adapting systems such as IDSs, classically viewed as applicable only to servers and networks, to embedded systems as well.

Our approach and our implementation feature the interesting highlight that the system runs on a standard PC, and the power traces are captured through the recording input of the sound card. Techniques where power traces are required, such as Power Analysis cryptographic attacks, usually rely on digital oscilloscopes or other expensive or bulky pieces of equipment. Given the standard quality of today's sound cards (typically 24-bit analog-to-digital conversion and 96 or 192kHz sampling rates), we found our approach to work perfectly well with our setup, which means that the approach should be suitable for a wide variety of target devices. Also worth noting, since our experiments produced good results even when using an inexpensive off-the-shelf sound card, we conclude that this technology is perfectly suitable for hobbyists as well as professional developers.

Experimental results confirmed the validity of our approach, showing very good performance when using part of the code base from the MiBench test suite. Many interesting insights were gained from the execution of the experiments that we believe can play an important role in the success of subsequent phases of this project. In that sense, the work presented in this thesis is only the initial phase of what we believe will be a much longer project with several opportunities for additional successful studies.

7.2 Summary of Contributions

We have presented several related contributions with the common theme of side-channel analysis and two distinct perspectives on this subject: we presented several countermeasures and a constructive application where the techniques used in SPA are used as a starting point to propose a technique applicable to embedded systems debugging.

We categorized these as main and secondary contributions, the former refers to the main elements or achievements presented in each work representing their claimed value. The latter are secondary elements; aspects that, though original ideas or results, are either not the centre or main aspect resulting from the work or not particularly extraordinary or remarkable achievements. Still, we presented and claimed these contributions with the attribute of "secondary" since they are essential elements supporting the main aspects of the works being presented.

7.2.1 Main contributions

The four main contributions corresponding to the four studies presented in this thesis are the following:

- An efficient countermeasure against timing attacks through idle-wait, hiding any useful patterns or statistical parameters of the timing that attacks could exploit. The proposed technique reduces the overhead introduced by the countermeasure with respect to existing solutions, and also, being idle-wait, reduces the computational cost, even at comparable overheads; this is an important aspect for devices relying on battery power, where lower computational cost translates into lower power consumption.

- An efficient exponentiation technique that is resistant to some forms of power analysis, at the cost of a small amount of storage. The main idea in this technique can be combined with several underlying exponentiation algorithms, adding resistance to power analysis while introducing zero computational overhead.

- In a follow-up contribution, we further improved our power analysis countermeasure by combining it with a modified form of an existing exponentiation algorithm, which in turn led to several new SPA-resistant algorithms with increased computational efficiency.

- A novel approach for non-intrusive program tracing of embedded devices through side-channel analysis; in particular, extending the ideas used in SPA through the use of pattern recognition techniques, combined with digital signal processing techniques to process the power traces. The intended application for this technique is

that of assisting embedded systems developers in the task of debugging at advanced stages or even after deployment.

7.2.2 Secondary Contributions

In addition to the already presented main contributions, the studies included the following secondary contributions:

- Identifying a subtle vulnerability in all forms of idle-wait countermeasures that could allow attackers to entirely bypass the idle-wait by requesting concurrent operations and measuring the throughput of decryption operations; we described the details of an attack that exploits this vulnerability, and described a correct way to implement idle-wait countermeasures to avoid this vulnerability.

- Presenting an analytical derivation for the mutual information between the data producing the leakage and the decryption time in the presence of our adaptive idle-wait countermeasure, relative to the mutual information in the absence of countermeasures.

- Presenting a two-thread parallel implementation of our SABM algorithm, which we claim is a nice and suitable implementation for modern processors, which are with virtually no exception, multi-core with at least two cores.

- Presenting a *correct* analytical derivation of the buffer space required for a given probability of buffer failure (including buffer overflow and buffer underflow) in contexts similar to the buffer usage in our SABM method. To the best of our knowledge, this derivation has not been done in the literature, and an incorrect derivation is presented in [83].

- Alternative analysis of some of the properties of NAF, based on an alternative algorithm to convert from binary to NAF — to the best of our knowledge, this algorithm is also our own original contribution.

- Operation of the power analysis tracing and debugging system based on a standard PC, capturing the power traces through the recording input of the sound card, avoiding the need of expensive and bulky pieces of equipment. To the best of our knowledge, this approach has not been presented in the existing literature. Though we applied it to the embedded systems debugging context, the value of this contribution covers the field of side-channel analysis, as this approach is suitable for SPA and possibly DPA on a wide range of target devices.

- We propose the idea of using our non-intrusive program tracing technique in the context of embedded systems security, in that it could be used as a monitoring

system that would detect anomalies in the execution, thus acting as an intrusion detection system (IDS) for embedded devices. This approach would have an important advantage over existing IDSs in that it is external to the system being monitored, making it a tamper-proof device from the point of view of remote attacks that operate by injecting unwanted code after the system is operating.

We believe that these additional contributions are important in the more general context of the fields where this thesis focuses. Each of the contributions are secondary in the context of the work where they are presented, but they certainly represent somewhat important contributions to the general areas of cryptology and embedded systems security.

7.3 Future Work

Several opportunities for future work arise from the results of the studies presented in this thesis.

For example, it was noted that an interesting highlight of our SABM method with respect to the theme of reduced computational overhead relates to the possibility that this method could be implemented in a way that it exhibits some level of resistance to DPA. Such implementation would be centered around the idea of randomizing both the order and the timing of executions of multiplications; depending on the source of randomness or pseudorandomness, this could lead to the possibility of adding DPA resistance with zero, or in any case very low, computational overhead, as no additional operations on the group elements would be necessary to randomize the data. It was noted, however, that further research is necessary to determine whether the method can indeed be implemented in a way that DPA attacks are completely defeated or at least slowed down to a point where they are rendered impractical. This aspect is related to the issue that the randomization is applied to the operations using the data on the buffer, and the buffer has in principle a small size, limiting the scope of the operations for which the order of execution can be randomized. We believe that this is a very important aspect to further investigate, since a positive result from such investigation would be extremely relevant.

This compounds with the results from our follow-up contribution, where exponent halves are processed simultaneously, with some of the techniques processing blocks of several bits. This introduces some level of parallelism that could contribute to "blur" any leaked signals and further slow down attacks based on statistical processing, in particular DPA.

It could also be interesting to investigate how well these techniques, applied as SPA and potentially DPA countermeasures, fit with our timing countermeasure. Do they help

slow down or completely defeat timing attacks as part of the effect of the randomization in the execution? Do they reduce the variance in the timing of the operations, so that the idle-wait countermeasure would have an even lower overhead? These are aspects worth investigating as follow-up work to the studies presented in this thesis.

For our most recent contribution, several opportunities for future work were discussed. Most notably, the use of static analysis techniques to assist in the classification process. In particular, using the control-flow graph (CFG) to narrow down the set of possible fragments of code to be identified from the power traces. The CFG indeed constraints the classification process, in that only feasible sequences are considered.

