A Network-based Asynchronous Architecture for Cryptographic Devices

Ljiljana Spadavecchia

Doctor of Philosophy

Institute for Computing Systems Architecture
School of Informatics
University of Edinburgh

2005

Abstract

The traditional model of cryptography examines the security of the cipher as a mathematical function. However, ciphers that are secure when specified as mathematical functions are not necessarily secure in real-world implementations. The physical implementations of ciphers can be extremely difficult to control and often leak so-called *side-channel* information. Side-channel cryptanalysis attacks have shown to be especially effective as a practical means for attacking implementations of cryptographic algorithms on simple hardware platforms, such as smart-cards. Adversaries can obtain sensitive information from side-channels, such as the timing of operations, power consumption and electromagnetic emissions. Some of the attack techniques require surprisingly little side-channel information to break some of the best known ciphers. In constrained devices, such as smart-cards, straightforward implementations of cryptographic algorithms can be broken with minimal work. Preventing these attacks has become an active and a challenging area of research.

Power analysis is a successful cryptanalytic technique that extracts secret information from cryptographic devices by analysing the power consumed during their operation. A particularly dangerous class of power analysis, differential power analysis (DPA), relies on the correlation of power consumption measurements. It has been proposed that adding non-determinism to the execution of the cryptographic device would reduce the danger of these attacks. It has also been demonstrated that asynchronous logic has advantages for security-sensitive applications. This thesis investigates the security and performance advantages of using a network-based asynchronous architecture, in which the functional units of the datapath form a network. Non-deterministic execution is achieved by exploiting concurrent execution of instructions both with and without data-dependencies; and by forwarding register values between instructions with data-dependencies using randomised routing over the network. The executions of cryptographic algorithms on different architectural configurations are simulated, and the obtained power traces are subjected to DPA attacks. The results show that the proposed architecture introduces a level of non-determinism in the execution that significantly raises the threshold for DPA attacks to succeed. In addition, the performance analysis shows that the improved security does not degrade performance.

Acknowledgements

I am deeply grateful to my husband, Joseph, for his love, patience and continuous support during the many difficult times of my PhD studies.

My beloved parents, grandmother and sister and all my relatives and friends from Serbia and the USA for their support and encouragement.

My first supervisor, D. K. Arvind, for his advice and comments.

My second supervisor, Dr. Murray Cole, for his advice and encouragement.

Joseph, Dr. Aris Efthymiou, Dr. Murray Cole, D. K. Arvind, Dr. Mary Cryan and Chris Bainbridge for proofreading the thesis material and for their helpful comments.

The Overseas Research Student (ORS) Award Scheme for covering the overseas tuition fees. To the Graduate School of Informatics for covering the home fees and partial maintenance. To the System Level Integration group for providing some of the maintenance funding. To the Informatics Teaching Organisation for teaching jobs.

The Orthodox Community of St. Andrew in Edinburgh, and in particular to Fr. John for being my dear spiritual guide.

My dear friends John, Spyros, Fotini, Katarina, Alin, Cornelia, Chris and Evie for making my stay in Edinburgh an indeed wonderful experience.

To my (at the time unborn) baby Tihomir for making the time during the thesis write-up the most memorable and wonderful time of my life.

Glory to God for all things!

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

Some of the work presented in this thesis has already been published in:
Ljiljana Dilparić* and D. K. Arvind. Design and Evaluation of a Network-Based Asynchronous Architecture for Cryptographic Devices. In *Proceedings of the 15th IEEE International Conference on Application-Specific Systems, Architectures, and Processors (ASAP 2004)*, 27-29 September 2004, Galveston, Texas, USA.

(*Ljiljana Spadavecchia*)

*Dilparić is the thesis author's maiden name.

To Joseph and Tihomir

Table of Contents

1 Introduction **1**

1.1 Thesis aims and contributions 4

1.2 Thesis structure 6

2 Cryptographic Algorithms **9**

2.1 Introduction 9

2.2 Data encryption standard - DES 10

2.2.1 History 10

2.2.2 Algorithm 10

2.2.3 Cryptanalysis of DES 14

2.3 Advanced encryption standard - AES 18

2.3.1 History 18

2.3.2 Algorithm 19

2.3.3 Cryptanalysis of AES 22

2.4 Summary 24

3 Side-channel Analysis **25**

3.1 Introduction 25

3.2 Timing analysis 26

3.2.1 Introduction 26

3.2.2 Attack details 27

3.2.3 Countermeasures 30

3.3 Power analysis 32

3.3.1 Introduction 32

3.3.2 Power dissipation 33

3.4 Simple power analysis 35

3.4.1 Attack details 35

3.4.2 Countermeasures 39
3.5 Differential power analysis 40
3.5.1 Introduction 40
3.5.2 Attack details 41
3.5.3 Increasing the magnitude of the bias signal 44
3.5.4 Higher-order DPA attacks 46
3.5.5 Variations of the DPA attack 48
3.5.6 Countermeasures 50
3.5.7 Software countermeasures 51
3.5.8 Hardware countermeasures 67
3.6 Electromagnetic emission analysis 71
3.6.1 Introduction 71
3.6.2 Attack details 72
3.6.3 Countermeasures 74
3.7 Fault analysis 75
3.7.1 Introduction 75
3.7.2 Attack details 76
3.7.3 Countermeasures 82
3.8 Summary 83

4 Asynchronous Architectures 87
4.1 Introduction 87
4.2 Asynchronous control 87
4.3 Asynchronous circuits 88
4.4 Communication in asynchronous circuits 89
4.4.1 Handshaking protocols 90
4.4.2 Encoding schemes 91
4.5 Advantages of asynchronous design 93
4.5.1 No clock skew 93
4.5.2 Low power consumption 93
4.5.3 Average-case instead of worst-case performance 94
4.5.4 Improved electromagnetic compatibility 95
4.5.5 Modularity of design 95
4.5.6 Simplified layout and improved robustness 96
4.6 Disadvantages of asynchronous design 96

4.6.1 Design complexity 96
4.6.2 Completion detection problems 97
4.6.3 Testing difficulties 97
4.6.4 Lack of tools 98
4.6.5 Performance measurement difficulties 98
4.7 Pipelines 98
4.8 Exploiting instruction level parallelism 100
4.9 Micronet 100
4.9.1 Introduction 100
4.9.2 Synchronous, asynchronous and micronet pipeline 101
4.9.3 Micronet as an asynchronous network of micro-operations . . 103
4.9.4 Micronet implementations 104
4.9.5 Summary 106
4.10 Side-channel analysis of asynchronous architectures 106
4.10.1 Motivation for using asynchronous architectures for cryptographic devices 106
4.10.2 Side-channel analysis of dual-rail asynchronous architectures 107
4.11 Summary 111

5 Design of the Network-based Asynchronous Architecture 113
5.1 Introduction 113
5.2 Design goals 114
5.3 Overview of the network-based architecture 116
5.4 Architectural components 119
5.5 Instruction execution 122
5.5.1 Instruction fetch 122
5.5.2 Instruction issue 123
5.5.3 Instruction compounding 127
5.5.4 Operand fetch-and-lock 132
5.5.5 Evaluation and write-back 144
5.6 Data-forwarding 149
5.6.1 The network topology 150
5.6.2 Data-forwarding and randomised routing 154
5.6.3 Data-forwarding and secret-sharing 157
5.6.4 On-chip random number generator 158

5.7 An example 159
5.8 Features 162
5.9 Summary 165

6 Evaluation **167**
6.1 Introduction 167
6.2 Evaluation framework 168
6.2.1 Asynchronous event-driven simulator 168
6.2.2 Parametric model 171
6.2.3 SUIF compiler 174
6.2.4 Power profiling 174
6.3 Security evaluation 175
6.3.1 Experimental setup 176
6.3.2 Covariance attack on AES 179
6.3.3 Differential power analysis of DES 193
6.4 Performance evaluation 195
6.5 Summary 199

7 Conclusions and Future Work **201**
7.1 Summary 201
7.2 Future work 204
7.3 Conclusions 206

A Published Paper **207**
A.1 Design and Evaluation of the Network-based Asynchronous Architecture for Cryptographic Devices 207

B Instruction Set **221**

C Rijndael and DES Tables **225**

Bibliography **229**

List of Figures

2.1 The Feistel structure of DES encryption algorithm. 11
2.2 The DES round function. 13
2.3 DES key selection function. 14
2.4 Rijndael round transformation. Obtained from http://home.ecn.ab.ca/∼jsavard/crypto/images/rijnov.gif 23

3.1 The timing analysis principle [94]. 28
3.2 The power analysis principle [94]. 33
3.3 CMOS inverter. 33
3.4 SPA attack on DES [78]. 36
3.5 Routines vulnerable to first and second-order DPA attacks. 47
3.6 The integration operation of the SW-DPA technique [39]. 68

4.1 Communication using handshake protocols. 90
4.2 Handshake protocols. 91
4.3 Dual-rail encoding scheme. 92
4.4 Four-stage pipeline. 99
4.5 Pipelines. 102
4.6 Micronet [105]. 105
4.7 Dual-rail encoding with alarm signal definition. 108

5.1 Execution times of the architectural configurations with (NET) and without (NO_NET) data-forwarding. 117
5.2 A block diagram of the network-based asynchronous architecture with four functional units. 119
5.3 Fetch-and-branch unit and the instruction fetch stage. 123
5.4 Instruction issue. 125
5.5 Instruction issue and completion order of the fetch-and-lock stage. . . 127

5.6 An example of instruction compounding. 130
5.7 The operand fetch-and-lock stage. 133
5.8 The three-step register lock procedure. 136
5.9 Register bank arbiter: `reserveLock` and `grantRead` queues. 136
5.10 The three-step register read procedure. 137
5.11 Instruction evaluation and write-back. 145
5.12 An example of memory data-hazards. 148
5.13 Memory access arbitration. 149
5.14 Binary hypercube H(3) and partial binary hypercube PH(6). 152
5.15 Directed binary de Bruijn graph DB(8). 153
5.16 Data-forwarding communication in a hypercube network configuration. 157
5.17 A sample execution of compounded instructions. 160
5.18 Hypercube H(2) organisation of functional units. 161

6.1 Delay distribution for different architectural components in virtual time units (VTUs). 173
6.2 Security evaluation process. 175
6.3 Distribution of functional units (FU) among arithmetic (AU), logic (LU), multiplier (MULT) and memory (MU) units. 177
6.4 A sample covariance plot for the PIPE configuration with the Hamming weight power model and non-variable delays, derived from 200 power profiles. 180
6.5 The covariance attack on the PIPE configuration with the Hamming weight power model and non-variable delays. 181
6.6 A sample covariance plot for the ASYNC4 configuration with the Hamming weight power model and non-variable delays, derived from 300 power profiles. 182
6.7 The covariance attack on the ASYNC4 configuration with the Hamming weight power model and non-variable delays. 182
6.8 A sample covariance plot for the ASYNC6 configuration with the Hamming weight power model and non-variable delays, derived from 300 power profiles. 183
6.9 The covariance attack on the ASYNC6 configuration with the Hamming weight power model and non-variable delays. 183

6.10 A sample covariance plot for the PH4 configuration with the Hamming weight power model and non-variable delays, derived from 5000 power profiles. 184

6.11 The covariance attack on the PH4 configuration with the Hamming weight power model and non-variable delays. 184

6.12 A sample covariance plot for the PH6 configuration with the Hamming weight power model and non-variable delays, derived from 25000 power profiles. 185

6.13 The covariance attack on the PH6 configuration with the Hamming weight power model and non-variable delays. COV1 and COV4 are covariance plots for the 1st (value 0) and the 4th key bit (value 1). . . 186

6.14 The covariance attack on the PH6 configuration with the Hamming weight power model and non-variable delays using 5000 power samples. COV1 and COV4 are covariance plots for the 1st (value 0) and the 4th key bit (value 1). 186

6.15 A sample covariance plot for the PH7 configuration with the Hamming weight power model and non-variable delays, derived from 50000 power profiles. 186

6.16 The covariance attack on the PH7 configuration with the Hamming weight power model and non-variable delays. COV1 and COV4 are covariance plots for the 1st (value 0) and the 4th key bit (value 1). . . 187

6.17 The covariance attack on the PHS4 configuration with the Hamming weight power model and non-variable delays derived from 35000 power profiles. COV1 and COV4 are covariance plots for the 1st (value 0) and the 4th key bit (value 1). 188

6.18 The covariance attack on the PHS6 configuration with the Hamming weight power model and non-variable delays, derived from 60000 power profiles. COV1 and COV4 are covariance plots for the 1st (value 0) and the 4th key bit (value 1). 188

6.19 The covariance attack on the PHS7 configuration with the Hamming weight power model and non-variable delays, derived from 75000 power profiles. COV1 and COV4 are covariance plots for the 1st (value 0) and the 4th key bit (value 1). 188

6.20 A sample covariance plot for the DB4 configuration with the Hamming weight power model and non-variable delays, derived from 35000 power profiles. 189
6.21 The covariance attack on the configuration DB4 with the Hamming weight power model and non-variable delays. COV1 and COV4 are covariance plots for the 1st (value 0) and the 4th key bit (value 1). . . 189
6.22 The covariance attack on the configuration DB6 with the Hamming weight power model and non-variable delays, derived from 85000 power profiles. COV1 and COV4 are covariance plots for the 1st (value 0) and the 4th key bit (value 1). 189
6.23 Number of power samples necessary to attack de Bruijn network configurations with the Hamming weight power model and non-variable delays. 190
6.24 Number of power samples used in the attacks on PIPE and PH configurations with the transition count power model and non-variable delays. 191
6.25 Number of power samples used to perform the covariance attack on AES run on different architectural configurations with non-variable delays. 192
6.26 Number of power samples necessary to attack hypercube network configurations with the Hamming weight power model. 192
6.27 The DPA attack on 35000 power profiles obtained from running DES on PH6 configuration with the Hamming weight power model and non-variable delays. 194
6.28 Number of power samples used to perform the DPA attack on DES run on the PIPE and PH configurations of the architecture with the Hamming weight power model and non-variable delays. 194
6.29 Relative execution times of PH and PHS configurations. DIST1, DIST2 and DIST3 represent different distribution of units. 195
6.30 Performance overheads of data-sharing for PH and PHS configurations. DIST1, DIST2 and DIST3 represent different distribution of units. 196
6.31 Variations in execution times of successive runs of the same algorithm for PH7 and PHS7 configurations. 197
6.32 Relative execution times of DB and DBS configurations. DIST1, DIST2 and DIST3 represent different distribution of units. 197

6.33 Performance overheads of data-sharing for DB and DBS configurations. DIST1, DIST2 and DIST3 represent different distributions of units. 198
6.34 Variations in execution times of successive runs of the same algorithm for DB7 and DBS7 configurations. 198
6.35 Performance comparisons of five architectural configurations. 199

C.1 Rijndael: Number of rounds as a function of the block and key length. 225
C.2 Rijndael: Shift offsets for different block lengths 225
C.3 DES: E bit-selection table. 226
C.4 DES: Key schedule permuted choice 1. 226
C.5 DES: Key schedule permuted choice 2. 226
C.6 DES: Key schedule left shift order. 227

List of Algorithms

1 DES encryption algorithm 12
2 Rijndael encryption algorithm 20
3 Repeated left-to-right square-and-multiply algorithm for modular exponentiation. 28
4 Repeated square-and-multiply algorithm for modular exponentiation, still vulnerable to timing attacks. 31
5 SPA-resistant repeated square-and-multiply algorithm. 39
6 Boolean-to-arithmetic masking. 53
7 Double-and-add algorithm for scalar multiplication 58
8 Addition-subtraction algorithm for scalar multiplication 59
9 Double-and-add scalar multiplication resistant to SPA attack. 60
10 Scalar multiplication using the Montgomery method. 62
11 Repeated left-to-right square-and-multiply algorithm for modular exponentiation, which models register faults. 78
12 Communication unit: operand `fetch` procedure. 139
13 Communication unit: operand `lock` procedure. 140
14 Register bank arbiter: `grantRead` procedure. 140
15 Register bank arbiter: `update` procedure. 141
16 Register bank arbiter: `reserveLock` procedure. 142
17 Register bank: `write` procedure. 142
18 Register bank: `read` procedure. 143
19 Register bank: `lock` procedure. 144

Chapter 1

Introduction

Cryptography in its traditional setting examines the security of the cipher as a mathematical function. In addition, it assumes that the secret information can be physically protected in tamper-proof locations and manipulated in closed, reliable computing environments. However, cryptographic systems are implemented on real electronic devices that process, transmit and store data. While operating, these devices interact with and influence the environment and leak a certain amount of information into so-called *side-channels*. An attacker can potentially compromise the secret cryptographic key stored in these devices by monitoring information that is leaked into side-channels. This type of cryptanalysis is known as *side-channel analysis*.

Numerous techniques for testing cryptographic algorithms in isolation have been designed. The most well known and studied methods, *differential cryptanalysis* [27] and *linear cryptanalysis* [90], can exploit extremely small statistical characteristics in the cipher's inputs and outputs. However, these methods analyse only one part of a cryptosystem's architecture: the algorithm's mathematical structure. On the other hand, by employing side-channel analysis the attacker is able to exploit weaknesses of physical implementations, rather than weaknesses of algorithmic aspects of a particular cryptosystem. Ongoing research in the last ten years (since 1995) has shown that the information transmitted via side-channels, such as execution time [76], computational faults [30, 28], power consumption [78] and electromagnetic emissions [113, 53, 13], can be detrimental to the security of ciphers.

Hundreds of millions of cryptographic devices, the vast majority being smart-cards, are used today in a variety of applications. These cards execute cryptographic computations based on the secret key stored in their memories. The goal of an attacker is to extract the secret key from a tamper-resistant card in order to modify its content, create

duplicate cards or perform an unauthorised transaction. Two general types of attacks can be distinguished:

1. *Invasive attacks* are attacks where the smart-card can be decomposed, its chip extracted, modified, probed, partially destroyed or used in a particular environmental setting. These attacks leave visible proof of tampering. They typically require a considerable amount of time, sophisticated (often very expensive) equipment and detailed knowledge of the card's internals. Due to these factors, invasive attacks are usually applied to extract information about the smart-card systems, and rarely to extract information about individual users. These attacks include fault attacks [30] and probing attacks [80].

2. *Non-invasive attacks* are attacks where the smart-card is passively monitored during its operation and communication with a (possibly modified) smart-card reader. No proof of tampering is evident from these attacks. They require minimal investment and can be carried out in relatively short amounts of time. These characteristics of non-invasive attacks have made them of great interest in recent years. Non-invasive attacks include side-channel attacks [76, 77] and glitch attacks [80]. The focus of this thesis is on side-channel attacks in particular.

Side-channel attacks were first discovered by Paul Kocher in 1995. The first side-channel discovery was the *timing attack* [76] which uses timing information to deduce the values of the secret keys. This attack exploits weaknesses in implementations of the observed cryptosystem, and correlates the time needed to perform the cryptographic operation with the operations performed and the input parameters. A typical example of these weaknesses are branches in the code that depend on the values of the secret key, found in square-and-multiply algorithm that is used in ciphers such as RSA [117].

The next attack to appear, the *power analysis attack* [78], was discovered in 1998 by Paul Kocher and his team of researchers from Cryptography Research in San Francisco. Kocher et al. described two types of attacks: *simple power analysis* (SPA) and *differential power analysis* (DPA). Basic to these attacks is the observation that the power consumed by the cryptographic device (in this case the smart-card) at any particular time during the cryptographic operation is related to the instruction being executed and to the data being processed. One of the ideas to prevent the timing attack on the square-and-multiply algorithm was to pad the code with dummy computations, such as empty loops. Kocher et al. noticed that the power consumption of these dummy

computations was different from the power consumption of meaningful ones. By simply observing the power traces obtained from the RSA coprocessor, they were able to determine which operations were performed, what enabled them to disclose the secret exponent. This is the basis of simple power analysis.

A far more powerful attack, the differential power analysis (DPA), is based on performing a statistical analysis of a large number of encryptions with known plaintexts (or ciphertexts). There are variants of this attack that do not require the knowledge of either plaintexts or the ciphertexts [29] and variants that use more sophisticated statistical methods, known as *higher-order DPA attacks* [78].

Another type of very powerful side-channel analysis attacks is based on measuring electromagnetic emissions, and is known as *electromagnetic emission analysis* (EMA) [53, 113]. The techniques used in electromagnetic analysis are very similar to those used in power analysis, although in some cases these attacks have proven to be even more threatening than power analysis attacks [115].

Probably the most threatening and well studied side-channel attack is the DPA attack. The DPA attack exploits the characteristic behaviour of transistor logic gates and software running on today's smart-cards and other cryptographic devices. The attack is performed by monitoring the electrical activity of a device, and then using advanced statistical methods secret information (such as secret keys and user PINs) stored in the device is determined. Far from being a theoretical attack DPA has been successfully carried out on a wide range of existing cryptographic devices and, therefore, represents a real threat to the security of modern cryptographic systems. What makes the DPA attack especially dangerous is the fact that it is inexpensive to perform (using cheap and readily available equipment) and most implementations are vulnerable, unless specific countermeasures are in place. The degree of security these countermeasures provide can be different, but any countermeasure is valuable because it increases the cost and the complexity of performing the attack. The complexity of power analysis attacks can be increased by introducing *software (algorithmic)* and *hardware (physical)* countermeasures. A general strategy to render side-channel attacks more difficult to apply is to *balance* and *randomise* major computations which involve the secret key. These attacks largely depend on the possibility to statistically correlate different runs of the same algorithm with the same key and different plaintexts. This means to correlate power consumption curves and the points on the curves that correspond to vulnerable operations (*i.e.* those that involve the secret key).

A number of countermeasures against the DPA attack and its variations have been proposed in recent years. However, the vast majority of these countermeasures do not guarantee security against these attacks, but rather raise the threshold for such attacks to succeed or force the use of more complex and costly techniques. A general observation concerning software countermeasures is that they are easy and inexpensive to implement (as they do not require the redesign of the existing hardware), but are not applicable to every cipher and are still susceptible to higher-order DPA attacks or signal processing analysis [94]. Hardware countermeasures, similarly to software countermeasures, focus on destroying the correlation between the power measurements and the values of the secret key. Another target of hardware countermeasures is the alignment of operations in power consumption curves, an important property used by DPA. Removing the correlation between features in the DPA profile and the algorithm source code makes retrieving useful information from the power traces significantly harder. Hardware countermeasures can generally provide a higher level of security but can also be costly in terms of performance, power efficiency and memory requirements.

1.1 Thesis aims and contributions

With the discovery of side-channel attacks security at the physical level of cryptographic hardware has become crucial. At the same time, low-power hand-held cryptographic devices, such as smart-cards, have become ubiquitous. Today smart-cards are used in a large number of applications including authentication and payment mechanisms. They are harder to crack than their magnetic strip predecessors, but are, however, still threatened by the wide range of invasive and non-invasive attacks. In addition, cracking smart-cards has become increasingly profitable. The wide-spread use of smart-cards provides those capable of reverse engineering or simply extracting the secret key material from smart-cards with new opportunities for theft and fraud [102]. This is the type of environment in which modern smart-cards need to survive.

A critical question, addressed in this thesis, is how to secure the physical layer of cryptographic devices against side-channel attacks without degrading performance. In that direction, this thesis concentrates on the design of an architecture that is robust to DPA attacks.

Asynchronous architectures have been suggested as an attractive platform for secure cryptographic devices [113, 102]. The reduced power consumption of these devices and the absence of the clock, the source of correlation in power consumption

curves, suggest that these architectures could exhibit improved security characteristics.

One of the proposed solutions to thwart the DPA attack was to introduce randomness and non-determinism in the execution [80, 78, 36, 91]. Due to the data-dependent nature of delays in asynchronous circuits, the precise ordering of events is usually non-deterministic. This thesis explores possibilities for increasing this already present level of non-determinism in the execution.

The main contribution of this thesis is a novel architectural approach to thwart DPA in the form of a *network-based asynchronous architecture*, in which the functional units in the processor datapath are themselves connected as an asynchronous network, rather than as a linear pipeline. The aim of this design is to decorrelate the power consumption measurements by exploiting the inherent non-determinism of instructions executing in parallel over a network in which routing of data is randomised. Data-dependencies between instructions are identified at run-time and the dependency information is used in *data-forwarding* in order to bypass the register file. The functional units are organised in a structure that belongs to so-called *graphs on alphabets* [81]. Now, each forwarding operation requires routing of the data through the network. Additionally, the routing is randomised and introduces random timing variations in the execution of the algorithm. The term non-determinism, used throughout the thesis, refers to the execution of instructions in a non-deterministic fashion, *i.e.*, randomising the order of instruction execution and, thus, their timings. Randomisation is achieved through a randomised data-forwarding process. This process introduces different timing interleavings and, thus, randomises (or adds non-determinism to) (1) the order of execution for different microinstructions and consequently instructions; (2) execution times, making them different for different runs of the code; and (3) execution power signatures, making them different for different runs of the code.

Similar concepts which use special mechanisms to randomise the execution of instructions to achieve similar goals, have been presented in [91, 92, 66]. But unlike [91, 92], in which the randomisation process is an overhead, the asynchronous network executes instructions in parallel to improve performance, while non-deterministic execution is a natural side-effect. The non-deterministic execution should result in power signatures that are harder to correlate using statistical methods, which provides a level of protection against power analysis attacks.

The main aim of this thesis is to investigate the validity of architectural ideas that aim at improving the security of cryptographic devices by introducing non-determinism

in the execution. In that direction, the main contribution of this thesis is provided evidence that the network-based asynchronous architecture does improve the resistance of cryptographic functions to DPA attacks. This makes the network-based asynchronous architecture an attractive platform for security-sensitive applications.

1.2 Thesis structure

The summary of the remaining chapters is given next.

Chapter 2 presents the details of the cryptographic algorithms that were used in the security investigations in this thesis. This includes the definition and specification of the Data Encryption Standard (DES) and the Advanced Encryption Standard (AES). It also presents well-known (non-side-channel) cryptanalytic methods for attacking these two important ciphers.

Chapter 3 provides details of the main background area, side-channel analysis. This includes details on three types of side-channel attacks: (1) timing analysis, (2) simple and differential power analysis, and (3) electromagnetic emission analysis; and the fault analysis as another important threat to cryptographic devices. This chapter also gives background on power dissipation, and covers some of the countermeasures proposed to defend cryptosystems against these attacks.

Chapter 4 introduces the second background area, asynchronous design. This chapter also reviews related work on the asynchronous network-based architecture and side-channel analysis attacks on asynchronous architectures.

Chapter 5 provides a detailed description of the design of the network-based asynchronous architecture. In particular, this chapter presents the architecture organisation and its building blocks, instruction execution through its stages, data-forwarding, routing in the network of functional units and data-sharing as used in this design. It also provides the details of the network topologies and the randomised routing techniques used in this design.

Chapter 6 presents the experimental evaluation of both security and performance of the proposed architecture. It gives a detailed description of the simulation environment, along with the results for several architectural configurations running DES and AES.

Chapter 7 summarises the work presented and discusses the contributions of the thesis. It also identifies overall conclusions are drawn and future work.

Chapter 2

Cryptographic Algorithms

2.1 Introduction

For more than 40 years Data Encryption Standard (DES) [10] has been the most widely used commercial encryption algorithm for protecting financial transactions and electronic communications worldwide. Developed by the US Government and IBM in the 1970s, DES was the government-approved symmetric algorithm for protecting sensitive information. The DES algorithm uses a 56-bit encryption key, which means that there are 72,057,594,037,927,936 possible keys. Considering the computational power level of the 1970s, exhaustive search on the key space of this size was infeasible. However, with the increase in computational power this has become feasible. A machine jointly built by Cryptography Research, Advanced Wireless Technologies, and Electronic Frontier Foundation can perform a fast key search on DES. This project developed purpose-built hardware and software to search 90 billion keys per second, and was able to determine the key after only 56 hours. This attack demonstrated that the exhaustive search on DES is possible and that the 56-bit key length is not sufficient. However, performing this attack is expensive. The major concern for smart-card manufactures are the attacks which can be performed with relatively inexpensive equipment in a small amount of time, such as side-channel attacks.

In 1997 the US National Institute of Standards and Technology (NIST) made the first call for proposals for an Advanced Encryption Standard (AES). The cipher key size were specified to be 128, 196 and 256 bits with block lengths of 128 bits. In October 2000, Rijndael [45] was announced as the choice for AES.

2.2 Data encryption standard - DES

2.2.1 History

In 1972, the NIST identified the need for a standard for encryption of unclassified, sensitive information. A cipher from IBM, based on an earlier algorithm Lucifer developed by Horst Feistel, was proposed. Although the cipher's short key length and the *S*-boxes were criticised, the algorithm was approved as a federal standard in 1976, under the name Data Encryption Standard (DES) and soon afterwards as the Federal Information Processing Standard (FIPS) PUB 46 [10]. Subsequent reaffirmation of the standard were published in 1983 (FIPS PUB 46-1), 1988 (FIPS PUB 46-2) and 1998 (FIPS PUB 46-3) also known as "triple DES". The most threatening theoretical attacks on DES were published in 1991, the differential cryptanalysis [27]; and in 1993, the linear cryptanalysis [90]. However, these attacks were only theoretical and it was the brute force attacks in 1998 and 1999 that demonstrated that DES can be attacked practically. These practical attacks also highlighted the need for a replacement algorithm. DES was replaced as a standard in 2002 with the Advanced Encryption Standard (AES) [9], but is, however, still in widespread use.

2.2.2 Algorithm

The DES algorithm uses 64-bit keys to encrypt and decrypt 64-bit blocks of data. The 56 bits of the key are generated randomly and used directly by the algorithm. The remaining 8 bits are used for error detection and are set to make the parity of each 8-bit byte of the key odd. The operations of encrypting and decrypting in DES are performed using the same key.

2.2.2.1 The overall structure

The algorithm's overall structure is shown in Figure 2.1. The algorithm consists of the following: the initial permutation (IP), 16 identical stages of processing called *rounds*, and the final permutation (FP), which is the inverse of the initial permutation. After the initial permutation, and before the main rounds, the resulting 64-bit block is divided into two 32-bit halves, left (L) and right (R), which are then processed alternately. This criss-crossing is known as the *Feistel structure*[1] and ensures that encryption and

[1] In a Feistel structure parts of the intermediate state are simply transposed unchanged to another position.

decryption are symmetric. Namely, the only difference between encryption and decryption is in the order in which the round keys are applied (during the decryption the round keys are applied in the reverse order). The advantage of the Feistel structure is that it simplifies the hardware implementation, as it removes the need for separate encryption and decryption algorithms.

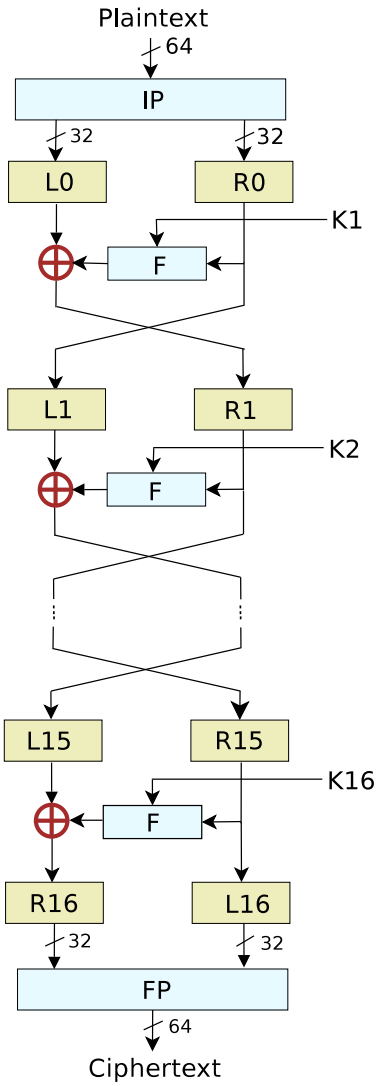


Figure 2.1: The Feistel structure of DES encryption algorithm.

The round function operates on two blocks: one consisting of the 32 bit right half of the intermediate result (R) and one consisting of 48 bits of the key K; and produces 32-bit output. The key used in each round represents the selection of 48 distinct bits from the original 64-bit key K, and is the product of the key schedule function (KS):

$$K_n = KS(K, n).$$

The round function updates the left and the right sides of the intermediate result according to the following rules:

$$\begin{aligned} L_n &= R_{n-1} \\ R_n &= L_{n-1} \oplus F(R_{n-1}, K_n) \end{aligned}$$

where $n = 1,\ldots,16$, and L_0 and R_0 are the left and the right half of the result of the initial permutation. Finally, the preoutput block $R_{16}L_{16}$ is subject to the final permutation, FP. The cipher's overall structure is also given in Algorithm 1.

Algorithm 1 DES encryption algorithm

INPUT: $PT(Plaintext), K(CipherKey)$

OUTPUT: $CT(Ciphertext)$

1: $L_0R_0 = InitialPermutation(PT)$
2: **for** $i = 1$ *to* 16 **do**
3: $\quad K_i = KS(K, i)$
4: $\quad L_i = R_{i-1}$
5: $\quad R_i = L_{i-1} \oplus F(R_{i-1}, K_i)$
6: **end for**
7: $CT = FinalPermutation(R_{16}L_{16})$

2.2.2.2 The round function

The round function (F) given in Figure 2.2, is defined as:

$$F(R_{i-1}, K_i) = P(S(E(R_{i-1}) \oplus K_i)).$$

The round function consists of four different stages:

***Expansion*:** in which the 32-bit half-block is expanded into 48 bits using the expansion permutation (E), in which some of the bits are duplicated. (The E table is given in Figure C.3 in Appendix C.)

***Key addition*:** in which the result of the expansion E is XORed with a *round key*. Sixteen 48-bit round keys (one for each round) are derived from the main key using the *key schedule*, described in Section 2.2.2.3.

***Substitution*:** in which the 48-bit block, result of the key addition, is divided into eight 6-bit portions that are subjected to the substitution boxes, S-boxes. The

transformation given by the S-boxes is a non-linear transformation, provided in the form of a look-up table, and represents the core of the security of DES. Without the S-boxes the cipher would be linear, and thus trivially breakable. Each of the 8 S-boxes replaces its 6 input bits with 4 output bits, as follows. Let S_k be one of the 8 selection boxes and b a 6-bit input. The first and the last bit of b represent, in base 2, a number i in the range 0 to 3. The middle 4 bits of the block b represent, in base 2, a number j in the range 0 to 15. The result of $S_k(b)$ is the 4-bit number given in row i and column j in the selection table S_k.

***Permutation*:** in which the 32-bit outputs from the S-boxes are subject to a fixed permutation P. This permutation is used to rearrange the outputs of the S-boxes in order to make the input bits to each of the S-boxes in the following rounds depend on the outputs of as many S-boxes as possible.

The alternation of substitution from the S-boxes, P-permutation of the bits and E-expansion provide the so-called ”confusion and diffusion”, a concept introduced by Claude Shannon [125], as a necessary condition for a secure and practical cipher.

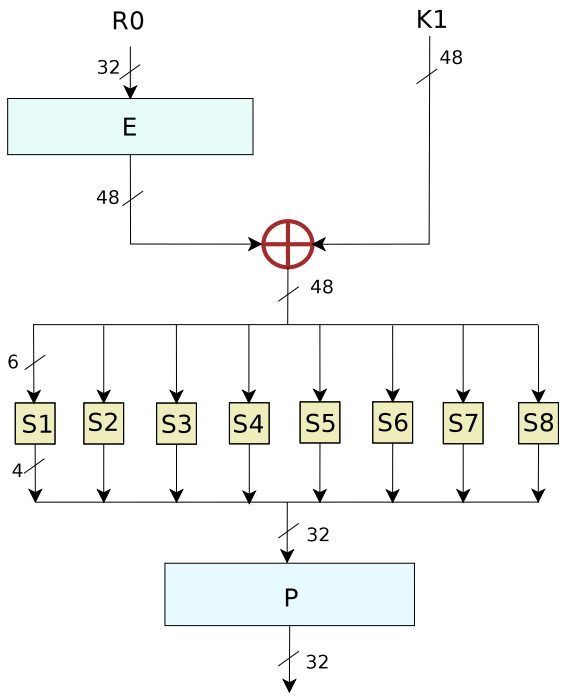


Figure 2.2: The DES round function.

2.2.2.3 Key schedule

The key schedule function (KS) is given in Figure 2.3. The function is defined by two *permuted choices*: $PC1$ and $PC2$. The two parts, C_0 and D_0, are defined according to the permuted choice $PC1$ (given in Figure C.4 in Appendix C). Permuted choice $PC1$ selects 56 bits of the 64 bits of the key, and splits the selection into two halves each containing 28 bits. In successive rounds, each half is rotated one or two bits to the left, depending on the round. Finally, the round key bits are chosen according to the permuted choice $PC2$, which selects 48 bits of the round key by selecting 24 bits from the left half (C) and 24 bits from the right half (D) (as shown in Figure C.5 in Appendix C).

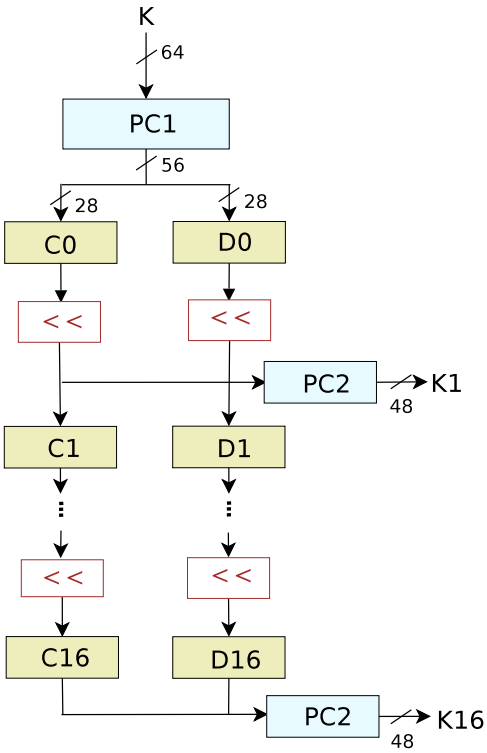


Figure 2.3: DES key selection function.

2.2.3 Cryptanalysis of DES

2.2.3.1 Exhaustive key search

The simplest method to break the DES cipher is to try to decrypt the given encrypted block with all possible keys. DES algorithm encrypts 64-bit blocks of data using 56-bit

secret keys, which means there are 2^{56} possible keys to be tried, making the average of 2^{55} trials. On a single PC, this would take hundreds of years to process.

In 1998, Cryptography Research, Advanced Wireless Technologies, and Electronic Frontier Foundation built a dedicated machine which demonstrated that exhaustive search for DES is feasible. This project was a part of the DES Key Search Project challenge, and developed purpose-built hardware and software to search 90 billion keys per second, being able to determine the key in 56 hours. Although this type of project may be possible only to well funded organisations, there are less expensive ways to crack the DES key. In January 1999, Distributed.Net broke a DES key in 23 hours, by using the idle times of the machines on the Internet donated by volunteers. More than 100,000 computers on the Internet received and computed part of the work, checking 250 billion keys per second.

2.2.3.2 Dictionary method and time-memory tradeoff

Although the exhaustive search is extremely time consuming, it is not as demanding in terms of memory requirements. Given a lot of memory, one can precompute all the possible keys, K, and the encrypted blocks, Y, corresponding to a given block of data, X, and store the pairs $\langle Y, K \rangle$. Given an encrypted block, Y', of the known block, X, with an unknown key, K', the right key could then be quickly found by searching this kind of dictionary.

In 1980, Hellman [63] proposed a time-memory tradeoff algorithm, which needs less time than the exhaustive search and less memory than the dictionary method.

2.2.3.3 Differential cryptanalysis

Biham and Shamir [27] in the late 1980s published a number of attacks against various block ciphers and hash functions, including DES, termed *differential cryptanalysis*. Differential cryptanalysis is a chosen plaintext attack which uses only the resulting ciphertexts. The attack uses a chosen ciphertext pair whose dedicated plaintexts have a particular difference. The two plaintexts do not have to be known to the attacker and can be chosen at random, but their difference has to satisfy a predefined condition. The differences in the plaintexts are used to assign probabilities to the possible keys and to locate the most probable key. The attacker selects the input difference for which the outputs difference occurs with high probability. In the case of DES, this difference is chosen to be a fixed XOR value of the two plaintexts.

In order to describe the attacks, recall that the round function (F) is defined as:

$$F(R_{i-1}, K_i) = P(S(E(R_{i-1}) \oplus K_i)).$$

Due to their linearity, the expansion function (E) and permutation (P) satisfy the following:

$$E(X) \oplus E(X^*) = E(X \oplus X^*)$$

$$P(X) \oplus P(X^*) = P(X \oplus X^*)$$

Considering that the S-boxes are non-linear, the knowledge of the difference of the input pair to the S-boxes does not guarantee the knowledge of the difference of the output pair. Usually several different outputs are possible. However, an important observation is that for any particular input XOR, not all the output XORs are possible. Furthermore, the possible ones do not appear uniformly, and some XORed values appear more frequently.

Important properties of the S-boxes are derived from the analysis of the tables that summarise the distribution of the input XORs and output XORs of all the possible input and output pairs. These tables are called the *pairs XOR distribution tables* of the S-boxes. In these tables each row corresponds to a particular input XOR and each column corresponds to a particular output XOR. The entries themselves count the number of possible pairs with such an input and such an output XOR. These tables are generated for all eight S-boxes. For a particular input XOR to an S-box, possible output XORs can also be determined.

The attack can be depicted with the following example, whose further details can be found in [27]. Assume that two plaintext outputs from the E transformation and the output from the first S-box are known. The XOR of two outputs from the E transformation is equal to the XOR of the two inputs to the S-box, and thus the input XOR for the first S-box can be determined. By consulting the XOR distribution table for the first S-box, it is possible to determine the number of possibilities for the input to the S-box, which also determines the number of possible keys. Next, the possibilities for the inputs and the corresponding keys can be determined, among which the right value of the key must occur. Using additional output pairs, additional candidates for the key can be obtained. Now the right key must occur among the possibilities for each chosen pair. This narrows down the number of possibilities for the key. Using a pair with a different input XOR helps determine the right key from the reduced set.

The differential cryptanalysis is, however, a theoretical attack and is infeasible to mount in practice. The main results of the findings of Biham and Shamir can be sum-

marised as follows: DES reduced to six rounds can be broken using 240 ciphertexts; DES reduced to eight rounds can be broken using 15000 ciphertexts chosen from a pool of 50000 candidate ciphertexts; DES reduced to up to 15 rounds can be broken faster than exhaustive search, but DES with 16 rounds still requires 2^{58} steps [27].

2.2.3.4 Linear cryptanalysis

Linear cryptanalysis is another theoretical attack on DES that was discovered by Matsui [90] in 1993. Linear cryptanalysis is a known-plaintext attack, although in certain cases can be applied as an only-ciphertexts attack. This method consists of obtaining a linear approximate expression of a given cryptographic algorithm. For that purpose, it constructs a statistical linear path between input and output bits for each S-box. This path is then extended to the entire algorithm reaching the linear approximate expression without any intermediate values.

The purpose of linear cryptanalysis is to find the following linear expression:

$$P[i_1, i_2, \ldots, i_a] \oplus C[j_1, j_2, \ldots, j_b] = K[k_1, k_2, \ldots, k_c] \tag{2.1}$$

where $A[a_1, a_2, \ldots, a_t]$ denotes $A[a_1] \oplus A[a_2] \oplus \cdots \oplus A[a_t]$; $A[a_i]$ is the i-th bit of A; i_1, i_2, $\ldots$, i_a, j_1,j_2, $\ldots$, j_b, k_1, k_2, $\ldots$, k_c denote fixed bit locations, and Equation 2.1 holds with probability $p \neq \frac{1}{2}$ for randomly given plaintext P and the corresponding ciphertext C. The magnitude of $|p - \frac{1}{2}|$ represents the effectiveness of Equation 2.1. Once the effective linear expression is obtained, one key bit $K[k_1, k_2, \ldots, k_c]$ can be determined following the algorithm based on the maximum likelihood method:

Step 1 – Let T be a number of plaintexts for which the left-hand side of Equation 2.1 is equal to zero.

Step 2 – If $T > N/2$, where N denotes the number of plaintexts, then guess

$$K[k_1, k_2, \ldots, k_c] = 0,\ if\ p > 1/2 \quad or \quad K[k_1, k_2, \ldots, k_c] = 1,\ if\ p < 1/2,$$

else guess

$$K[k_1, k_2, \ldots, k_c] = 1,\ if\ p > 1/2 \quad or \quad K[k_1, k_2, \ldots, k_c] = 0,\ if\ p < 1/2.$$

To solve the problem, Matsui first studied the linear approximation of S-boxes. The taken approach was to investigate the probability that a value of an input bit coincides with a value of an output bit. Next, the effective approximation of the cipher is obtained.

For a practical known-plaintext attack on n-round DES cipher, the best expression of $(n-1)$-round DES cipher is used. This is equivalent to regarding the final round as having been deciphered using K_n. A term of F function is accepted in the linear expression, and consequently the following form of expression is obtained:

$$P[i_1, i_2, \ldots, i_a] \oplus C[j_1, j_2, \ldots, j_b] \oplus F_n(R_{n-1}, K_n)[l_1, l_2, \ldots, l_d] = K[k_1, k_2, \ldots, k_c] \quad (2.2)$$

If an incorrect candidate is substituted for K_n in Equation 2.2, the effectiveness of this equation decreases. Based on this fact a maximum likelihood method to deduce K_n and $K[k_1, k_2, \ldots, k_c]$ is applied. Next, the linear approximation of the S-boxes and the F function is extended to the entire algorithm. Detailed examples of this extension to the 3-, 7- and 8-round DES are given in [90].

Although this attack is a theoretical one, it is the most powerful attack on DES that is faster than the brute force attack. The main results presented in [90] can be summarised as follows: DES reduced to 8 rounds can be broken with 2^{21} known plaintexts; DES reduced to 12 rounds can be broken with 2^{33} known plaintexts and the full 16 round DES can be broken with 2^{47} known plaintexts.

Matsui noticed that if the plaintexts are not random, there might even be a linear approximate expression that does not have a plaintext bit in it. This suggests that this method finally leads to an only-ciphertext attack. If the attack is regarded as only-ciphertext attack then the results of [90] can be summarised as follows: if plaintexts consists of natural English sentences, DES restricted to eight rounds can be broken with 2^{29} ciphertexts; if the plaintexts are random, DES restricted to eight rounds can be broken with 2^{37} ciphertexts only. The author also illustrated the situation in which 16-round DES is breakable faster than an exhaustive search for 56 key bits using the only-ciphertext attack.

2.3 Advanced encryption standard - AES

2.3.1 History

In 1997 NIST announced the *Advanced Encryption Standard* (AES) development effort and made a formal call for algorithms. The call stated that the AES would specify an “unclassified, publicly disclosed encryption algorithm(s), available royalty-free, worldwide. In addition, the algorithm(s) would implement symmetric key cryptography as a block cipher and (at a minimum) support a block size of 128-bits and key sizes of 128, 192, and 256 bits” [6].

In 1998, fifteen AES candidates were announced at the First AES Candidate Conference [2]. The Second AES Candidate Conference [4] was held in 1999. The results and comments of this meeting were used to reduce the number of candidates to five algorithms: MARS, RC6, Rijndael, Serpent, and Twofish. On October 2, 2000, NIST announced that it had selected Rijndael (a portmanteau name composed of the names of the inventors - two Belgian cryptographers - Joan Daemen and Vincent Rijnmen), a refinement of an earlier design Square [7], as the new standard. Rijndael was pronounced as a new standard (AES) on November 26, 2001 as FIPS PUB 197 [9], and effectively became a new standard on May 26, 2002.

2.3.2 Algorithm

AES Rijndael [9] is a symmetric block cipher that processes block lengths of 128 bits and key length that can be independently specified to 128, 192 and 256 bits. Actually, AES is not precisely Rijndael [45], as Rijndael supports a larger range of block and key sizes. Namely, the key and block sizes in Rijndael can be any multiple of 32 bits, with a minimum of 128 bits and a maximum of 256 bits.

2.3.2.1 The overall structure

Unlike most ciphers, DES for instance, Rijndael does not have a Feistel structure, but it is a so-called *substitution-permutation network.* A substitution-permutation network is a series of linked mathematical operations used in block ciphers that consist of *S*-boxes and *P*-boxes that transform blocks of input bits into output bits. AES operates on a 4×4 array of bytes, termed the `State`. Each round of transformation is composed of three different layers, which are designed to provide resistance against differential and linear cryptanalysis [45]. These layers are:

***Linear mixing layer*:** which guarantees a high degree of diffusion over multiple rounds.

***Non-linear layer*:** which consists of parallel application of substitution tables (*S*-boxes) that have optimum worst-case non-linearity properties.

***Key addition layer*:** which involves a simple XOR of the round key to the intermediate cipher result, called the `State`.

For encryption each round transformation is composed of four different stages:

1. `BytesSub` – a non-linear substitution step where each byte of the `State` is replaced with another according to the lookup table.
2. `ShiftRows` – a transposition step where each row of the `State` is shifted cyclically a certain number of steps.
3. `MixColumns` – a mixing operation which operates on the column of the `State`, combining the four bytes in each column using a linear transformation.
4. `AddRoundKey` – each byte of the `State` is combined with the `RoundKey`, which is derived from the `CipherKey` using a key schedule.

In order to make the decryption process symmetrical, the final round omits the `MixColumns` stage. Finally, the cipher consists of the following steps (also given in Algorithm 2):

- Initial round key addition;
- $N_r - 1$ rounds, where N_r represents the total number of rounds and depends on the key size (number of rounds for the original Rijndael is given in Figure C.1 in Appendix C); N_b in Algorithm 2 represents the block length divided by 32. The round transformation is given in Figure 2.4.
- Final round.

Algorithm 2 Rijndael encryption algorithm

INPUT: $State(Plaintext), CipherKey$

OUTPUT: $State(Ciphertext)$

1: $KeyExpansion(CipherKey, ExpandedKey)$;
2: $AddRoundKey(State, ExpandedKey)$;
3: **for** $i = 1$ *to* Nr **do**
4: $\quad Round(State, ExpandedKey + Nb * i)$;
5: **end for**
6: $FinalRound(State, ExpandedKey + Nb * Nr)$;

The steps of the round transformation can be combined together in a single set of table lookups, allowing faster implementation on 32-bit processors and considerable parallelism in the round transformation. As a result the number of operations used in the cipher can be reduced to two: table lookups and XORs [45].

2.3.2.2 The `ByteSub` transformation

The `ByteSub` transformation is a non-linear byte substitution, operating on each of the `State` bytes independently. The substitution table (S-box) is invertible and is constructed by composing the following two transformations:

1. Taking the multiplicative inverse in $GF(2^8)$.
2. Applying affine transformation over $GF(2^8)$:

$$b(x) = (x^7 + x^6 + x^2 + x) + (x^7 + x^6 + x^5 + x^4 + 1) \cdot a(x) \ mod \ (x^8 + 1).$$

The inverse of `ByteSub` is the byte substitution with the inverse table applied, which is obtained by the inverse of the affine transformation followed by taking the multiplicative inverse in $GF(2^8)$.

2.3.2.3 The `ShiftRow` transformation

In the `ShiftRow` transformation each row of the `State` is cyclically shifted over different offsets: row 0 is not shifted, row 1 is shifted by $C_1 = 1$ bytes, row 2 by $C_2 = 2$ bytes and row 3 by $C_3 = 3$ bytes. (In the original Rijndael, the values of C_1, C_2 and C_3 depend on the block length as shown in Figure C.2 in Appendix C.)

The inverse of `ShiftRow` is a cyclic shift of the three bottom rows by $4-1=3$, $4-2=2$, and $4-3=1$ bytes, respectively. (In the original Rijndael, the values of offsets for the inverse operations are $N_b - C_1$, $N_b - C_2$, $N_b - C_3$, N_b represents number of columns in the block and is equal to the block length divided by 32.)

2.3.2.4 The `MixColumn` transformation

In the `MixColumn` transformation the columns of the `State` are considered as polynomials over $GF(2^8)$, and multiplied, modulo x^4+1, with a fixed polynomial $c(x)$, given by:

$$c(x) =' 03' x^3 +' 01' x^2 +' 01' x +' 02'$$

The inverse transformation is similar to `MixColumn` transformation, except the polynomial used in the inverse operation is:

$$d(x) =' 0B' x^3 +' 0D' x^2 +' 09' x +' 0E'$$

and satisfies $c(x) \cdot d(x) =' 01'$.

After two rounds of Rijndael, `ShiftRow` and `MixColumn` transformations provide *full diffusion*, in the sense that every bit in the `State` depends on all state bits from two previous rounds.

2.3.2.5 The AddRoundKey transformation

In the `AddRoundKey` transformation the `RoundKey` is simply XORed with the `State`. The `RoundKey` is derived from the `CipherKey` by means of a key schedule. The length of `RoundKey` is equal to the size of the `State`. The total length of all round keys is equal to $4 \cdot (N_r + 1)$, where N_r represents the number of rounds. The `CipherKey` is first expanded into the `ExpandedKey` and each `RoundKey` is derived from the `ExpandedKey` in the following way: the first 4 words of the `ExpandedKey` represent the first `RoundKey`, and each further block of 4 words represent the second and subsequent keys.

2.3.3 Cryptanalysis of AES

The most common way to attack block ciphers is to try various attacks on versions of the cipher with a reduced number of rounds. AES has 10 rounds for 128-bit keys, 12 rounds for 192-bit keys, and 14 rounds for 256-bit keys. According to [1], the best known attacks are on 6 rounds for 128-bit keys, 6 rounds for 192-bit keys, and 7 rounds for 256-bit keys.

2.3.3.1 The XSL attack

Courtois and Pieprzyk [43] in 2002 published a theoretical attack against Rijndael and Serpent [5]. The attack expresses the entire algorithm as multivariate quadratic polynomials, and uses an innovative technique to treat the terms of those polynomials as individual variables. It relies on first analysing the internals of a cipher and deriving a system of quadratic simultaneous equations. These systems of equations are very large, for example 8000 equations with 1600 variables for 128-bit AES. The variables represent not just the plaintext, ciphertext and key bits, but also various intermediate values within the algorithm. In the XSL attack a specialised algorithm, termed as eXtended Sparse Linearization (XSL), is applied to solve these equations and recover the key. In this attack, unlike other forms of cryptanalysis such as differential and linear cryptanalysis, only one or two known plaintexts are required.

However, the analysis given in [43] in not universally accepted. The complicated technical details of the paper raised suspicions about the accuracy of the underlying

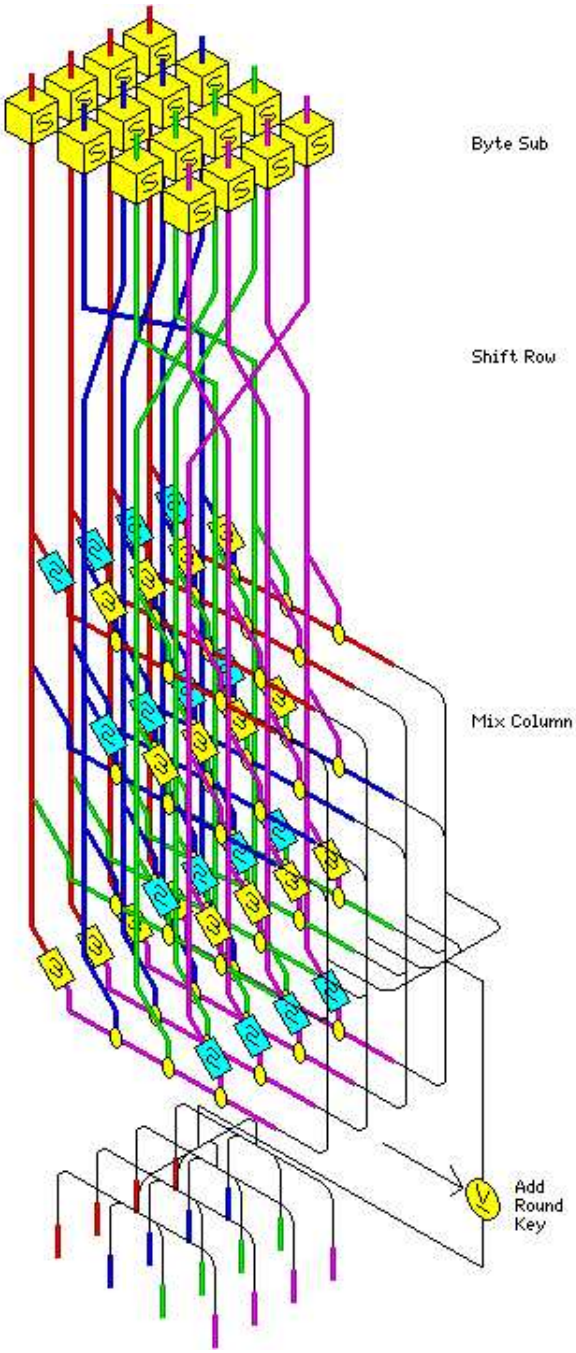


Figure 2.4: Rijndael round transformation. Obtained from http://home.ecn.ab.ca/∼jsavard/crypto/images/rijnov.gif

mathematics. Furthermore, several cryptography experts have found problems in the underlying mathematics of the proposed attack, suggesting that the authors had made a mistake in their calculations. These findings have led to the general belief that this attack is speculative and impractical.

2.4 Summary

This chapter provided an overview of two important cryptographic algorithms, DES and AES, the former standard and the new standard. It also presented the most well known cryptanalytic techniques used in theoretical and practical attacks on these two cryptographic standards. The experimental security investigations presented in Chapter 6 are based on investigating the security against differential power analysis of these two important cryptographic algorithms when run on different configurations of the network-based architecture.

In the next chapter an overview of new and very powerful cryptanalysis techniques that, unlike the attacks reviewed in this chapter, do not depend on the mathematical characteristics of the cryptographic algorithm, but on the implementation and physical characteristics of the device the algorithm is implemented on is given. This type of analysis is known as side-channel analysis. Countermeasures proposed to thwart these attacks are also reviewed in the next chapter.

Chapter 3

Side-channel Analysis

3.1 Introduction

Cryptographic operations are physical processes in which data is represented by physical quantities in physical structures. These are then stored, sensed and combined by the elementary logic devices (gates). At any point in the evolution of technology, the smallest logic device must have a definite physical extent, require a certain amount of time to perform its function and dissipate switching energy when transiting from one state to another [93]. A corollary of the second law of thermodynamics states that in order to introduce direction into transition between states, energy must be lost irreversibly. A system that does not dissipate energy cannot make a transition and therefore cannot compute [93]. It has been shown that this energy can be correlated with the operations performed and the data that is being processed.

While operating, electronic devices interact and influence the environment. Besides consuming and emitting power, these devices emit electromagnetic radiation and react to temperature changes. This information leakage is intrinsic to the physical implementation of the device, and is characterised as the *side-channel*. If observed and recorded, information leaked into side-channels can be used to recover compromising information (secret keys for example) about the device in question. This is particularly true for cryptographic devices for which the secrecy of the key is imperative (Kerchkoff principle[1]). This type of analysis defines the branch of cryptanalysis known as *side-channel analysis*. According to the type of information used, side-channel analysis attacks can be classified into three main categories:

[1]Kerchkoff principle: The security of cryptographic algorithms must be based on the secrecy of the key not on the secrecy of the algorithm.

- Timing analysis
- Power analysis
- Electromagnetic emission analysis

Considering the rapid development of electronic business and different kinds of digital communication systems the electronics industry as well as the academic community were alarmed by the discovery of side-channel attacks. It became crucial to protect cryptographic systems against these new and powerful types of attacks. A number of countermeasures were proposed for each of these attacks. However, according to the research currently conducted in this area, it is hard to come up with a general countermeasure that guarantees that the cryptosystem is secure against all side-channel attacks. The current definition of side-channel security says that a cryptosystem is secure if it is secure against all known side-channel attacks. Although this does not guarantee the security against attacks that are yet to be discovered, this notion of security is generally accepted. Some side-channel attacks can be completely prevented by using clever implementations of cryptographic algorithms. To prevent against the most powerful side-channel attacks, power analysis, most practical solutions rely on increasing the complexity of the attack. This increase in complexity is equivalent to complicating the statistical analysis and increasing the number of necessary readings of the side-channel data to the extent that the attack is not feasible or is too expensive to perform. The complexity of side-channel attacks can be increased on two levels: by introducing *software (algorithmic)* and/or *hardware (physical)* countermeasures. The general strategy to increase the complexity of side-channel attacks involves balancing and randomising major computations which involve the secret key.

3.2 Timing analysis

3.2.1 Introduction

When designing a commercial cryptographic scheme cryptographers have always been concerned with the execution time of their implementations. The amount of time needed to encrypt or decrypt a message or produce a digital signature is often used as a benchmark when comparing different cryptographic schemes. The fastest scheme, under the same conditions and with the same parameters, is considered to be the most efficient and, therefore, the most appealing to the demands of the market.

The actual timing of a cryptographic function does not only depend on the operations performed, but also on the parameters passed to it: both the secret key and the plaintext (ciphertext) data. Cryptosystems often take slightly different times to process different input parameters. The timing variations are due to different performance optimisations that are used to bypass unnecessary operations, branching and conditional statements. A good portion of these variations are due to processor instructions, such as multiplications and divisions, that run in variable times [76].

In 1995, Paul Kocher from Cryptography Research in San Francisco [76], demonstrated that the timing variations can be used to deduct secret exponents used in systems such as RSA [3], DSS [8], Diffie-Hellman [48], and others. He outlined a simple and inexpensive attack which enables an attacker to discover the fixed (secret) exponents used in these cryptosystems. The attack exploits certain engineering aspects involved in the implementation of cryptosystems which succeeded even against cryptosystems that have remained impervious to sophisticated cryptanalytic techniques, such as differential [27] and linear cryptanalysis [90]. With the growing popularity of electronic commerce this discovery drew the attention of both industry and academia. The cryptographic community became aware that some widely used standards (such as SSL) are vulnerable to this new attack. This led to the discovery of *timing attacks* and opened a completely separate and new area of cryptanalysis, known as *side-channel analysis*. Kocher's discovery even made it to the front page of New York Times [86].

3.2.2 Attack details

Private-key operations in RSA or Diffie-Hellman consist of performing modular exponentiations of the form: $S = M^d \ mod \ N$. As suggested in [117], this operation can be implemented using a *repeated square-and-multiply algorithm* given in Algorithm 3. In this algorithm, S can be thought of as a digital signature, M is a message, N is public, and d is the private (secret) exponent which can be represented using at most n bits, where n is the length of S. Kocher noticed that the execution path of the algorithm depends on the value of the private exponent d. Namely, in a loop iteration, if the corresponding bit of d is equal to 1, then both the modular squaring and multiplication are performed (lines 3 and 5, respectively); otherwise, if the bit is equal to 0, then only the modular squaring is performed. Therefore, the number of operations that are performed and the overall execution time depend on the value of the private exponent. If an attacker could observe and compare the execution times of several loop iterations

(Figure 3.1) then he would be able to deduce the values of bits of the private exponent d for each of the iterations [76].

Algorithm 3 Repeated left-to-right square-and-multiply algorithm for modular exponentiation.

INPUT: $M, N, d = (d_{n-1}, \ldots, d_1, d_0)_2$

OUTPUT: $S = M^d \ mod \ N$

1: $S \leftarrow 1$
2: **for** $j = n-1 \ to \ 0$ **do**
3: $\quad S \leftarrow S^2 \ mod \ N$
4: $\quad$ **if** $d_j = 1$ **then**
5: $\quad\quad S \leftarrow S \cdot M \ mod \ N$
6: $\quad$ **end if**
7: **end for**

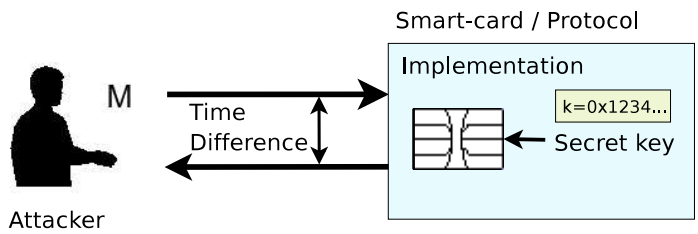


Figure 3.1: The timing analysis principle [94].

Kocher [76] explained how the overall running time of the algorithm can be used to deduce the bits of the private exponent d. The timing attack allows someone who knows bits $0 \ldots k-1$ of the private exponent to discover the bit k. The attack proceeds as follows. By knowing the first k bits, the attacker can compute the first k iterations of the for-loop and find the value of S after that iteration. In the next iteration, the value of the unknown bit of d will be used. The squaring in line 3 will be performed regardless of the value of the bit, but the multiplication in line 5 is performed only if the value of the unknown bit is equal to 1. The difference in timing of this iteration when zero and one are the bits in question, enables the attacker to determine the value of the unknown bit. Starting from $k = 0$ and following this fashion, all bits of the secret exponent can be discovered.

An interesting property of the timing attack, observed by Kocher [76], is its *error-detection* property. Namely, if at any point the k-th bit was guessed incorrectly, then

the values of S computed in consecutive iterations will be essentially random and the timings following the error will not be reflected in the overall exponentiation time. Therefore, after the error occurred, no more meaningful correlations can be observed. This property can be used for error correction [76]. Each timing measurement is equal to $T = e + \sum_{i=0}^{n-1} t_i$, where times t_i are required for multiplication and squaring for each bit d_i, and time e includes measurement error and loop overhead. Given a guess of the k-th bit, d_k, the attacker can find $\sum_{i=0}^{k-1} t_i$. If d_k was correct, subtracting from T yields $e + \sum_{i=k}^{n-1} t_i$. The relative independence of modular multiplications from each other and from the measurement error, yields the variance of $e + \sum_{i=k}^{n-1} t_i$ to be $Var(e) + (n-k)Var(t)$. If only $l < k$ bits were guessed correctly, then the expected variance should be $Var(e) + (n-k+2l)Var(t)$. Therefore, iterations done with a correctly guessed key decrease the variance by $Var(t)$, while the iterations following the incorrectly guessed key increase the variance by $Var(t)$. This is an easy to compute test which provides a good way to identify if the bit was guessed correctly.

3.2.2.1 Attacks on other systems

Almost any implementation that runs in variable amounts of time could be vulnerable to timing analysis [104]. Most public key systems and signature schemes, such as ECC, RSA and ElGamal, use algebraic operations that often run in variable times. Block ciphers, such as IDEA and AES Rijndael, are also vulnerable to timing attacks because they use multiplications [72, 79]. The bit rotations, used in ciphers such as RC5 and DES, when implemented using shift and conditional "wrap around" can leak Hamming weights of the operands. (Hamming weight represents the number of ones in the binary representation of the data.) For example, in the software implementations of DES, the 28 bits of C and D values in the DES key schedule (see Section 2.2 for the description of DES) are often rotated using a conditional which tests whether the bit that must be wrapped around is equal to 1. The additional time required to "wrap around" non-zero bits could introduce slight timing variations, which could reveal the Hamming weight of the key.

Naive implementations of AES Rijndael [9] are also at risk, as described by Koeune and Quisquater [79]. The AES encryption consists of the initial round key addition followed by a number of round transformations (see Section 2.3 for the description of AES). The different transformations during each round operate on an array of bytes, called the `State`. This attack focused on a particular round transformation, the `MixColumn` transformation. In the `MixColumn` transformation, the columns of the

`State` are considered as polynomials over $GF(2^8)$, and multiplied, modulo x^4+1, with a fixed polynomial $c(x) =' 03'x^3 +' 01'x^2 +' 01'x +' 02'$. This operation can be implemented very efficiently, since $'03' =' 02' +' 01'$, the only multiplications that will actually have to be performed are those by $'02'$. In addition, the multiplication in $GF(2^8)$ can be implemented very efficiently by following two simple steps: (1) shift the byte one position left, (2) if a carry occurs, XOR the result with $'1B'$ [9]. Therefore, in careless implementations, this operation could show timing variations, as it can take longer when the carry actually occurs.

Timing attacks have been successfully performed against a number of cryptographic functions, but also against some Internet protocols such as SSL [32].

3.2.3 Countermeasures

Naturally, there is a question of protecting cryptosystems against timing attacks. Kocher noticed that the most obvious method would be to make sure all operations run in constant time. Doing this at the implementation level is often difficult in view of all the possible factors that can introduce variations in timing (such as compiler optimisations, different platforms, RAM cache hits and instruction timings). Even if this is achieved, for example by withholding the result of an operation until a specified amount of time is expired, other information, such as power consumption or CPU usage, can reveal sensitive information [76]. In addition, performance of such systems would be considerably degraded as all operations will take the same amount of time as the slowest one, while performance optimisations are not allowed for obvious reasons. This would imply a severe performance drawback, especially for asymmetric cryptosystems, since this constant time would be that of the slowest possible case.

Daemen and Rijmen [46] similarly suggested that cryptographic implementations can be protected against timing attacks by ensuring that the cipher execution time is independent of the value of the key, by inserting NOP operations in the shortest path of the conditional statement until all paths take the same time. However, they also noticed that this solution might be vulnerable to power analysis (described in Section 3.3).

Even ensuring that the same set of operations is performed in each iteration of the algorithm (an example of such an implementation for modular exponentiation is given in Algorithm 4), does not make the execution time constant. This is a general misconception about the timing attack. The timing attack does not only discover the path of execution, but also the operands that are used [104]. Multiplication with zero

would take different time when compared to multiplication with one. If, however, in the case of modular exponentiation, squaring and multiplication are implemented to run in constant time, then the modular exponentiation would only be correlated with the Hamming weights of the secret exponent, which in some cases can reveal the secret exponent [104]. For example, Montgomery multiplication runs in almost constant time but there are small variations due to conditional subtraction which implies that Montgomery multiplication is vulnerable to timing attacks [47]. Both squaring and multiplication operations in the square-and-multiply algorithm could be performed using Montgomery multiplication. If the squaring part is attacked, then even keys of length 512 can be efficiently discovered. The timing attack can also be applied to RSA implementations with the Chinese Reminder Theorem as shown in [119].

Algorithm 4 Repeated square-and-multiply algorithm for modular exponentiation, still vulnerable to timing attacks.

INPUT: $M, N, d = (d_{n-1}, \ldots, d_1, d_0)_2$

OUTPUT: $S = M^d \ mod \ N$

1: $S \leftarrow 1$
2: **for** $j = n-1 \ldots 0$ **do**
3: $\quad S \leftarrow S^2 \ mod \ N$
4: $\quad T \leftarrow S \cdot M \ mod \ N$
5: $\quad$ **if** $d_j = 1$ **then**
6: $\quad\quad S \leftarrow T$
7: $\quad$ **end if**
8: **end for**

Another suggested approach to prevent timing attacks is to add random delays to execution and make timing measurements imprecise. However, this can be overcome by increasing the number of samples so that the added noise is filtered out. The number of samples required increases roughly as the square of the timing noise [76].

Kocher [76] proposes using blinding techniques by which the attacker would be prevented from knowing the input to the modular exponentiation. Prior to computing the modular exponentiation, pair (v_i, v_f) is chosen, such that $v_f^{-1} = v_i^d \ mod \ N$, where this relation might be different for different cryptosystems. For example, in the case of RSA, it is faster to choose random v_f relatively prime to N and then compute $v_i = (v_f^{-1})^e \ mod \ N$, where e is the private exponent. Before the modular exponentiation, the message should be multiplied by $v_i \ mod \ N$ and the result is subsequently

corrected by multiplying it with $v_f \ mod \ N$. Pairs (v_i, v_f) should not be reused, since they themselves could be subjected to timing analysis, compromising the secret exponent. On the other hand, calculating inverses is expensive, so it is impractical to generate a new pair for each exponentiation. Moreover, the inverse operation itself can be subjected to timing analysis. For those reasons it was suggested that v_i and v_f are updated before each modular multiplication by calculating $v_i = v_i^2 \ mod \ N$ and $v_f = v_f^2 \ mod \ N$. In this way, the blinding pair is not reused and the total performance cost is kept small. This countermeasure makes the internal computations impossible to simulate by the attacker, thereby preventing the exploitation of the knowledge of the running times. Although it does not guarantee elimination of all possible timing attacks, this type of countermeasures is nonetheless efficient [76]. In addition, blinding techniques have also been proven efficient against other types of side-channel attacks, as described in Section 3.5.7.

In summary, in order to defeat the timing attack, implementors should prevent an attacker from knowing the inputs to vulnerable operations. For example, in the square-and-multiply algorithm, if the attacker does not know the base of the modular operation, timing information is not useful. Blinding techniques proposed by Kocher [76] have been successful in preventing timing attacks, but the suitability of blinding depends entirely on the details of the cryptosystem. However, the majority of public key cryptosystems have the required algebraic structure for applying this countermeasure.