We also noted that future studies should include the option of a "reject" output in the classification process, since the study in its current stage assumes that the set of possible fragments of executed code is known with certainty given that developers have access to the source code. For a practical application, however, this is not a reasonable assumption in that it disregards the possibility of random execution of code as the result of stack corruption, invalid pointer operations, etc. We believe that this is one of the important aspects to be tackled through future research.

We also believe that it is important to evaluate the technique in a wider range of target devices; on one hand, we claimed that it is reasonable to extrapolate from the success of power analysis attacks on a variety of devices. On the other hand, since we introduced the use of a standard PC sound card for the purpose of capturing the power traces, it seems important to evaluate the applicability of our approach for a variety of target devices.

We emphasize again that this project is only the initial phase of a longer project, where we studied the feasibility of our approach. We believe that these improvements and opportunities for future work that we discussed will lead to substantial improvements in the performance and the range of target devices for which our technique is suitable.

Appendices

Appendix A

Proof for Lemma 4.1

For convenience, we use the abbreviations O (Overflow), EO (Early Overflow), and prefixes L or NL (Legitimate or Non-Legitimate). The probabilities of legitimate and non-legitimate early overflow are given by

$$\begin{aligned} \Pr\{\text{LEO}\} &= \Pr\{\text{Not O} \mid \text{EO}\}\ \Pr\{\text{EO}\} \\ \Pr\{\text{NLEO}\} &= \Pr\{\text{O} \mid \text{EO}\}\ \Pr\{\text{EO}\} \end{aligned}$$

Clearly, $\Pr\{\text{Not O} \mid \text{EO}\} + \Pr\{\text{O} \mid \text{EO}\} = 1$, so it suffices to show that the conditional probability of overflow (at ℓ bits) given that early overflow occurs is greater than $\frac{1}{2}$.

Let m denote the bit at which early overflow occurs (thus, $m < \ell$), and let $\boldsymbol{\delta}_m$ denote the random variable corresponding to deviation from the mean for the remaining $\ell - m$ bits (that is, if $\mathbf{k}_m$ is the random variable representing the number of nonzero bits in the remaining $\ell - m$ bits, then $\boldsymbol{\delta}_m = \mathbf{k}_m - p(\ell - m)$).

We note that overflow occurs at ℓ bits if and only if $\boldsymbol{\delta}_m \geqslant 0$, so we have:

$$\begin{aligned} \Pr\{\text{O at } \ell \mid \text{O at } m\} &= \Pr\{\boldsymbol{\delta}_m \geqslant 0\} \\ &= \Pr\{\boldsymbol{\delta}_m = 0\} + \Pr\{\boldsymbol{\delta}_m > 0\} \end{aligned}$$

If $p = \frac{1}{2}$, then the probability mass function for $\boldsymbol{\delta}_m$ is symmetric; thus:

$$\Pr\{\boldsymbol{\delta}_m > 0\} = \Pr\{\boldsymbol{\delta}_m < 0\}$$

but

$$\Pr\{\boldsymbol{\delta}_m < 0\} + \Pr\{\boldsymbol{\delta}_m = 0\} + \Pr\{\boldsymbol{\delta}_m > 0\} = 1$$

and

$$\Pr\{\boldsymbol{\delta}_m = 0\} \quad \begin{cases} > 0 & \text{if } \ell - m \text{ is even} \\ = 0 & \text{if } \ell - m \text{ is odd} \end{cases}$$

Thus, on average (taken over m),

$$\Pr\{\boldsymbol{\delta}_m = 0\} > 0 \implies \Pr\{\boldsymbol{\delta}_m \geqslant 0\} > \frac{1}{2}$$

For $p = \frac{1}{3}$, the distribution is, even for small values of $\ell - m$, near-symmetric (we recall that for reasonably large values of $\ell - m$, the distribution is closely approximated by a Gaussian, which is symmetric). This means that the argument used for the case $p = \frac{1}{2}$ is valid for $p = \frac{1}{3}$ as well — at least in the context of deriving an approximation for the probability of buffer failure. $\square$

The statement does hold in a strict and more rigorous sense, but the details of the proof get unnecessarily long and involved, given the context in which we make use of the lemma. A sketch of the necessary steps to complete the proof covering the case where $\ell - m$ takes small values is as follows: For $p = \frac{1}{3}$, we have to consider three different cases; m being congruent to 0, 1, or 2 (mod 3). That is, we consider the cases where m has the form $3n$, $3n+1$, and $3n+2$. Since we are interested in the average case (average taken over m), we work with the terms $\Pr\{\mathbf{k} \geqslant n;\ 3n\}$, $\Pr\left\{\mathbf{k} \geqslant n + \frac{1}{3};\ 3n+1\right\}$, and $\Pr\left\{\mathbf{k} \geqslant n + \frac{2}{3};\ 3n+2\right\}$ (where $\Pr\{\mathbf{k} \geqslant x;\ y\}$ denotes the probability that x or more nonzero bits occur in y bits), to show that their average is greater than $\frac{1}{2}$.

We observe that for the cases $3n+1$ and $3n+2$, since $\mathbf{k}$ is integer, equality to the mean can not occur, and therefore $\Pr\left\{\mathbf{k} \geqslant n + \frac{1}{3};\ 3n+1\right\} = \Pr\{\mathbf{k} \geqslant n+1;\ 3n+1\}$ and $\Pr\left\{\mathbf{k} \geqslant n + \frac{2}{3};\ 3n+2\right\} = \Pr\{\mathbf{k} \geqslant n+1;\ 3n+2\}$

We can either proceed by induction on n, or obtain a recurrence formula for the average of the above expressions, considering the effect of adding three additional random bits; the probabilities of adding $0, 1, 2$, and 3 nonzero bits are $\frac{8}{27}$, $\frac{12}{27}$, $\frac{6}{27}$, and $\frac{1}{27}$, respectively. This allows us to express the probabilities for $m = 3n+3$, $3n+4$, and $3n+5$ in terms of the above probabilities for $3n$, $3n+1$, and $3n+2$, and obtain the required result.

Appendix B

Online Computation of the NAF of a Non-Negative Integer

We describe a simple online procedure to obtain the NAF representation of a non-negative integer with ℓ-bit binary representation, processing each input bit independently, updating the output accordingly. Thus, we treat the conversion from standard binary to NAF as a transformation applied to the output of a random source of independent and uniformly distributed bits.

After processing bit b_{n-1}, we have output $d_{n-1}d_{n-2}\cdots d_1d_0$ with the possibility of a carry (at the very end, after processing bit $b_{\ell-1}$, this carry would correspond to digit d_ℓ). We observe that this output after processing bit b_{n-1} is the NAF representation of the n-bit non-negative integer $b_{n-1}b_{n-2}\cdots b_1b_0$ that has been processed so far.

Since the values considered are always non-negative, the most-significant digit at the end of each iteration can not be $\overline{1}$ — if it was, then there would be no carry, since the output is a valid NAF representation, and NAF does not allow adjacent non-zero digits, and thus the represented value would be negative. Also, for the same reason, a carry can only occur if $d_{n-1} = 0$.