3.3 Power analysis

3.3.1 Introduction

Power analysis attacks were discovered by Kocher, Jaffe and Jun [78] in 1998. One proposed way to counteract timing attacks was to introduce “dummy” computations, such as empty loops, in the execution of the cryptographic algorithm. Kocher et al. noticed that this might be insufficient defence, as the power consumption of “dummy” computations is different from the power consumption of meaningful ones. They have spent several months exploring this idea, and finally, by using relatively inexpensive equipment, managed to discover secret keys from a number of smart-cards. They claimed that for some devices, a power trace (where a *trace* is a set of power consumption measurements taken across the cryptographic operation) of a single cryptographic operation can reveal the value of the secret key. They also claimed that by examining

as few as 1000 power traces and applying statistical analysis on the obtained data (Figure 3.2), they could break any smart-card on the market [78]. This drew the attention of both the smart-card vendors and the cryptographic community, and yet again featured in the New York Times [134].

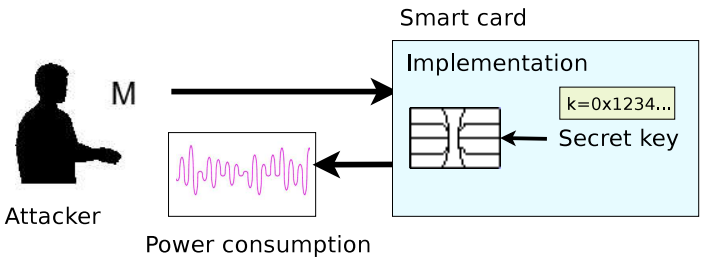


Figure 3.2: The power analysis principle [94].

3.3.2 Power dissipation

Most modern cryptographic devices are implemented using Complementary Metal Oxide Semiconductor (CMOS) technology. The main characteristic of this technology can be demonstrated with inverters or NOT gates (Figure 3.3). The inverter has two transistors that act as voltage controlled switches. When the inverter input is high, the top switch opens and the bottom closes. This grounds the inverters output and it goes low. On the other hand, when the input voltage is low, the top switch closes, and the bottom switch opens setting the output to high.

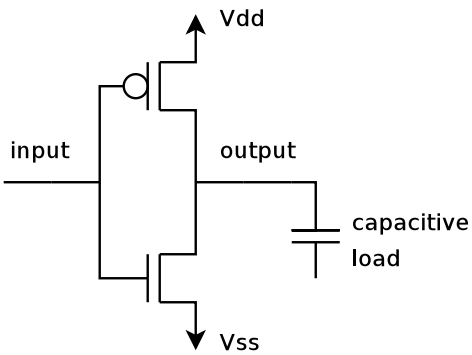


Figure 3.3: CMOS inverter.

Power dissipation in most CMOS circuits can be divided into three parts [135]: (1) static dissipation, (2) dynamic dissipation and (3) short-circuit dissipation.

Static dissipation (P_s): is due to the leakage of current drawn continuously from the power supply, and is equal to:

$$P_s = I_{leak} \cdot V_{dd}$$

where I_{leak} is the leakage current and V_{dd} is the supply voltage.

Dynamic dissipation (P_d): is due to the current that is required to charge and discharge the capacitive load, and is the dominant source of power dissipation in current CMOS technologies [135]. Dynamic power dissipation can be seen as:

$$P_d = f \cdot C_l \cdot V_{dd}^2 \cdot A_c$$

where A_c is the circuit activity, f is frequency of switching, C_l is circuit capacitance and V_{dd} is power supply voltage.

Short-circuit dissipation (P_{sc}): is due to the short current flowing from V_{dd} to V_{ss}. This occurs during the short period of time in the transition from 0 to 1 or, alternatively, from 1 to 0, during which both transistors are on, and is given by:

$$P_{sc} = I_{mean} \cdot V_{dd}$$

where I_{mean} is the mean current and V_{dd} is the supply voltage.

The total power dissipation can be obtained from the sum of the three dissipation components:

$$P_{total} = P_s + P_d + P_{sc}$$

However, the dynamic power dissipation is the most dominant in this formula [135, 136], which reduces the total dissipation estimate to:

$$P_{total} \approx P_d = f \cdot C_l \cdot V_{dd}^2 \cdot A_c \tag{3.1}$$

Equation 3.1 shows that for given voltage supply V_{dd} and capacitive load C_l power dissipation of the circuit is proportional to the circuit activity A_c. In other words, the more capacitance that is switched, the more power is dissipated. This is the key characteristic of CMOS technology that has inspired the attackers of cryptographic systems to develop power-based side-channel attacks.

The source of current for most devices is supplied by the constant voltage, and the power dissipation of these devices is proportional to the flow of current through them

($P = I \cdot V$, where P is the power). It is now clear how power consumption/dissipation can be correlated to the calculations performed in the cryptographic device.

The basic assumption in power analysis is that the device dissipates more power when processing a 1 in comparison to the power that is dissipated when processing a 0. For example, the multiplier would do more work when the multiplier bit is 1 than it would do when the bit is 0. The power dissipation is useful to the attacker because it is correlated with the operations performed in the device.

Power analysis attacks are based on measuring the power that is consumed by the devices. Kocher et al. [78] in their experiments measured the circuit's power consumption by inserting a small (50 Ω) resistor in series with the power or ground input. The voltage difference across the resistor divided by its resistance yielded the current. They sampled voltage differences at a high rate and then processed the data using statistical methods. Depending on the way in which the obtained data is seen and processed, two general types of power analysis can be distinguished:

1. Simple power analysis (SPA)

2. Differential power analysis (DPA).

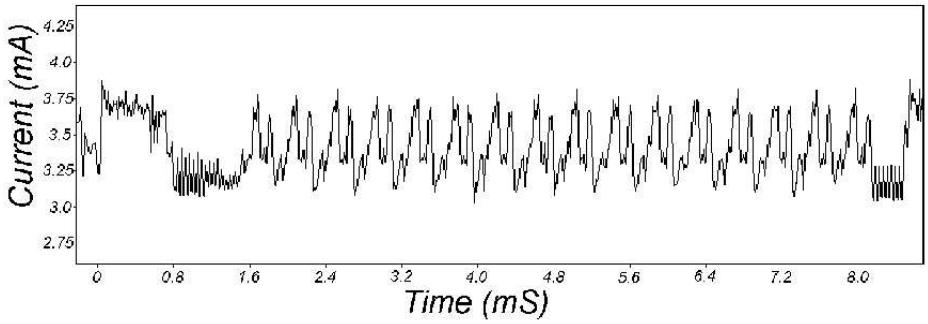
3.4 Simple power analysis

3.4.1 Attack details

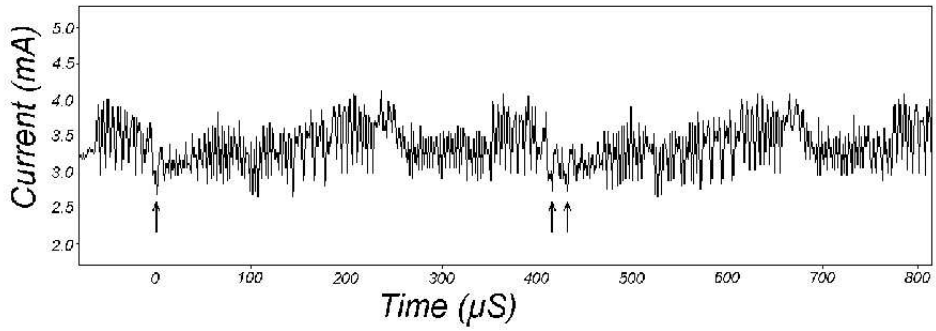
Simple power analysis (SPA) is a cryptanalytic technique whereby information about the operation performed in the device, or the operands manipulated in the operation, can be directly interpreted from a single power trace. Often this single trace is replaced with the average of a number of traces in order to reduce the measurement noise. The success of this approach and the techniques used in the attack depends on the implementation of the cryptographic algorithm and the operations used in it.

The SPA attack was first performed by Kocher, Jaffe and Jun [78] from Cryptography Research in San Francisco. In the power traces obtained from the power consumption measurements taken across a cryptographic operation of DES, 16 rounds of DES were clearly visible (Figure 3.4(a)). A detailed analysis of the traces, revealed the number of rotations performed on C and D registers (see Section 2.2 for the description of DES) that are used during the key schedule. Figure 3.4(b) shows that in round 2 (left arrow) registers C and D were rotated once and in round 3 (two right arrows) they were

rotated twice. Small variations between the rounds turned out to be SPA weaknesses caused by conditional jumps based on key bits and computational intermediates. An even closer look, enabled Kocher et al. to distinguish between microprocessor instructions.



(a) Power trace of the entire DES encryption.



(b) Power trace of DES rounds 2 and 3.

Figure 3.4: SPA attack on DES [78].

A SPA attack can reveal the sequence of operations executed and, therefore, can be used to break cryptographic implementations in which the execution path depends on the data being processed, more specifically the secret data. For example, some points for the attack, as Kocher et al. [78] suggested, could be:

DES key schedule: The DES key schedule involves rotating 28-bit key registers. Rotations are most commonly implemented by shifting one bit of one end and appending a 0 at the other. If the bit shifted off is a 1 then the appended 0 is flipped. This conditional operation may be detected in the power trace, as the resulting trace for bits 1 and 0 will have different characteristics, and can be distinguished.

DES permutation: DES implementations usually contain a number of bit permutations. Conditional branching in software can cause power consumption differences when bits zero and one are manipulated.

Comparisons: Memory comparisons or string comparisons can cause SPA and timing vulnerabilities, because they usually perform a conditional branch when a mismatch occurs.

Multipliers: Circuits that implement modular multiplications tend to take into account a great deal of information about the data being processed, and therefore leak a substantial amount of information about that data. The existence of weaknesses depends on the multiplier implementation but weaknesses are often correlated to operand values and Hamming weights.

Exponentiators: Modular exponentiation functions are usually implemented to scan across the exponent from left to right, performing a squaring operation in every iteration and an additional multiplication for each value 1 bit of the exponent. If the squaring and multiplication operations have different power signatures, exponentiation operation can be compromised. This analysis is similar to that of the timing attack. Modular operations that operate on two of more bits at the time may have more complex leakage characteristics.

Given the architectural details, SPA can more easily interpret the power consumption curve. Generally, any implementation where the path of execution depends on the values of the bits of the secret key is potentially vulnerable to this attack.

Besides exploiting the correlation of power consumption with the operations performed in the device, SPA can also exploit the correlation between the power consumption and the operands. Typically the power consumption is correlated with the Hamming weight of operands. The Hamming weight represents the number of 1s in the binary representation of data. This type of correlation occurs when the power consumption varies with the number of 1s that change.

In a typical smart-card microprocessor a large portion of the power dissipation occurs in the gates attached to internal buses. Experiments reported by Messerges et al. [97] have shown that the activity on data- and address-buses is the dominant cause of power consumption changes. There are two types of correlation between power and the operands [97, 14]. Which of the two types is observed depends on the design of a particular device in question [104].

The two types of correlation are as follows:

- *Hamming weight correlation*, which occurs when the power consumption and the number of 1s written on the bus are directly correlated.
- *Transition count correlation*, which occurs when the power consumption and the number of bits that change on the bus are correlated. In other words, the power is correlated to the Hamming weight of the XOR of the current and the previous value on the bus.

The Hamming weight of a 56-bit DES key conveys an average of:

$$\sum_{i=0}^{56} \frac{\binom{56}{i}}{2^{56}} \log\left(\frac{2^{56}}{\binom{56}{i}}\right) \approx 3.95$$

bits of information about its value [76]. Physical implementations of many cryptosystems operate on 8-bit data blocks, enabling the power analysis to potentially reveal

$$\sum_{i=0}^{8} \frac{\binom{8}{i}}{2^{8}} \log\left(\frac{2^{8}}{\binom{8}{i}}\right) \approx 2.54$$

bits of information about each byte of the DES key. In total this is $7 \cdot 2.54 \approx 17.8$ key bits. In this way the key space is reduced to approximately 2^{38} possibilities [104], which further facilitates the brute force attack to which DES is already susceptible[2].

Messerges et al. [97] noticed that the Hamming weight information is best used with some information about the cipher implementation. For example, in the case of DES, in the PC1 phase, if the attacker has information about the key bytes and also about shifted versions of the key bytes, a more powerful attack can be mounted. Namely, given the Hamming weight of each byte for eight of the C and D shifts, there is enough information to discover every key bit using the equation:

$$A\vec{k} = \vec{w}, \tag{3.2}$$

where $\vec{w}$ is a 56×1 vector of Hamming weights, w_i; $\vec{k}$ is a 56×1 binary vector of the key bits, k_j; and A is a 56×56 binary matrix such that A_{ij} is 1 if and only if weight w_i includes key bit k_j. Using this technique, even algorithms that use more than 56 key bits, such as triple DES, could be vulnerable [97].

Further, in [97] it is explained how a SPA attack can be mounted if the transition count rather than Hamming weight information is available. Namely, the attacker

[2] For ciphers with longer key sizes (such as triple DES) even the Hamming weight information cannot facilitate the brute force attack [104].

would need to know what is on the data-bus before and after the data being observed is accessed. In many cases this data is a fixed address or an instruction opcode, which can facilitate the situation. If the attacker can access this data, then he would be able to set up a system of equations similar to those given in Equation 3.2. Should the attacker not have all the necessary information, then he might need to resort to trial and error methods to determine the correct equations. As the number of such possibilities is limited, this approach is considered feasible.

Experiments reported in [97] have confirmed that poor implementations of DES almost always leak sufficient information to mount the SPA attack. Shifting the key bytes or the use of conditionals is especially vulnerable. Also, if the portion of the code runs in variable time, power analysis could be used to mount the timing attack.

3.4.2 Countermeasures

Kocher et al. [78] concluded that, in general, it is not particularly difficult to build an SPA resistant device. Avoiding the usage of secret values to perform conditional operations can prevent SPA from correlating power consumption and the operations that are performed. This can be seen in the example of modular exponentiation. Algorithm 5 shows the implementation of modular exponentiation which is resistant to simple power analysis (quoted from [104]).

Algorithm 5 SPA-resistant repeated square-and-multiply algorithm.

INPUT: $M, N, d = (d_{n-1}, \ldots, d_1, d_0)_2$

OUTPUT: $S = M^d \ mod \ N$

1: $S \leftarrow 1$
2: **for** $j = n-1 \ to \ 0$ **do**
3: $\quad S_0 \leftarrow S^2 \ mod \ N$
4: $\quad S_1 \leftarrow S_0 \cdot M \ mod \ N$
5: $\quad S \leftarrow S_{d_j}$
6: **end for**

In algorithms that inherently assume this type of key-dependent branching, it might not be possible to achieve this type of protection, or it can require creative coding and introduce a serious performance penalty [78]. However, the size of SPA characteristic could be reduced by moving the operations with large power characteristics, multiplications for instance, outside of conditional branches [104] as shown in Algorithm 5.

In some cases even constant execution path code can demonstrate serious SPA vulnerabilities, if the nature of the device is such that the attacks could make correlations between the power and the operands [78]. Chari et al. [36] suggested that a way to counteract this problem is to randomly split every bit of the original computation into k shares. Every share is equally probable and every subset of $k-1$ shares is statistically independent of the encoded bit. Computations are then performed on shares, without ever reconstructing the original bit. The shares are refreshed after each operation that involves them, to prevent information leakage. The method to encode the bit into shares must be chosen according to the computation being protected. For example, bit b can be encoded as k random shares: $b \oplus r_1, r_2, \ldots, r_1 \oplus r_2 \oplus \cdots \oplus r_{k-1}$, where r_is are randomly chosen bits. This approach increases the amount of noise by obscuring the value of sensitive data. To deduce the sensitive data the attacker must now combine multiple power measurements from various locations within the power trace.

3.5 Differential power analysis

3.5.1 Introduction

Differential power analysis (DPA) is a class of side-channel attack that is more powerful than simple power analysis. Actually, DPA is believed to be the most threatening attack that resulted from Kocher's research. This is primarily because the attacker does not need to know as many details about the algorithm implementation in order to perform this attack. Moreover, this attack gains additional strength by using statistical analysis to help recover the secret information from the side-channel.

To carry out a DPA attack, an attacker must have a number of power consumption curves (PCC) collected from a device that has repeatedly executed a cryptographic operation with different inputs and the same key. It is crucial that PCCs contain information about the secret key that can be deduced using statistical methods. The algorithmic condition, the so-called *fundamental hypothesis*, states that for a DPA attack to be successful the following must be true (quoted from [57]):

Definition 3.5.1 (*Fundamental Hypothesis*). *There exists an intermediate variable, that appears during the computation of the algorithm, such that knowing a few bits (in practice less than 32 bits) allows to decide whether two inputs (respectively two outputs) give or not the same value for this variable.*

In other words, the (only) algorithmic condition is that there must exist an intermediate variable whose value is dependent upon an easily accessible subset of the secret key and upon known plaintext or ciphertext data. The key bits involved may then be reconstructed by partitioning the power curves according to the chosen intermediate value, and comparing the calculated average curves. The attack is successful if there is a significant difference between two average curves at one or more points in time. What makes the attack particularly interesting, and at the same time powerful, is that most cryptographic algorithms satisfy the fundamental hypothesis.

3.5.2 Attack details

The attack is performed as follows. Let $T_1, T_2, \ldots, T_n$ be power traces collected from a cryptographic device that is repeatedly performing a cryptographic operation using different inputs and the same key. Each trace is an array of k power consumption measurements and represents the power consumed during each cryptographic operation. The general assumption is that the power consumption is different depending on whether zero or one is the calculated value of the specific bit (b) that is being taken into consideration. The bit b is determined by a selection function, D, which depends on the cipher. It is crucial that the key bits are manipulated during the operation defined by the selection function.

The attacks in [78] were performed on the DES encryption algorithm, because of its widespread use. In each of its 16 rounds, the DES performs eight S-box lookup operations. Each of the 8 S-boxes takes as input 6 bits and outputs 4 bits of data. The input 6 bits are generated by XORing 6 bits of the key with 6 bits of the R register (see Section 2.2 for the description of DES).

The selection function for DES can be defined as follows. If the attacker has knowledge of the plaintexts, the selection function D can be defined as the first bit of the register R_1, which can be calculated as:

$$R_1 = L_0 \oplus F(R_0, K_1) \tag{3.3}$$

where

R_1 – is the register containing the rightmost 32 bits of the results of the first round transformation.

L_0 – is the register containing the leftmost 32 bits of the results of the initial permutation of the plaintext - a value known to the attacker.

R_0 – is the register containing the rightmost 32 bits of the results of the initial permutation of the plaintext - a value also known to the attacker.

K_1 – is the first round key.

F – is the round function.

The only unknown in Equation 3.3 is the value of round key K_1. From the definition of the round function (F) and the S-boxes, it follows that the first bit of R_1 is influenced by only 6 bits of the first round key K_1. The attacker does not know these bits, but can use brute force and try all of the 2^6 possibilities.

To generalise, for each guess of the key, K_s, the PCCs are partitioned into two groups according to the intermediate value of the bit b calculated during the operation (using the selection function D), as follows. Let $\mathcal{T}_0$ and $\mathcal{T}_1$ be the two partitions of the power traces:

$$\mathcal{T}_0 = \{T_i : b = 0\}$$

$$\mathcal{T}_1 = \{T_i : b = 1\}$$

For $j = 1 \ldots k$, the average traces are calculated:

$$A_0[j] = \frac{1}{|\mathcal{T}_0|} \sum_{T_i \in \mathcal{T}_0} T_i[j]$$

$$A_1[j] = \frac{1}{|\mathcal{T}_1|} \sum_{T_i \in \mathcal{T}_1} T_i[j]$$

where $|\mathcal{T}_0| + |\mathcal{T}_1| = n$, and $T_i[j]$ is the j-th power consumption measurement in the i-th power trace. Next, the attacker computes the differential trace Δ:

$$\Delta[j] = A_1[j] - A_0[j]$$

$\Delta[j]$ is therefore the average over n plaintexts of the effect on the power consumption measurements at point j, due to the value represented by the selection function D.

If K_s is incorrectly guessed, the bit computed using D will differ from the actual target bit for about half of the plaintexts. The selection function is, therefore, uncorrelated to what is actually computed in the device. In that case, function D is a random function that is used to divide the set of all power samples into two subsets and therefore it should be that:

$$\lim_{n \to \infty} \Delta[j] = 0.$$

In practice, however, this differential trace might not be completely flat and it might even show a slight correlation to the trace for the correct key guess [78].

If, on the other hand, the guess K_s was correct, the computed value of the selection function should be equal to the value calculated in the target device. Therefore, the selection function is actually correlated to the value manipulated in the device and the power consumption and the selection function are also correlated. If the bit b was manipulated at times j', then the expected difference in power when two values of b are manipulated should be:

$$E[T_i[j']|b=1] - E[T_i[j']|b=0] = \varepsilon > 0. \tag{3.4}$$

When j is not equal to j', the power dissipation should be independent of the value of the bit b, and therefore this difference should be zero, *i.e.*:

$$E[T_i[j]|b=1] - E[T_i[j]|b=0] = 0, \forall j \neq j'. \tag{3.5}$$

As the number of traces grows, values $A_1[j]$ and $A_0[j]$ converge to $E[T_i[j]|b=1]$ and $E[T_i[j]|b=0]$, respectively. Therefore, according to Equations 3.4 and 3.5:

$$\lim_{n\to\infty} \Delta[j] = \lim_{n\to\infty} (A_1[j] - A_0[j]) = \begin{cases} \varepsilon, & j = j' \\ 0, & j \neq j' \end{cases} \tag{3.6}$$

Equations 3.4, 3.5 and 3.6 show that if enough plaintext samples are used, $\Delta[j]$ will show a power bias (spike) of ε at time j', and will converge to zero at all other times. However, due to small statistical biases in the outputs from the S-boxes, the Equation 3.6 is not entirely correct, and in practice $\Delta[j]$ will not always converge to zero, but a larger bias will occur at time j' [97]. If the key used for the cryptographic operation was correct, a spike will appear where the selection function is correlated to the value of the bit being manipulated.

In summary, the attack works as follows. For each guess the attacker constructs a new partition for the power traces and a new differential trace, $\Delta[j]$. If the appropriate selection function was chosen, the differential trace should show biases whenever the bit defined by the selection function was manipulated. If the selection function was not chosen correctly, then the resulting $\Delta[j]$ will not show any biases. In this way the attacker can determine the six bits of the first round key, K_1. Repeating this approach for seven other S-boxes the attacker can learn the entire 48 bits of the first round key. The remaining 8 bits of the DES key can be discovered by brute force, or by applying the attack on the successive rounds.

The attack can also be performed on the last round of DES encryption. This attack works similar to the case when the first round is attacked, except that in this case the attacker needs to know the ciphertexts. The selection function can be in this case defined as the first bit of the register L_{15} which can be calculated as follows:

$$\left.\begin{aligned} R_{16} = &\quad L_{15} \oplus F(R_{15}, K_{16}) \\ L_{16} = &\quad R_{15} \end{aligned}\right\} \Rightarrow \quad R_{16} = L_{15} \oplus F(L_{16}, K_{16})$$
$$\Rightarrow \quad L_{15} = R_{16} \oplus F(L_{16}, K_{16}) \tag{3.7}$$

where

R_{16} – is the register containing the rightmost 32 bits of the results of the last round transformation of DES and can be obtained by applying the inverse of the final permutation on the ciphertext. Since the ciphertext is known to the attacker, this value can be easily obtained.

L_{16} – is the register containing the leftmost 32 bits of the results of the last round transformation of DES and can be obtained by applying the inverse of the final permutation on the ciphertext. Since the ciphertext is known to the attacker, this value can also be easily obtained.

L_{15} – is the L register after round 15.

K_{16} – is the key used in round 16.

F – is the round function.

The D function in Equations 3.3 and 3.7 was chosen because at some point during DES encryption, the value of the specified bit must be calculated. When this occurs, there will be a difference in the amount of power dissipated/consumed when this value is equal to zero compared to the case when this value is equal to one.

3.5.3 Increasing the magnitude of the bias signal

The number of traces (n) that are required to successively perform the DPA attack, depends on the noise in the power consumption measurements (σ) and on the size of the bias (ε) attributed to the partitioning bit [104]. The spike is identified if:

$$\varepsilon > \frac{2\sigma}{\sqrt{n}}.$$

Therefore, the number of power traces necessary for the attack must be:

$$n > \left(\frac{2\sigma}{\varepsilon}\right)^2.$$

Messerges et al. [97] suggested that a way to decrease the number of power consumption measurements is to use *multiple-bit DPA attacks*, in which the magnitude of the bias (ε) is increased. The magnitude of the bias depends on the number of bits that are defined by the selection function D. In devices which show Hamming weight correlation, for the selection function D that outputs d bits: $\varepsilon = dl$, where l is a constant equal to the instantaneous power difference between two data words with Hamming weights i and $i+1$. Experiments have shown [97] that this difference can be considered approximately equal, for all i.

When performing the multiple-bit DPA attack, three sets are used for partitioning:

$$\mathcal{T}_0 = \{T_{ij} : D(.,.,.) = 0^d\}$$

$$\mathcal{T}_1 = \{T_{ij} : D(.,.,.) = 1^d\}$$

$$\mathcal{T}_2 = \{T_{ij} : T_{ij} \notin \mathcal{T}_0, \mathcal{T}_1\},$$

while the rest of the attack remains unchanged and the power traces in set $\mathcal{T}_2$ are not used in the analysis.

An example of a 4-bit DPA attack on DES was outlined in [97]. The 4-bit selection function is based on the 4-bit output from S-box. The power traces are partitioned according to whether the output from the first S-box in the DES round functions is 0000 or 1111. The expected size of the bias signal is 4ε. The problem with this approach is that each of the partitions should have about $n/2^4$ power traces. In that way the averages of the partitions should have higher levels of noise. The experiments in [97] have shown that this can introduce difficulties in recognising the bias of the correct guess as the incorrect solution might have the same magnitude as the correct key, in which case a small brute-force search could be used to discover the correct key.

Another way to mount a multiple-bit DPA attack is to partition the power traces into $\mathcal{T}_0$ and $\mathcal{T}_1$ by not using the output, but the input to the S-box. In this way one partition maximises the number of address-bus transitions and the other minimises the number of address-bus transitions. The rest of the attack is repeated as usual.

As noted in [97], when mounting the d-bit DPA attack, the attacker may need to use more power traces. This is due to the fact that a large portion of the power traces, approximately $n(1-2^{1-d})$ power traces placed in the partition $\mathcal{T}_2$, are not used in the

attack. As the averaging decreases noise, this would mean that the attacker would need to obtain a higher number of power traces in order to reduce noise in partitions $\mathcal{T}_0$ and $\mathcal{T}_1$. This implies that an attacker cannot arbitrarily increase the number of output bits of the selection function, as the requirement for the number of power traces increases significantly. Messerges et al. [97] have shown that if an attacker running a 1-bit DPA attack needs n power samples to mount the attack, then for a d-bit DPA attack the number of power samples needed to maintain the same signal-to-noise ratio is equal to $n_d = 2^{d-1}n/d^2$. It can be seen from this relation that for the 8-bit attack, only twice as many power samples are needed. The advantage is that the resulting power bias in d-bit attacks should be magnified d times. Experiments in [97] have confirmed that the signal levels for multiple-bit DPA attacks are indeed much stronger.

3.5.4 Higher-order DPA attacks

In the DPA attack explained in Section 3.5.2, also known as the first-order DPA attack, the attacker records the power consumption and computes their particular statistical property for each individual instant of the computation. This attack does not require any knowledge about the individual electric consumption of each instruction, nor the positions in time in which each of these instructions occurred. It only relies on the fundamental hypothesis given in Definition 3.5.1. This makes the (first-order) DPA attack particularly impressive, since using simple mathematical tools and techniques that are independent of the algorithm's implementation, the values of the secret keys can be relatively easily revealed.

The variations of this method which use more than one intermediate result in the statistical analysis are called *higher-order DPA attacks*. Higher-order DPA attacks are a generalisation of the (first-order) DPA attack, in which PCCs are analysed using joint statistics applied to a collection of points in time. The higher-order attacks are more powerful, but also more complicated as the choice of joint statistic and the points in time may depend on the specifics of the cipher implementation.

Higher-order DPA attacks were defined in [78] as DPA attacks that use combinations of one or more samples within a single power trace. During a first-order DPA attack, the attacker calculates the statistical properties of each power trace at each sample point. In a higher-order DPA attack, the attacker calculates joint statistical properties of the power consumption traces at multiple sample points within each trace. The definition of the n-th order DPA attack (quoted from [95]) is given next.

Definition 3.5.2. *An n-th order DPA attack makes use of n different samples in the power consumption signal that correspond to n different intermediate values calculated during the execution of the algorithm.*

Higher-order DPA attacks depend on the following fundamental hypothesis (quoted from [15]):

Definition 3.5.3 (*Fundamental Hypothesis (order n)*). *There exists a set of n intermediate variables, that appear during the computation of the algorithm, such that knowing a few key bits (in practice less than 32 bits) allows to decide whether two inputs (respectively two outputs) give or not the same value for a known function of these n variables.*

Chari et al. [36] have shown that the complexity of performing a higher-order DPA attack increases with the exponent of the number of points used in the joint statistic.

In practice, higher-order DPA attacks are more difficult to mount than first-order DPA attacks. For example, the second-order DPA attack requires more complex analysis, increased memory and processing requirements and increased number of power consumption patterns [44]. Messerges [95] demonstrated that a second-order DPA attack can be performed on the software implementation of the cipher which uses random masking. Two implementations of a cipher are observed, pseudo-codes for which are given in Figure 3.5.

```
W_1(plaintext)
{
  A: result = plaintext ^ secret_key;
  ...
  // other operations
  ...
  return ciphertext;
}
```

```
W_2(plaintext)
{
  B: r_mask = random();
  m_plaintext = plaintext ^ r_mask;
  C: result = m_plaintext ^ secret_key;
  ...
  // other operations
  ...
  return ciphertext;
}
```

Figure 3.5: Routines vulnerable to first and second-order DPA attacks.

In the routine W_1 the operation at line A is vulnerable to the first-order DPA attack. The routine W_2 is resistant to first-order DPA attack, but the joint statistic applied at

the operations on lines B and C mounts the second-order DPA attack. The DPA bias statistic for routine W_2 is defined as:

$$\Delta = \bar{S_0} - \bar{S_1}$$

where $\bar{S_b}$, for $b = \{0,1\}$ is an average statistic calculated as $\bar{S_b} = |P_B - P_C|$, over all plaintext input values in which the i-th bit is set to either 0 or 1, and P_B and P_C are instantaneous power consumptions at points B and C, respectively. If $\Delta > 0$, then the key bit is equal to one, otherwise it is equal to zero. Further details of the correctness of this approach are given in [95]. The author also points out that in this attack the knowledge of the cipher implementation is important, since otherwise the attacker would need to resort to additional statistical analysis to find the critical points.

3.5.5 Variations of the DPA attack

Biham and Shamir [29] studied the power analysis of key scheduling in AES candidates. They noted that an important disadvantage of the DPA attack was that the attacker has to know all the plaintexts (for the attack on the first round) or all the ciphertexts (for the attack on the last round). The authors point out that, in reality, the attacker does not necessarily know the precise values of the inputs or the outputs, as these could be additionally protected with an unknown protocol. Biham and Shamir introduced a variant of power analysis in which the attacker does not have to know either the inputs or the outputs of the encryption operations, the precise timing in which they were carried out, or the details of their software implementations in the smart-cards. Their technique performs the power analysis of the key scheduling part of the block cipher, as an important target for power analysis because its power consumption is a function of the key, and not of the data, and thus could be used to attack smart-cards in which the plaintext, ciphertext, protocols and implementation details are unknown to the attacker. It is, however, assumed that the protocol always performs the same subroutines in the same order, and that they always require the same number of clock cycles. As a result, the power consumption graphs of different executions of the protocol can be aligned and compared at the level of single instructions.

The first part of the attack is to discover the portion of the power trace which corresponds to the key scheduling operation, and is performed in two steps:

Step 1 – The attacker executes the protocol many times on a single smart-card in different contexts (*i.e.* executing different operations). The attacker then aligns the

power curves at each clock cycle, and compares them. Next, those clock cycles in which the graphs show a large variability in power consumption, due to differences in the processed data, are eliminated from further consideration. The remaining clock cycles represent operations that are data-independent.

Step 2 – The attacker repeats Step 1 for several smart-cards, which contain different cryptographic keys, and finds their common data-independent regions. Regions which show small variability among different cards and which are likely to be standard system-wide operations are eliminated.

The attacker performs a filtering process identifying the cycles in which each card behaves in the same way (regardless of the values being processed), but different cards behave in different ways (depending on the card's unique identity). These periods will include the key scheduling elements of the encryption process, but not the routine book-keeping operations or the actual data encrypting steps.

The next step is to study the individual steps of the key scheduling algorithm. Software implementations of iterated block ciphers usually compute each subkey just before it is used in the appropriate round (usually due to small RAM size). The computed subkey is stored in the RAM in chunks of 8 bits. The amount of current consumed depends on the number of 1s among the 8 written bits. Even though the measurements are likely to contain many errors and provide only partial information about the key, they can be smoothed by averaging the power consumed by the large number of executions with a common key. The crucial observation is that even imperfect measurements of the Hamming weights of a small number of bytes generated during the key scheduling part of the block cipher can completely reveal the key.

Biham and Shamir performed the analysis of the feasibility of applying this attack to each of the AES candidates. In particular, the authors note that Rijndael applies several layers of XORing each key word into the next word, with almost no other information in between. There is mixing in the key bytes, but the authors claimed that the derivation of Hamming weights is not expected to be very complicated.

Clavier et al. [39] also proposed an improvement to the general DPA technique, called *Hamming integration variant method*. They have noticed that if more than one output bit of S-boxes are observed, more information can be extracted in comparison to a DPA attack in which only one output bit is observed. In the case of DES, the basic idea is that instead of considering one output bit of the S-boxes, all four output bits are considered. The Hamming integration method assumes that the device's power

consumption is proportional to the Hamming weights of the observed data. PCCs are classified according to the Hamming weights of the *S*-box outputs (details can be found in [39]). The signal-to-noise ratio is increased this way, which should have a consequence of enabling a successful attack with a reduced number of PCCs. The authors also observe that very strong correlation exists between the chip's power consumption and the operations being executed. This value is high during data transfers between the CPU and external RAM, so the operation which is performed after the *S*-box output is determined, can be targeted for DPA. They notice that, in this case, the power consumption does not depend only on the value of the bit that is output from the *S*-box, but also on the transitions that occur on the data-bus. A high power consumption is expected when a 1 is written on the bus previously discharged, or similarly when a 0 is written on the bus previously charged. The authors noticed that the status of the bus must be considered when all four output bits from the *S*-box are observed simultaneously, as the information could be lost otherwise. This means that the previous value of the bus line must be determined. If the attacker does not know the previous value of bus line, all possibilities must be examined experimentally.

3.5.6 Countermeasures

The fact that DPA attacks represent a real threat to cryptographic devices has sparked huge interest in preventing these attacks and protecting cryptographic devices. DPA was demonstrated as being able to break almost any implementation of both symmetric and asymmetric ciphers and the majority of the proposed countermeasures were shown not to be sufficient. Rather than preventing the attack, countermeasures tend to increase the complexity of performing the attack to the point that it becomes infeasible or too expensive to perform it.

The complexity of differential power analysis can be increased by introducing software (algorithmic) and hardware (physical) countermeasures. Software countermeasures tend to be more desirable, since they can be implemented on existing architectures. Hardware countermeasures, although generally more costly to implement, might be necessary depending on the desired level of security.

DPA is an implementation-oriented attack that exploits the fact that the power consumption pattern is correlated to the value of at least one bit of the secret key. A general strategy to render DPA more complex to apply, is to *balance* and *randomise* major computations which involve the secret key. This in turn is equivalent to increas-

ing the amount of noise that is already present in the power traces, and in that way hiding the DPA bias signal.

3.5.7 Software countermeasures

Although considered inexpensive, software countermeasures can result in significant memory and execution time overheads [35]. The overhead depends on the type and arrangement of the fundamental operations used in an algorithm. Some of the proposed software countermeasures are summarised in the following sections.

3.5.7.1 Desynchronisation

The major strength of the DPA attack comes from the assumption that the sequence of instructions is fixed, and that in adding power consumptions up, the instructions are paired nicely. Daemen and Rijmen [46] suggested that inserting dummy instructions based on some modifying parameter, could make the straightforward attack not work anymore. However, dummy instructions, as any other instruction, have characteristic power signatures, which weakens this countermeasure.

3.5.7.2 Software balancing

Software balancing was also proposed by Daemen and Rijmen [46]. This technique specifies that the program is written in a such way that during the execution both the data and its complement are processed. In this way the correlation between data and power consumed is decreased, but overheads could be significant.

3.5.7.3 The ”duplication” method

Goubin and Patarin [57] studied how DPA attacks may be prevented by using the following idea: replace each variable V that occurs during the computation and that depends on the inputs or the outputs, with k variables $V_1, \ldots, V_k$, and an appropriately chosen function f for which $V = f(V_1, \ldots, V_k)$ that satisfies two additional conditions:

1. It is not possible to deduce information about the set of k values of variables V_i, $i = 1 \ldots k$, from any subset of $k-1$ values. In other words, the knowledge of $k-1$ variables V_i does not give any information about V itself.

2. Transformations to be performed on variables V_i during the computation can be implemented without calculating the value of V.

The cryptographic algorithm is transformed by replacing each intermediate variable V depending on the inputs (or influencing the outputs) with k variables $V_1, \ldots, V_k$. The authors have demonstrated this approach in the cases of DES and RSA algorithms.

- For DES, each intermediate variable V is replaced with two variables V_1 and V_2. They choose the function $f(v_1, v_2) = v = v_1 \oplus v_2$, where $\oplus$ is the XOR operation, which obviously satisfies conditions 1 and 2.

- In the case of RSA, each intermediate variable V whose values lie in the multiplicative group of Z/nZ is replaced with two variables V_1 and V_2 and the choice of function $f(v_1, v_2) = v = v_1 \cdot v_2 \; mod \; n$. It is easy to see that this function also satisfies the conditions 1 and 2.

The authors claim that it is possible to show that when each variable is split into k variables, the complexity of the implementation increases in $O(k)$, while the complexity of the attack should increase exponentially in k.

3.5.7.4 Random masking

Messerges [96] proposed a way of protecting vulnerable operations in cryptographic algorithms by using a strategy that employs *random masks*. A *random boolean and arithmetic masking* was proposed for masking fundamental operations used in implementation of cryptographic algorithms vulnerable to power analysis. The random masks decorrelate power consumption signals from the secret key and the input and output data, thereby forcing the DPA to use joint probability distribution.

Each of the masking strategies consists of masking word x with random mask r_x resulting in x'. Two types of masking are distinguished:

- *Boolean masking*, which uses bitwise XOR and masks x with r_x as follows: $x' = x \oplus r_x$. This type of masking can be used in DES.

- *Arithmetic masking*, which uses addition and subtraction modulo 2^{32} (it is assumed that the data is represented with 32-bit words) and masks x with r_x as follows: $x' = (x - r_x) \; mod \; 2^{32}$. This type of masking can be used in RSA.

The basic strategy is to mask the key and the input operation prior to algorithm execution. In that case, since the algorithm is executed on masked data, all the intermediate results are also masked. As the new mask is randomly chosen for each new run of the algorithm, the statistical correlation of the power consumption measurements is

not sufficient for a successful attack. The attacker would have to look at joint probability distributions of multiple points in the power signal in order to successfully mount the power analysis attack.

Boolean and arithmetic masking operations are possible only if all the fundamental operations of the algorithm can work with masked input and produce masked output. Different classes of cryptographic algorithms use different masking depending on the kind of operations performed in the algorithm. For example, DES requires only boolean masking; while RSA uses only arithmetic operations and, therefore, needs only an arithmetic mask. On the other hand, some algorithms require more than one mask, for instance algorithms which perform both arithmetic and boolean operations, *boolean-to-arithmetic* and *arithmetic-to-boolean* maskings are required.

The boolean-to-arithmetic masking is given in Algorithm 6 (quoted from [96]), in which x represents the unmasked data, x' the masked data, and r_x a random mask. The algorithm works by unmasking x using the XOR operation and then arithmetically masking x using modular subtraction. Since the unmasked x can be vulnerable to power analysis attack, a random value C is used to randomly select whether x or $\overline{x}$ is unmasked. Arithmetic-to-boolean masking algorithm is very similar to boolean-to-arithmetic algorithm (see [96] for further details).

Algorithm 6 Boolean-to-arithmetic masking.

INPUT: (x', r) *such that* $x = x' \oplus r$

OUTPUT: (A, r) *such that* $x = (A - r)$ *mod* 2^k

1: *Randomly select* : $C = 0$ *or* $C = -1$

2: $B = C \oplus r$ /* $B = r$ *or* $B = \overline{r}$ */

3: $A = B \oplus x'$ /* $A = x$ *or* $A = \overline{x}$ */

4: $A = A - B$ /* $A = x - r$ *or* $A = \overline{x} - \overline{r}$ */

5: $A = A + C$ /* $A = x - r$ *or* $A = \overline{x} - \overline{r} - 1$ */

6: $A = A \oplus C$ /* $A = x - r$ */

This technique shows better performance characteristics when compared to secret-sharing heuristic (reviewed in Section 3.5.7.5), but should be carefully implemented. Coron and Goubin [41] proved that the approach given in [96] is not sufficient to prevent DPA. According to [96], the basic strength of the masking algorithm is in the fact that the attacker never knows whether variable x or $\overline{x}$ is being calculated; therefore, the attacker cannot guess the bits of the key. In [41] it is pointed out that this is true if only one bit is observed (one-bit DPA attack). If two bits are observed, then a DPA-like

attack can be applied. This attack is based on the fact that if two bits of x are equal then those two bits are also equal in $\overline{x}$. Therefore, in the attack, all PCCs are divided into two groups according to this criteria. If the power consumption when two bits are equal differs from the case when these bits differ, then the 2-bit DPA attack can be applied, and the expected DPA bias amplitude should be equal to:

$$\varepsilon = \frac{\mu_{00} + \mu_{11} - \mu_{01} - \mu_{10}}{2}$$

where μ_{ij}, $i, j \in \{0, 1\}$, is the mean value of power consumption when the pair of observed bits is equal to $'ij'$.

The authors provide no experimental results for their analysis, but point out that this is not a higher-order DPA attack, because the latter consider joint probability distributions of multiple points in the power signal. The higher-order DPA attack, as shown in [36], requires a number of experiments that is exponential in the number of points considered. The authors argue that since this attack concentrates only at one point in the power signal, the required number of experiments should be of the same order as in the case of a single-bit DPA.

3.5.7.5 Secret-sharing

Chari et al. [36] proposed a general technique to counter statistical attacks in devices where the power model is reasonable and the source of randomness exists. Their technique is based on a well known secret-sharing scheme, which randomly splits every relevant bit into several (k) shares. Each of k shares and furthermore, every collection of $k-1$ shares should be statistically independent of the bit required for computation. In this way only shares of the sensitive data are manipulated and never the data itself. This share-based technique needs to be applied for a sufficient number of steps into the computation until the attacker has a very low probability of predicting bits, *i.e.*, until sufficient secret key operations have been carried out. Similar splitting has also to be performed at the end of computation if attacker can access its outputs. This generally increases the noise and complicates the analysis of data, as the attacker has to analyse joint distribution functions on multiple points in the power trace. The authors claim that the number of power traces needed to perform the attacks increases exponentially with the noise. This countermeasure eliminates the threat of the first-order DPA attack and forces the attacker to mount a second-order DPA attack. However, the performance penalty of the secret-sharing scheme could make this countermeasure impractical [104, 103].

3.5.7.6 Elliptic curve cryptography

The use of elliptic curves in cryptography was first proposed by Koblitz [74] and Miller [99] in 1985. Since then, elliptic curves (EC) over large finite fields have become a common way to implement public-key protocols. Elliptic curve cryptosystems (ECC) can use much smaller key sizes, typically around 160 bits, providing the same security level as 1024 key bits of RSA. In addition, ECCs show better performance and computation speed than other multiplication groups such as RSA and ElGamal at the same security level [60]. This makes ECCs very attractive for implementations on devices with limited memory and computation capabilities, such as smart-cards.

Definition 3.5.4. *An elliptic curve over field $\mathbb{K}$ is a pair (E, O) where E is a non-singular curve of genus one over $\mathbb{K}$ with a point $O \in E$. The set of points $(x,y) \in \mathbb{K} \times \mathbb{K}$ verifying the (non-singular) Weierstraß equation*

$$E_{/\mathbb{K}} : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \quad (a_i \in \mathbb{K}) \tag{3.8}$$

together with O, form an elliptic curve over $\mathbb{K}$. The point O is called the point of infinity.

If *char* $\mathbb{K} \neq 2$ and *char* $\mathbb{K} \neq 3$, Equation 3.8 can be reduced to the form:

$$E_{/\mathbb{K}} : y^2 = x^3 + a_x^2 + b \quad (a, b \in \mathbb{K}).$$

In the field $\mathbb{K} = GF(2^n)$ of characteristic 2, Equation 3.8 can be reduced to the form:

$$E_{/\mathbb{K}} : y^2 + xy = x^3 + a_x^2 + b \quad (a, b \in \mathbb{K}).$$

The set of points (x,y) on an elliptic curve, together with point O can be equipped with an Abelian group structure by the following operation.

Definition 3.5.5 (*Addition formula for* *char* $\mathbb{K} \neq 2,3$). *Let $P = (x_1, y_1) \neq O$ be the point on the elliptic curve. The inverse of P is:*

$$-P = (x_1, -y_1).$$

Let $Q = (x_2, y_2) \neq O$ be a second point on the elliptic curve. The sum $P + Q = (x_3, y_3)$ can be calculated as:

$$\begin{aligned} x_3 &= \lambda^2 - x_1 - x_2 \\ y_3 &= \lambda(x_1 - x_3) - y_1 \\ \lambda &= \begin{cases} \frac{y_2 - y_1}{x_2 - x_1}, & if\ P \neq Q \\ \frac{3{x_1}^2 + a}{2y_1}, & if\ P = Q \end{cases} \end{aligned} \tag{3.9}$$

Definition 3.5.6 (*Addition formula for* *char* $\mathbb{K} = 2$). *Let $P = (x_1, y_1) \neq o$ be the point on the elliptic curve. The inverse of P is:*

$$-P = (x_1, x_1 + y_1).$$

Let $Q = (x_2, y_2) \neq o$ be a second point on the elliptic curve. The sum $P + Q = (x_3, y_3)$ can be calculated as:

$$\left.\begin{array}{rcl} x_3 & = & \lambda^2 + \lambda + x_1 + x_2 + a \\ y_3 & = & \lambda(x_1 + x_3) + x_3 + y_1 \\ \lambda & = & \frac{y_1 + y_2}{x_1 + x_2} \end{array}\right\} \quad , \ if \ \ P \neq Q, \tag{3.10}$$

and as

$$\left.\begin{array}{rcl} x_3 & = & \lambda^2 + \lambda + a \\ y_3 & = & x_1^2 + (\lambda + 1)x_3 \\ \lambda & = & x_1 + \frac{y_1}{x_1} \end{array}\right\}, \ if \ \ P = Q. \tag{3.11}$$

One of the main drawbacks of using ECs in cryptography is the fact that adding two points requires too many field multiplications. Certain special curve types allow for faster addition, but this choice has to be carefully made as shown in [55].

Koblitz [75] suggested an alternative, which he called *anomalous binary curves* (ABC curves) or Koblitz curves. These are the curves of types:

$$E_1 : y^2 + xy = x^3 + x^2 + 1$$

$$E_2 : y^2 + xy = x^3 + 1$$

over $GF(2^n)$. Cryptosystems based on these curves offer significant advantage in terms of reduced processing time. This fact, together with the short key size, has made Koblitz curve cryptosystems attractive for practical implementations.

Similarly, in order to ease additions, Montgomery [101] considered the family of the elliptic curves of the form:

$$E_{/\mathbb{K}} : By^2 = x^3 + Ax^2 + x,$$

where $B(A^2 - 4) \neq 0$ and *char* $\mathbb{K} \neq 2$, and the cardinality of $E(\mathbb{K})$ is divisible by 4.

Hessian-type elliptic curves were also considered because they provide unified formula for adding and doubling. They are defined as the intersection of the two quadrics in $\mathbb{P}^3$, and can be given in the following form:

$$E_{/\mathbb{K}} : x^3 + y^3 + 1 = 3Dxy,$$

where $D \in \mathbb{K}$, $D^3 \neq 1$, and the cardinality of $E(\mathbb{K})$ is divisible by 3.

Projective coordinates. The addition formulae, given in Equations 3.9, 3.10 and 3.11, require a number of arithmetic operations, namely, additions, squaring, multiplications and inversions over $GF(2^n)$. The representation of points of the elliptic curves can influence the cost of point adding and doubling [61]. To avoid costly inversions, it is convenient to use projective coordinates. Among numerous possibilities, the two most commonly used are *homogeneous* and *Jacobian* projective coordinates.

- *Homogenous projective coordinates* are obtained by setting $x = X/Z$ and $y = Y/Z$, whereby the general Weierstraß equation becomes:

  $$E_{/\mathbb{K}} : Y^2Z + a_1XYZ + a_3YZ^2 = X^3 + a_2X^2Z + a_4XZ^2 + a_6Z^3.$$

  The point of infinity $\mathcal{O}$ is represented by $(0,\theta,0)$ for some $\theta \in \mathbb{K}^*$, and the affine point (x,y) is represented by $(\theta x, \theta y, \theta)$ for some $\theta \in \mathbb{K}^*$, and a projective point $(X,Y,Z) \neq \mathcal{O}$ corresponds to affine point $(X/Z, Y/Z)$.

- *Jacobian projective coordinates* are obtained by setting $x = X/Z^2$ and $y = Y/Z^3$, whereby the general Weierstaß equation becomes:

  $$E_{/\mathbb{K}} : Y^2 + a_1XYZ + a_3YZ^3 = X^3 + a_2X^2Z^2 + a_4XZ^4 + a_6Z^6.$$

  The point of infinity $\mathcal{O}$ is represented by $(\theta^2, \theta^3, 0)$ for some $\theta \in \mathbb{K}^*$, and the affine point (x,y) is represented by $(\theta^2 x, \theta^3 y, \theta)$ for some $\theta \in \mathbb{K}^*$, and a projective point $(X,Y,Z) \neq \mathcal{O}$ corresponds to affine point $(X/Z^2, Y/Z^3)$.

Scalar multiplication. The basic operation performed in ECC algorithms is the *elliptic curve scalar multiplication* over a large finite field (definitions and details regarding finite fields can be found in [84]). Elliptic curve scalar multiplication (or just scalar multiplication) is the operation of adding a point P to itself d times and is denoted as dP. Scalar multiplication of a point on an EC is analogous to modular exponentiation in a multiplicative group.

The group structure of the ECs enables the translation of existing discrete logarithm cryptosystems into the context of elliptic curves [40]. The discrete logarithm problem in group G of order n and generator g, is the problem of finding x given an element $y = g^x$ of G. The basic strength of symmetric encryption algorithms is in the discrete logarithm problem. Finding d such that $Q = dP$ seems to be much harder in

the group defined over the elliptic curve when compared to other groups, such as the multiplicative group over the finite field.

The scalar multiplication used in ECCs is analogous to modular exponentiation in multiplicative groups. The basic algorithm used for this operation is the *double-and-add algorithm* given in Algorithm 7. Computing dP is based on a binary expansion $d = (d_{l-1}, \ldots, d_0)$ where d_{l-1} is the most significant bit of d, and it is sometimes known as the *binary algorithm*.

Algorithm 7 Double-and-add algorithm for scalar multiplication

INPUT: $P, d = (d_{l-1}, \ldots, d_0)$

OUTPUT: $Q \leftarrow dP$

1: $Q \leftarrow P$
2: **for** $i = l-2$ *to* 0 **do**
3: $\quad Q \leftarrow 2Q$
4: $\quad$ **if** $d_i = 1$ **then**
5: $\quad\quad Q \leftarrow Q + P$
6: $\quad$ **end if**
7: **end for**

Various techniques have been suggested to speed-up scalar multiplications. They tend to reduce the number of point operations. For example, one idea is to precompute the table of multiplies of the point P, for a known P. The feasibility of this approach depends on the memory capabilities of the particular system.

Addition-subtraction chains. Choosing a representation of d that has fewer non-zero elements, reduces the number of EC additions and improves the scalar multiplication. The conventional binary system is non-redundant and d has only one representation. Using different number systems which have redundancy in them, the integer d can be represented in more than one way. One such approach uses *signed binary expansion* of d and the fact that the addition and the subtraction have the same cost.
The exponent d is seen as:

$$d = \sum_{i=0}^{l-1} c_i 2^i, \;\; c_i \in \{-1, 0, 1\}. \tag{3.12}$$

The algorithm that uses this representation is called *addition-subtraction algorithm* and is given in Algorithm 8.

Algorithm 8 Addition-subtraction algorithm for scalar multiplication

INPUT: $P, d = (d_{l-1}, \ldots, d_0)$

OUTPUT: $Q \leftarrow dP$

```
1:  Q ← P
2:  for i = l − 2 to 0 do
3:      Q ← 2Q
4:      if c_i = 1 then
5:          Q ← Q + P
6:      end if
7:      if c_i = −1 then
8:          Q ← Q − P
9:      end if
10: end for
```

The *non-adjacent form* (NAF) of d is a signed binary expansion of d where $c_i c_{i+1} = 0$ for all $i > 1$. It can be shown [55] that each positive integer has a unique NAF with the minimal number of non-zero c_is in the representation given by the Equation 3.12.

The problem of computing dP using the fewest number of EC group operations for a given d, is equivalent to finding a shortest addition-subtraction chain for d [55]. An addition chain for d is a sequence of positive integers:

$$a_0 = 1 \rightarrow a_1 \rightarrow a_2 \rightarrow \cdots \rightarrow a_r = d$$

such that $a_i = a_j + a_k$ for $k \leq j < i$ and for all $i = 1, 2, \ldots, r$. An addition-subtraction chain is defined as the extended addition chain, where $a_i = \pm a_j \pm a_k$ for some $k \leq j < i$, for all $i = 1, 2, \ldots, r$. The shortest addition-subtraction chain for d gives the scalar multiplication for the fewest number of EC group operations. Now, dP is calculated by computing a_1P, a_2P, ..., $a_rP = dP$.

Resistance to SPA. Naive implementations of elliptic curve cryptosystems are highly vulnerable to power analysis as Coron first suggested in [40]. If the scalar multiplication is performed using the double-and-add algorithm, then SPA can be applied. In Algorithm 7, the step $Q \leftarrow Q + P$ in line 4 is only processed if $d_i = 1$. The attacker can attempt to determine the bits of d by seeing how the program behaves at the if-statement. The test is always carried out, but the point addition is performed only when $d_i = 1$. This operation requires a number of time and power consuming operations, such as $GF(2^n)$ multiplications and inversions as shown in Equations 3.9, 3.10

and 3.11. The attacker could try to spot this jump in the power trace, which would enable him to discover the bits of the secret scalar d.

The major threat of SPA attack on ECCs comes from the fact that addition and doubling operations can be distinguished. The power signal can show the difference between point doubling and point addition in Algorithm 7, and in that way discover the secret exponent d. The attacker only needs to observe the power consumption of the device to identify those parts of the power curve that correspond to additions and duplications, which trivially gives the key d.

Coron [40] suggested a simple solution to protect EC implementations against SPA. The execution of the EC addition is made independent of the values of d, by removing branches in the code that depend on the values of the secret scalar, as shown in Algorithm 9. The drawback is that this solution penalises the running time. More efficient SPA resistant algorithms have been suggested in [61].

Algorithm 9 Double-and-add scalar multiplication resistant to SPA attack.

INPUT: $P, d = (d_{l-1}, \ldots, d_0)$

OUTPUT: $Q[0] \leftarrow dP$

1: $Q[0] \leftarrow P$
2: **for** $i = l-2$ *to* 0 **do**
3: $\quad Q[0] \leftarrow 2Q[0]$
4: $\quad Q[1] \leftarrow Q[0] + P$
5: $\quad Q[0] \leftarrow Q[d_i]$
6: **end for**

It has been proposed that a model for the EC in which addition and doubling are given by the same formula would not suffer from the side-channel analysis of the code dependent nature of the operation. One common idea to make point addition and point doubling indistinguishable is to unify the common part of the code for both operations and add dummy code to balance the difference between the two operations [83]. However, this defence is unsuitable since the point addition and point doubling would still have separate execution profiles, and would reside in different areas of memory and execute in different parts of the hardware [83]. Even if the same code was used for point addition and doubling operations, the test in the scalar multiplication could still leak some information about the key. Ideally, one would have to execute the same code at the same address with different results. For standard elliptic curves of the Weierstraß model one cannot use the same formula for point addition and point doubling, since

the addition formula has a singularity when the two points are the same. Liardet and Smart [83] suggested that this can be achieved by using different forms of the curve, where mathematically the point addition and point doubling use the same formula. Their approach involves the representation of an elliptic curve as the intersection of two quadratic surfaces in $\mathbb{P}^3$. The price of this technique is that the performance is affected as the number of multiplications and squarings in the finite field increases. They also combined their approach with the random windowing method (*e.g.* [55]) for point multiplication and point blinding. However, even after using the various optimisations such as Jacobi form and the signed windowing method (details in [83]), the total increase in the computation cost is 70% in comparison with a standard projective coordinate method. In addition to the cost problems, Walter [133] proved that this method is vulnerable to certain attacks.

Joye and Quisquater [69] investigated the cubic form of the EC, known as Hessian form. The symmetry of the Hessian form, provides the same algorithm for point-addition and point-doubling. This solution provided an improvement over Jacobian curves of 33%. In a similar way, Brier and Joye in [31] showed how to perform point doubling and point addition with the same formula with the general Weierstraß parametrisation.

Oswald and Aigner [110] have proposed randomisation of the binary algorithm (double-and-add algorithm in which d is represented using binary expansion) in order to resist the power analysis. Most of the proposed countermeasures that use randomisation as a way of resisting the power analysis [40, 61, 98] randomised the input parameters (point P or the secret scalar exponent d) of the binary algorithm. In the solution proposed in [110] the binary algorithm itself is randomised. This is achieved by inserting random decisions into addition-subtractions chains, a technique which was originally used to speed-up the binary algorithm [60].

Gebotys and Gebotys [54] investigated a modification of adding and doubling routines for security against power attacks, by inserting redundant operations into point-addition and point-doubling routines so that the order and type of field operations were identical, thereby making the signatures for doubling and addition not easily distinguishable in the power traces. Addition is split into two routines to help this, as the code for addition is approximately twice as long as the code for doubling.

Okeya and Sakurai [107] took the approach of using a variant of Montgomery's point multiplication algorithm [101] for suitable curves over fields with odd characteristic. Montgomery method is similar to Coron's double-and-add method, as it al-

ways performs double and add, and holds the two points during the computation. The choice of point to add depends on the value of the bit of the secret exponent d (see Algorithm 10). A method is shown to implement a scalar multiplication resistant to power analysis using a combination of the Montgomery method, secure against SPA, and Coron's randomisation technique.

Algorithm 10 Scalar multiplication using the Montgomery method.

INPUT: $P, d = \sum_{i=0}^{n} d_i 2^i, d_i \in \{-1, 0, 1\}$

OUTPUT: $Q[0] \leftarrow dP$

1: $Q[0] \leftarrow P$
2: $Q[1] \leftarrow 2P$
3: **for** $i = n-2$ *to* 0 **do**
4: $\quad Q[1-d_i] \leftarrow Q[0] + Q[1]$
5: $\quad Q[d_i] \leftarrow 2Q[d_i]$
6: **end for**

Möller [100] noticed that the previously mentioned methods, although efficient, can be implemented only for specific ECs. Namely, ECs suitable for those in [83] (curves with Jacobi form) have to have a group order that is divisible by 4; ECs suitable for those in [69] (curves with Hessian form) have to have a group order that is divisible by 3; and ECs suitable for those in [107] (curves with Montgomery form) have to have a group order divisible by 4. Even though one can use an EC of the required form, it is very likely that people will use the recommended standard. For example, over a large prime field, NIST suggests using a curve of the large prime order and the order of the above mentioned ECs is a small number.

Möller [100] suggested a technique that does not impose limitations to specific chosen curves. In this solution scalar multiplication is done in such a way that point-addition and point-doubling occur in a fixed pattern. The method used randomised projective coordinates and the precomputed multiples of the point P, together with the recoding of the multiplier d in a special signed-digit encoding. A 2^w-ary representation of the scalar d, where $w \geq 2$ is a small integer is given by:

$$d = \sum_{i=0}^{k'} b'_i 2^{wi}$$

where $b'_i \in \{0, 1, \ldots, 2^w - 1\}$. The k' is chosen to be minimal, *i.e.* $b'_{k'} \neq 0$, and for

$i > k'$, $b'_i = 0$. This representation is then converted into:

$$d = \sum_{i=0}^{k} b_i 2^{wi}$$

such that $b_i \in \{-2^w, 1, \ldots, 2^w - 1\}$. The method precomputes points P, $2P$, $3P$, $\ldots$, $(2^w - 1)P$, $-2^w P$ and uses these points to unify the point-addition and point-doubling procedures. This method is more time-efficient than the ones described in [69] and [83]. While the method in [107] has performance advantages, its shortcoming is that it requires specific curves. By implementing the method using randomised projective coordinates and storing precomputed points in extended point representation, the information leakage is limited. Möller's technique is a combination of the Coron's double-and-add and the windowing method. The downfall is that it requires additional table look-ups and a table size of 2^w. This additional memory space is not always available, particularly for devices with scarce memory, such as smart-cards.

Improvements of the scalar multiplications using windowing method were proposed by Okeya and Tagaki [108, 109]. Both of these solutions based their improvements on reducing the size of the look-up table by an appropriate choice of the windowing method. The sizes of look-up tables are reduced to 2^{w-1} [109] and 2^{w-2} [108].

Another solution based on Montgomery multiplication that is applicable not only to Montgomery curves but to any other type curve was suggested by Izu and Takagi [67]. This solution improves the addition chain as well as the addition formula. Izy and Takagi also suggested addition formulae that use only the x-coordinates for the Weierstraß form curve, that are also suitable for parallel execution.

In summary, the countermeasures against SPA on ECCs can be divided into three main categories: (1) indistinguishable addition formulae that use one formula for both elliptic addition and doubling [83, 69, 31]; (2) addition chain that always computes elliptic addition and doubling for each bit [40, 106, 107, 31, 108]; (3) window-based addition chain with a fixed pattern [106, 107, 100, 109, 108].

Resistance to DPA. Although the algorithms mentioned previously provide more or less secure implementations of scalar multiplication against SPA attacks, they do not necessarily provide security against more powerful attacks - DPA attacks. For example, the double-and-add algorithm (Algorithm 7) which is resistant to SPA, does not defend scalar multiplication against the DPA. The step $Q[0] \leftarrow Q[d_i]$ in line 4 can consume slightly different amounts of power, depending on whether the bit d_i is a one or a zero. Even though it is very hard to find such a small difference using SPA, DPA is a powerful

technique that exploits secret information by statistical analysis of power consumption. Coron [40] has demonstrated that this SPA resistant implementation of double-and-add algorithm is vulnerable to DPA attack. Namely, it was noticed that at step j the processed point Q depends only on the first bits $(d_{l-1},\ldots,d_j)$ of d. If the selection function for the DPA attack is chosen to be a specific bit in the memory representation of point Q, then when Q is processed, power consumption will be correlated to this specific bit of Q. In order to recover the bits of the secret scalar d, the point being processed in the card needs to be guessed. If the second most significant bit of d, d_{l-2} was a zero, then point $4P$ would be computed and the power consumption correlated to any specific bit of the point $4P$. Otherwise, if $d_{l-2}=1$, $4P$ is never computed and there would be no correlation between power consumption and the bits of point $4P$. This could be used to recover d_{l-2}. By recursively applying the same logic to successive bits of d all of them can be discovered.

This attack can be extended to any scalar multiplication algorithm. In the case of the addition-subtraction algorithm, the point dP is calculated by computing the sequence of points:

$$a_0P = P \rightarrow a_1P \rightarrow a_2P \rightarrow \cdots \rightarrow a_rP = dP$$

such that $a_i = \pm a_j \pm a_k$, for some $k \leq j < i$, for all $i = 1,2,\ldots,r$. The attack consists of guessing the a_i's, starting from $a_0 = 1$ to $a_r = d$. At step $i \geq 1$, the set A_i of all possible $a_i = \pm a_j \pm a_k$, $k \leq j < i$, is guessed. Now, for each $a_i' \in A_i$, power consumption is correlated to the point $a_i'P$, and the peak can be observed only for $a_i'P$ which was actually calculated.

Coron [40] proposed the following countermeasures to prevent DPA attacks:

- *Randomisation of the private exponent d.*

  ```
  Let #ε be the number of points on the curve.
  Select a random number k.
  Compute d' = d + k#ε.
  Compute the point Q = d'P.
  ```

  Since $\#\varepsilon P = 0$, then $Q = dP$. Now, in every calculation a different d' is used to calculate $Q = d'P$.

- *Blinding the point P.* Point P is blinded by adding a random variable R, for which S, such that $S = dR$, is known. Values $Q' = d(P+R)$ and $Q = Q' - S$ are calculated. The value of R and S is updated for each calculation using formulae:

$$R = (-1)^b 2R$$

$$S = (-1)^b 2S$$

  where b is a random value. Values R and S can be stored outside the card, and refreshed for each calculation, in this case, each time a different Q' is calculated.

- *Random projective coordinates.* This approach consists of randomising projective coordinates of point P:

$$(X,Y,Z) = (\lambda X, \lambda Y, \lambda Z)$$

  for some random $\lambda \neq 0$, and calculating $Q = dP$. Projective coordinates (X,Y,Z) of point $P = (x,y)$ are given by:

$$x = \frac{X}{Z},\ y = \frac{Y}{Z}.$$

Okeya and Sakurai [107] criticised Coron's [40] first countermeasure against DPA as not secure, since it fails to break the dependencies between the executing procedure and the secret key. They also criticised the second countermeasure and claimed that it does not manage to randomise the expression of computed objects. As for the third countermeasure, they claim that careless implementations of this countermeasure are vulnerable to the attack, but the countermeasure is otherwise safe.

Hasan [61] proposed a countermeasure to resist DPA attack on ECCs based on Koblitz curves. The underlying principle is that d is randomly changed each time it is used in the cryptosystem, so that the averaging out technique, used in DPA, does not converge to an identifiable differential signal and the DPA attacks are expected to fail. The major idea was to change d to a pseudo-random number (at a reasonable cost) and still provide the correct result. The suggested masking techniques are summarised in [61].

Joye and Tymen [70] proposed to protect ECCs from DPA by transforming the curve through various random morphisms to provide a non-deterministic execution of the algorithm. In this solution the ECC execution of some operations in the group of a curve E, is transposed to a curve $\phi(E)$, where ϕ is a random isomorphism. The rich algebraic structure of the elliptic curves enables a number of possible choices for such

isomorphism. Another way to achieve the security against DPA is to randomise the base point by using the isomorphism of the finite field. A specific method for ABC curves is also suggested.

Ciet et al. [38] suggest a method that is valid for ECs over fields of any characteristic. This countermeasure combines the point-blinding and randomised scalar multiplication mentioned earlier, into a unique algorithm, in a way which benefits from the computational speed-up of the method. This method is viewed as two-dimensional generalisation of Coron's [40] method.

Ha and Moon [60] suggested randomising the NAF algorithm to resist DPA and employ a simple SPA-immune system. The NAF representation of the secret exponent in the form of the addition-subtraction chain can reduce the number of operations that need to be performed compared to ordinary binary representation. As the result of randomisation, the signed scalar representation is not in the NAF form anymore. Resistance against SPA comes from the fact that the attacker cannot distinguish addition from subtraction. The authors also suggest a countermeasure against SPA. Their SPA-immune addition-subtraction scalar algorithm makes the power consumption independent of the secret key. The authors notice the importance of building a good SPA countermeasure, as any SPA weakness completely destroys the DPA countermeasure.

It is generally thought that the ECCs are easy to protect against DPA, using randomisation over an implementation that is already secure against SPA. According to these countermeasures, an inexpensive and secure way to protect the scalar multiplication against DPA is achieved either by using (1) random projective coordinates, (2) random elliptic curve isomorphism, or (3) random field isomorphism. Goubin [56] has shown that for many ECCs such a protection against DPA is not sufficient. It is shown that for a large class of ECCs the power analysis still works even if one of the three mentioned techniques (together with SPA countermeasure) is implemented. This attack is based on the chosen message. The author points out that this result means that the randomisation of the secret scalar should be performed more extensively and this problem should be taken seriously. Also this attack shows the importance of blinding the message before it enters the scalar multiplication, as it would prevent the attacker from choosing the message for this attack.

3.5.8 Hardware countermeasures

In general software countermeasures are easy and inexpensive to implement, but are not applicable for each cipher and are still susceptible to higher-order DPA attacks or signal processing analysis. Hardware countermeasures, in a manner similar to software countermeasures, focus on destroying the correlation between the power measurements and the values of the secret key. Another target of hardware countermeasures is the alignment of operations in power consumption curves, an important property used by DPA. Some of the hardware countermeasures proposed in the literature are summarised next.

3.5.8.1 Power randomisation

Interleaving random computations into the execution of the cryptographic operations has been often suggested as a defence against DPA. Daemen and Rijmen [46] proposed *power randomisation* generated by a hardware module that is built into the chip and adds noise to power consumption. Power randomisation reduces signal-to-noise ratio of the attack and as such makes the attack more difficult. The authors notice that a way to prevent (or at least seriously complicate) the attack is to minimise the correlation between the data and the power consumption. If the cipher is implemented in software, then this can be achieved by modifying the ALU hardware so that all instructions that operate on sensitive data are balanced.

3.5.8.2 Random process interrupts

Some of the first countermeasures against DPA used *random process interrupts* (RPIs). If the encryption operation is interrupted at random times with random operations, then the times at which critical operations (those that manipulate the secret key) are carried out, will vary from encryption to encryption. Instead of executing all the operations sequentially, the CPU interleaves the execution with dummy instructions so that corresponding executions do not match because of the time shifts. Power traces collected from the devices protected using this countermeasure will not be aligned with respect to operations that the device is performing. This is one of the basic strengths of the straight-forward DPA attack. As a result, spikes which would normally appear thin and tall in the differential trace, now appear shorter and are smeared across the trace. This method increases the amount of noise and should therefore result in the greater number of power traces necessary to mount some version of the DPA attack.

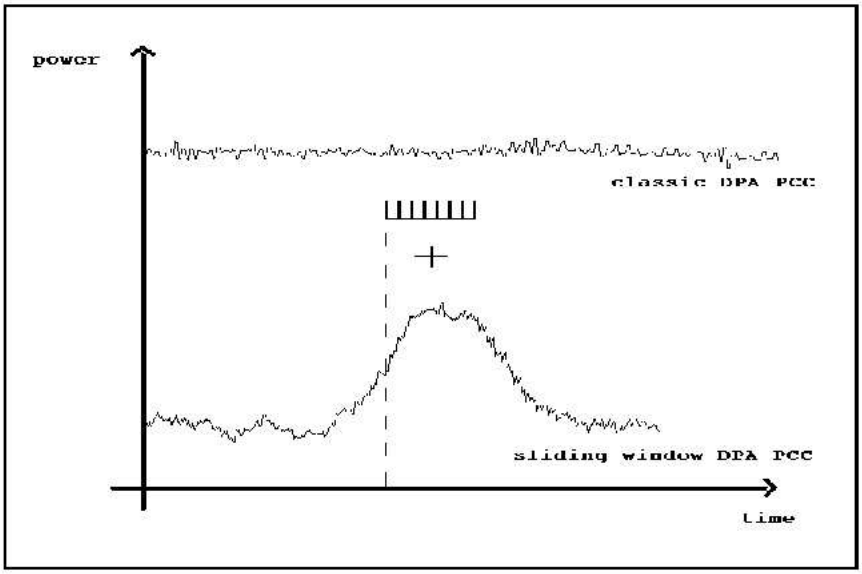


Figure 3.6: The integration operation of the SW-DPA technique [39].

Clavier et al. [39] showed that the approach of introducing dummy computations and randomising the noise is not effective enough. One of the important properties of DPA attacks is that operations are lined-up. In the presence of RPIs the peaks are smeared across the differential trace, which has a desynchronisation effect and can be considered as adding noise. The authors showed that RPIs do not make the DPA attack infeasible, but instead they just considerably increase the number of PCCs necessary for a successful attack. As the spike's amplitude is distributed over a certain number of consecutive cycles, the technique called *sliding window DPA* (SW-DPA) was used in order to restore it (Figure 3.6). The method has two steps: (1) the classical (Kocher-style) differential curve is obtained, (2) PCCs are integrated by adding points from k consecutive cycles from the differential trace obtained in the first step. The authors also show that the number of experiments required increases with a factor of k, *i.e.*, for the spike to be identified the number of power readings must be equal to $N' = kN$, where N is the number of power readings necessary to perform the attack without the RPI countermeasure.