Thus, after processing bit b_{n-1}, the output digit d_{n-1} and carry c_{n-1} can only be

$$(d_{n-1}, c_{n-1}) = \begin{cases} (0,0) \\ (0,1) \\ (1,0) \end{cases}$$

We now consider the effect of processing bit b_n. If $b_n = 0$, then clearly $d_n = c_{n-1}$ (the carry from the previous iteration), and no carry can result from processing bit b_n. If $b_n = 1$, then, if $d_{n-1} = 0$ with no carry, we would have $d_n = 1$ with no carry produced. If there is a carry from the previous iteration, then we add b_n and the carry, obtaining a value 10_2 aligned at position n — that is, $d_n = 0$ with a carry produced at this iteration.

Finally, if $d_{n-1} = 1$, then we have to substitute the resulting 11_2, since NAF precludes it. This is fixed by substituting 11_2 by its NAF equivalent, $10\overline{1}$ aligned at the same position; that is, we would replace the value of d_{n-1} with $\overline{1}$, $d_n = 0$ and a carry is produced at this iteration.

The procedure is necessarily correct given that: (1) it does output a valid signed-digit representation of the value represented by $b_{\ell-1}b_{\ell-2}\cdots b_1b_0$ in standard binary — indeed, every operation that modifies the output replaces blocks of digits with a different block representing the same value and aligned at the same position; and (2) by construction, this output does not have adjacent non-zero digits. Since we know that NAF representation is unique [5], then the output of this procedure must be *the* NAF representation of the input value.

Appendix C

Source Code for Generation of Sequence of Random Calls

```
/*********************************************************************

   This program is executed "offline" to generate the source code
   for experiment 2 on the target device --- it "hardcodes" a
   sequence of 500 randomly selected function calls (among the
   subset from MiBench being used).

   The random selection is done using /dev/urandom as a source of
   random data.

*********************************************************************/
#include <iostream>
#include <fstream>
#include <vector>
#include <string>
using namespace std;

enum Function
{
    ADPCM = 0,
    FFT = 1,
    RANDOM = 2,
    AES = 3,
    CRC32 = 4,
    ADPCM_encode = 10,
    ADPCM_decode = 11
};

void write_function_call (Function function, int offset, ostream & out);

    // Draw a pseudorandom int between 0 and range-1
int random_int (int range, ifstream & dev_urandom);


int main()
{
    const int TRACE_SIZE = 500;     // Number of function calls

    int counter[5] = {0,0,0,0,0};
```

```
    ifstream dev_urandom ("/dev/urandom");
    if (! dev_urandom)
    {
        cerr << "Could not open /dev/urandom" << endl;
        return 1;
    }

        // File that contains the sequence of function calls:
    ofstream calls ("randomized_function_calls.h");
    if (!calls)
    {
        cerr << "Coult not open randomized_function_calls.h for output" << endl;
        return 1;
    }

        // Depending on how many calls to each function, output
        // the #defined sizes to this file:
    ofstream defines ("buffer_sizes.h");
    if (!defines)
    {
        cerr << "Coult not open buffer_sizes.h for output" << endl;
        return 1;
    }

    bool called_ADPCM_encode = false;
    for (int i = 0; i < TRACE_SIZE; i++)
    {
            // If ADPCM_encode has not been called, then exclude
            // calls to ADPCM_decode (since there would be nothing
            // to decode) --- this behaviour cycles, of course
            // (to alternate encode/decode calls)
        const int rnd = random_int (5, dev_urandom);

        if (rnd == ADPCM)
        {
            if (called_ADPCM_encode)
            {
                write_function_call (ADPCM_decode, 0, calls);
            }
            else
            {
                write_function_call (ADPCM_encode, counter[rnd]++, calls);
            }
            called_ADPCM_encode = !called_ADPCM_encode;
        }
        else
        {
            write_function_call (rnd, counter[rnd]++, calls);
                // Call uses an offset for the random input data;
                // leave that offset incremented for next call
        }
    }

    defines << "#define ADPCM_COUNT " << counter[ADPCM]
            << "\n#define FFT_COUNT " << counter[FFT]
            << "\n#define AES_COUNT " << counter[AES]
            << "\n#define CRC_COUNT " << counter[CRC32] << '\n';

    return 0;
}

void write_function_call (Function function, int offset, ostream & out)
{
```

```
    switch (function)
    {
        case ADPCM_encode:
            out << "adpcm_coder(pcmdata + " << offset << "*PCMSIZE, adpcmdata, PCMSIZE, "
                   "&coder_1_state);" << endl;
            break;

        case ADPCM_decode:
            out << "adpcm_decoder(adpcmdata, pcmdata_2, PCMSIZE, &decoder_state);" << endl;
            break;

        case FFT:
            out << "fft_float (FFTSIZE, 0, real_in + " << offset << "*FFTSIZE, imag_in + "
                << offset << "*FFTSIZE, real_out, imag_out);" << endl;
            break;

        case RANDOM:
            out << "nothing = (random() ^ random()) & 0xFFFF;" << endl;
            break;

        case AES:
            out << "encrypt (plaintext + "<< offset << "*AESSIZE, ciphertext, &ctx);" << endl;
            break;

        case CRC32:
            out << "rc = crc32buf (crcdata + " << offset << "*CRCSIZE, CRCSIZE);" << endl;
            break;
    }
}

int random_int (int range, ifstream & dev_urandom)
{
    unsigned int rnd;
    dev_urandom.read(reinterpret_cast<char *>(&rnd), sizeof(unsigned int));

    return rnd % range;
}
```