3.5.8.3 Detached power supplies

Shamir [124] proposed decorrelating the external power supplied to the smart-card chip from the internal power consumed by the chip, using so-called *detached power*

supplies. This method uses so-called “air-gap” to decorrelate the power supplied to the card from the power consumed by the card. The design uses two capacitor as the power isolation elements. During half of the time the first capacitors is regularly charged by the external power supply and the second capacitor is irregularly discharged by supplying power to the smart-card chip. During the other half of the time, the roles of the two capacitors are switched. With this method the smart-card chip is always powered by at least one capacitor but the external power supply is never directly connected to the internal chip. Messerges [94] pointed out that this is an easy to implement and simple technique, it has been observed that the signal can be still leaked through other means.

3.5.8.4 Non-deterministic execution

Removing the correlation between features in the DPA profile and the algorithm source code makes retrieving useful information from the power traces significantly harder. A commonly proposed solution for achieving this, is to introduce a level of randomness and non-determinism in the execution of the cryptographic algorithm. May et al. [91] proposed a new processor design to counteract side-channel attacks, called *non-deterministic instruction stream computer* (NDISC). Their design introduced simple additions to a processor with either single or multiple execution units, which allow randomised execution of instructions on an instruction-by-instruction basis. Like a super-scalar processor, a set of independent instructions is selected and executed at random. On the other hand, unlike a super-scalar processor, those instructions are not executed in parallel, but out-of-order on an instruction-by-instruction basis. In this way, correlation between different runs of the same program is removed, thereby making the DPA attack more difficult. This technique, called *instruction descheduling*, creates a level of non-determinism. There is a block of logic that determines conflicts between instructions, resulting in a set of instructions that is executable. The instruction is considered executable if it does not depend on any result that is not yet available and does not overwrite any data that has yet to be used by other instructions yet to be executed or currently executing. Two techniques for increasing non-determinism were introduced:

1. *Multi-source reduction operation*, which is applied on operations that are both associative and commutative and executed on the set of data. Typical examples are multiplication and XOR, used in DES and RSA.

2. *Ignore-depend technique*, which introduces two extra instructions: `IGNORE` and `DEPEND`. Nesting a set of instructions between `IGNORE R` and `DEPEND R`, tells the instruction descheduler to ignore the dependences of the register `R`.

3. *Standard compiler techniques* to increase concurrency by minimising dependences are also employed to increase non-determinism. The result is that the correlations between different runs of the same program are removed, thereby making the DPA attack harder.

May et al. [92] combined this approach with *random register renaming*. Register renaming is a common technique used to improve the performance of processors. The strength of random register renaming comes from the fact that it changes over time. They observed that a large fraction of the power trace is produced by overwriting register values. Each time a register value is overwritten, the power consumption is related to the number of bits that were flipped. They randomly rename the registers in such a way, that every time a register is overwritten, a non-predetermined value is overwritten. This technique can be applied to any type of processor. Renaming works by defining a set of virtual register identifiers and physical register identifiers. The virtual set is used in the instruction set of processors and the physical set is used in the execution time. At any time, each virtual identifier uniquely corresponds to one physical identifier. This binding is unique at any time, but changes over time. When a virtual register is overwritten, a physical register is assigned to the virtual register. In standard processors this improves the performance since this physical register can be used immediately for storing new values, whereas the old register can still hold values that are used by instructions that are still in execution, and thus enable out-of-order parallel execution. Random register renaming expands the ideas presented in the NDISC processor and allows non-deterministic altering of the register-to-register or memory-to-register transfers. In this way, the registers are randomly renamed, and as a result, the program execution is also randomised. The advantage is that program code need not to be modified and neither speed nor power consumption are compromised.

Irwin et al. [66] proposed the addition of a specialised processor pipeline stage to the NDISC, called the *mutation unit*, that is placed just before the execution unit in the processor pipeline. A mutation unit implements the ideas related to adding random calculations to the algorithm that is currently in execution. The purpose of this specialised pipeline stage is to alter the instruction stream in a way that the meaning of the program is maintained but the instructions used to implement it change each

time it is run. This is achieved by using the knowledge of register liveness, obtained at compile-time. The register liveness information allows the processor to alter register usage and mapping patterns at run-time, by describing which registers contain useful values and when they can be safely overwritten. The liveness status of each physical register is stored in the so-called *liveness table* in the mutation unit, and is determined by the compiler or the programmer. The table content is kept valid by two extra instructions: `LIVE` and `DEAD`, which set and clear, respectively, bits in the liveness table. The task of keeping the liveness table valid is achieved by inserting extra instructions into the instruction stream, and is the responsibility of the compiler or the programmer. The compiler uses the combination of the liveness information and the resulting register alternations to decide which registers are alive and which ones are dead. The instances of `LIVE` and `DEAD` instructions are then inserted into the instruction stream to maintain the correctness of liveness table deduced by the compiler analysis. Using the liveness table, the mutation unit can either delete, insert, or alter instructions from the instruction stream. These operations are implemented using the concept of random instruction generation and random register remapping. Using the liveness table the generation of random instruction is facilitated, as at any time it is possible to generate instructions that target dead registers and sources either dead or live registers. The authors claim that using this technique the meaning of the application remains unchanged, but the power consumption and the running time are variable. Their experiments have confirmed that this technique increases the level of non-determinism and manages to foil the straight-forward attack when performed for a relatively low number (10000) of available power profiles.

3.6 Electromagnetic emission analysis

3.6.1 Introduction

The idea that a processor can leak information by different means was first suggested by Anderson [17]. That the information can be deduced from timing and power consumption was first demonstrated by Kocher et al. [76, 77]. Quisquater and Samyde [113] demonstrated that the electromagnetic emission can be another means of secret information leakage through a side-channel.

Electric current flowing through a conductor induces electromagnetic emanations, and is a source of side-channel information. As the power consumption in a device

varies while the data is being processed, so does the electromagnetic field and one can expect to extract secret information from a relevant analysis.

Quisquater and Samyde [113] introduced *electromagnetic analysis* (EMA) as a more general type of analysis than timing and power analysis. They noticed that the timing attack [76] is limited to mono-dimensional data processing. The differential power analysis [77] used a two-dimensional matrix to visualise the correlation of parameters featured in the attack. Electromagnetic analysis can be treated in a similar manner to the previous cases, but it also holds three-dimensional information linked to the volume [113] (locality information).

3.6.2 Attack details

In the experimental setup for the attack Quisquater and Samyde [113] placed a simple flat coil close to the chip. The coil is sensitive to the electromagnetic field which changes induce the current at the bounds. This current resembles the field radiated by the chip. They have found that all the treatments for DPA are possible with EMA since this analysis includes at least the same information and the electromagnetic radiation is directly connected to the current consumption.

There are two types of EM analysis:

- *Simple electromagnetic analysis* (SEMA), in which information is extracted from a single EM sample. Just as in SPA, if a computation makes use of conditional branches based on secret information, then this can be observed as relative shifts in the distances between major computational structures. If this shift is enough to identify that the branch is taken, then this would be enough to identify the value of the secret information.

- *Differential electromagnetic analysis* (DEMA), analogous to DPA, where the secret information is deduced from a number of power traces using statistical analysis of the data.

EMA attacks became very interesting in cases when this attack was more powerful than power analysis. Research presented in [53, 115] has shown that there are such situations. Gandolfi et al. [53] noticed that in some cases power curves appear to convey no information, as the power does not vary or does vary but in a way that seems to be uncorrelated to the secret data. The power consumption of the device was observed

at the global level and in that case no information can be obtained. However, locality information can be revealed by eavesdropping local electromagnetic emanations. Gandolfi et al. conducted practical experiments on various devices and algorithms and concluded that electromagnetic measurements, although noisy, yield better differential trace than power signals. Signal-to-noise ratio for DEMA was higher than in the case of DPA, and it was easier to identify the correct guess as there were no false alerts due to erroneous peaks. The locality of EMA is pointed out as its advantage. The probes used in the experimental setup can be positioned in the neighbourhood of a region that radiates while the device is executing. Different areas radiate with different intensities and different code dependencies. The most active areas appear to be located near the CPU, data-buses and power supplies. The experimental results have shown that the CPU stands out by radiating the most informative signal. This confirms that the locality of the EMA attack is its major strength over power analysis attack as it allows pinpointing the problematic areas of the smart-cards that leak secret information. On the other hand, the authors also point out that definite advantage of power analysis over electromagnetic analysis is the simplicity of electric measurements.

Agrawal et al. [12, 13] and Rao and Rohatgi [115] also studied leakage of compromising information via electromagnetic emissions. They have shown that these emanations consist of multiple signals, each leaking somewhat different information about the underlying computation. They noticed, that in addition to the direct emanations, EM signals consist of several different signals which are unintentional. These are often the result of the increased miniaturisation and the complexity of modern CMOS devices which results in electrical and electromagnetic coupling between components in close proximity. An example of unintentional EM emanations are amplitude modulated EM signals (present even in the power traces), and angle modulated EM signals. Exploring unintentional emanations can be much more effective than doing the same with the direct emanations, and in some cases can enable the attacks to be carried out.

Various experiments were conducted [12, 13, 115] to illustrate different types of EM emanations and the information that can be deduced from them. The quantitative comparison of these types of emanations and power consumption information leakage has also been conducted. SEMA and DEMA attacks were performed on straightforward implementations of DES, RSA and COMP128 on smart-cards and various other cryptographic devices. Experimental results showed that in some cases EM side-channel attacks are more powerful than other side-channel attacks. The investigation showed that although the electromagnetic side-channel resembles the power

side-channel, there are situations where the electromagnetic side-channel can carry much more useful information than the power side-channel. Furthermore, not only can EM emanations be used to attack cryptographic devices where the power side-channel is unavailable, but they can be used to break power analysis countermeasures. The authors claim that this is due to the existence of a class of so-called *bad instructions* which leak much more information to the electromagnetic side-channel than to the power side-channel. These instructions have to be identified for each smart-card separately. If the bad instruction for a particular smart-card is identified, major countermeasures, suggested in [57, 36, 78], can be defeated. The type of EM analysis necessary for this might depend on the countermeasure and the knowledge of the card details and algorithm implementation.

3.6.3 Countermeasures

Quisquater and Samyde [113] propose that adding noise would be an elegant way for disturbing EMA. If each instruction has a specific electromagnetic signature, then the attacker could build a dictionary for each instruction used in the processor, and that way reverse the part of the code. They propose and discuss several possible countermeasures for preventing EMA, and point out that if the countermeasures are purely hard then the circuit and the countermeasure have to be overlapped. Some of the countermeasures they proposed are listed next.

- *Reduction of the EM field*. Designers can try to block or reduce the electromagnetic field, and that way block the information leakage or reduce it to non-measurable form, by using layers of non-ferro-magnetic metals. In this way, at least one of the two components of the radiation is reduced, which could cause a loss of information.

- *Block the EM radiation.* A simple way to block the EM radiation is to put the device in a Faraday cage. This is realisable but does not guarantee the security as the attacker could still find a way to introduce a reason to leak the information.

- *Design of very low consumption processors*. This is an important direction in micro-electronics research. If the current consumed by the processor is decreased so is the electromagnetic radiation.

- *Asynchronous processors*. The clock in synchronous architectures allows for signals to be resynchronised and reamplified. Therefore, synchronous architectures

maximise the electromagnetic field. The presence of the clock signals also facilitates the process of distinguishing the fundamental signal from the harmonics. As asynchronous architectures do not have a clock, the electromagnetic signal is significantly reduced.

- *Dual-rail logic*. It has been suggested that by using dual-rail logic the electric transitions, which are clearly visible in the electromagnetic traces, will be balanced out. In the dual-rail design each wire is replaced with two wires, and the two significant states are opposite to each other.

- *New architecture designs*. The strength of EMA lies in its locality property. Therefore, in order to counteract the attack, it would be useful to block this principle. The authors suggest that a reasonable way to achieve this is to build a new architectural design in which the locality cannot be exploited. They note that distributed parallel architectures, in which significant instructions are distributed, could block the locality property and help prevent the attack.

3.7 Fault analysis

3.7.1 Introduction

In 1996, Boneh, De Millo and Lipton [30] announced that faults during computation can have severe consequences to the strength of cryptographic systems. They have shown that for many digital signatures and identification schemes faulty outputs, caused by hardware malfunctioning, completely expose the secret key stored in the device. Since then, different implementations of RSA, DES, variety of identification schemes, ECCs and even unknown ciphers have been successfully attacked. These attacks exploit computational errors that occur during cryptographic operations. These errors are usually introduced from outside of the device and tend to flip a register bit in the device, thereby affecting its security.

Boneh, De Millo and Lipton have shown that the secret signing key used in implementations of RSA based on Chinese Reminder Theorem (CRT) can be completely exposed using a single erroneous RSA signature. They have also presented an attack on the RSA implementation without CRT which exposed the key from about 1000 erroneous signatures. Their results also contain effective attacks on identification schemes that expose the secret key from as little as 10 faulty signatures. These results are how-

ever theoretical and the estimates for the number of necessary faulty signatures are based on standard security parameters [30]. Practical results of similar investigations are given in [18, 17, 80, 126].

Most of the previously known attacks on RSA cryptosystems were based on factoring the modulus. The inability of the attacker to factor the modulus is the strength of the RSA algorithm. The fault analysis attacks, similarly to the timing and power analysis attacks, avoid this difficult task. They use the fact that from time to time, due to malfunctioning hardware, an error can occur in the computation. The malfunctioning of the hardware could be introduced in different ways, such as power glitches or temperature changes. For example, the device might transmit a ciphertext or plaintext according to a value of a single register bit. If that bit was flipped (due to for example a power surge) the device would send the plaintext instead of the ciphertext. In this example, the power surge changed the mode of operation, having nothing to do with the underlying strength of the cryptographic algorithm. These results have demonstrated that faults can increase cipher's vulnerability, and that the engineers of cryptographic devices have to take care that the device behaves correctly in the presence of faults, as specified by the FIPS 140-2 standard [11].

3.7.2 Attack details

Boneh, De Millo and Lipton [30] presented fault attacks on several cryptographic schemes. The following sections give details of the attacks on RSA with and without Chinese Reminder Theorem (CRT), DES and other cryptographic systems.

3.7.2.1 An attack on RSA with CRT

Let $N = pq$ be the product of two large prime numbers each $n/2$ bits long. Signing a message $M \in \mathbb{Z}_{/N}$ using RSA, involves computing $S = M^d \ mod\ N$, where d is the secret exponent. The expensive part of this computation is the modular exponentiation. This operation can be speeded-up by using the CRT. Namely, the modular exponentiation can be split into two operations by calculating: $S_1 = M^{d \ mod\ p-1} \ mod\ p$ and $S_2 = M^{d \ mod\ q-1} \ mod\ q$. The final S is calculated as a combination of S_1 and S_2: $S = aS_1 + bS_2$, where $a, b \in \mathbb{Z}_{/N}$ satisfy the following:

$$a = \begin{cases} 1, & mod\ p \\ 0, & mod\ q \end{cases} \quad b = \begin{cases} 0, & mod\ p \\ 1, & mod\ q \end{cases}$$

The values of a, b, $d \ mod \ p-1$ and $d \ mod \ q-1$ can be precomputed to additionally speed-up the computation. The speed-up of using CRT comes from the fact that doing two modular exponentiations using the modulo of the size $n/2$ is quicker than doing one modular exponentiation with the modulo size n. Namely, since the square-and-multiply algorithm for calculating the modulo $M^d \ mod \ N$ requires $O(n^3)$ operations, the speed-up of using two modular exponentiations with modules half the size, is of factor $n^3/(2\frac{n^3}{8}) = 4$.

If exactly one of the computed values S_1, S_2 was faulty, then the attacker who has two signatures of the same message, one correct and the other faulty, can factor N, as described next. Let $\hat{S}$ be a faulty signature. In order to calculate $\hat{S}$ the attacker has to calculate $\hat{S}_1$ and $\hat{S}_2$ first. If only one of $\hat{S}_1$, $\hat{S}_2$ is faulty, it can be assumed without loss of generality that $\hat{S}_1$ is faulty. *i.e.*, $S_1 \neq \hat{S}_1 \ mod \ p$. Therefore, $\hat{S} \neq \hat{S} \ mod \ p$ but $\hat{S} = \hat{S} \ mod \ q$, or in other words:

$$gcd(S - \hat{S}, N) = q.$$

As an improvement to this method, Lenstra [82] noticed that one faulty signature and the message M contained enough information to factor N. Consider the value $M - \hat{S}^e$, where e is the public exponent used to verify the signature. Since:

$$\begin{aligned} M - \hat{S}^e \ mod \ p &= M - \hat{S}_1^e \ mod \ p \\ &\neq 0 \ mod \ p \\ M - \hat{S}^e \ mod \ q &= M - \hat{S}_2^e \ mod \ q \\ &= 0 \ mod \ q, \end{aligned}$$

in order to factor N the attacker only needs to calculate:

$$gcd(M - \hat{S}^e, N) = q.$$

This attack works under a very general fault model, *i.e.*, it makes no difference what type of error or how many errors occurred in the computation of S_1. The only important point is that the fault occurs during the modulo computation of only one of the primes [30].

3.7.2.2 An attack on RSA without CRT

Although the usage of the CRT in RSA can improve performance, not all implementations of RSA use the CRT. This section gives a description of the fault analysis attack

presented in [30], which exploits register faults that occur during the modular exponentiation. Although this attack is not as practical as the attack on RSA with CRT, it nevertheless illustrates the vulnerability of non-CRT implementations [30].

Let N be a RSA composite of length n and d a secret exponent. Further, suppose that the observed implementation of RSA uses the right-to-left repeated square-and-multiply algorithm for modular exponentiation given in Algorithm 11. This algorithm requires at least two registers to store intermediate values of variables z and S. The variable z is used to store the values M, M^2, M^{2^2}, $\ldots$, $M^{2^{n-1}}$ needed to calculate $M^d \ mod \ N$. If a register fault flips a bit of register z, then a factor used in this product and, thus, the produced signature will be faulty. This enables an attacker with an access to the number of erroneous signatures to deduce the value of d. It is important that the fault occurs at a random iteration during the modular exponentiation, and that it flips only one bit of register z.

Algorithm 11 Repeated left-to-right square-and-multiply algorithm for modular exponentiation, which models register faults.

INPUT: $M, N, d = (d_{n-1}, \ldots, d_1, d_0)_2$

OUTPUT: $S = M^d \ mod \ N$

1: $z \leftarrow M$
2: $S \leftarrow 1$
3: **for** $j = 0 \ to \ n-1$ **do**
4: /* *register fault:* $z \leftarrow z \pm 2^w$ */
5: **if** $d_j = 1$ **then**
6: $S \leftarrow S \cdot z \ mod \ N$
7: **end if**
8: $z \leftarrow z^2 \ mod \ N$
9: **end for**

The attacker obtains signatures for a number of messages $M_1, M_2, \ldots, M_l$, until sufficiently many erroneous signature $\hat{S}_i$ are collected. The pairs $\langle M_i, \hat{S}_i \rangle$ are then used to deduce the secret signing key d. Starting with the most significant bits, the attack recovers blocks of bits from the binary representation of d. As an illustration of this technique suppose that during signing of the message M the fault occurred when $j = n-2$. This error affects two intermediate values of register z. Thus, the sequence of values stored in register z is:

$$M, \ M^2, \ \ldots, \ M^{2^{n-3}}, \ \widetilde{M}, \ \widetilde{M}^2$$

where $\widetilde{M} = M^{2^{n-3}} \pm 2^w$, *i.e.*, the fault that occurred in the register z is caused by the bit flip at position w. If $\langle M, \hat{S} \rangle$ is the erroneous signature obtained after this operation, then $\hat{S}$ can be seen as:

$$\hat{S} = M^{\Sigma_{i=0}^{n-3} d_i 2^i} \widetilde{M}^{\Sigma_{i=n-2}^{n-1} d_i 2^i} \ mod\ N,$$

or just using the binary notation, in a simple form:

$$\hat{S} = M^{d_{n-3} \ldots d_0} \widetilde{M}^{d_{n-1} d_{n-2}} \ mod\ N.$$

Next, the following equivalences modulo N can be derived [104]:

$$\begin{aligned}
(M^{2^{n-2}})^{d_{n-1}d_{n-2}} \hat{S} &\equiv (M^{2^{n-2}})^{d_{n-1}d_{n-2}} M^{d_{n-3}\ldots d_0} \widetilde{M}^{d_{n-1}d_{n-2}} \ (mod\ N) \\
&\equiv M^{d_{n-1}d_{n-2}d_{n-3}\ldots d_0} \widetilde{M}^{d_{n-1}d_{n-2}} \ (mod\ N) \\
(M^{e2^{n-2}})^{d_{n-1}d_{n-2}} \hat{S}^e &\equiv M^{ed} (\widetilde{M}^{d_{n-1}d_{n-2}} \ (mod\ N) \\
&\equiv M(\widetilde{M}^e)^{d_{n-1}d_{n-2}} \ (mod\ N) \\
&\equiv M(M^{2^{n-2}} \pm 2^w)^{e \cdot (d_{n-1}d_{n-2})} \ (mod\ N).
\end{aligned}$$

Therefore, with the knowledge of $\langle M, \hat{S} \rangle$ and that $\widetilde{M} = M^{2^{n-2}} \pm 2^w$ for some w, an attacker can exhaust the values of w, d_{n-1} and d_{n-2} until this condition holds, and in that way discover two bits of d. Since $\langle M, \hat{S} \rangle$ is erroneous, $\hat{S}^e \not\equiv M\ mod\ N$ and therefore $d_{n-1}d_{n-2} \neq 00$. Thus, there are three values for the pair of bits $d_{n-1}d_{n-2}$ that need to be considered. This leads to the conclusion that it takes at most $3n$ trials to recover the correct values of $d_{n-1}d_{n-2}$ and w [104].

In practice however, the attacker does always know the value of j^*, at which the register fault might have occurred. With some trials and errors this value would have to be determined as explained in [104]. The interesting part is that identifying the value of $j = j^*$ also reveals the value of bits $d_{n-1} \ldots d_j$. Knowledge of these bits reduces the effort required to identify j^* for other erroneous signatures since there are now fewer unknown bits of d.

3.7.2.3 An attack on DES

It is obvious that the techniques previously described cannot be directly applied to ciphers that do not utilise modular exponentiation. For instance, typical operations of symmetric ciphers use bit-oriented transformations such as AND, XOR and rotations. Biham and Shamir [28] introduced a technique that easily breaks the implementation of DES when it is subject to the same random register faults that Boneh et al. considered.

Their method is a combination of differential cryptanalysis [27] and fault analysis, that was simply termed as *differential fault analysis*.

It was assumed that the DES encryption algorithm is implemented on a smart-card with random transient faults (the type of faults considered so far) in its registers occurring at each bit with some small probability, and that during each encryption only a small number of faults (typically one) appears. Further, it is assumed that the malfunctions are realised as single bit inversions in the registers which store 32-bit values of L_{i-1} and R_{i-1}. These faults can cause erroneous output as they influence the intermediate values during the DES computation. Since $R_i = L_{i-1} \oplus F(R_{i-1}, K_i)$, the only consequence of a single bit error in L_{i-1} is the identical single bit error in R_i. Due to this one-to-one correspondence, the model can be simplified to consider only the errors in register R_{i-1}, in which a single bit of this register is flipped. In the attack, the attacker obtains two encryptions of the same (possibly unknown) plaintexts from the smart-card. One encryption is performed in errorless conditions and computes the ciphertext C, and the other one is performed while the smart-card was malfunctioning, computing the ciphertext $\hat{C}$. It was assumed that the fault occurred only once during the encryption, in round $i = i^*$. The first step was to identify this round. The ciphertext $\hat{C}$ was corrupted by a single bit error in register R_{i^*-1}. From $C = FP(R_{16}L_{16})$ follows that $R_{16}L_{16} = FP^{-1}(C)$ and from $\hat{C} = FP(\hat{R}_{16}\hat{L}_{16})$ follows that $\hat{R}_{16}\hat{L}_{16} = FP^{-1}(\hat{C})$, where FP denotes the final permutation in DES. Further, since $L_{16} = R_{15}$, the attacker also knows R_{15} and $\hat{R}_{15}$. If the fault occurred in round $i^* = 16$, then $R_{15} \oplus \hat{R}_{15}$ reveals the exact bit at which the fault occurred. In addition, $L_{15} = \hat{L}_{15}$. Now the calculation:

$$\begin{aligned} R_{16} \oplus \hat{R}_{16} &= (L_{15} \oplus F(R_{15}, K_{16})) \oplus (\hat{L}_{15} \oplus F(\hat{R}_{15}, K_{16})) \\ &= F(R_{15}, K_{16}) \oplus F(\hat{R}_{15}, K_{16}) \end{aligned}$$

reveals the difference in the two outputs of the round functions. Moreover, this gives the difference in the outputs from the S-boxes, as:

$$P^{-1}(R_{16} \oplus \hat{R}_{16}) = S(E(R_{15} \oplus K_{16})) \oplus S(E(\hat{R}_{15} \oplus K_{16})). \tag{3.13}$$

Since the S-boxes are designed to be non-linear, the influence of K_{16} in Equation 3.13 is not cancelled out and this calculation can be used to reveal the bits of key K_{16}. The difference of the inputs to the S-boxes can be calculated as $E(R_{15} \oplus \hat{R}_{15})$. Differential fault analysis uses differences of input and outputs from the S-boxes to derive the information about the 48 bits of the key K_{16}.

As with the fault analysis of RSA without CRT, it cannot be known at which round the fault has occurred. If the fault occurred at round $i^* = 15$ the analysis would be

similar. Again, $P^{-1}(R_{15} \oplus \hat{R}_{15})$ could be calculated, which in turn reveals which S-boxes were affected by the fault in $\hat{R}_{14}$.

With enough ciphertext pairs $\langle C, \hat{C} \rangle$ all 48 bits of the key K_{16} can be revealed. The remaining 8 bits can be discovered using brute force. Alternatively, the last round of DES could be peeled back using the knowledge of K_{16} and the differential fault analysis could be reapplied to the previous rounds. This technique could be used to attack the triple DES as well. An interesting fact is that Biham and Shamir managed to recover the complete 48 bits of key K_{16} using only 50 to 200 ciphertext pairs [28].

3.7.2.4 Intrusive fault analysis

Biham and Shamir [28] have assumed that during any encryption a fault occurs at only one bit of register R_i and only during one round $i = i^*$. One criticism of differential fault analysis was that this model for faulty behaviour is unrealistic as it does not include more permanent faults. Biham and Shamir, consequently, presented alternative attacks which exploit permanent or stuck faults in hardware registers. In these attacks the attacker was required to physically intrude into the circuitry of the cryptographic hardware and set the values of certain memory cells. Anderson and Kuhn [17] have shown that this is possible and that the circuitry of the embedded chip can be exposed. Biham and Shamir have shown how the DES hardware implementations can be attacked using only the erroneous ciphertext outputs. However, the attack can be facilitated if the correct $\langle plaintext, ciphertext \rangle$ pair can also be obtained (for example, before any damage is done to the hardware) [28].

3.7.2.5 Attacks on elliptic curve cryptosystems

Joye and Quisquater [68] noted that the CRT attacks (described in Section 3.7.2.1) can be mounted against several elliptic curve systems. Biehl et al. [26] also described the fault attacks on elliptic curve cryptosystems. The danger of faults in these systems comes from the fact that the register faults can transform a cryptographically strong elliptic curve into a less strong curve in which the discrete logarithm problem can be solved, and consequently the secret key revealed. Three different types of attack that can be used to derive the information about the secret key when bit errors are inserted, were presented.

3.7.3 Countermeasures

All the attacks presented so far relied on the device outputting the erroneous ciphertext. An obvious and very simple defence against fault analysis would be to improve the implementation of the cipher, so that it does not output the erroneous outputs, by checking the output for correctness. However, this could be costly, depending on the cryptographic system [30].

In the case of RSA, the correctness of the signature is verified using the public exponent e. The cost of checking the correctness of the signature depends on the value of the public exponent e. This value is usually specified to be small (*e.g.* $e = 3$) to exploit this fact. Therefore, the cost of verifying the signature might not be the same as the cost of generating it. In the case when e is large, Shamir [123] proposed the correctness check for RSA with CRT that is less costly than the full signature verification. Namely, he suggested choosing a random, 32-bit long value r and calculating values: $S_{pr} = M^d \ mod \ pr$ and $S_{qr} = M^d \ mod \ qr$. If $S_{pr} \ mod \ r = S_{qr} \ mod \ r$, then the exponentiations were carried out correctly and the signature can be reconstructed from the linear combination of $S_p = S_{pr} \ mod \ p$ and $S_q = S_{qr} \ mod \ q$.

In the case of DES, checking for correctness could be achieved by calculating the same output twice and comparing the results. However, this solution would reduce the efficiency of the implementation by a factor of two. Moreover, this countermeasure does not work in every fault model [104]. For the random transient fault model, there are $32 \cdot 16 = 512$ bit positions where the fault can occur. Therefore, the probability that the same fault occurred in each of the two encryptions is equal to $\frac{1}{512}$, which means that the number of encryptions an attacker must perform in order to obtain the required number of faulty signatures increases by a factor of 512. On the other hand, in the intrusive fault model this countermeasure would fail completely, as the error occurs in the same manner for both encryptions. A more sensible solution seems to use the decryption to verify the correctness of the ciphertext in this case [104].

Bellare and Rogaway [23] proposed the use of *randomisation* technique for preventing fault attacks. In schemes that support randomisation, RSA is applied on $F(M,r)$, where F is some formating function and r is a random string. This technique is equivalent to padding the message M with random bits r before encryption. This should defeat all fault analysis attacks, apart from those that exploit intrusive faults, as it ensures that the signer never signs the same message twice. In this way, given an erroneous signature, the attacker does not know the full plaintext message $F(M,r)$.

However, for DES and other ciphers with small key sizes this might not be feasible since some number of input bits would have to be used for storing the random bits. For RSA, however, this technique has been implemented with small overheads [23].

3.8 Summary

Side-channel analysis, in particular power analysis, have been extremely effective in attacking implementations on simple hardware platforms. Chari et al. [36] noticed that a general countermeasure against DPA would be to modify the code to ensure that the attacker cannot predict any relevant bit in any part of the computation, without making several other run-specific assumptions. However, they also noticed that is unclear how one can perform serious computations if this requirement is to be satisfied, since no bits that depend directly on the data and the key can be manipulated in any cycle. In RSA-like ciphers, for example, this could be done and the actual value could be blinded, but such a structure is unlikely to be present in block ciphers [36]. These authors propose a generic technique to create provably resistant implementations. They assume that the device has reasonable power properties and a source of randomness exists. Their method consists of creating a model for the physical characteristics of the device and then designing implementations provably secure in that model. These implementations resist generic attacks with an *a priori* bound on the number of experiments. In such a setting, the lower bound on the number of experiments to mount a statistical attack can be proved. Other attempts have been made in order to formalise the approach of protecting cipher implementations against side-channel attacks. Coron et al. [42] gave a definition of leakage immunity and presents several leakage detection tests. However, research in this direction is still in its initial stages and substantial effort is required to find more effective and general countermeasures against side-channel attacks [36].

Daemen and Rijnmen [46] suggested that the cryptographic systems can be protected by combination of two approaches. The first approach consists of preventing, or rather complicating the exploration of correlations in the power traces by using techniques such as desynchronisations, software balancing or power randomisation. The second approach is to minimise the correlation between the data and the power consumption, by manipulating both the data and its complement in major operations. The suggestion is that if the cipher is implemented in software, then the hardware can be modified so that all instructions that operate on cipher key and input are balanced. Similarly, the hardware implementation would have to be balanced also. When considering

the possibilities for balancing the authors found that arithmetic operations are problematic to balance. The balancing in the case of other instructions involves manipulation of both the data and its complement and modification of the hardware implementation of circuits so that the outputs also contain the data and their complement.

Chari et al. [37] introduced an interesting class of side-channel attacks termed as *template attacks*. Template attacks have been characterised as the strongest form of side-channel attacks, from the information theoretic point of view [37], because they can extract sensitive information from a far fewer number of power samples than other known side-channel attacks. They claim that these attacks can break countermeasures whose security is based on the assumption that the attacker cannot obtain more than one or a limited number of side-channel samples. These attacks require the attacker to have access to identical experimental device that can be programmed to his willing. The template attacks derive their power from the ability to closely model the noise in the device. This is in contrast to other attack approaches which focus on removing the noise by averaging. The precise models of signal and the noise are referred to as *templates* of computation. Although the possession of identical devices that is used to model the noise is an advantage, if the attacker is not able to control the possible source of randomness in the device, the attack would be less dangerous [37].

On the other hand, due to the wide-ranging impact of these attacks, in areas such as credit cards, mobile phone SIMs and pay-TV, several proposed commercial implementations are described as *ad-hoc* [36] and most of them can be broken using a higher number of experiments or by using smarter statistical methods. As seen in this chapter no generic solutions protect against these attacks. In general this seems to be a difficult problem to solve, and until a generic solution is found this remains an active area of research.

This chapter has presented several ways in which an attacker can use side-channel information to attack well known and well studied ciphers and their implementations. Although some of the techniques are described in the example of well known ciphers, such as RSA and DES, these attacks could be readily applied to other ciphers also. Out of all side-channel attacks considered in this chapter, power and electromagnetic analysis seem to be the most powerful ones. Most software countermeasures, such as blinding and randomisation, have been effective against attacks such as timing analysis, simple power analysis or fault analysis. However, in the case of attacks that exploit the statistical properties of a large number of side-channel information samples, software techniques have been shown to be less effective. Securing ciphers against the

statistical attacks might require the combination of both hardware and software countermeasures.

The next chapter presents the overview of asynchronous architecture design. The overview of the design of the micronet-based architecture, which is used as an inspiration for the network-based architecture presented in this thesis, is also given. The next chapter also presents the related work towards securing asynchronous architectures against side-channel analysis attacks.

Chapter 4

Asynchronous Architectures

4.1 Introduction

Asynchronous design is not new, it was used in the early days of computers when the machines were built using discrete components. With the discovery of integrated circuits the synchronous design approach attained popularity and became the dominant style. The major difference between the two design styles is in the way the timing is managed: in the synchronous case the clock system provides global time reference whereas asynchronous systems operate without the reference to a global clock signal. In the last two decades there has been a revival of interest in asynchronous circuits [33]. This is partially due to serious design and implementation problems that the high performance synchronous systems are facing [93].

4.2 Asynchronous control

Changes on a wire that occur on non-clock signals in synchronous architectures do not influence their functionality. On the other hand, the change on a wire in an asynchronous system, that occurs whenever a transition from one logic level to another takes place, is termed an *event*. The interface between control circuits and the datapath in asynchronous circuits is said to be an *event-driven* one.

Due to absence of centralised control, communication and control sequencing in asynchronous designs are implemented explicitly. Communication is usually implemented through the use of handshaking protocols [122]. These protocols perform local synchronisations, and therefore the control and the control circuits are distributed throughout the system. The correct operation of such a system does not dependent

on meeting strict timing requirements, but is based on controlling the order in which events occur. Due to data-dependent nature of delays, this order in which events occur is usually non-deterministic [105].

The granularity of instruction parallelism in synchronous systems is usually limited to the cycle level, which simplifies the implementation of control. The manner of operation of synchronous architectures is deterministic, and enables the control and datapath operations to run in parallel. The performance of asynchronous circuits largely depends on the organisation of control [105]. In order to exploit local timing variations, asynchronous designs must exhibit fine-grain parallelism. Also it must be ensured that the handshaking and other control signals themselves do not represent considerable performance overheads. These issues influence the design of both circuits and the architecture, and trade-offs are often necessary between parallelism, circuit complexity and the resulting speed of circuits [105].

4.3 Asynchronous circuits

In the past two decades there has been much work in synthesis and analysis of asynchronous circuits, since this class of circuits promises a number of advantages over synchronous circuits. Some of the most often cited advantages are increased modularity, better tolerance to variations in power supply voltages and temperatures, low power consumption, higher performance, and no problems with clock skew [65].

Variable delays in asynchronous circuits can introduce so-called *hazards*. These occur when an intermediate value on the input is presented on the output of logic gates. Potential hazards limit the number of asynchronous circuits that operate correctly without any assumptions about wire and gate delays. According to the assumptions made about wire and gate delays, asynchronous circuits can be classified as follows:

***Delay-insensitive (DI) circuits*:** A circuit whose operation is independent of the delays in both gates and wires, except that the delays are finite and positive, is said to be *delay-insensitive (DI)*. The class of true delay-insensitive circuits that can be implemented in CMOS is very restricted [88].