Appendix D

Source Code for Continuous Classification Program

```
/*********************************************************************

   This program performs "continuous" classification to determine
   a sequence of function calls from a single power trace.  It
   has to perform "segmentation" of the trace, identifying the
   position and length of each of the function calls being
   identified.

   It corresponds to "experiment 2" as described in the text.

*********************************************************************/
#include <iostream>
#include <fstream>
#include <string>
#include <sstream>
#include <vector>
#include <map>
#include <algorithm>
#include <iterator>
#include <numeric>
#include <utility>
#include <cmath>
#include <complex>
#include <inttypes.h>
using namespace std;

#include <fftw3.h>

#include "fft_tools.h"

const double dc_weight = 0; // Feature not enabled for now

struct Training_sample  // Notice the different definition for NN rule
{
    string name;
    int size;
    vector< complex<double> > dft;

    Training_sample() : size(0) {}
```

```
    Training_sample (const string & name, int size,
                     const vector< complex<double> > & dft)
        : name(name), size(size), dft(dft)
    {}
};

ostream & operator<< (ostream & out, const Training_sample & s)
{
    out << s.name << " with size " << s.size;
    return out;
}

vector<Training_sample> read_training_samples (const char * train_samples_filename);
vector<double> read_trace (const char * trace_filename);


template <typename ConstIterator>
double distance (ConstIterator trace_begin, // assumed sufficient space
                                            // starting at this position
                 const Training_sample & sample,
                 map< int, vector< complex<double> > > & trace_dfts);

struct Near_neighbor
{
    string name;
    int start;
    int length;

    Near_neighbor (const string & name, int start, int len)
        : name(name), start(start), length(len)
    {}
};

ostream & operator<< (ostream & out, const Near_neighbor & nn)
{
    out << nn.name << ", at " << nn.start << ", length = " << nn.length;
    return out;
}


// This function returns only the nearest neighbours of the
// winning class --- For example, if k = 5, and the 5 nearest
// neighbours include 3 of class A, 1 of class B, and 1 of
// class C, then the function will only return thr three of
// class A.
//
// NOTE: It returns them SORTED --- the first one is the
// nearest (within the winning class --- not necessarily
// the nearest neighbour among all training samples)
template <typename ConstIterator>
vector<Near_neighbor> kNN  (unsigned int k,
                            ConstIterator trace_begin, ConstIterator trace_end,
                            const vector<Training_sample> & training_samples,
                            const string & excluded_from_training_db,
                            map< int, vector< complex<double> > > & trace_dfts,
                            bool first_segment);  // Just for the function to know whether it has to
                                                  // try starting positions before the point given

Near_neighbor closest (const vector<Near_neighbor> & nearest_neighbours);


int main (int argc, const char * arg[])
```

```
{
    if (argc < 3)
    {
        cerr << "Usage: " << arg[0] << " <k (the k in k-NN)> <trace filename> [<start pos>]" << endl;
        return 1;
    }
    const unsigned int k = atoi(arg[1]);
    if (k == 1)
    {
        cout << "Warning -- k = 1, boiling down to simple nearest-neighbour rule" << endl;
    }
    else if (k < 1)
    {
        cerr << "ERROR --- parameter k must be a positive integer" << endl;
        return 1;
    }

    const char * const train_samples_filename = "training-samples.txt";
    const int start_pos = (argc >= 4) ? atoi(arg[3]) : 0;

    const vector<Training_sample> & training_samples
        = read_training_samples (train_samples_filename);

    cout << "Read " << training_samples.size() << " samples" << endl;

    const vector<double> & trace = read_trace(arg[2]);

    map< int, vector< complex<double> > > trace_dfts;
        // To "memoize" the computation of the FFT for various sizes
        // (many samples have repeated size, so no need to re-compute
        // the FFT to match the sample size)

    string excluded_from_training_db; // Feature not used for now

    vector<double>::const_iterator start = trace.begin() + start_pos;
    while (start != trace.end())
    {
        const vector<Near_neighbor> & nearest_neighbours
            = kNN (k, start, trace.end(),
                   training_samples, excluded_from_training_db,
                   trace_dfts, start == trace.begin());

        const Near_neighbor & closest_match = closest (nearest_neighbours);

        cout << "Executed " << closest_match.name
             << " at time " << ((start - trace.begin()) + closest_match.start)
             << endl;

             // Need to advance one by one  (adding could skip over
             // trace.end() and produce an invalid iterator for which
             // even a less-than comparison could fail)
        for (int k = 0; k < (closest_match.length + closest_match.start) && start != trace.end(); ++k)
        {
            ++start;
        }
    }

    return 0;
}


template <typename ConstIterator>
struct Neighbour
```

```
{
    double dist;
    vector<Training_sample>::const_iterator trn_sample;
    int start;

    Neighbour()
    {}

    Neighbour (double dist,
               vector<Training_sample>::const_iterator trn_sample,
               int start)
        : dist(dist), trn_sample(trn_sample), start(start)
    {}
};

template <typename ConstIterator>
ostream & operator<< (ostream & out, const Neighbour<ConstIterator> & n)
{
    out << "Sample " << *(n.trn_sample) << " at position " << n.start
        << " (distance: " << n.dist << ')';
    return out;
}


// This function returns only the nearest neighbours of the
// winning class --- For example, if k = 5, and the 5 nearest
// neighbours include 3 of class A, 1 of class B, and 1 of
// class C, then the function will only return thr three of
// class A.
//
// NOTE: It returns them SORTED --- the first one is the
// nearest (within the winning class --- not necessarily
// the nearest neighbour among all training samples)
template <typename ConstIterator>
vector<Near_neighbor> kNN  (unsigned int k,
                            ConstIterator trace_begin, ConstIterator trace_end,
                            const vector<Training_sample> & training_samples,
                            const string & excluded_from_training_db,
                            map< int, vector< complex<double> > > & trace_dfts,
                            bool first_segment)
{
    if (trace_begin == trace_end)
    {
        throw runtime_error("");
            // Temp ...  At some point, should check that trace_end
            // is not being exceeded
    }

    // typedef pair<double,vector<Training_sample>::const_iterator> Neighbour;
    vector< Neighbour<ConstIterator> > nearest_neighbours;
    nearest_neighbours.reserve(k);

    nearest_neighbours.push_back (Neighbour<ConstIterator>(distance (trace_begin,
                                                                     training_samples.front(),
                                                                     trace_dfts),
                                  training_samples.begin(), 0));

    for (ConstIterator start = (first_segment ? trace_begin : trace_begin - 100);
                       start != trace_begin + 100;
                       ++start)
    {
        trace_dfts.clear();
        for (vector<Training_sample>::const_iterator s = training_samples.begin();
```

```
                                            s != training_samples.end();
                                            ++s)
    {
        if (s->name != excluded_from_training_db)
        {
            double cur_dist = distance (start, *s, trace_dfts);

            if (cur_dist < nearest_neighbours.back().dist ||
                nearest_neighbours.size() < k)
            {
                if (nearest_neighbours.size() < k)
                {
                    nearest_neighbours.push_back (Neighbour<ConstIterator>(cur_dist, s,
                                                                           start - trace_begin));
                }
                else
                {
                    nearest_neighbours.back() = Neighbour<ConstIterator>(cur_dist, s,
                                                                         start - trace_begin);
                }

                    // Send the newly-added item to its position
                    // (to keep the elements sorted)
                for (int i = nearest_neighbours.size() - 1;
                        i >= 1 && nearest_neighbours[i].dist < nearest_neighbours[i-1].dist;
                        --i)
                {
                    std::swap (nearest_neighbours[i], nearest_neighbours[i-1]);
                }
            }
        }
    }
}

    // Now determine majority vote

if (nearest_neighbours.size() != k)
{
    throw runtime_error ("ERROR --- nearest_neighbours does not have k elements");
}

map<string,int> votes;

for (unsigned int i = 0; i < k; i++)
{
    votes[nearest_neighbours[i].trn_sample->name]++;
}

int max_votes = 0;
string winner;
for (map<string,int>::const_iterator i = votes.begin();
                                     i != votes.end();
                                     ++i)
{
    if (i->second > max_votes)
    {
        max_votes = i->second;
        winner = i->first;
    }
}

vector<Near_neighbor> knn;
for (typename vector< Neighbour<ConstIterator> >::const_iterator
```

```
            n = nearest_neighbours.begin();
            n != nearest_neighbours.end();
            ++n)
    {
        if (n->trn_sample->name == winner)
        {
            knn.push_back (Near_neighbor(winner, n->start, n->trn_sample->size));
        }
    }

    return knn;
}

template <typename ConstIterator>
double distance (ConstIterator trace_begin,
                 const Training_sample & sample,
                 map< int, vector< complex<double> > > & trace_dfts)
{
    if (static_cast<unsigned int>(sample.size / 2 + 1) != sample.dft.size())
    {
        ostringstream error;
        error << "Inconsistent FFT size --- sample.size: " << sample.size
              << ", sample.dft.size: " << sample.dft.size() << endl;
        throw runtime_error (error.str());
    }

    const double mean = accumulate(trace_begin, trace_begin + sample.size, 0.0) / sample.size;

    if (trace_dfts[sample.size].empty())
    {
        double * data = static_cast<double *>(fftw_malloc (sample.size * sizeof(double)));

        copy (trace_begin, trace_begin + sample.size, data);

            // Remove DC and (conditionally) apply Hamming window
            // before computing FFT
        for (int i = 0; i < sample.size; ++i)
        {
            data[i] -= mean;
#ifdef WINDOWED_FFT
            data[i] *= hamming_window (i, sample.size);
#endif
        }

        fftw_complex * dft
           = static_cast<fftw_complex *>(fftw_malloc ((sample.size / 2 + 1) * sizeof(fftw_complex)));

        fftw_plan plan = fftw_plan_dft_r2c_1d (sample.size, data, dft, FFTW_ESTIMATE);
        fftw_execute (plan);

        vector< complex<double> > & trace_dft = trace_dfts[sample.size];
        trace_dft.clear();
        trace_dft.push_back (0.0);
        for (size_t i = 1; i < (sample.dft.size() / 2); ++i)
        {
            trace_dft.push_back (log(complex<double>(dft[i][0],dft[i][1])));
        }

        fftw_destroy_plan (plan);
        fftw_free (dft);
        fftw_free (data);
    }
```

```
    double dist = 0;
    const vector< complex<double> > & trace_dft = trace_dfts[sample.size];
    for (size_t i = 1 /* ignore DC */; i < (sample.dft.size() / 2); ++i)
    {
        complex<double> diff = trace_dft[i] - sample.dft[i];
        dist += square(diff.real()) + square(diff.imag());
        // double diff = trace_dft[i].real() - sample.dft[i].real();
        // dist += square(diff);
    }

    // Disregard DC in the FFT, but consider it (treated separately,
    // to compare against the measurements from the training data,
    // and give it a separate weight of its own)

    dist /= (sample.dft.size() / 2 - 1);

       // Adjust factor to favor longer traces when approx.
       // equally close match otherwise (after having normalized
       // to obtain square distance per dimension, etc.
    if (sample.dft.size() > 700) dist *= 0.85;

    return dist;
}


Near_neighbor closest (const vector<Near_neighbor> & nearest_neighbours)
{
    return nearest_neighbours.front();
}


vector<Training_sample> read_training_samples (const char * filename)
{
    ifstream file (filename);
    if (!file)
    {
        throw runtime_error (string("Could not open file ") + filename);
    }

    vector<Training_sample> training_samples;

    ostringstream sfn;
    string line;
    while (getline (file, line))
    {
        if (line != "" && line[0] != '#')
        {
            istringstream buf(line);
            int num_samples;
            string name;
            if (buf >> name >> num_samples)
            {
                for (int s = 1; s <= num_samples; ++s)
                {
                    sfn.str("");
                    sfn << name << "/trace-" << s << ".txt";
                    const int size = read_fft (sfn.str().c_str()).size();

                    sfn.str("");
                    sfn << name << "/trace-" << s << ".txt";
#ifdef WINDOWED_FFT
                    sfn << ".fft";
#else
```

```
                    sfn << "-non-windowed.fft";
#endif
                    const vector< complex<double> > & dft = read_complex_fft (sfn.str().c_str());

                    if (dft.size() != (size / 2 + 1))
                    {
                        cerr << "Inconsistent sizes on training samples FFTs for "
                             << sfn.str() << ": Trace size = "
                             << size << ", DFT size = " << dft.size() << endl;
                    }
                    training_samples.push_back (Training_sample(name, size, dft));
                }
            }
            else
            {
                ostringstream error;
                error << "Invalid format, file " << filename << ", line: `" << line << '\'';
                throw runtime_error (error.str());
            }
        }
    }

    return training_samples;
}

vector<double> read_trace (const char * trace_filename)
{
    ifstream file (trace_filename);
    if (! file)
    {
        ostringstream error;
        error << "Could not open data file " << trace_filename;
        throw runtime_error (error.str());
    }

    vector<double> trace;
    trace.reserve (4192);
    copy (istream_iterator<double>(file), istream_iterator<double>(),
          back_inserter(trace));

    return trace;
}
```