***Speed-independent (SI) circuits*:** If wire delays in the circuits are assumed to be insignificant or zero, and the correct operation of the circuit does not depend on the delays in any circuit elements, then the circuit is said to be *speed-independent (SI)*. Although it seems unrealistic, this assumption is realistic when applied lo-

cally, to so-called *equipotential regions*, inside which delays are negligible. An example of a scheme that adopts this approach are *self-timed* circuits which were introduced by Seitz in [122].

***Quasi delay-insensitive (QDI) circuits*:** This class of circuits is similar to the first two. They are often characterised as delay-insensitive circuits with *isochronic forks*, which is in practice very close to speed independence [87]. The wires of a set of interconnecting wires are set to form an isochronic fork if the difference between signal propagation delays in those wires is negligible compared to the delays of the gates connected to them [88]. Circuits created using the DI design style augmented with the isochronic fork assumption are said to be *quasi delay-insensitive (QDI)*.

The isochronic fork was introduced by Burns and Martin [34, 87] and is considered to be an essential and the "weakest possible" compromise to true delay insensitivity [88] if such circuits are to be realised in CMOS [112].

***Bounded delay*:** If the circuit operates correctly only when the delays are below some predefined limit, then the circuit is said to be *bounded delay* circuit.

4.4 Communication in asynchronous circuits

Due to the absence of global synchronisation, communication in asynchronous systems is achieved through handshaking mechanisms [122]. Data is passed between modules using a group of wires collectively known as a *channel*. In channels, data usually flows in one direction (*i.e.* channels are unidirectional) between two modules (1) the *sender*, that delivers the data into the channel; (2) the *receiver*, that accepts the data from the channel. If the cause of the transfer is taken into account, the following definition of the two parties participating in the transfer can be made [22]: (1) the *initiator* is the device that initiates the transfer; (2) the *target* is the device towards which the transfer is initiated. Communication provided by a handshaking scheme is decentralised and the only two parties taking part in it are the sender and the receiver. The sender is responsible for the start of the transaction and the receiver responds when it is ready to receive.

4.4.1 Handshaking protocols

Local synchronisations in asynchronous systems are achieved through handshaking protocols. A handshake involves two modules: the *initiator* and the *target*; which exchange so-called *request* and *acknowledge* signals. The target module (the receiver) responds to a received request by sending an acknowledge signal. The initiator module (the sender) starts the communication by sending a request and then waits for the corresponding acknowledge signal to arrive (Figure 4.1). A request signal is used to indicate the wish to communicate, while an acknowledge signal indicates that the communication has been completed. This combination of a request and an acknowledgement signal is called a *handshake*. Consequently, the communication medium between the two modules is called a *handshake channel* [112].

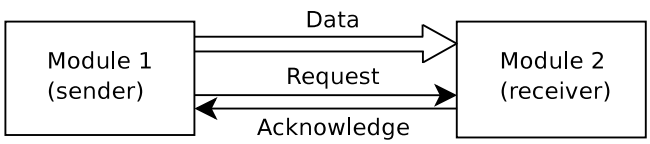
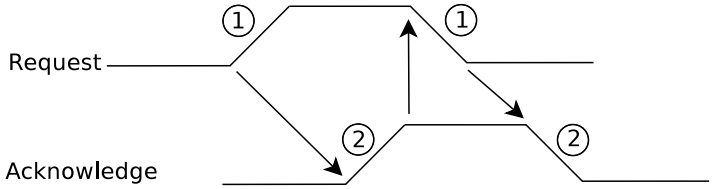
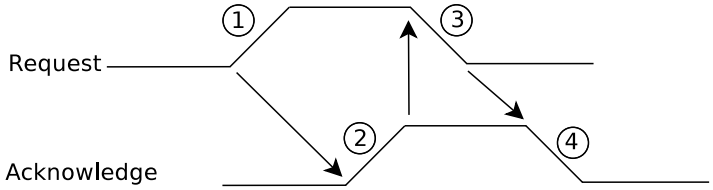


Figure 4.1: Communication using handshake protocols.

If the positive and the negative edge transitions of a signal are treated equally during a handshake, the handshake is called *two-phase* or *transition signalling*. Figure 4.2(a) illustrates a two-phase handshake. In the two-phase handshake, the sender initiates the communication by generating a raising or falling request event that is sent to the receiver. The receiver acknowledges the receipt of the data by generating a rising or falling acknowledge event. If, on the other hand, only the positive edge transitions of a signal are used in the handshake, the handshake is called *four-phase* or *level signalling*. Figure 4.2(b) illustrates a four-phase handshake. The two protocols differ in the number of signal transitions used to complete the communication. In the two-phase protocol, a single event is used to generate a request inducing a single acknowledge event. As a consequence, no meaning is assigned to the levels on the request and acknowledge wires. In the four-phase signalling protocol both request and acknowledge wires are returned-to-zero (RTZ) at the end of each communication. In that case, the levels on the request and acknowledge wires indicate a particular phase of the handshake, and both request and the acknowledge signals must be reset before new data can be transmitted.



(a) Two-phase handshake.



(b) Four-phase handshake.

Figure 4.2: Handshake protocols.

The events that take place in the two protocols can be summarised as follows. In the-two phase handshake, two events take place (Figure 4.2(a)) : (1) the sender requests to transfer the data; and (2) the data transfer is completed. In the four-phase handshake, there are accordingly four events (Figure 4.2(b)) : (1) the sender sends a request to start the transaction; (2) the receiver acknowledges; (3) the sender stops sending the data; (4) the receiver completes the handshake.

The design of the interface circuits between modules involved in the communication is influenced by the choice of the handshaking protocol. It is generally accepted that the four-phase protocol results in smaller implementations of the interface circuits, whereas the fewer number of transactions in the two-phase approach can show power and performance advantages (*e.g.* [105]). However, the second half of the four-phase handshake protocol (phases 3 and 4) can be done in parallel with the computation, which means that, overall, the four-phase protocol can achieve higher performance and lower costs than the two-phase protocol in CMOS designs [50].

4.4.2 Encoding schemes

Handshake protocols are responsible for both control related tasks and data communication. Since the simultaneous arrival of data and request signals cannot be guaranteed under a delay-insensitive model, a technique is required to detect the presence of data.

Detecting the arrival of the signal is only possible if it produces an event on a wire. This means that in the case of data, only those bits whose values have changed could be detected. For this reason a number of encoding schemes has been invented:

***Single-rail encoding*:** In the single-rail encoding scheme [112] one wire is used for each bit of data. It is assumed that the data is available prior to the control signal from the sender, or equivalently, it is assumed that the delay of the request is longer than the delay of the data-bus. This encoding scheme is also known as *bundled-data*. This approach assumes that a safety margin is introduced by delaying the request event, which ensures that the request arrives after the new data values have been established at the receiver's interface, thus conveying the readiness of data.

***Dual-rail encoding*:** In the dual-rail encoding scheme two wires are used to encode each bit of data, whereby n bits of data are represented using $2n$ wires. The transfer of each bit of data involves the activity of only one of the two wires (as illustrated in Figure 4.3), which allows the presence or absence of data to be established. The data is considered to be available by detecting a level (for four-phase signalling) or an event (for two-phase signalling) on one of the two rails for every bit of data. Therefore, a separate signalling wire to convey the readiness of data is not required.

Data	**D0**	**D1**
Clear	0	0
Logic 0	0	1
Logic 1	1	0
Illegal	1	1

Figure 4.3: Dual-rail encoding scheme.

***One-hot encoding*:** In the one-hot encoding scheme 2^n wires are used to represent n bits of information. Each line represents one n-bit code, thus one line transmits n-bits of information and the data validity, *i.e.*, timing information.

***N-of-M encoding*:** Examples of N-of-M encodings when $N = 1$ are dual-rail and one-hot encodings. Systems coded using N-of-M encoding where $M > N$, operate correctly regardless of delays in wires and gates, and are therefore considered

delay-insensitive (*e.g.* [22]). Codes in which $N > 1$ are more complex, and use more than one wire in a group to indicate one of a number of possible codes. These codes can offer better utilisation of the available wires. For example, 2-of-7 encoding can transmit 4 bits of data over 7 wires in a delay-insensitive manner. However, these codes result in larger arithmetic circuits and higher cost of conversion between the coded form and a single-rail form, when compared to 1-of-M codes.

4.5 Advantages of asynchronous design

Synchronous control has been a norm in processor design for a number of years. The improvements in integrated circuit technology have set forth new constraints on the design of synchronous processors. Serious design and implementation problems that the high performance synchronous systems are facing, were partially the reason for the renewal of interest in asynchronous circuits. The main advantages of asynchronous circuits that have motivated researchers to investigate the asynchronous processor design are summarised next.

4.5.1 No clock skew

The global clock signal is used in synchronous systems to regulate their operation, with all system state changes occurring at the change of the clock signal level. Scaling down of physical sizes of both transistors and interconnections, has an affect on physical delays on the wires in a chip, causing some components of the system to see the global clock signal change before others do. This difference in arrival times of the clock signal at different parts of the circuit is known as the *clock skew* [62]. In order to ensure correct operation, the clock period must be increased, which in turn imposes a limit on the maximum clock frequency. This problem can be made less severe through careful engineering of the clock distribution network, but solutions can be expensive in terms of area and power consumption (*e.g.* [22]). On the other hand, the absence of the clock in asynchronous designs completely avoids the problem of clock skew.

4.5.2 Low power consumption

Power consumption is an important factor when designing computing systems. In simple hardware systems, such as embedded systems, the power consumption is important

as the power supply of these systems is usually limited. In larger, high-performance systems, the power consumption can effect the cost of packaging due to both problems of power supply and the cost of cooling. Asynchronous designs have the ability to reduce power consumption by avoiding the following two problems of synchronous design: (1) all parts of the synchronous design are clocked, regardless on whether they perform useful functions or not; (2) a significant amount of power is required just for driving the clock line.

One of the proposed solutions to reduce the superfluous circuit activity in synchronous systems is *clock-gating*, a technique which isolates inactive circuits from the clock. Similarly, *signal-gating* can be used for non-clock signals. Although these techniques can improve power consumption, the main drawback is complicated circuit design and timing analysis [59]. Power consumption can also be reduced by lowering the power supply voltage, but it might be difficult to make transistors operate at low supply.

The event-driven nature of asynchronous circuits naturally avoids any unnecessary circuit activity. The absence of the global clock signal in asynchronous circuits causes power consumption to be more evenly distributed over time, resulting in voltage variances that are smaller when compared to the synchronous case. Namely, transistors fire only when they contribute to the computation, in contrast to synchronous case, where most transistors fire simultaneously at all rising and falling clock edges. Also, an asynchronous system activates only those parts of the circuit which are required and so does not dissipate power in the rest of the circuit that is not being used. These facts lead to the conclusion that asynchronous circuits should generally have lower power consumption than synchronous circuits.

4.5.3 Average-case instead of worst-case performance

Components in the asynchronous design do not have to wait until after they complete a transaction, but rather, if requested they can proceed to the next operation. This characteristic of asynchrony is often cited as one of its main advantages over the synchronous designs. The speed of asynchronous circuits thus depends on the *average* speed of the components, whereas synchronous designs are optimised for the *worst-case* conditions [33]. In synchronous systems, the clock period, and therefore its maximum frequency, is limited by the operation that takes the longest time to complete. This is determined by the slowest component, its slowest operation, its worst-case

data inputs, and worst-case operating conditions (such as supply voltage, temperature and transistor speed). The worst-case combination is, however, typically met very infrequently [22]. Asynchronous circuit, by not being restricted to run in fixed clock periods, can potentially realise better than worst-case performance whenever all the worst-case conditions do not coincide. Asynchronous circuits are optimised for the *average-case* and the worst-case operations just take longer [33]. The fact that there is no fixed clock period during which the operation must complete, enables the delays to be as long as necessary. Although, this may in some cases be slower than the synchronous clock period, considering that the asynchronous circuits operate at the speed determined by the current operation and therefore are effectively limited by its average (or typical) delay, they are potentially faster.

4.5.4 Improved electromagnetic compatibility

Clock-orchestrated operation of the synchronous designs has for a consequence that the radiated energy emissions of the circuits are concentrated at the harmonic frequencies of the clock. If the circuit includes analog or RF circuits this could cause a problem as the high frequency harmonics can be confused for a proper signal. Attempts to solve this problem suggest varying the clock period in order to spread the radiated energy across the spectrum. However, these solutions can have a negative effect on the performance as the clock period can only be prolonged in order to maintain the correct operation of the circuit (*e.g.* [22]). The asynchronous circuits, on the other hand, produce broadband distributed interference spread across the entire spectrum [52]. Due to the absence of a clock, asynchronous circuits may have better noise and electromagnetic compatibility properties than synchronous systems [132].

4.5.5 Modularity of design

In synchronous designs the improved performance can often be achieved only by increasing the global clock frequency, which usually requires most of the design to be reimplemented. On the other hand, the performance of asynchronous designs can be improved by modifying only the most active parts of the circuit, while obeying the communication protocol on the interface of the redesigned block. A designer of an asynchronous system has an advantage of being able to simply replace one block by another with different characteristics and evaluate performance changes.

4.5.6 Simplified layout and improved robustness

The independence of circuit correctness from the delays on the wires in the asynchronous circuits, causes the chip layout to be simplified. Delay-insensitive circuits are also tolerant to implementation parameters such as fabrication process and transistor scaling (*e.g.* [116]). Asynchronous circuits are also more tolerant to physical environment variations, such as temperature, power supply and fabrication specification. In synchronous systems, these parameters have to be taken into account, since the system must work under worst-case operating conditions. Asynchronous circuits, on the other hand, in general do not have critical timing requirements to match, and may therefore exhibit different timing variations and run as fast as their operating conditions allow. Furthermore, their functionality and correctness will be preserved regardless of the operating conditions.

4.6 Disadvantages of asynchronous design

Despite the aforementioned advantages, the asynchronous designs also have a number of disadvantages when compared to synchronous. These disadvantages serve as a motivation for further research, but may also account for the unwillingness of industry to adopt such techniques.

4.6.1 Design complexity

The operation of synchronous circuits is fully orchestrated by the clock: every processing stage must complete its activity in less than the duration of the clock period. This characterises a simple and so far largely successful scheme. Not having any global timing restrictions poses a problem in the design of asynchronous circuits. It has introduced numerous styles to describe asynchronous circuits, from speed-independent to delay-insensitive [33]. In order to ensure the correct behaviour, these designs have to deal with the sequentiality of signals and avoid hazards and races, caused by non-determinism. These problems present the designers with difficult tasks of specifying the behaviour of the circuit and the values of signals at every moment in time.

4.6.2 Completion detection problems

The lack of timing references in asynchronous circuits makes the completion detection complicated, as the task of determining when the completion signal should be generated is difficult. This should not be too early nor too late as the successor stage might read the incorrect data, or it can reduce the performance. This is usually manifested with asynchronous design requiring extra hardware to allow each block to perform local synchronisations to pass the data to other blocks. This added complexity results in larger circuits and more difficult design process.

4.6.3 Testing difficulties

Although a considerable amount of work is put into specification and design of asynchronous circuits, testing them has received relatively little attention [64]. Testing asynchronous circuits has traditionally been considered difficult, particularly if compared to synchronous circuits [131]. A number of aspects of asynchronous circuits have made them harder to test than synchronous circuits: the existence of a large number of state-holding elements, considerably complicates testing; it is not possible to "freeze" and "single step" from a particular state of the circuit, which is commonly done when testing synchronous circuits; there is a large overhead of logic around the asynchronous circuit; and finally, detecting hazards and races is a difficult problem that needs to be solved in the first place [64].

Although these difficulties are largely challenging, there are examples where testing can be implemented as a part of control, with some probability of success. Namely, *self-checking* circuits have an interesting characteristic, seen only in self-timed circuits, that makes them fully testable [64]. During the handshake a request from the sender initiates the process and the receiver must eventually acknowledge back the sender, when ready to receive. If this process is never completed, it is likely that there is a fault in the circuit, and the circuit halts.

Although the methods for testing synchronous circuits are not directly applicable (partly due to the absence of the global clock signal in the asynchronous circuits), some techniques (*e.g.* partial scan [73]) have been adopted for use in asynchronous circuits.

4.6.4 Lack of tools

Years into the design of synchronous systems has yielded a large body of tools for design, simulation, synthesis, routing and verification. Considering that the revival of interests for asynchronous architectures appeared relatively recently, a good amount of tools has appeared for asynchronous design, from high-level languages for hardware description to verification and synthesis [114]. However, these tools are still immature, when compared to the rich infrastructure available for synchronous designs.

4.6.5 Performance measurement difficulties

The task of measuring the performance of synchronous systems consists of simply measuring the length of the critical path and then counting the number of clock cycles. In asynchronous design, this task is not as trivial as the length of time in which a task needs to completewe depends on both the delays and the input data. This implies that the performance measurements are inclined to be variable, and the performance measure is actually an average measure.

Some of the alternative methods for measuring the performance of asynchronous circuits are reviewed in [118]. One of them is MIPS/Watt metric, which measures how fast the processor runs, in terms of power consumption, and shows that the performance and the power consumption can be correlated.

It has been experimentally shown in [89], that the higher speed at higher voltage, and therefore higher power consumption, can be traded for lower voltage with lower power consumption, but also lower performance.

4.7 Pipelines

Pipelining is an implementation technique in which instruction execution is divided into a number of distinct stages. It is by far the most popular technique for improving performance in CPUs, which enables a new task to start before an old one has been completed. Instructions at different stages of execution are allowed to progress concurrently, thus allowing the execution of a number of instructions to be overlapped. Figure 4.4 illustrates a sample organisation of a four-stage pipeline. The throughput of a pipeline is determined by the frequency of results coming out of the pipeline. In the synchronous case, the time needed to move data one stage down the pipeline (the machine cycle) is determined by the slowest pipeline stage. If there are no data

dependencies, the throughput is equal to one instruction per machine cycle. In the asynchronous case, the movement of data between pipeline stages is controlled by the handshaking protocols. The data can move forward only when the succeeding stage is empty and ready. Therefore, the asynchronous pipelines may have variable throughput and have the ability to exploit the actual delays, where synchronous architectures are optimised for the worst-case.

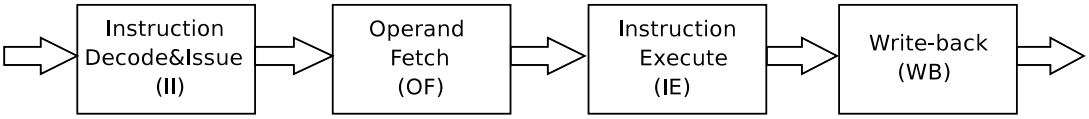


Figure 4.4: Four-stage pipeline.

In a pipeline organisation, as the one given in Figure 4.4, the basic mechanism for naming operands and communicating results is provided through the register bank. Data and control dependencies in the datapath enforce sequentiality in the instruction execution order. Therefore, the pipelined architecture must ensure that the correct order of instructions is preserved. In a non-pipelined architecture, at the beginning of execution of an instruction, all the values in the register file are up-to-date. In a pipelined architecture, however, data values are read and written at different stages, and therefore the immediate correctness of values in physical registers is no longer guaranteed.

Register-based communication can be adapted for use in asynchronous environments with the addition of a *register locking mechanism* [111], which provides the synchronisation required at the register file. The register locking mechanism allows for reads to a particular register to be stalled until all the pending writes have been completed. A simple way to implement this mechanism it to use a single additional bit for each physical register. Typically, during the decode, the destination register of the instruction is locked by setting the lock bit, and during the write-back, the register is unlocked by unsetting the lock bit. Any attempts to read the register while the lock bit is set are stalled. Locks for an already locked register can be treated in several ways. For example, they can be stalled until the register is unlocked, which is a solution that provides correct execution even in the case of out-of-order write-backs [116]. A *lock FIFO* [111] organises multiple lock bits and allows a single register to be locked multiple times and also generates write-select signals during the write-back stage.

4.8 Exploiting instruction level parallelism

Exploiting instruction level parallelism (ILP) in synchronous architectures suffers from control overheads. In addition, the centralised control forces complex designs to operate below their technological best by always assuming worst-case behaviour. On the other hand, the benefit of centralised approach, in which operations complete in fixed delays, is that it leads to deterministic behaviour of the architecture, where the state of the system can be predicted at any time. In contrast, in asynchronous systems operations take only as long as it is necessary and even the execution times of identical instructions may show variations. This characteristic of asynchronous systems can have unfavourable effects on instruction generation and scheduling. On the other hand, exploiting concurrent behaviour is more efficient under distributed control, whereas enforcing synchronisations and sequentiality of operations in asynchronous environment increases the control complexity [116].

Pipelined architectures typically exploit parallelism that is available between stages of the pipeline. However, this does not make the full use of the available ILP, that can be exploited to the fullest by allowing more instructions to execute concurrently. Available instruction level parallelism can be utilised in one of the following two ways:

1. Introducing deeper pipelines that make better use of resources over time, known as *temporal parallelism*. Architectures that exploit temporal parallelism are called super-pipelined architectures.

2. Allowing the operations to be performed in parallel by duplicating resources, known as *spatial parallelism*. Architectures that fetch and execute multiple instructions in parallel are called super-scalar or Very Long Instruction Word (VLIW) architectures.

The following section gives details of the asynchronous architecture that exploits both temporal and spatial parallelism, but belongs to neither of the aforementioned architectural types.

4.9 Micronet

4.9.1 Introduction

A feature common to all of the asynchronous designs is their view of the datapath [116]. In most of the asynchronous designs the datapath is, like in synchronous

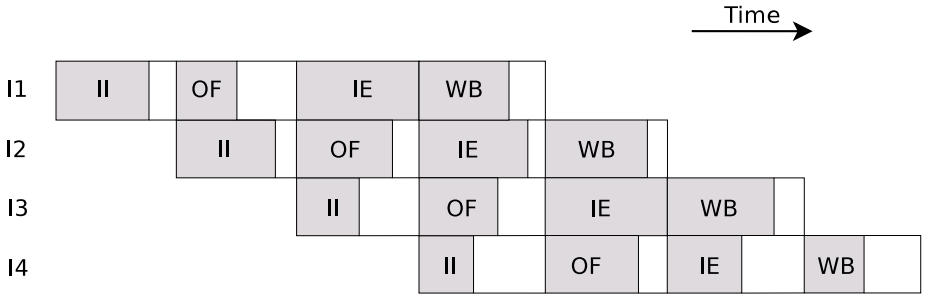
designs, seen as a single linear pipeline. An alternative, as shown in [116], is to implement the datapath as a so-called *micronet*. Micronet is a network of asynchronous communicating pipelines, that can be seen as the generalisation of the Sutherland's micropipelines [129]. It allows a higher degree of fine-grain concurrency to be exploited, both between and within instructions, a feature that is considered to be very expensive to achieve in the equivalent synchronous design [116].

4.9.2 Synchronous, asynchronous and micronet pipeline

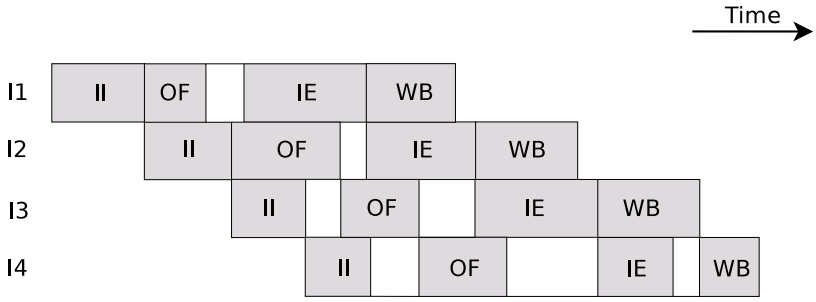
The delay of the slowest stage of the synchronous pipeline defines its clock period. The imbalance between delays of different pipeline stages results in the existence of idle periods which lead to poor utilisation of the physical resources (*e.g.* [116]). Figure 4.5(a) illustrates a four-stage synchronous pipeline datapath that exploits temporal parallelism. The shadows in the figure represent the activity of the stage and the white spaces the idle times.

The performance of the asynchronous pipeline is determined by the average speed of its components. Figure 4.5(b) shows an asynchronous *linear* pipeline, which like the synchronous one, exploits only temporal parallelism, but in a more efficient manner. The instructions spend in each stage of the pipeline only as long as necessary. If the current stage is completed and the next stage is free, the instruction can proceed to the next stage. Micropipelines are representatives of asynchronous linear pipelines. In these pipelines, only different pipeline stages can operate concurrently. The average throughput of an asynchronous pipeline is bounded by the stage with the slowest isolated average throuput. Idle times can be reduced by introducing buffers between stage, but can also cause increase in pipeline latency and reduce performance when compared to synchronous case. In [116] it is suggested that exploiting spatial parallelism might be a way to reduce the number of buffers required to maintain isolated average performance.

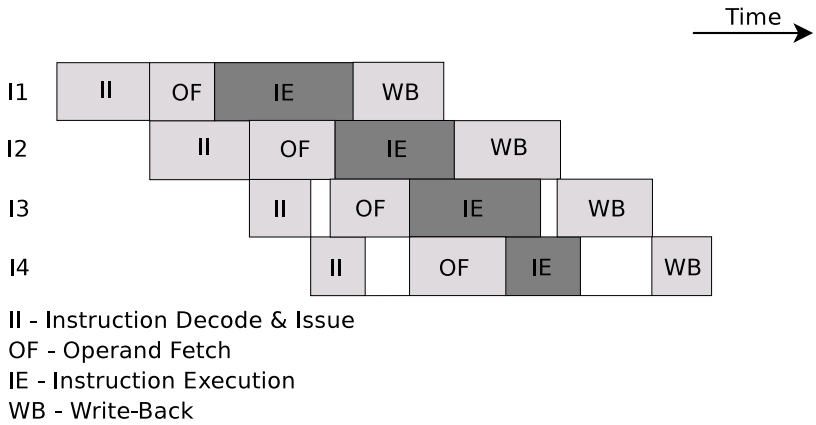
Figure 4.5(c) shows an asynchronous datapath which exploits both temporal and spatial parallelism. The figure shows an asynchronous *non-linear* pipeline, the so-called *micronet*, in which stages from different instructions can overlap. The datapath is no longer modelled as a true pipeline, but rather as a network of communicating stages. In micropipelines, although each stage may consist of a number of different resources, generally only one (or a subset) of them can be active at any time for a given instruction. In case of the micronet, successive instructions which utilise different



(a) Synchronous pipeline.



(b) Asynchronous (linear) pipeline.



(c) Micronet pipeline.

Figure 4.5: Pipelines.

resources within a stage are able to execute concurrently. In the example illustrated in Figure 4.5(c), the instruction execution (IE) stage has two concurrent resources and two instructions take place concurrently. Instructions can share resources in any of the stages, which enables further resource utilisation [118]. For example, while an instruction is stalled waiting for an operand on one bus, another instruction could use the other bus to fetch its operand. The amount of spatial parallelism which can be achieved in practice depends on the relative delays of the functional units within the datapath [116].

4.9.3 Micronet as an asynchronous network of micro-operations

A micronet-based processor exhibits fine-grain concurrency both at spatial and temporal level [21]. In a synchronous design the centralised control forces each instruction to go through all the stages of the pipeline regardless of the need to do so, with the time spent in each stage being determined by the clock period. The execution cost is therefore determined by worst-case delay of the slowest stage. The similar is true for micropipeline-based datapath [51], except that the cost is this case determined by the actual delay of the slowest stage. In the micronet, on the other hand, considering that the control is decentralised and distributed amongst the communicating functional units which operate concurrently, the ILP can be achieved implicitly without extra cost [116]. The micronet datapath is modelled as a network of functional units, in which each instruction visits the appropriate functional units, and for as long as is necessary to execute a particular part of the instruction. Different program instructions may execute concurrently within the same stage. There are several instructions active at a same time, and they compete for functional unit resources, and may even overtake each other. In order to maximise the utilisation of datapath resources, the micronet architecture is designed to enforce only minimal constraints on their use [105]. This is achieved by breaking each individual instruction into a number of *micro-operations*, each scheduled independently. This allows micro-operations from different instructions to exists in the same logical pipeline stage. An instruction uses only those micro-operations that are required for its execution. This enables other instructions to use the resources within the same stage that are not used. The micronet architecture exposes resources such as register read ports and operand-buses and allows different instructions to use them concurrently. For example, an instruction that has only one operand, and therefore requires only one read-bus, will leave the other read-bus available. The

second read-bus can be used concurrently by another instruction that also has only one operand. Similarly, if an instruction does not need any of the resources within a stage, it will skip it.

In asynchronous pipelines the number of active instructions at any time is bounded with the number of pipeline stages, while in the micronet architecture the number of active instructions is bounded with the number of functional units [118]. Furthermore, the time an instruction spends in a micronet stage is variable. The fast instructions can overtake the slower ones and, effectively, instructions compete for resources. In practice, all instructions do not necessarily have identical execution times and therefore their results may be ready out of program order. In-order write-back is not enforced as it can degrade the performance by stalling the functional units and increasing the instruction evaluation times. In the micronet architecture the strict ordering of instruction completions can be relaxed and that way ILP further exploited. This has an effect of increased utilisation of functional units. Therefore, the micronet datapath offers a finer-level of concurrency than the level offered by micropipelines [21].

4.9.4 Micronet implementations

Micronet asynchronous processor (MAP) architecture was developed at the University of Edinburgh [21, 19, 116]. Rebello [116] studied the conversion of a synchronous processor architecture into an efficient micronet-based architecture. This process was described in a series of refinement steps, each introducing additional scope for exploiting fine-grain parallelism and decentralised control.

The development of scheduling algorithms targeted at micronet processors was studied in [20], where an additional scheme for exploiting information obtained at compile-time was also suggested. This theme of exploiting a close interaction between compiler and architecture was continued in the work of Mullins [105], who explored super-scalar asynchronous architectural designs. Further investigation of scheduling algorithms and problems of optimisations for ILP in micronet-based processor were studied by Sotelo-Salazar [118], who introduced a scheme for scheduling instructions within basic blocks, that minimises stalls cause by data-dependencies and resource contention. The issue of efficient scheduling of instructions with uncertain latencies in the micronet-based asynchronous architecture was also addressed in [118].

Data-forwarding is a technique commonly used in pipelined architectures to minimise the cost of functional unit stalls (due to data-dependencies) by redirecting data

that is being written to the registers, to the waiting functional unit(s). In synchronous ILP designs, the cost of maintaining the correct operation increases the complexity of control which in turn affects the clock period and thus the performance [116]. Micronet datapath design can utilise the existing handshaking mechanisms, together with register locking mechanisms, in order to achieve the same effect but with trivial hardware overheads. Exploiting concurrency in a micronet architecture is aided by the distributed nature of the control strategy and by the fact that the movement of data is controlled locally [116]. In the micronet architecture, data-forwarding is implemented by tagging each result with the destination register identifier. During the write-back stage the tag may be matched to operands currently being fetched. This allows the operand fetch stage to obtain the data from the write-back bus before the result is available from the register file. In such an event a second handshake is sent to register file to cancel the pending read request.

A block diagram of an example RISC architecture developed using a micronet model is given in Figure 4.6. A small token ring provides arbitrated access to a single write-back bus. Write-after-write (WAW) hazards are avoided by ensuring that an instruction's destination register is unlocked before write-back can take place. This way the instruction issue stalls are avoided, and only the write-back stage can be stalled while the destination register is locked. The *go-write* signal shown in the figure is generated at the register file and indicated when it is safe to write-back a particular result [105].

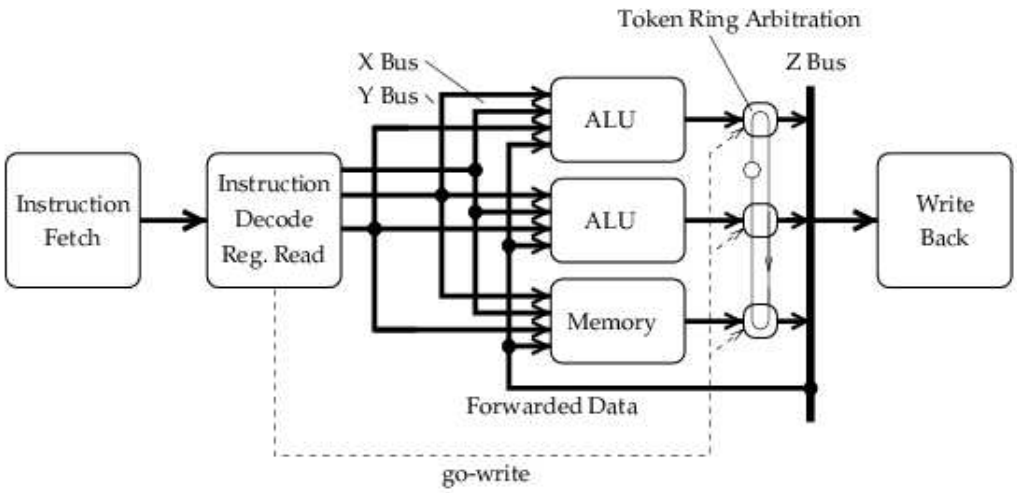


Figure 4.6: Micronet [105].

4.9.5 Summary

The previous sections presented an overview of the basic concepts regarding asynchronous architectures: asynchronous control, asynchronous circuits, and some of the advantages and disadvantages of asynchronous control. In the next section, side-channel analysis of devices implemented using asynchronous paradigm are reviewed.

4.10 Side-channel analysis of asynchronous architectures

Apart from the various design problems that the global clock introduces in synchronous designs, the clock is also seen as a source of information that makes the modern cryptographic devices vulnerable to side-channel analysis. Recently, asynchronous circuits have been suggested as a promising platform for security sensitive applications due to a number of their advantages.

4.10.1 Motivation for using asynchronous architectures for cryptographic devices

Most smart-cards use CMOS technology which consumes power only when the logic state of the chip changes; at other times only quiescent current is drawn. Synchronous architectures are clock-driven; all activity in the chip is driven by an internal or external clock edge and nearly all activity ceases before the next clock edge is due. This results in periodic power consumption curves and the existence of large peaks that correspond to clock signals, which facilitate the power analysis attacks. In contrast, operations in asynchronous architectures are initiated in a data-driven fashion making gate-switching more distributed. Circuits employ handshaking protocols which work as local synchronisation signals. This allows the circuits to operate independently, as the activity in each individual subcircuit only depends on their local handshake signals. The global timing reference in the power consumption signals that is used in power analysis are no longer present and the analysis is more difficult [136].

The absence of the clock in the asynchronous architectures reduces high cycle-to-cycle power variations and clock-driven electromagnetic emissions, which are sources of side-channel information. As the subcircuits are independent, the contribution of each subcircuit to the overall power consumption does not have consistent relative

timings [136]. The power consumption curves (PCCs) of asynchronous circuits appear smoother and represent agregation of many small peaks appearing at arbitrary times according to their handshake signals.

Timing variations of asynchronous circuits make correlating power traces more difficult and signal averaging problematic [49]. This further complicates the statistical analysis of the power traces obtained from asynchronous architectures. Handshakes dominate the behaviour of every part of the circuit, but also the behaviour of one stage might be affected by the behaviour of the previous stage and a number of other stages. This results in power curve overlaps and shifts which raise a problem during the statistical analysis. Also, the cycle time for the same instruction can vary in different periods. Therefore, different power consumption curves might not necessarily have the same periodicity. Correlating power traces and signal averaging is made difficult due to these issues. On the other hand, the different execution times for the same instructions can introduce fine-grain timing noise, which can be a potential source of information for the attacker [102, 136].

4.10.2 Side-channel analysis of dual-rail asynchronous architectures

Recent investigations into the information leakage in asynchronous architectures [102, 136, 103, 49] have shown that asynchronous logic with dual-rail encoding improves resistance to side-channel attacks.

Moore et al. [102] suggested that $1-of-n$ encoded speed-independent circuits could provide a good framework for smart-card devices that are resistant to side-channel analysis and fault injection. They notice that one of the ways to protect smart-cards is to eliminate the data-dependent power consumption. The idea to achieve this is to use $1-of-n$ encoding scheme in which each bit of data is represented with n wires and at any point only one of them is high. Since the power consumption is directly proportional to the circuit's switching activity, a simple binary encoding of data (unlike $1-of-n$), where one wire is used for each bit of data, results in power consumption proportional to the number of state changes. $1-of-n$ encoding consumes constant power to transmit data since only one wire transitions for every symbol encoded. However, it turned out that this is not enough to guarantee the data-independent power consumption. A number of issues, ranging from the layout problems, logic complexity variation and the data-dependent control had to be resolved during the de-

sign [102]. Other ideas towards protecting smart-cards against side-channel analysis include diffusing data-dependent timing, which is already present in asynchronous architectures due to data-dependent execution. $1-of-n$ encoding scheme also allows the protection against fault injection as it allows for the alarm signal to be generated and propagated as the internal part of the data.

Dual-rail ($1-of-2$) encoding is often used to construct speed-independent circuits. The two wires in this scheme are traditionally used to encode three states: *clear*, *logic-0* and *logic-1*. In this design the fourth state is used to explicitly encode the error state, or the *alarm*, as shown in Figure 4.7. This enables dual-rail circuits to deadlock when faults are injected. In order to ensure the data-independent power consumption the return-to-zero (RTZ) signalling was required.

D0	**D1**	**Meaning**
0	0	Clear
0	1	Logic 0
1	0	Logic 1
1	1	Alarm

Figure 4.7: Dual-rail encoding with alarm signal definition.

In the experiments, reported in [49], the design of secure dual-rail XAP asynchronous processor was compared with a synchronous XAP, bundled data XAP, 1-of-4 XAP and 1-of-2 (dual-rail) XAP. Power analysis of secure dual-rail processor revealed that small imbalances in the design of dual-rail gates allowed some data-dependent power leakage to be observed. Although the amount of leakage in the secure dual-rail XAP was significantly lower, when compared to synchronous XAP, this reduction in power leakage was not enough to protect against DPA. Authors noticed that although, in theory, secure dual-rail XAP should be breechless thanks to its dual-rail encoding and RTZ signalling, the manufactured chip showed small weaknesses to DPA. This turned out to be due to implementation tools that were used, which tent to optimise the space, and also the bit encoded into two wires might have one wire longer than the other. This creates the imbalance and produces the power leakage.

Experiments involving electromagnetic analysis (EMA) included similar tests as the power analysis. The collected electromagnetic signals had exploitable magnitudes which allowed for successful DEMA attack to be carried out on the collected reference curves. Again, the imbalance introduced by the design tools has been identified as the

reason for data-dependent EM leakage. The important property of EMA, its locality, has also been demonstrated. The ability of EMA to isolate fine circuit areas has an implication that the balanced design is of great importance. Furthermore, in clocked designs the clock usually adds noise to the EM signal and in asynchronous design this is no longer true and this inconvenience of EM analysis is no longer faced. As a defence against EMA, the authors suggest the use of the top level metal grid which can help mask the underlying activity. The authors also suggest that non-determinism may be used to make data correlation more difficult.

The analysis of the resistance of the secure dual-rail XAP to fault injection is also performed. The suggested design strategy towards the resistance to fault injections was in ensuring that all the attempts to introduce faulty behaviour should result in a deadlock. However, the experimental investigation revealed weaknesses to optical analysis [126] of secure dual-rail XAP processor, which were again the result of some design concessions. Details of these faulty behaviours and the proposed countermeasures are given in [49].

Fournier et al. [49] also notice the importance of the design-time security evaluation and validation. They suggest that the systematic simulation during the design time can reveal data-dependent leakages. Also, the range of physical phenomena that can trigger the faulty behaviour can be simulated. They notice that these simulations can be performed in a way similar to traditional fault analysis simulations.

Yu et al. [136] investigated the security of self-timed ARM-compatible processor, that was designed specifically to explore the benefits of asynchronous design for security sensitive applications. Their secure asynchronous design consists of the following features, which were introduced to improve security: (1) dual-rail encoding, used to reduce data-dependent power consumption; (2) “null” insertion between consecutive data bits, used to ensure that the power consumption of the new data does not depend on the previous data value; (3) return-to-zero dual-rail encoding extended to storage elements for on-chip registers and state-holding storage; (4) design of functional units with data-independent response times. Yu et al. have compared their dual-rail secure design with the single-rail asynchronous design. The findings were that the dual-rail design has shown better resistance to power and timing analysis than its single-rail equivalent. The power analysis experiments have shown that the signal-to-noise ratio of the data-dependent operations was smaller in the secure dual-rail processor than in the single-rail processor. The analysis has also identified the instructions that were still vulnerable to power analysis attacks.

Experiments reported in [136] also included the timing analysis. The authors noticed that a system that takes different time when processing different data values can potentially leak timing information. This information could be compromising and the attacker who could extract the timing variation could also extract the data value. The larger the variation, the easier would be to extract the information. The timing variations in single-rail processor were much larger than in the secure dual-rail processor. Furthermore, processing the same instruction with the same data does not always consume the same amount of time. In synchronous cases this is usually the result of noise, whereas in asynchronous case this is the effect of handshaking protocols. Each handshake is influenced by the neighboring stages and therefore processing the same instruction with the same data can consume slightly different times. This suggests that the self-timed behaviour of asynchronous circuits contributes to making attacks more difficult. Using the secure dual-rail design, the timing variation for some instruction was almost eliminated where for some other instructions it was still present.

Sokolov et al. [127] presented a way to improve the security of (synchronous) dual-rail circuits. The dual-rail protocols implemented in most designs allow switching from *all-zero* (“00”) to a code-word, *i.e.*, “01” or “10” (see Figure 4.7), and back to all-zero. The “00” state is used to indicate the absence of data and is usually called a *spacer*. The authors suggest introducing an additional spacer, *all-one* (“11”) spacer, in the dual-rail protocol and in that way guarantee that all gates switch in each clock cycle regardless of the transmitted data values. The dual-rail protocol now alternates between the two spacers. Reported experimental results show that the data-dependent switching activity is eliminated, but the price of this effect is in increased switching activity and area.

The asynchronous processors show interesting security and tamper-resistance properties. Although the investigations into their security have shown some weaknesses, the leakage of information through power consumption was significantly reduced. The combination of techniques might have to be used when securing these processors against side-channel attacks and fault injection. Another problem with asynchronous design is that secure designs have a cost of doubled or tripled area penalty [49]. This can be balanced against large memory size savings that are possible due to reduced need for software countermeasures in asynchronous design. The authors of [49] concluded that pure balancing on its own is insufficient and suggested that introducing non-determinism in the execution might be a way to remove vulnerabilities introduced in this way.

4.11 Summary

This chapter has presented background concepts of the second of the two major back-grount areas: asynchronous processor design. This information is intended to set the context for the overlap of two design themes: secure-cryptosystem design and asynchronous design. This chapter has also covered basic concepts and design ideas of the micronet-based asynchronous architecture that shares some of the design ideas with the network-based asynchronous architecture introduced in the next chapter. Finally, this chapter reviewed the research done in evaluating the asynchronous designs for side-channel analysis.

The next chapter gives a detailed description of the design of the *network-based asynchronous architecture* that aims at achieving improved resistance to side-channel analysis and in particular differential power analysis.

Chapter 5

Design of the Network-based Asynchronous Architecture

5.1 Introduction

Chapter 3 reviewed side-channel attacks and countermeasures for preventing these attacks that were proposed in the literature. Section 3.5.8 reviewed hardware countermeasures for preventing side-channel attacks. Some of these solutions [91, 92, 66] demonstrated a hardware paradigm for randomising the execution, which assists in making the side-channel attacks more difficult to apply. These solutions exploit the available instruction level parallelism in order to randomise the execution, achieve non-deterministic execution, and hide compromising information in the power traces.

A completely different approach for counteracting side-channel attacks, reviewed in Section 4.10, uses balanced asynchronous logic [136, 103, 49] to balance the power consumption of the device and decorrelate the data being processed from the power consumption. Research in this direction has confirmed that the use of balanced asynchronous circuits is beneficial for protecting cryptographic devices, but perfectly balanced systems, in which the power consumption and the data are completely decorrelated, are difficult to design. The authors of [49] concluded that pure balancing on its own is insufficient and suggested that introducing non-determinism in the execution might be a way to remove vulnerabilities introduced through balanced design.

Due to the absence of centralised control the execution in asynchronous architectures is naturally non-deterministic [116]. In the previous chapter the micronet asynchronous architecture was reviewed. In this architecture the control is distributed in order to exploit both temporal and spatial parallelism. The micronet model features

single instruction issue and out-of-order write-back, and is capable of issuing instructions one at a time, rapidly, allowing different instruction execution stages to execute concurrently.

This chapter concentrates on the design and implementation of the *network-based asynchronous architecture* in which execution is *non-deterministic*. Different runs of the same code have different order of instruction execution, execute in different amounts of time and have different power signatures. This type of behaviour leads to decorrelation of power consumption measurements and, therefore, the increase in complexity and the cost of applying side-channel analysis. The non-deterministic execution is achieved by: (1) exploiting available instruction level parallelism, (2) allowing concurrent execution of both independent and dependent instructions, and by (3) performing data-forwarding using randomised routing of register values through the network. The non-deterministic execution renders the architecture more robust to side-channel analysis, and in particular differential power analysis, as one of its most powerful forms. The focus here is on the design of an architecture in which a number of control issues resulting from data and structural dependencies have to be addressed efficiently at run-time.

5.2 Design goals

Due to the absence of a global clock and centralised control, execution in asynchronous architectures is inherently non-deterministic. This thesis investigates ways in which non-deterministic execution, often cited as a possible way to render the side-channel analysis more difficult to apply, can be further exploited. A major contribution of this thesis is the introduction of a novel architectural idea in which the functional units in the processor datapath are themselves connected as a network, rather than as a linear pipeline. The aim is to decorrelate the power consumption measurements by exploiting the inherent non-determinism of instructions executing in parallel over the network of functional units in which data-forwarding is realised through randomised routing. This investigation focuses on exploring architectural principles which could be beneficial for efficiently and effectively decorrelating power consumption measurements, and not on the low-level implementation of the architecture.

The network-based asynchronous architecture is a network of functional units. One of the most popular ways to increase the performance of processors is to organise functional units into a linear pipeline. A new approach which sees the processor datapath as

a micronet, *i.e.*, network of communicating stages, was first introduced in [21]. One of the fundamental differences between the network-based architecture and the micronet architecture is in the topology of the network. In the network-based architecture the network has the topology of a family of graphs known as *graphs on alphabets*. The interesting question, investigated here, is how the control can be distributed over the network, in order to make the execution both concurrent and randomised, and thus non-deterministic.

In the network-based architecture the distributed control strategy and the fact that data movement is controlled locally aid in exploiting concurrency. Central to the performance of the architecture is the ability to issue instructions rapidly and keep the functional units busy. The instructions are issued in order, at a fast rate. The time an instruction spends in different functional units may vary, and depends on a number of static and dynamic factors [118]. Furthermore, computations may overtake each other leading to out-of-order completion and non-deterministic execution.

In order to improve performance and to achieve concurrent execution, techniques of instruction compounding and data-forwarding are implemented. These are important features for exploiting instruction level parallelism and achieving high performance. A novel implementation of instruction compounding technique is presented, in which data-dependencies between instructions are recognised during the instruction issue stage. This information is then used to initiate data-forwarding operations, which route operand values from one functional unit to another, thus avoiding the register file. This leads to efficient execution that is achieved by distributing both state and control and minimising the need for high-level synchronisations.

The number and functionality of functional units can be changed without influencing the overall behaviour of the architecture. The network itself does not depend on the types of functional units that form it, but the execution of a particular code can be influenced by the distribution of the functional units among different types. The overall behaviour of the network and the architecture depends on the code that is executed, the distribution of different instructions within the code, the ordering of instructions in the instruction stream, and the distribution of different units within the network. For a particular code the distribution of units could be adjusted to match its specific security and performance requirements.

5.3 Overview of the network-based architecture

In most architectural implementations the exploitation of instruction level parallelism (ILP) is limited by data-dependencies and structural hazards. Asynchronous designs that aim at exploiting ILP face a number of control issues that result from data and structural dependencies. These issues need to be addressed efficiently so that the execution does not suffer from control overheads. Even when instruction scheduling techniques for avoiding such dependencies are considered, a good statically generated instruction schedule is not always possible, and techniques are required to resolve them at run-time (*e.g.* [116]). A commonly used solution for reducing stalls caused by data-dependencies is data-forwarding. Data-forwarding is traditionally done by redirecting the data being written to the register bank to the unit that requires that data.

In the design of the network-based asynchronous architecture techniques for exploiting fine-grain ILP and efficiently resolving control issues that result from data-dependencies and structural hazards are explored. The execution paths of instructions without data-dependencies are independent and these instructions are allowed to execute concurrently, at their own pace. There is no communication or synchronisation imposed between different execution stages of these instructions. Communication of execution stages of different instructions in some cases requires a level of synchronisation which is undesirable in asynchronous design. When there are dependencies between instructions (not only between consecutive instructions), these are recognised at run-time during the instruction issue stage, and are used to initialise data-forwarding operations. In this implementation data-forwarding involves communication between units in the network through which the data is routed in a randomised fashion from the producer unit to the consumer unit. In traditional implementations data-forwarding requires synchronisation between the consumer's operand fetch stage and the producer's write-back stage. This synchronisation inevitably slows down the faster stage (the fetch stage) when two dependent instructions are fetched and issued one after another. Data-forwarding, as implemented here, does not require these stages to be synchronised. Furthermore, the consumer does not issue the operand fetch for the register that is obtained through data-forwarding and the producer's write-back stage is completely decoupled from the data-forwarding operations as the latter is realised through the network. Although the dependencies between instructions do mean that the consumer might need to wait for the result to be ready and forwarded, this is still faster than writing the result to the register bank and reading it during the operand fetch stage. In this

case, the operand fetch stage is replaced by waiting for the data to be forwarded. During this period the network routes the data to the consumer unit. The routing module of the unit operates concurrently with the unit's operand fetch-and-lock module. Data-forwarding implemented in this way reduces the register bank traffic, as the number of fetch requests is reduced with the number of forwarding operations. This in turn reduces register bank response times and improves performance. Figure 5.1 compares the execution times of running AES on architectural configurations with and without data-forwarding as the number of functional units is increased. This confirms that data-forwarding in the network-based asynchronous architecture improves the performance. The results presented in Chapter 6 confirm that it also improves security against differential power analysis, which is the primary goal of this investigation.

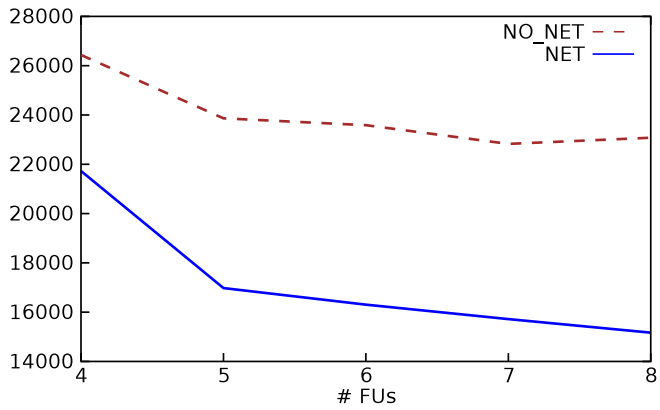


Figure 5.1: Execution times of the architectural configurations with (NET) and without (NO_NET) data-forwarding.

Although the performance gains of data-forwarding are evident (Figure 5.1), these are not the primary focus of this design. The focus is on the variation of execution times that this approach introduces. Namely, the network routes the data using a randomised routing protocol, which is specific for the implemented network topology. The routing consists of two parts: in the first part the data is routed to a randomly chosen unit; whereas in the second part the data follows the shortest path to its destination. The number of nodes a packet visits on its path from the source to the destination varies between routes, which introduces variations in the execution times. Different units may now receive different amounts of traffic that also influences the time required to route the packet through the network. Although one might argue that this is a serious

shortcoming and that the units can become overwhelmed with routing, in practice this is not a real threat because the randomised part ensures that units get balanced amounts of traffic. However, some routes could be more likely than others due to the fact that some units could be utilised more than others. For example, the arithmetic unit may be utilised more than the memory unit, as the former would need to route other network traffic while also receiving a higher amount of its own traffic. However, this can be avoided by having the appropriate number of each unit type in order to balance the computations. In summary, data-forwarding leads to the fulfillment of the first design goal, non-deterministic execution, what makes the implementation of data-forwarding one of the key features of this architectural design.

The fact that data-forwarding is realised through randomised routing is not the only differentiating characteristic of this design. In this implementation data-forwarding is not requested either by the consumer or the producer unit, as seen in some other implementations (*e.g.* [105]). Instead, it is initiated by the issue unit, which does so even before the consumer instruction is issued (further details are given in Section 5.5.2). This enables the producer to start off with data-forwarding and reduce possible stalls in the consumer's operand fetch stage. Further details on the data-forwarding process are given in Section 5.6.

Non-determinism in the execution is further increased by out-of-order instruction completion. Although instructions are issued in-order, there are many instructions in flight at the same time and their progress during the execution is independent. Non-deterministic execution introduces different execution times even for the same instruction with the same input data. The order in which different micro-operations are performed changes from one run to another. Different runs of the same program code execute in different amounts of time and the order of particular micro-operations performed changes from a run to a run. This means that the order in which values are read from or written to the register file also changes from run to run. In practice, all instructions do not have identical execution times and, thus, the results of instructions may be available out of program order. In-order write-back to the register bank is not enforced as this would be a performance overhead, and it would introduce functional units stalls and reduction in performance. Supporting out-of-order execution is beneficial both from the points of view of the non-deterministic execution and the performance, as it allows instructions to proceed at their own pace and complete when the results are ready.

All these natural side-effects of randomised routing help toward decorrelating the power traces. It is especially important that the reference time points (on which many side-channel attacks rely on) are removed and that the signal is smeared across the power trace. Using randomised routing the standard execution routine is disrupted, it changes with time, and is unpredictable. This is the primary goal of this research: to investigate how and to what extent these architectural ideas contribute to decorrelating the power consumption traces of cryptographic algorithms running on the architecture.

5.4 Architectural components

Figure 5.2 shows a block diagram of the network-based architecture with four functional units. The architecture is composed of a Fetch and Branch Unit (FBU), the Issue Unit (IU), a number of Functional Units (FU) each with a Communication Unit (CU), the Register Bank (RB), the Register Bank Arbiter (RBA) and the Memory Arbiter (MA). The architecture can be configured to have different numbers (≥ 1) and distribution of functional units from the following four types: arithmetic, logic, multiplier and memory units.

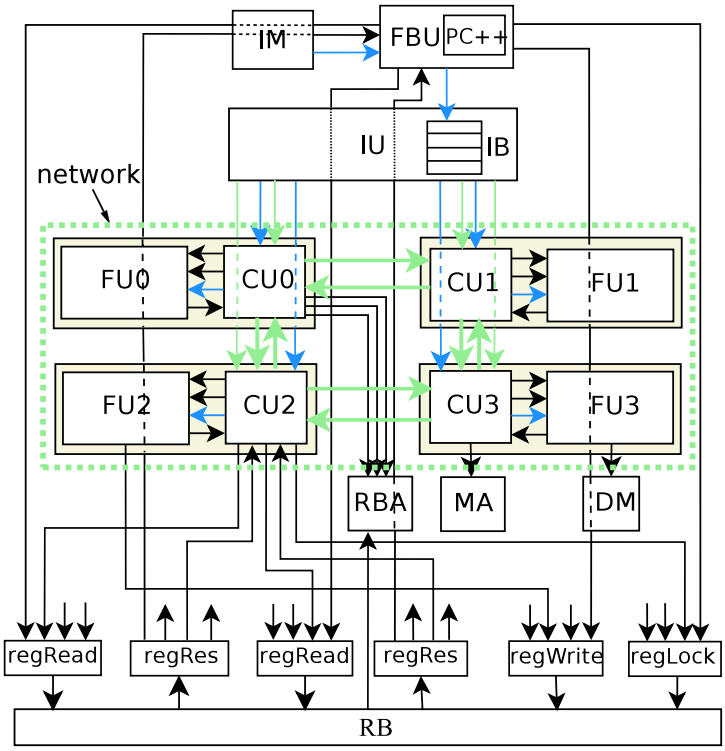


Figure 5.2: A block diagram of the network-based asynchronous architecture with four functional units.

The term *"network-based"* is derived from the property of the architecture that CUs form a network. CUs carry out most of the communication within the architecture, including communication with other CUs, the dedicated FU, the IU, the RB, the RBA and the MA. Each CU is paired with a FU which evaluates the instruction and writes back the result to the RB. The RBA and the MA are responsible for resolving structural hazards caused by data and memory dependencies. The goal behind such an organisation is to decentralise the control and achieve higher levels of non-deterministic execution through concurrent execution and randomised routing of register values in the network.

This section gives a brief introduction to the composite parts of the architecture depicted in Figure 5.2, with more details in the following sections. The major building blocks of the architecture are as follows:

Instruction Memory (IM) and Data Memory (DM): which store the program instructions and data, respectively.

Fetch and Branch Unit (FBU): which fetches the instructions from the instruction memory (IM) and places them into the instruction buffer (IB). The FBU also executes the control transfer instructions. Detailed description of the operation of this unit is given in Sections 5.5.1 and 5.5.5.2.

Issue Unit (IU): which issues instructions to the functional units of the appropriate type in an in-order basis. The issue of an instruction initiates a series of micro-operations in the respective CUs that deal with the given instruction. The task of identifying data-dependencies and initialising data-forwarding micro-operations, is also performed by the IU. Further details of these processes are given in Section 5.5.2.

Communication Units (CU): which represent the communication part of the functional units and form a network. The task of CUs is to perform fetch-and-lock operations and data-forwarding in the network. In this way the control, that is in other architectures (*e.g.* micronet architecture [116]) performed by a semi-centralised control unit, is now distributed between CUs. Detailed description of the CUs and their functionality is given in Section 5.5.4.

Functional Units (FU): which evaluate the instructions and perform the write-back operations. They are effectively isolated from the network and communicate only with the CU and the RB. Further details on the FUs and the instruction set are given in Section 5.5.5.

Register Bank (RB): which consists of 32 registers, two operand read ports (`regRead` in Figure 5.2), one operand lock port (`regLock`) and one operand write port (`regWrite`). Read and write accesses can be performed in parallel if they refer to different register locations. The RB communicates with CUs, the part of the FBU that evaluates the branch instructions (PC++ module in Figure 5.2), FUs, and the RBA. The communication with different functional blocks is controlled through a number of arbiters that sit on its read, lock and write ports, and schedule requests to and from the RB, one at the time, in the order of their arrival. Further details on the functionality of the RB is given in Section 5.5.4.

Register Bank Arbiter (RBA): which is responsible for resolving structural hazards which arise from data-dependencies. In an architectural organisation in which control and sequencing of operations are distributed and decentralised, some arbitration is necessary to ensure that the data-dependencies are preserved and that operand fetch-and-lock operations are scheduled in the correct order. The RBA is an active part of the register locking mechanism and communicates with each of CUs granting read and lock requests to the RB. Full explanation of the operation of this unit is given in Section 5.5.4.1.

Memory arbiter (MA): which is responsible for memory disambiguation. It is implemented as a reorder buffer and ensures that memory instructions access the data memory (DM) in order of their issue. The MA can be implemented in a way similar to RBA, due to the similarity of dependencies when referencing the RB and the DM. However, given that, in practice, there is a need for no more than one or two memory units, the implementation cost can be avoided at a small performance penalty. Further details are given in Section 5.5.5.3.

Network: is formed between CUs. The network topology can be configured to be a binary hypercube or a binary de Bruijn graph. Further details are available in Section 5.6.1.

5.5 Instruction execution

The execution of individual instructions, can be divided into four stages:

- Instruction fetch
- Instruction issue
- Operand fetch-and-lock
- Evaluation and write-back

5.5.1 Instruction fetch

Implementation of certain instructions, and in particular control transfer instructions, can cause the issue stage to be starved of instructions. This could have a damaging effect on exploitation of the concurrency and the efficient utilisation of functional units. For this reason, all the Program Counter (PC) related instructions are executed in the FBU. The FBU is responsible for fetching instructions from the IM and processing control transfer instructions, during which the PC register is updated directly.

The FBU continuously fetches instructions and places them in the instruction buffer (IB) until either the buffer is full, or the fetch stalls waiting for a control transfer to be resolved. If the value of the PC is valid, and there is space in the IB, the next instruction is fetched (Figure 5.3). If this instruction is a PC-related instruction (*i.e.* control transfer instruction), the FBU will execute it, update the value of the PC and fetch the next instruction. If, on the other hand, the fetched instruction is not a PC-related instruction, it is added to the IB, the value of the PC is incremented and the process is repeated. (Note that PC-related instructions are not added to the IB.)

Since the implementation of the FBU does not support branch prediction, some stalls might be possible though they are minimised in this implementation. Unconditional branches are executed by the FBU directly updating the PC. Conditional branches are also executed by the FBU but may cause stalls in the instruction fetch stage, and consequently in the issue stage. It is assumed that the fetch and decode mechanisms are in place and operating so that the IB is always full, except possibly when a branch instruction occurs. This ensures that the issue unit is not starved of instructions.

The task of the fetch-and-branch unit is, therefore, to fetch and decode instructions, and then either execute the control transfer instruction or place the instruction into the instruction buffer. The hardware required for execution of control transfer instructions

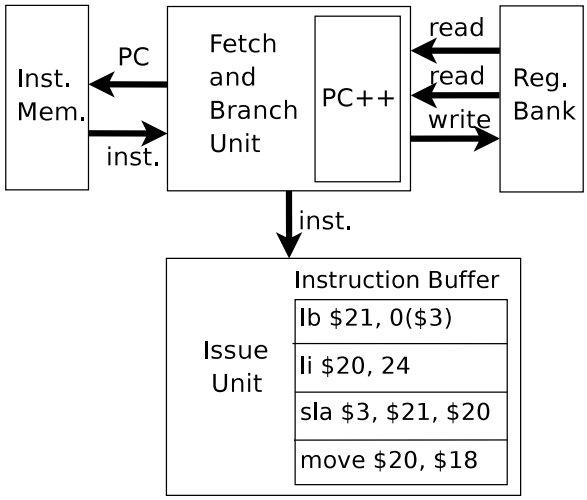


Figure 5.3: Fetch-and-branch unit and the instruction fetch stage.

would need to be similar to that of an arithmetic unit due to the nature of the performed operations. The fetch-and-branch unit is organised so that it is capable of decoding the instruction and consecutively executing the control transfer instruction or placing the instruction in the instruction buffer, depending on its type. The fetch and decode stages for the consecutive instruction are stalled until the control transfer instruction is executed. This organisation assumes that the fetch, decode and PC-update operations are faster then the operations performed in the instruction issue stage, and therefore, the instruction issue stage is not starved of instructions. The functionality of the instruction issue unit is given in Section 5.5.2. The description of the tasks involved in the issue stage leads to the conclusion that the assumption about the fetch unit is realistic, and that the process of fetching, decoding and calculating the next PC value is faster than the instruction issue stage. This process might be slower when there is a control transfer instruction to be executed in the fetch stage, but it is still faster than waiting for the PC-related instructions to be executed in the functional units. Furthermore, as the organisation of the architecture evolved, it became apparent that having the FBU executing the PC-related instructions is beneficial and keeps the IU busy, but is also the simplest design, as the PC value is now updated directly by the FBU which speeds-up the fetch and consequently the instruction issue stage.

5.5.2 Instruction issue

The IU has the task of sequencing the instruction issue operations and initialising data-forwarding operations. In order to increase temporal ILP the issue has to be fast and

attention is paid so that the instructions can be issued in the quickest possible way. The instructions are issued in an in-order fashion to the appropriate FUs. The IU also initialises data-forwarding operations, through which operand values are obtained. The respective CU performs the rest of the required communication to complete the operand fetch-and-lock stage. The control is next transfered to the FU that evaluates the instruction and writes the result back to the RB. The task of the control unit, responsible for sequencing of operations in the micronet architecture [116], is here distributed between the IU, CUs and the RBA. Mutual operation and communication of these units ensures that the operation sequencing is done in a correct manner and in accordance with data-dependencies.

The instruction decode stage is distributed between the FBU and the IU. Each of these units decodes only the portion of the instruction that is needed for the execution in that particular unit. Thus, the FBU decodes the mnemonic part to recognise the control transfer instructions, while the IU decodes the operands. The process of issuing the instruction involves the following four steps (Figure 5.4):

Step 1 – The instruction is assigned to a FU of the appropriate type.

Step 2 – The dependencies between the current instruction and a number of previously-issued instructions are identified .

Step 3 – Forwarding requests (if any) to the producer units are generated and queued for dispatch.

Step 4 – The issue handshake with the CU of the target FU is initialised.

After the instruction is fetched, the next step is to check if there is a unit of the appropriate type that is free and ready to accept the next instruction (Step 1). The IU keeps a `ready` bit for each of the functional units in the architecture. The `ready` bit is set if the respective CU has completed the operand fetch-and-lock stage, and is therefore free and ready to accept the next instruction. This step simply associates an instruction with a functional unit. If the functional units are duplicated, then instructions are assigned in around-robin fashion, subject to the availability of functional units at the given time.

After the instruction is assigned a unit, the next step consists of identifying data-dependencies between the current instruction and a number of previously issued instructions (Step 2). For each of the issued instructions, the information about the destination register and the unit that the instruction was issued to are kept in the *dependency*

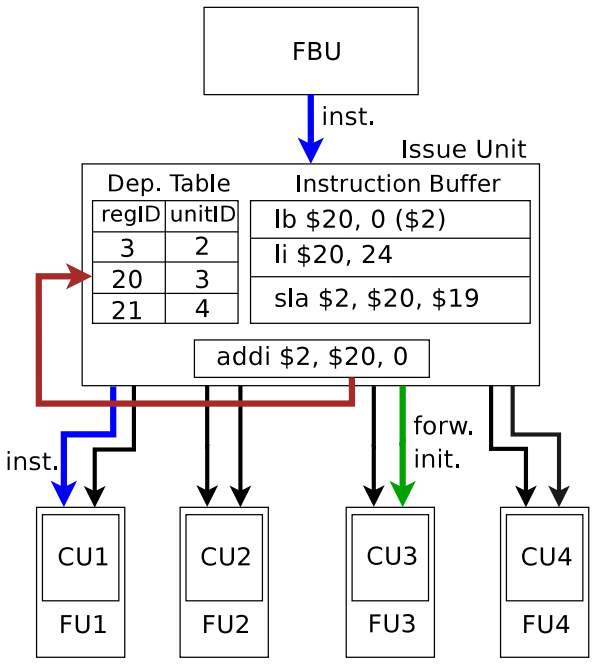


Figure 5.4: Instruction issue.

table. Entries in the dependency table are tuples of the form: ⟨`regID`, `unitID`⟩, where `regID` denotes the resulting register identifier, and `unitID` the identifier of the unit the instruction was issued to. In other words, for each register `regID` in the dependency table, the `unitID` points at the unit which has the most recently generated its value and can forward it. For each of the current instruction's operands, the existence of such a unit in the dependency table is first checked. If a producer unit is found, the operand's *compounding bit* is set. The next step is to generate the forwarding signal to the producer unit (Step 3). If the producer unit is not found in the table, the compounding bit is left unset and the data will be obtained from the register bank.

Before the issue proceeds to Step 4, in which the instruction is issued to a unit, it is first ensured that already generated data-forwarding requests to the destination unit are issued. This precaution is necessary, as the current instruction could overwrite the previous register value, stored in the unit, for which a data-forwarding operation is generated. Finally, when the instruction is issued the information about its resulting register and the unit is added to or updated in the dependency table, while ensuring that there can only be one entry per register in the dependency table.

The small amount of information maintained in the dependency table allows the issue unit to initiate data-forwarding operations. In [105] the explicit dependency information was used to forward the data and the compounded instructions encapsulated the

information about the use of results of instructions within the compound. This information was used in order to enable the producer of a result to initiate data-forwarding operations. In some other implementations (reviewed in [105]) it is the consumer of the result which requests the data from the producer. In the implementation of the network-based architecture, the issue unit initialises data-forwarding operations in the producer units, while the producer units together with the rest of the network realise actual operations.

The data-forwarding initialisation signals are queued and issued one at a time to their destinations. Each of these signals contains a tuple ⟨`regID`, `unitID`⟩, which represent identifiers of the register and the consumer unit that the register value should be forwarded to. The IU does not wait for the handshake to complete before it starts the next data-forwarding initialisation handshake (except when the same unit is in question). Data-forwarding signals are initiated and left to complete at their own speed.

The IU issues instructions one at a time in an in-order manner. The entire process of instruction issue and signalling of the completion of the operand fetch-and-lock stage in the corresponding CU, is done as one handshake[1]: (1) the IU request high means that the instruction is issued, *i.e.*, the data is available at the instruction bus; (2) the CU acknowledges that the instruction is received; (3) the IU lowers the request and the data is no longer available; (4) the CU lowers the acknowledge to signal that the operand fetch-and-lock stage has been completed and the CU is free to accept the next instruction. A high `request` edge during the issue handshake resets the `ready` bit for that unit and the low `acknowledge` edge sets it. In this way, the different edges of the handshake are exploited to signal different events, thus avoiding the need for an additional line to signal the ready state of the CU.

Although the instructions are issued to the respective CUs one at a time, the IU does not wait for the handshake on one issue port to complete before it starts an issue handshake with another CU. Furthermore, although each issue handshake is initiated one at the time and in-order, their completion and, therefore, the completion of the operand fetch-and-lock stage in the CUs is out-of-order. An example of issue and completion orders are shown in Figure 5.5, and were obtained from a simulation of AES algorithm on the architectural configuration with six functional units. (The AU, LU, MULT and MU in Figure 5.5 stand for arithmetic, logic, multiplier and memory unit, respectively.)

[1]The communication protocol in the implementation uses four-phase handshaking bundled data protocol.

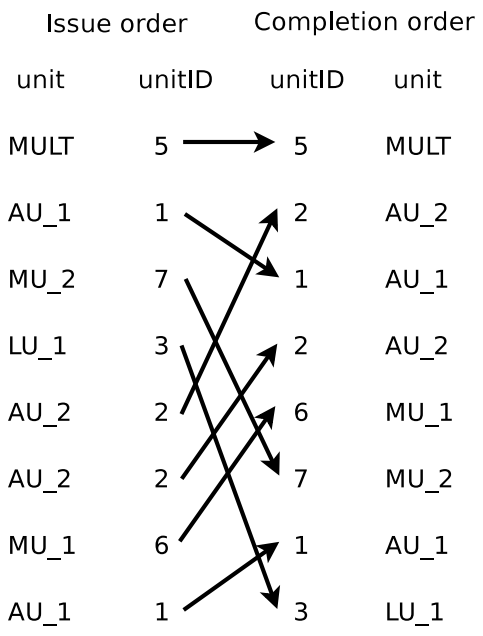


Figure 5.5: Instruction issue and completion order of the fetch-and-lock stage.

The issue stage is not concerned with the availability of operand values before it issues an instruction. Once the instruction is issued, the progress of the operand fetch depends on the availability of the operand values from the register file or through data-forwarding. This allows the issue stage to issue instructions at a fast rate, enabling the functional units to process more instructions concurrently, as the stalls during the issue stage that are due to unavailability of results are completely removed. The only stalls in the issue stage can result from resource contention (all the units of the right type are busy). This problem can be alleviated by carefully balancing the number of units in the particular architectural configuration. This not only improves the performance but, as shown in Chapter 6, also improves the security.

5.5.3 Instruction compounding

This section demonstrates how the explicit dependency information generated at run-time may aid in exploiting fine-grain instruction level parallelism. The ability to append the dependency information to the instruction prior to issue and to use that information in later stages of instruction execution, can help to further exploit asynchrony and solve some of the problems normally caused by data-dependencies.

In contrast to VLIW architectures, in which grouping independent instructions is beneficial, the aim here is to exploit the explicit dependencies between instructions.

This involves identifying groups of dependent instructions at run-time, during the issue stage, and forwarding register values directly between them, thereby bypassing the register file. The small amount of information maintained locally in the issue unit, allows a novel implementation of data-forwarding that is specifically aimed at asynchronous design with highly distributed control. The obtained information naturally flows from older to younger instructions and has no limitations regarding control transfer instructions, seen in some other implementations of the same concept (*e.g.* [105]).

5.5.3.1 Related work

The technique of *instruction compounding* generally consists of identifying groups of dependent instructions in order to forward register values between them. Different implementations of this concept vary in the way in which the dependency information is obtained, compounded instructions are issued, or forwarding requests are generated and handled. Depending on whether the dependency information is generated at compile-time or at run-time, two general approaches, *static* and *dynamic*, can be recognised.