Appendix E

Source Code for Processing of Classifier Output

```
/*****************************************************************

   This program compares/matches the output from the classifier
   against the actual source code, prompting the user for
   assistance in the non-easy cases.

*****************************************************************/
#include <iostream>
#include <string>
#include <fstream>
#include <sstream>
#include <vector>
#include <map>
#include <cstdlib>
#include <algorithm>
#include <iterator>
using namespace std;

string abbrev_function (const string & line);
    // Transforms an actual line of code calling a function
    // to a short single-word name for the function being
    // called --- for example, it converts a line
    // fft_float (FFTSIZE, 0, real_in .... ) into just "FFT"

struct Misclassification
{
    Misclassification (const string & actual,
                       const string & classif,
                       size_t at, size_t len,
                       int num_functions)
        : actual_function(actual),
          classified_as(classif),
          at(at), length(len),
          num_functions_covered(num_functions)
    {}

    string actual_function;
    string classified_as;
    size_t at;
```

```
    size_t length;
    int num_functions_covered;
};

ostream & operator<< (ostream & out, const Misclassification & misclass)
{
    out << misclass.actual_function << " classified as " << abbrev_function (misclass.classified_as)
        << " at " << misclass.at << " (recovered after " << misclass.length << " trace samples)";
    return out;
}

string trivial_answer (const vector<string> & src_lines, size_t line_src,
                       const vector<string> & classif_lines, size_t line_classif);

void output_stats (const vector<Misclassification> & misclassifications);

int main (int argc, const char * arg[])
{
    string actual_sequence_filename, classif_out_filename;

    if (argc == 3)
    {
        actual_sequence_filename = arg[1];
        classif_out_filename = arg[2];
    }
    else
    {
        cout << "Tip: usage: " << arg[0] << " <actual seq from src (abbreviated)> <classif output>"
                "\n\n";

        cout << "File with actual execution sequence: " << flush;
        getline (cin, actual_sequence_filename);

        cout << "File with classifier output (incl. timing): " << flush;
        getline (cin, classif_out_filename);
    }

    ifstream actual_sequence (actual_sequence_filename.c_str()),
             classifier_out (classif_out_filename.c_str());
    if (!actual_sequence)
    {
        cerr << "Error: could not open file " << actual_sequence_filename << endl;
        return 1;
    }
    if (!classifier_out)
    {
        cerr << "Error: could not open file " << classif_out_filename << endl;
        return 1;
    }

    vector<string> src_lines, classif_lines;
    string line;
    while (getline(actual_sequence, line))
    {
        src_lines.push_back (line); // it's already in abbrev. form
    }
    while (getline(classifier_out, line))
    {
        classif_lines.push_back (line);
    }

    size_t line_src = 0, line_classif = 0;
    bool reached_end = false;
```

```
vector<Misclassification> misclassifications;

while (!reached_end)
{
    while (line_src < src_lines.size()
           && line_classif < classif_lines.size()
           && src_lines[line_src] == abbrev_function(classif_lines[line_classif]))
    {
        ++line_src; ++line_classif;
    }

    if (line_src == src_lines.size() || line_classif == classif_lines.size())
    {
        reached_end = true;
    }
    else
    {
        system ("clear");

        cout << "Difference (at src_line " << line_src
             << ", classif_line " << line_classif << "):\n"
             << "#   Src Code        Classifier output\n";

        size_t s, c;

        if (line_src >= 2 && line_classif >= 2)
        {
            s = line_src - 2, c = line_classif - 2;
        }
        else
        {
            s = c = 0;
        }

        for (int i = -2;
                  i <= 15 && s < src_lines.size() && c < classif_lines.size();
                  ++i)
        {
            cout << i << "  ";
            if (0 <= i && i < 10) cout << ' ';

            cout << src_lines[s]
                 << string(16 - src_lines[s].length(), ' ')
                 << classif_lines[c] << endl;
            ++s; ++c;
        }
    }

    bool invalid_input;
    do
    {
        line = trivial_answer (src_lines, line_src,
                               classif_lines, line_classif);

        const bool trivial_case = (line != "");
        if (! trivial_case)
        {
            cout << "\nEnter number of skip lines (skip src <space> skip classif"
                    " -- END if at end): " << flush;
            getline (cin, line);
        }
```

```
            if (line != "END")
            {
                istringstream buf (line);
                size_t skip_src, skip_classif;
                buf >> skip_src >> skip_classif;

                string d;
                int offset_start, offset_end;

                istringstream buf2(classif_lines[line_classif]);
                buf2 >> d >> d >> d >> d >> offset_start;
                istringstream buf3(classif_lines[line_classif + skip_classif]);
                buf3 >> d >> d >> d >> d >> offset_end;

                string confirm = "y";
                if (! trivial_case)
                {
                    cout << "Resync at " << src_lines[line_src + skip_src]
                         << " // " << classif_lines[line_classif + skip_classif]
                         << ";  misclassif range: " << offset_start << " to " << offset_end
                         << " --- correct ([y]/n)? " << flush;

                    getline (cin, confirm);
                }

                if (confirm != "n" && confirm != "N")
                {
                    const Misclassification misclass (src_lines[line_src],
                                                      classif_lines[line_classif],
                                                      offset_start,
                                                      offset_end - offset_start,
                                                      skip_src);
                    misclassifications.push_back (misclass);

                    line_src += skip_src;
                    line_classif += skip_classif;
                    invalid_input = false;

                }
                else
                {
                    invalid_input = true;
                }
            }
            else
            {
                reached_end = true;
            }
        }
        while (invalid_input);
    }

    output_stats (misclassifications);

    return 0;
}

string abbrev_function (const string & line)
{
    if (line.find("crc32") != string::npos)
    {
        return "CRC32";
    }
```

```
    else if (line.find("adpcm-encode") != string::npos
             || line.find("adpcm_coder") != string::npos)
    {
        return "adpcm-encode";
    }
    else if (line.find("adpcm-decode") != string::npos
             || line.find("adpcm_decoder") != string::npos)
    {
        return "adpcm-decode";
    }
    else if (line.find("fft") != string::npos)
    {
        return "FFT";
    }
    else if (line.find("encrypt") != string::npos
             || line.find(" aes ") != string::npos)
    {
        return "AES";
    }
    else if (line.find("random") != string::npos)
    {
        return "random";
    }

    return "";
}


void output_stats (const vector<Misclassification> & misclassifications)
{
    cout << "Processing with training-samples.txt:" << endl;
    system ("cat training-samples.txt");
    cout << endl;
    // Output all misclassifications first
    copy (misclassifications.begin(), misclassifications.end(),
          ostream_iterator<Misclassification>(cout, "\n"));

    map<string,int> misclass_count, intrusion_count;
        // misclass counts how many times a function was missed
        // intrusion counts how many times a function shows up when it shouldn't

    int total_misclass = 0;
    size_t total_misclass_len = 0;

    for (vector<Misclassification>::const_iterator m = misclassifications.begin();
                                                   m != misclassifications.end();
                                                   ++m)
    {
        misclass_count[m->actual_function]++;
        intrusion_count[abbrev_function (m->classified_as)]++;
        total_misclass += m->num_functions_covered;
        total_misclass_len += m->length;
    }

    cout << "Total misclassification length: " << total_misclass_len
         << "\nMisclassifications per function:\n";

    for (map<string,int>::const_iterator m = misclass_count.begin();
                                         m != misclass_count.end();
                                         ++m)
    {
        cout << m->first << string(16 - m->first.length(), ' ') << m->second << endl;
    }
```

```
    cout << "\nIntrusions (function showing up when it shouldn't) per function:\n";
    for (map<string,int>::const_iterator m = intrusion_count.begin();
                                         m != intrusion_count.end();
                                         ++m)
    {
        cout << m->first << string(16 - m->first.length(), ' ') << m->second << endl;
    }
}


string trivial_answer (const vector<string> & src_lines, size_t line_src,
                       const vector<string> & classif_lines, size_t line_classif)
{
    if (line_src < 5 || line_classif < 5
        || line_src > (src_lines.size() - 10)
        || line_classif > (classif_lines.size()))
    {
        return "";
    }

    const string & src = src_lines[line_src];
    const string & classif = abbrev_function(classif_lines[line_classif]);
    if (src == "FFT" || classif == "FFT")
    {
        return "";
    }

    bool single_func = true;
        // Single-function switch
    for (int i = 1; i <= 5; i++)
    {
        if (src_lines[line_src+i] != abbrev_function(classif_lines[line_classif+i]))
        {
            if (line_classif == 15 && line_src == 15)
            {
                cout << "NO SINGLE FUNCTION SWITCH -- i = " << i
                     << ", difference:\n"
                     << src_lines[line_src+i] << endl
                     << classif_lines[line_classif+i] << " -- abbrev: "
                     << abbrev_function(classif_lines[line_classif+i]) << endl;
            }
            single_func = false;
            break;
        }
    }
    if (single_func)
    {
        return "1 1";
    }

    bool skip_one = true;
        // Check if skip due to several consecutive classified random()
    if (abbrev_function(classif_lines[line_classif]) == "random"
        && abbrev_function(classif_lines[line_classif-1]) == "random")
    {
        for (int i = 0; i < 5; i++)
        {
            if (src_lines[line_src+i] != abbrev_function(classif_lines[line_classif+i+1]))
            {
                skip_one = false;
                break;
            }
        }
```

```
        if (skip_one)
        {
            return "0 1";
        }
    }

    return "";
}
```

References

[1] Milton Abramowitz and Irene A. Stegun (Editors). *Handbook of Mathematical Functions*. Dover Publications, 1965.

[2] Dakshi Agrawal, Bruce Archambeault, Josyula R. Rao, and Pankaj Rohatgi. The EM Side-Channel(s). *Cryptographic Hardware and Embedded Systems – CHES 2002*, pages 29–45, 2003.

[3] Dakshi Agrawal, Josyula Rao, and Pankaj Rohatgi. Multi-Channel Attacks. *Cryptographic Hardware and Embedded Systems – CHES 2003*, pages 2–16, 2003.

[4] C. Archambeault, E. Peeters, F.-X. Standaert, and J.-J. Quisquater. Tempate Attacks in Principal Subspaces. *Cryptographic Hardware and Embedded Systems – CHES!2006*, pages 1–14, 2006.

[5] Steven Arno and Ferrell Wheeler. Signed Digit Representations of Minimal Hamming Weight. *IEEE Transactions on Computers*, 42(8):1007–1010, 1993.

[6] Elaine Barker and Allen Roginsky. Transitions: Recommendation for trasnsitioning the use of cryptographic algorithms and key lengths. NIST Special Publication 800-131A, 2011. http://csrc.nist.gov/publications/nistpubs/800-131A/sp800-131A.pdf.

[7] Matt Bishop. *Computer Security: Art and Science*. Addison-Wesley, 2003.

[8] David Brumley and Dan Boneh. Remote Timing Attacks are Practical. *Proceedings of the 18th USENIX Security Conference*, 2003.

[9] David R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley, 1997.

[10] Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. Template Attacks. *Cryptographic Hardware and Embedded Systems – CHES 2002*, pages 13–28, 2003.

[11] Zhimin Chen and Yujie Zhou. Dual-Rail Random Switching Logic: A Countermeasure to Reduce Side Channel Leakage. *Workshop on Cryptographic Hardware and Embedded Systems*, 2006.

[12] B. Chevallier-Mames, M. Ciet, and M. Joye. Low-cost Solutions for Preventing Simple Side-Channel Analysis: Side-Channel Atomicity. *IEEE Transactions on Computers*, 53(6):760–768, 2004.

[13] Jim Cooling. *Software Engineering for Real-Time Systems*. Addison-Wesley, 2003.

[14] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Cliffor Stein. *Introduction to Algorithms*. The MIT Press, Third edition, 2009.

[15] Jean-Sébastien Coron. Resistance Against Differential Power Analysis for Elliptic Curve Cryptosystems. *Workshop on Cryptographic Hardware and Embedded Systems*, 1999.

[16] Atmel Corporation. AVR 8- and 32-bit Microcontrollers, 2012. http://www.atmel.com/products/microcontrollers/avr/.

[17] Intel Corporation. Intel Atom Processor N270, 2008. http://ark.intel.com/products/36331?wapkw=atom%20n270%20specifications.

[18] T. M. Cover and Joy A. Thomas. *Elements of Information Theory*. Wiley-Interscience, Second edition, 2006.

[19] Whitfield Diffie and Martin E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976.

[20] Harold M. Edwards. A Normal Form for Elliptic Curves. *Bulletin of the American Mathematical Society*, 44(3):393–422, 2007.

[21] Taher ElGamal. A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. *IEEE Transactions on Information Theory*, IT-31(4), 1985.

[22] Eric Jacobsen and Richard Lyons. The Sliding DFT. *IEEE Signal Processing Magazine*, 20(2):74–80, 2003.

[23] Eric Young et al. OpenSSL. http://www.openssl.org.

[24] Torbjörn Granlund et al. GNU Multi-Precision Arithmetic Library (GMP). http://www.gmplib.org.

[25] Junfeng Fan, Xu Guo, Elke De Mulder, Patrick Schaumont, Bart Preneel, and Ingrid Verbauwhede. State-of-the-art of Secure ECC Implementations: A Survey on Known Side-Channel Attacks and Countermeasures. *2010 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, pages 76–87, 2010.

[26] Junfeng Fan and Ingrid Verbauwhede. An Updated Survey on Secure ECC Implementations: Attacks, Countermeasures and Cost. *Cryptography and Security: From Theory to Applications*, pages 265–282, 2012.

[27] Pierre-Alain Fouque and Frédéric Valette. The Doubling Attack: Why Upwards is Better than Downwards. *Cryptographic Hardware and Embedded Systems – CHES 2003*, pages 269–280, 2003.

[28] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[29] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. Electromagnetic Analysis: Concrete Results. *Workshop on Cryptographic Hardware and Embedded Systems*, 2001.

[30] Catherine Gebotys. Design of Secure Cryptography Against the Threat of Power-Attacks in DSP Embedded Processors. *ACM Transactions on Embedded Computer Systems*, 3(1):92–113, 2004.

[31] Catherine Gebotys. *Security in Embedded Devices*. Springer, 2009.

[32] Benedikt Gierlichs. DPA-Resistance Without Routing Constraints? – A Cautionary Note About MDPL Security. *Workshop on Cryptographic Hardware and Embedded Systems*, 2007.

[33] Benedikt Gierlichs, Lejla Batina, and Pim Tuyls. Mutual Information Analysis. *Workshop on Cryptographic Hardware and Embedded Systems*, 2008.

[34] Daniel M. Gordon. A Survey of Fast Exponentiation Methods. *Journal of Algorithms*, 27(1):129–146, 1998.

[35] Jorge Guajardo and Christof Paar. Efficient Algorithms for Elliptic Curve Cryptosystems. In *Advances in Cryptology – CRYPTO '97*, volume 1294, pages 342–356. Springer Berlin / Heidelberg, 1997.

[36] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, WWC '01, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society.

[37] Jae Cheol Ha and Sang Jae Moon. Randomized Signed-Scalar Multiplication of ECC to Resist Power Attacks. *Workshop on Cryptographic Hardware and Embedded Systems*, 2002.

[38] Carl Hamacher, Zvonkp Vranesic, and Safwat Zaky. *Computer Organization*. McGraw-Hill, Fifth edition, 2002.

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

[40] M.A. Hasan. Power Analysis Attacks and Algorithmic Approaches to Their Countermeasures for Koblitz Curve Cryptosystems. *IEEE Transactions on Computers*, 50:1071–1083, 2001.

[41] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Fourth edition, 2007.

[42] Johann Heyszl, Stefan Mangard, Benedikt Heinz, Frederic Stumpf, and Georg Sigl. Localized electromagnetic analysis of cryptographic implementations. In Orr Dunkelman, editor, *Topics in Cryptology CT-RSA 2012*, volume 7178 of *Lecture Notes in Computer Science*, pages 231–244. Springer Berlin Heidelberg, 2012.

[43] HT Omega. Claro Plus – Online specifications. http://www.htomega.com/claroplus.html.

[44] Gareth A. Jones and J. Mary Jones. *Elementary Number Theory*. Springer, 1998.

[45] Marc Joye. Recovering Lost Efficiency of Exponentiation Algorithms on Smart Cards. *Electronic Letters*, 38(19):1095–1097, 2002.

[46] Neil Koblitz. Elliptic Curve Cryptosystems. *Mathematics of Computation*, 48(177):203–209, 1987.

[47] Paul Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. *Advances in Cryptology*, 1996.

[48] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. *Advances in Cryptology – CRYPTO' 99*, pages 388–397, 1999.

[49] Konrad Kulikowski, Alexander Smirnov, and Alexander Taubin. Automated Design of Cryptographic Devices Resistant to Multiple Side-Channel Attacks. *Workshop on Cryptographic Hardware and Embedded Systems*, 2006.

[50] Ralph Langner. Stuxnet: Dissecting a Cyberwarfare Weapon. *IEEE Security & Privacy*, 9(3):49–51, May-June 2011.

[51] Jane Liu. *Real-Time Systems*. Prentice Hall, 2000.

[52] François Macé, François-Xavier Standaert, and Jean-Jacques Quisquater. Information Theoretic Evaluation of Side-Channel Resistant Logic Styles. *Workshop on Cryptographic Hardware and Embedded Systems*, 2007.

[53] Edgar Mateos and Catherine Gebotys. Side Channel Analysis Using Giant Magneto-Resistive (GMR) Sensors. *Constructive Side-Channel Analysis and Secure Design (COSADE)*, 2011.

[54] Dominic Mazzoni. Audacity: Free Audio Editor and Recorder. http://audacity.sourceforge.net.

[55] Marcel Medwed and Elisabeth Oswald. Template Attacks on ECDSA. *Information Security Applications*, pages 14–27, 2009.

[56] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996. http://www.cacr.math.uwaterloo.ca/hac/.

[57] Nele Mentens, Benedikt Gierlichs, and Ingrid Verbauwhede. Power and Fault Analysis Resistance in Hardware through Dynamic Reconfiguration. *Workshop on Cryptographic Hardware and Embedded Systems*, 2008.

[58] Victor S. Miller. Use of Elliptic Curves in Cryptography. *Advances in Cryptology*, 1986.

[59] Amir Moradi, Nima Mousavi, Christof Paar, and Mahmoud Salmasizadeh. A comparative study of mutual information analysis under a gaussian assumption. In HeungYoul Youm and Moti Yung, editors, *Information Security Applications*, volume 5932 of *Lecture Notes in Computer Science*, pages 193–205. Springer Berlin Heidelberg, 2009.

[60] Carlos Moreno, Sebastian Fischmeister, and M. Anwar Hasan. Non-Intrusive Program Tracing and Debugging of Deployed Embedded Systems Through Side-Channel Analysis. 2013. To appear in Languages, Compilers and Tools for Embedded Systems – LCTES-2013.

[61] Carlos Moreno and M. Anwar Hasan. An Adaptive Idle-Wait Countermeasure Against Timing Attacks on Public-Key Cryptosystems. 2010. http://cacr.uwaterloo.ca/techreports/2010/cacr2010-16.pdf.

[62] Carlos Moreno and M. Anwar Hasan. Fast SPA-Resistant Exponentiation Through Simultaneous Processing of Half-Exponents. 2011. http://cacr.uwaterloo.ca/techreports/2011/cacr2011-13.pdf.

[63] Carlos Moreno and M. Anwar Hasan. SPA-Resistant Binary Exponentiation with Optimal Execution Time. *Journal of Cryptographic Engineering*, pages 1–13, 2011. 10.1007/s13389-011-0008-9.

[64] Carlos Moreno and M. Anwar Hasan. SPA-Resistant Binary Exponentiation with Optimal Execution Time. 2011. http://cacr.uwaterloo.ca/techreports/2011/cacr2011-03.pdf.

[65] Katsuyuki Okeya and Tsuyoshi Takagi. The Width-w NAF Method Provides Small Memory and Fast Elliptic Scalar Multiplications Secure against Side Channel Attacks. In *Topics in Cryptology – CT-RSA 2003*, volume 2612 of *Lecture Notes in Computer Science*, pages 328–343. Springer Berlin / Heidelberg, 2003.

[66] Christof Paar and Jan Pelzl. *Understanding Cryptography: A Textbook for Students and Practitioners*. Springer, 2010.

[67] Athanasios Papoulis and S. Unnikrishna Pillai. *Probability, Random Variables and Stochastic Processes*. McGraw-Hill, Fourth edition, 2002.

[68] Prakash Aayush and Hiren D. Patel. An Instruction Scratchpad Memory Allocation for the Precision Timed Architecture. Technical report, 2012. Computer Architecture and Embedded Systems Group Technical Report CAESR-TR-2012-04, University of Waterloo.

[69] W. Press, S. Teukolsky, W. Vetterling, and B. Flannery. *Numerical Recipes in C*. Cambridge University Press, Second edition, 1992.

[70] John G. Proakis and Dimitri G. Manolakis. *Digital Signal Processing: Principles, Algorithms, and Applications*. Prentice Hall, Fourth edition, 2006.

[71] Jean-Jacques Quisquater and David Samyde. Electromagnetic Analysis (EMA): Measures and Counter-measures for Smart Cards. *Smart Card Programming and Security (E-Smart2001)*, 2001.

[72] Christian Rechberger and Elisabeth Oswald. Practical Template Attacks. *Information Security Applications*, pages 440–456, 2005.

[73] Ronald Rivest, Adi Shamir, and Leonard Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[74] Werner Schindler. A Timing Attack Against RSA with the Chinese Remainder Theorem. *Workshop on Cryptographic Hardware and Embedded Systems*, 2000.

[75] Werner Schindler. Optimized Timing Attacks Against Public Key Cryptosystems. *Statistics and Decisions*, 2002.

[76] Adi Shamir and Eran Tromer. Acoustic Cryptanalysis on Nosy People and Noisy Machines, 2004. Preliminary Proof-of-Concept Presentation.

[77] Jerome A. Solinas. Low-Weight Binary Representations for Pairs of Integers. *Centre for Applied Cryptographic Research Technical Report CORR 2001-41*, 2001. http://www.cacr.math.uwaterloo.ca/techreports/2001/corr2001-41.ps.

[78] Douglas R. Stinson. *Cryptography: Theory and Practice*. Chapman & Hall / CRC, Third edition, 2006.

[79] Da-Zhi Sun, Jin-Peng Huai, Ji-Zhou Sun, and Zhen-Fu Cao. An Efficient Modular Exponentiation Algorithm against Simple Power Analysis Attacks. *IEEE Transactions on Consumer Electronics*, 53(4):1718–1723, 2007.

[80] Nicolas Veyrat-Charvillon and François-Xavier Standaert. Mutual information analysis: How, when and why? In Christophe Clavier and Kris Gaj, editors, *Cryptographic Hardware and Embedded Systems - CHES 2009*, volume 5747 of *Lecture Notes in Computer Science*, pages 429–443. Springer Berlin Heidelberg, 2009.

[81] John Viega and Gary McGraw. *Building Secure Software*. Addison-Wesley, 2002.

[82] Andrew R. Webb and Keith D. Copsey. *Statistical Pattern Recognition*. Wiley, Third edition, 2011.

[83] Peter Williams and Radu Sion. Usable PIR. In *Proceedings of the 2008 Network and Distributed System Security (NDSS) Symposium*, 2008.