In the static approach implemented in [105], the instruction compounds are formed at compile-time and then scheduled atomically. Data is forwarded only between consecutive instructions within a compound, while the communication between compounds is performed through the register file. Instruction compounds consist of a number of instructions, each being dependent on the previous one. Compounding is restricted to basic blocks to simplify the execution, as compounding instructions from different blocks in this implementation can introduce control hazards. The compiler restriction when forming instruction compounds is that the resulting compound dependency graph is an acyclic graph, as otherwise, it would not be possible to schedule the compounds as atomic units (*i.e.* it would result in deadlocks). Membership of a particular compound is indicated by a single additional bit appended to each instruction at compile-time. If the particular compounding bit is set, then the next instruction is a member of the same compound and requires the result of the preceding one. The consequence of this is that each member of the compound must be scheduled in-order and consecutively, thereby enabling the consumer of a result to be identified. The explicit dependency information appended to the producer of the result allows the producer to make a request to forward its data to the consumer.

The dynamic approach for instruction compounding in [105] is achieved by maintaining a so-called forwarding table within the processor, which records pending results

and a list of potential consumers. The table is used to construct compounds dynamically and free from the restrictions imposed on the compiler. Each instruction accesses the table twice: once to indicate that it has generated a result and again, prior to the generation of a forwarding request, to obtain the location of a consumer. At this point, the entry is reset to indicate that the result is no longer available via the forwarding mechanism. A consumer instruction records its destination in the same table entry and sets its compounding bit. The forwarding request is then generated by the producer after reading the location of its recorded consumer from the forwarding table. As in the static case, dynamic compounding is restricted to only recording a single consumer for each result.

5.5.3.2 Dynamic compounding in the network-based asynchronous architecture

The instruction compounding as implemented in the network-based asynchronous architecture allows a novel approach for data-forwarding. The compounding is performed at run-time recognising not only the dependencies between consecutive instructions of the basic block, but also between the instruction that is currently in the issue stage and a (relatively small) number of previously-issued instructions. This number depends on the size of the *dependency table* that is maintained in the IU. This table records pending or already calculated results and the units that have generated them. Each instruction compound consists of a consumer instruction and one or two producer instructions (the number of source operands is at most two). The dependency table holds pairs, $\langle$`regID`, `unitID`$\rangle$, which identify the register and the unit that has produced the register value.

The dependency table is accessed by each instruction once for each operand in order to locate the potential producer for that operand and to initiate the forwarding operation. Finally, the dependency table is accessed once more to add the information about the resulting register and the unit that the instruction is being issued to. This information is used to construct compounds dynamically, free from the limitation of only forming the compounds between instructions that belong to the same basic block, and free from the limitation that each result can have only one consumer.

The dependency table is updated each time an instruction is issued by adding the information about the resulting register and the unit that the instruction is issued to. The dependency table holds at most one entry per register, pointing to the unit that has the most recently generated its value. When the instruction is issued, an older entry for the same register is updated, if such an entry exists, or otherwise, the information

about the resulting register and the destination unit are added to the table, removing the oldest entry from the table.

Forming instruction compounds is equivalent to identifying units which can forward register values between them. In [105] the compounded instructions are issued as atomic units. The execution of instruction within a compound is mutually exclusive, while instructions from different compounds can execute concurrently. In the implementation of the network-based architecture, the instructions are issued on an instruction-by-instruction basis, enabling instructions from the same compound to execute concurrently subject to availability of results.

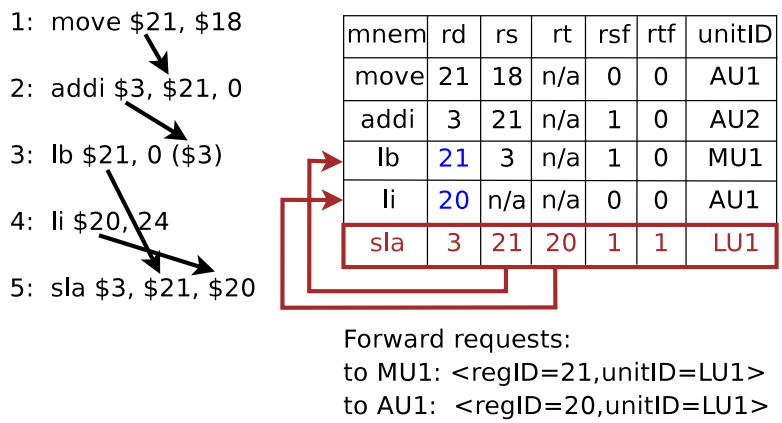


Figure 5.6: An example of instruction compounding.

Figure 5.6 illustrates instruction compounding. Instructions 3 and 2 are compounded, where instruction 2 is the producer for register 3 and the instruction 3 is the consumer. Instruction 5 is compounded with instruction 3, which produces its first operand (register 21), and instruction 4 which produces its second operand (register 20). Here, instruction 5 is a consumer and instructions 3 and 4 are the producers.

The dependency table holds at most one entry per register corresponding to the instruction which has most recently produced its value. If the information about the operand exists in the table, then its value will not be obtained from the RB but through data-forwarding. This is signalled to the CU, which initiates the operand fetch operations, by setting a specific bit in the instruction, the *compounding bit*, associated with each of the operands. (Compounding bits are denoted with `rsf` and `rtf` in Figure 5.6.) If this bit is set, then the operand should not be read from the RB, but its value is forwarded to the unit. The unit, therefore, does not initiate the operand fetch operation but waits for the corresponding producer unit to forward its result. These bits are the only additional information attached to the instruction during instruction compounding.

When the instruction's operand is located in the dependency table, a forwarding request to the producer unit is sent. This signal contains a tuple ⟨`regID`, `unitID`⟩, which denotes the result register and the unit this result should be forwarded to. These two pieces of information are used to identify the register value in the locally maintained *forwarding table* in the producer unit, and to generate the path from the producer to the consumer unit. The forwarding table in each unit contains results that were recently calculated in that unit.

In summary, the key features of instruction compounding implemented in this architectural design, are as follows:

- The dependencies between instructions are identified at run-time, during the instruction issue stage.
- The dependencies are recognised between the current instruction and a (small) number of previously issued instructions. Depending on the size of the dependency table, possibilities for forming compounds can vary, as the information about the producers varies. Namely, the bigger the table, the more possibilities for compounding.
- Information obtained from the dependency table is appended to the consumer instruction in the form of two compounding bits (one for each source operand). Each of the two compounding bits specifies whether the corresponding operand values should be obtained from the register file or through data-forwarding.
- Explicit dependency information obtained at run-time, allows the issue unit to initiate data-forwarding operations in producer units.
- Instructions of a compound are not issued as atomic unit, but one at a time, allowing for a degree of concurrent execution of these instructions.
- Instructions within a compound are not restricted to basic blocks.
- Each result can have more than one consumer.

The proposed compounding mechanism is dynamic and hardware-based. It works by recognising the dependencies between the instruction currently in the issue stage and a number of previously issued instructions (not only consecutive instructions). However, modern optimising compilers try to separate dependent instructions. The dynamic compounding technique implemented here recognises dependent instructions

that fit into a window of the size of the dependency table. Therefore, the impact of optimising compilers could have a negative effect on the implemented dynamic compounding mechanism if the size of the dependency table and the way in which the compiler separates the instructions is such that the dependent instructions exceed the size of this window. For a small dependency table an optimising compiler could make the dynamic compounding technique less effective. However, the proposed architectural design is not a general purpose one and is targeted for specific applications for which the compilation process and the size of the dependency table could be adjusted so that this issue is not problematic.

5.5.4 Operand fetch-and-lock

Instruction level parallelism in the network-based asynchronous architecture is exploited by allowing instructions both with and without data-dependencies to execute in parallel. The mutual exclusion of instructions with data-dependencies is not imposed, although the progress of the consumer is subject to the availability of operands. In this way fine-grain ILP is exploited and the control is further distributed.

As noted earlier, an operand value can be obtained in two ways: through data-forwarding or from the register file. Data-coherence of the register file is based on the *register locking scheme* [111], which has been a common solution in many asynchronous implementations. The locking scheme guarantees correct operation during the asynchronous access to the register file. The mechanism consists of a device that contains an individual *lock* bit for each register, which indicates that a register is yet to be written by a pending instruction. If the lock bit is set, it implies that the register cannot be read from or written to by any subsequent instruction. If the lock bit is unset, then the content of the register can be read by any instruction until it is locked again.

In the micronet architecture [116] the instructions are issued in-order and one at a time, and the operand fetch-and-lock stages are performed by a specialised control unit during the issue stage. Since it cannot be known for how long the register will contain valid data, the issue unit cannot issue instructions that depend on a locked register, *i.e.*, if there is a read-after-write (RAW) or write-after-write (WAW) dependency. Therefore, the issue unit is stalled until the register value of the operand is available, *i.e.*, the register is unlocked. In such a setting the above scheme is enough for maintaining the data-coherence of the register file.

In network-based architecture responsibility for fetching the operand values and locking the result register is transfered to communication units. This allows for faster issue rate but additional care has to be taken to avoid structural hazards. Although instructions are issued in-order, the rest of the communication towards the completion of the instruction execution progresses at different speeds and effectively out-of-order, thereby complicating the structural hazard avoidance mechanisms. Therefore, the order in which fetch-and-lock operations are processed in the RB has to be arbitrated. A specialised arbitration unit, the *Register Bank Arbiter* (RBA) was implemented, which schedules fetch-and-lock operations according to data-dependencies, thus avoiding structural hazards and preserving data-coherence of the register file.

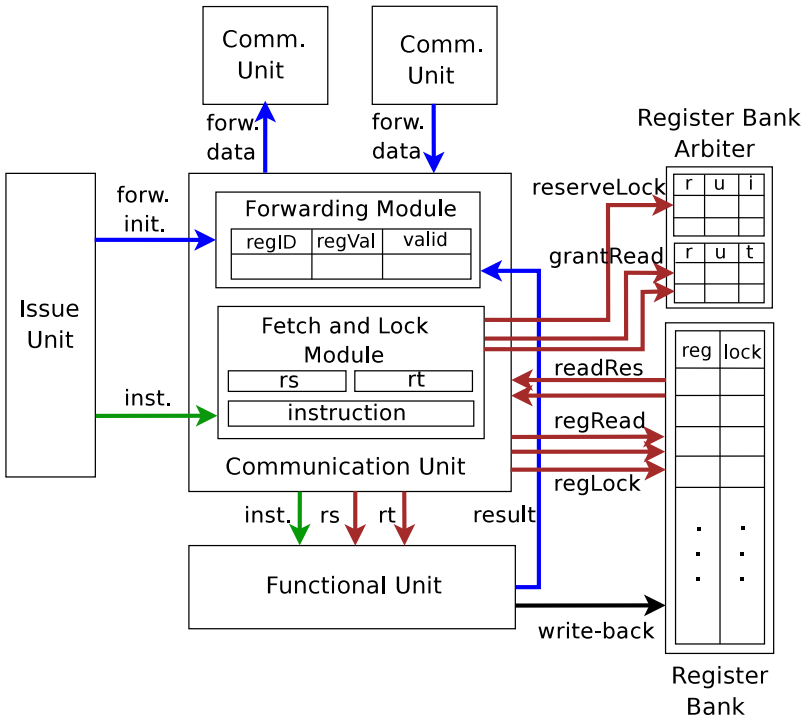


Figure 5.7: The operand fetch-and-lock stage.

Figure 5.7 shows the organisation of communication between the CU, RB, RBA and other CUs during the operand fetch-and-lock stage. As the data can be obtained in two ways, major building blocks of a communication unit are the *forwarding module* and the *fetch-and-lock module*. These two modules are implemented to operate independently and communicate only when the forwarding module receives the operand value. The forwarding module is responsible for performing data-forwarding operations and also maintains the *forwarding table*, which contains the results of a number of previously executed instructions, that can be forwarded on request. If the compound-

ing bit for the operand for the instruction is set, then the operand value is not obtained from the register bank, but through forwarding. The forwarding module performs routing of data as a part of the network. It has a number of “in” and “out” connections to other CUs, depending on the topology of the network, which correspond to the vertices of the network graph. Data is forwarded through the network, from the producer to the consumer unit.

The fetch-and-lock module is responsible for locking the resulting operand and fetching the operand values that are not obtained through data-forwarding. If the compounding bit for an operand is set, then the module does not initiate the operand fetch operation. As the fetch operations for the operands that are being forwarded are not performed, the problem of issuing fetch-and-lock operations in accordance with data-dependencies is this way removed (this namely concerns RAW hazards). This is true under the condition that for the set of instructions that are concurrent in execution all the data-dependencies are captured in the issue stage, which is subject to the size of the dependency table. The maximum number of instructions that could be concurrent in execution, and for which RAW hazards can occur, is equal to the number of CUs. It seems that if the size of the dependency table is at least the number of CUs, then the RAW hazards would be removed. However, the control transfer instructions are not considered during the issue stage. Namely, they are evaluated in the fetch-and-branch unit, and thus do not take part in instruction compounding. Therefore, data-dependencies between control transfer instructions and those evaluated in the rest of the architecture are not captured during the issue stage. RAW hazards due to these instructions remain, and the mechanism for preventing them still needs to be implemented. An obvious solution for removing these hazards would be to move the execution of control transfer instructions with source operands out of the FBU. However, in the current instruction set (covered in Section 5.5.5.1) the number of such instructions is small and the improvement of such an approach would be negligible, if compared to one in which control transfer instructions are not executed in the FBU at all. In addition, the size of the dependency table dictates the size of forwarding tables maintained in each of the CUs. As the size of the forwarding table cannot be too large, if the number of CUs is high the approach for removing the RAW hazards (by making the size of the dependency table at least equal to the number of CUs) cannot be followed.

5.5.4.1 Register bank arbitration for fetch-and-lock operations

Although the instructions are issued in-order, the operand fetch-and-lock operations in the RB could be performed out-of-order. The idea is that the order is imposed only when data-coherence is in danger.

When the instruction is received by the CU the compounding bit for each of its operands are first checked. If the compounding bit for an operand is set, then the operand value is forwarded and the CU does not initiate the operand fetch operation. Otherwise, the CU needs to obtain this operand value from the RB. The RBA arbitrates in this process. This arbitration is realised through granting the CU with permissions to perform fetch-and-lock operations. Therefore, before the CU can issue a fetch or a lock request to the RB, permission has to be obtained from the RBA.

Write-after-write (WAW) data-hazards are prevented by ensuring that the lock requests for the same register are issued in program order. This is achieved by granting instructions with exclusive permissions to lock a particular register in the order of their issue, while locks for other registers can proceed out-of-order. For each register in the RB, the RBA stores the *lock reserve status* (`lockStatus`). This status is binary and can be set to `RESERVED` or `NOT_RESERVED`. The status is set to `RESERVED`, if there is an instruction that has reserved the right to lock the register, but has not locked it yet. Otherwise, the status is set to `NOT_RESERVED`. The `NOT_RESERVED` status means that the previous instruction has locked the register and no other instruction has reserved the right to lock it next. There is no direct correspondence between the lock status of the register in the RB, and the `lockStatus` of the register in the RBA. For a particular register, only one lock request can be issued, and only one instruction can lock the register. If the register is locked, and since an already locked register cannot be locked again, the next lock can be reserved but it will be stalled in the RB until the previous instruction unlocks the register. The `lockStatus` is set (`RESERVED`) when the lock is reserved in the RBA, and reset (`NOT_RESERVED`) after the instruction locks the register in the RB.

This process of locking a register is shown in Figure 5.8. Algorithms 13, 15, 16, 17 and 19 describe the procedures performed in the CU, the RBA and the RB during this operation. The summary of this process follows next:

Step 1 – Obtain the exclusive right to lock the register. After the instruction is received, the CU initiates the `reserveLock` handshake with the RBA (see Algorithm 16). When a request is received, if the register's `lockStatus` is set to

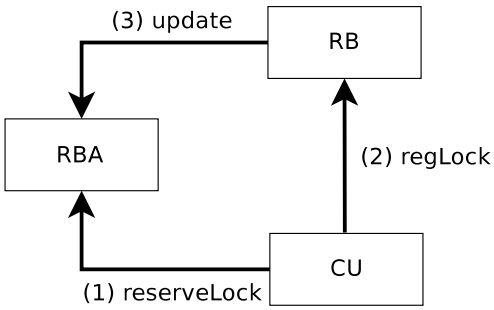


Figure 5.8: The three-step register lock procedure.

`NOT_RESERVED`, then the lock is reserved, *i.e.*, its `lockStatus` is set to `RESERVED`. Even if the `lockStatus` of a register is `NOT_RESERVED`, it does not mean that the register is unlocked, but rather, that there is no other instruction that has reserved the right to lock it in the future, and that the previous instruction has already locked it. The reservation of the lock for a register is signalled with the high acknowledge `reserveLock` signal. If, on the other hand, register's `lockStatus` is `RESERVED`, then the request is indexed and queued, as shown in Algorithm 16 and Figure 5.9.

Step 2 – Send the lock request to the RB. After the lock is reserved, the CU initiates the `regLock` handshake with the RB, which requests the register to be locked, as shown in Algorithm 13.

Step 3 – Lock the register and update its status. If the register is unlocked, the RB locks the register and sends the `update` signal (see Algorithm 19) to the RBA which resets its `lockStatus` (see Algorithm 15). An awaiting lock request for the same register can now be reserved. If the register is locked, the lock request is stalled until it is unlocked.

reserveLock queue

regID	unitID	index
3	AU2	1
21	MU1	2
20	AU1	3
3	LU1	4

grantRead queue

regID	unitID	tag
21	AU2	0
3	MU1	1
21	LU1	2
20	LU1	3

Figure 5.9: Register bank arbiter: `reserveLock` and `grantRead` queues.

If the register is locked by the previous instruction, the next lock request to the same register can be issued. This request will not be able to lock the register until

it becomes unlocked, but no other lock request could overtake it. It is clear that this locking mechanism speeds up the register locking procedure. This is another example of how fine-grain instruction level parallelism can be exploited. At any time, only one instruction can lock a register and one more instruction can have the lock request to the RB initialised although not completed. Once locked the register remains locked until the result is written to it. The write-back of the result will cause the register to be unlocked, awaiting reads to be performed and the pending lock-request to lock the register.

Read-after-write (RAW) data-hazards are prevented by ensuring that the operand fetch request to the RB is issued after the instruction that writes that register has locked it. If the size of dependency table is large enough to gather all the RAW dependencies and perform data-forwarding between instructions with these dependencies, then RAW dependencies can occur only when executing the control transfer instructions which are not included in data-forwarding.

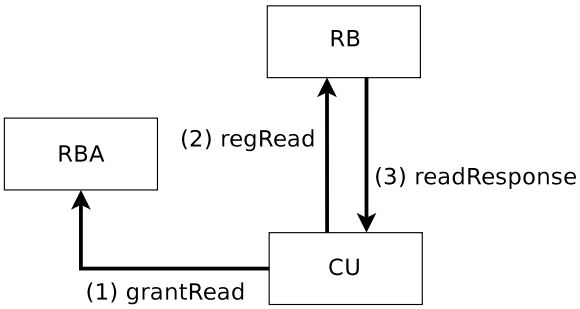


Figure 5.10: The three-step register read procedure.

The three steps when fetching operands are illustrated in Figure 5.10, given in Algorithms 12, 14 and 18, and also be summarised as follows:

Step 1 – Obtain the grant to read (fetch) the register value. If the compounding bit for the operand is not set, the CU will initiate the `grantRead` handshake with the RBA (see Algorithm 12). This read request is granted if the status of the register is `NOT_RESERVED`. Otherwise, the `grantRead` request is tagged with the most recent entry index for the same register in the `reserveLock` queue, if one exists, or with zero otherwise, and queued in the `grantRead` queue as shown in Figure 5.9 and Algorithm 14. The request tagged with a zero is granted after the `update` signal resets the status of the register, *i.e.*, sets it to `NOT_RESERVED` (see Algorithm 15). Each time a lock request is reserved all reads with the same

index have their indices updated to 0 and will be granted next (after the register is locked and its `lockStatus` set to `NOT_RESERVED`).

Step 2 – Send the read request to the RB. After the RBA grants the fetch operation, the CU initiates the `regRead` request with the RB (see Algorithm 12).

Step 3 – Send the read response to the CU. The register value is read from the RB only if it is valid, *i.e.*, the register is written and unlocked. If the register is locked, then the read requests will wait for this status to change (see Algorithm 18).

Although care is taken when issuing read and lock requests so that WAW and RAW hazards are prevented, the order in which they are processed in the RB is still not guaranteed. The hazardous write-after-read (WAR) situation occurs when the subsequent lock request manages to lock the register before the reads are performed. As the register cannot be read unless it is unlocked, the incorrect register value is eventually read. The order in which read and lock requests are processed in the RB depends on the number of these requests and the speed with which the RB can serve them. The RB records the number of register reads performed for each register. Similarly, the IU records the number of register reads that ought be performed before the register is locked (with respect to data-forwarding). This information is attached to the instruction on issue and sent as a part of the lock request to the RB. A simple comparison then prevents WAR hazards. This solution is satisfactory, albeit ad-hoc, given that the validity of architectural principles is being explored in this study.

Algorithm 12 Communication unit: operand `fetch` procedure.

```
1:  /*for each of the operands */
2:  for operand = current_inst.rs, current_inst.rt do
3:      /*if the operand value is not obtained through data-forwarding */
4:      if operand.compounding_bit == false then
5:          /*send grantRead request to RBA */
6:          grantRead[operand].data.regID = operand.regID;
7:          grantRead[operand].req = high;
8:      else
9:          /*send regRead request to RB */
10:         regRead[operand].data.regID = operand.regID;
11:         regRead[operand].req = high;
12:     end if
13:     /*if the read is granted */
14:     if grantRead[operand].ack == high then
15:         /*send regRead request to RB */
16:         regRead[operand].data.regID = operand.regID;
17:         regRead[operand].req = high;
18:         grantRead[operand].req = low;
19:     end if
20:     /*complete the regRead handshake */
21:     if regRead[operand].ack == high then
22:         regRead[operand].req = low;
23:     end if
24:     /*read response received */
25:     if readResponse[operand].req == high then
26:         operand.value = readResponse[operand].data.regValue;
27:         readResponse[operand].ack = high;
28:     end if
29:     /*complete the readResponse handshake */
30:     if readResponse[operand].req == low then
31:         readResponse[operand].ack = low;
32:     end if
33: end for
```

Algorithm 13 Communication unit: operand `lock` procedure.

```
1:  /*send reserveLock to RBA */
2:  reserveLock.data.regID = current_inst.rd.regID;
3:  reserveLock.req = high;
4:  /*if the lock is reserved */
5:  if reserveLock.ack == high then
6:      /*send regLock to RB */
7:      regLock.data.regID = current_inst.rd.regID;
8:      regLock.req = high; grantRead.req = low;
9:  end if
10: /*destination register is locked */
11: if regLock.ack == high then
12:     current_inst.rd.locked = true;
13:     regLock.req = low;
14: end if
```

Algorithm 14 Register bank arbiter: `grantRead` procedure.

```
1:  /*for each of the units */
2:  for all unitID do
3:      /*if grantRead request is received */
4:      if grantRead[unitID].req == high then
5:          if lockStatus[grantRead[unitID].data.regID] == NOT_RESERVED then
6:              grantRead[unitID].ack = high;
7:          else if  rL = reserveLockQueue.find(grandRead[unitID].data.regID) != NULL then
8:              grantReadQueue.add(grantRead[unitID].data, rL.index);
9:          else
10:             grantReadQueue.add(grantRead[unitID].data, 0);
11:         end if
12:     end if
13:     /*complete the grantRead handshake */
14:     if grantRead[unitID].req == low then
15:         grantRead[unitID].ack = low;
16:     end if
17: end for
```

Algorithm 15 Register bank arbiter: `update` procedure.

```
1:  /*received an update request from the RB */
2:  if update.req == high then
3:      lockStatus[update.data.regID] = NOT_RESERVED;
4:      for all gR = grandReadQueue.getNext() do
5:          if gR.tag == 0 & gR.regID == update.data.regID then
6:              grantRead[gR.unitID].ack = high;
7:              grantReadQueue.delete(gR);
8:          end if
9:      end for
10:     /*find the first lock request in the reserveLock queue */
11:     if rL = reserveLockQueue.find(update.data.regID) != NULL then
12:         lockStatus[update.data.regID] = RESERVED;
13:         reserveLock[rL.unitID].ack = high;
14:         reserveLockQueue.delete(rL);
15:         for all gR = grantReadQueue.getNext() do
16:             if gR.tag == rL.index then
17:                 gR.tag = 0;
18:             end if
19:         end for
20:     end if
21: end if
22: /*complete the update handshake */
23: if update.req == low then
24:     update.ack = low;
25: end if
```

Algorithm 16 Register bank arbiter: `reserveLock` procedure.

```
1:  /*for each of the units */
2:  for all unitID do
3:      /*if reserveLock request is received */
4:      if reserveLock[unitID].req == high then
5:          if lockStatus[reserveLock[unitID].data.regID] == NOT_RESERVED then
6:              lockStatus[reserveLock.data.regID] = RESERVED;
7:              reserveLock.ack = high;
8:          end if
9:      else
10:         reserveLockQueue.add(reserveLock.data, reserveLockQueue.index++);
11:     end if
12:     /*complete the reserveLock handshake */
13:     if reserveLock[unitID].req == low then
14:         reserveLock[unitID].ack = low;
15:     end if
16: end for
```

Algorithm 17 Register bank: `write` procedure.

```
1:  /*if regWrite is received */
2:  if regWrite.req == high then
3:      /*write the value and unlock the register */
4:      registers[regWrite.data.regID] = regWrite.data.regValue;
5:      locked[regWrite.data.regID] = false;
6:      regWrite.ack = high;
7:  end if
8:  /*complete the regWrite handshake */
9:  if regWrite.req == low then
10:     regWrite.ack = low;
11: end if
```

Algorithm 18 Register bank: `read` procedure.

```
1:  /*for each of the read ports */
2:  for all read_port do
3:      /*if regRead is received and the register is not locked */
4:      if regRead[read_port].req == high &
        locked[regRead[read_port].data.regID] == false then
5:          /*send the read response and increment number of reads for the register */
6:          readResponse.data.regID == regRead[read_port].data.regID;
7:          readResponse[read_port].data.regValue = registers[regID];
8:          reads[regRead[read_port].data.regID]++;
9:          readResponse[read_port].req = high;
10:         regRead[read_port].ack = high;
11:     end if
12:     /*complete the regRead handshake */
13:     if regRead[read_port].req == low then
14:         regRead[read_port].ack = low;
15:     end if
16:     /*complete the readResponse handshake */
17:     if readResponse[read_port].ack == high then
18:         regRead[read_port].req = low;
19:     end if
20: end for
```

Algorithm 19 Register bank: `lock` procedure.

```
1:  /*if regLock is received */
2:  if regLock.req == high then
3:      /*if the register in not locked and all the reads are performed */
4:      if locked[regLock.data.regID] == false &
        reads[regLock.data.regID] == regLock.data.reads then
5:          /*lock the register and send the update to RBA*/
6:          locked[regLock.data.regID] = true;
7:          update.data.regID = regLoc.data.regID;
8:          update.data.req = high;
9:          regLock.ack = high;
10:     end if
11: end if
12: /*complete the regLock handshake */
13: if regLock.req == low then
14:     regLock.ack = low;
15: end if
```

5.5.5 Evaluation and write-back

Upon the completion of the operand fetch-and-lock stage, the CU transfers the control of instruction execution over to the FU (when the FU is free and ready). The CU can now complete the issue handshake with the IU and label itself as free and ready for the next instruction. Note that operand fetch-and-lock in the CU and the evaluation stage in the FU can belong to different instructions, although CU and FU are considered an atomic functional unit. This means that the maximal throughput of the architecture is twice the number of functional units.

The instruction evaluation stage (performed in the FU) consists of the following tasks:

- Evaluate the instruction;
- Write the result to the RB;
- Write the result to the forwarding table in the CU.

Functional units are effectively isolated from the network and communicate only with the CU and the RB. The communication with the CU consists of obtaining the instruction and operands, and sending the result back to the CU. In a similar manner to

the communication between the CU and the IU, the complete communication between the FU and the CU is performed in a single handshake. The low acknowledge signal means that the FU has evaluated the instruction and is now free and ready for the next instruction. The write-back handshake with the RB is initiated in parallel with the handshake that writes the result into the forwarding table in the CU (Figure 5.11).

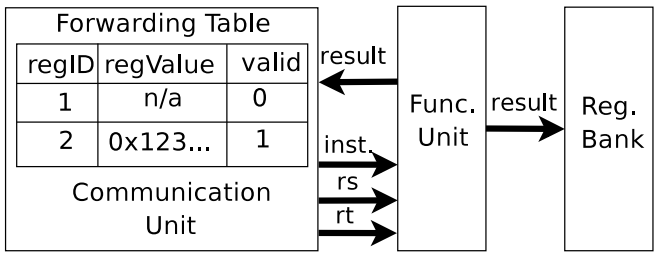


Figure 5.11: Instruction evaluation and write-back.

The FUs are divided into four types:

1. Arithmetic Units (AU)
2. Logic Units (LU)
3. Multipliers (MULT)
4. Memory Units (MU)

The number of each of these four types can vary and can be adjusted to match the specific needs of a particular code that is run on the architecture. Applications in which arithmetic operations are dominant can achieve better performance if the architecture is configured with a number of arithmetic units. In the case of cryptographic applications, arithmetic and logic operations are predominant and, therefore, multiple arithmetic and logic units would lead to improvement in both performance and security, as instructions execute in parallel, contributing towards the non-deterministic order of their execution.

5.5.5.1 Instruction set

The instruction set implemented in the architecture is a subset of the MIPS instruction set, and is divided into five groups, which are summarised below (a complete list of all the instructions is given in Appendix B).

Group 1: Arithmetic instructions with zero, one or two source operands that are executed in arithmetic units (AUs) and include additions and subtractions.

Group 2: Logic instructions with one or two source operands that are executed in logic units (LUs) and include bitwise operations, bit shifting, and logic operations such as ORs, ANDs and XORs.

Group 3: Multiplier instructions with one or two source operands that are executed in multiplier units (MULTs) and include multiplication, division and modulo operation.

Group 4: Memory instructions with one or two source operands that are executed in memory units (MUs) and include load from and store to the memory operations.

Group 5: Branch and jump instructions with zero, one or two source operands that are executed in a specialised module in the FBU and include jump instructions, conditional and unconditional branches, call and return instructions.

5.5.5.2 Control transfer operations

In order to ensure fast instruction issue rate and prevent the issue stage from being starved of instructions, control transfer instructions are implemented in the specialised module in the FBU. This module is capable of executing jump, call, return, and both conditional and unconditional branches, and directly updates the value of the PC register (`pcReg`), and the link register (`linkReg`) that is used for storing the PC value in the call instructions. The operands values (for the control transfer instructions with source operands) are obtained from the RB through the RBA arbitration. The resulting `pcReg` and the `linkReg` values are both maintained locally in the unit and in the RB.

The *jump* instruction updates the `pcReg` with the address specified by its operand, and the FBU then resumes the instruction fetch from the new PC. A *call* instruction causes the `pcReg` to be updated in the same way as jump instructions, with the difference that the value of `pcReg` is saved in the `linkReg`, before its value is updated. A *return* instruction simply restores the `pcReg` value from the `linkReg`. Finally, branch instructions modify the PC depending on the result of the register comparison. Once the result of the comparison is committed, the FBU issues the next instruction determined by the new PC value.

5.5.5.3 Memory operations

Memory operations can restrict the available parallelism as they introduce implicit data-dependencies through memory locations, *i.e.*, a data dependency exists when two

memory instructions refer to the same location. Therefore, if the store is ahead of a load in the program order, the load cannot access the memory ahead of a store when they both refer to the same memory address. Similarly, a store cannot access the memory before a load when they refer to the same memory address, if the load is ahead in the program order. Again, two stores cannot access the memory out of program order, if they are referring to the same memory address.

The process of determining if two instructions access the same memory location is called *memory disambiguation*. Memory disambiguation can be implemented either at run-time, the *dynamic memory disambiguation*; or at compile-time, the *static memory disambiguation*. The requirement for memory disambiguation is seen in architectures in which instructions are either issued or executed out of program order, *i.e.*, instructions can overtake while executing.

Dynamic memory disambiguation schemes keep track of the memory instructions in the order in which they are decoded. When a memory instruction is to be issued, its address must be compared to the addresses of all previously-decoded memory operations, to check whether the address has been referenced. If such an entry exists, the order of execution of the corresponding instructions has to be preserved. The disadvantage of this approach is that the hardware can become slow and complex as the number of entries grows (*e.g.* [118]). Static memory disambiguation on the other hand, has the flexibility of affording more aggressive algorithms to disambiguate the memory references, but the problem with this approach is that it is not always possible to statically disambiguate memory references [118].

Maintaining data-coherence for memory access in the network-based asynchronous architecture with multiple memory units is similar to maintaining register file data-coherence. An obvious way to avoid memory disambiguation is to consider only one memory unit. Since the instructions are issued in-order, this would ensure that loads and stores access memory in program order. Multiple memory units, executing in an independent manner make the situation of loads overtaking the stores and vice-versa possible. Such situations are hazardous when loads and stores refer to the same memory location. If a store overtakes a load, it means that there is a true dependency (RAW), whereas if a load overtakes a store, it represents an anti-dependency (WAR). If two stores refer to the same memory address, it means that there is an output dependency (WAW). Figure 5.12 provides an example of potential dependencies and the memory data-hazards that can occur if the instructions with these dependencies are not executed in program order.

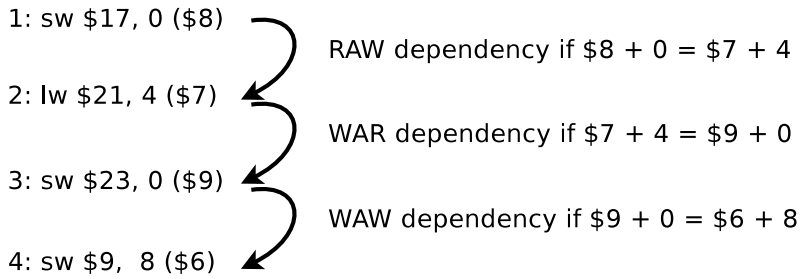


Figure 5.12: An example of memory data-hazards.

Preserving the data-coherence of the memory file in the presence of multiple memory units requires additional arbitration. The mechanism implemented for this purpose is called the *Memory Arbiter* (MA). With the MA in place the memory instruction datapath has an additional stage during which the memory operation is granted. One idea to implement this additional stage would be similar to that of the RBA. This implementation would allow memory instructions to access memory out-of-order except when the same memory location is to be accessed. Data-dependencies between instructions that access the register file can be detected by simply comparing register identifiers. For instructions that access memory this kind of comparison can be made only after each instruction has calculated its memory address.

The frequency at which the same memory location is accessed should in practice be small as such communication usually takes place through the register file [105]. Therefore, the simplest and the most economic way to implement the memory arbiter was to enforce in-order memory access. This allows memory instructions to execute in parallel with other instructions, but access memory in the order of their issue. The rest of their execution, *e.g.*, write-back stage in the case of loads, can progress out-of-order.

The MA is implemented as a reorder buffer providing in-order of issue memory access. Each memory instruction is tagged with the small counter which loops around when it reaches the number of memory units. The MA keeps its own record of the instruction tag that has accessed the memory last and it will allow only the next increment to proceed. The process of granting a memory operation, shown in Figure 5.13, can be summarised as the following three steps:

Step 1 – Obtain permission to execute the memory operation. The CU initiates a memory access handshake with the MA.

Step 2 – MA grants the operation. If the memory tag of the instruction in question is equal to the instruction tag of the next memory instruction that is to be granted,

then the memory access is granted. Otherwise, the request will wait for its turn.

Step 3 – Transfer of control to the FU. After the memory operation is granted, the control is transfered to the FU that evaluates the memory address[2] and accesses the memory.

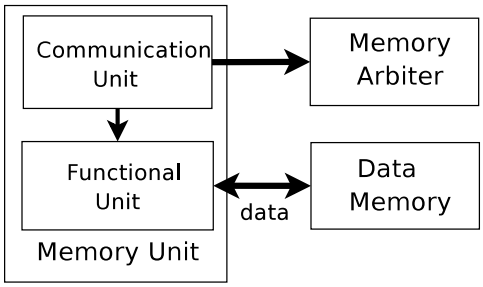


Figure 5.13: Memory access arbitration.

5.6 Data-forwarding

Data-forwarding is a commonly used technique in pipelined architectures for minimising the cost of functional unit stalls due to data-dependencies. It works by redirecting the result of an instruction to be written in the register file to the following instruction in the pipeline. In addition, data-forwarding may be preferable to obtaining operands from the register file from the power consumption perspective even when performance is unaffected [105]. Data-forwarding results in a reduction in register file traffic, which reduces the total number of requests at read ports and thereby leads to a decrease in their access times. The extent of forwarding may be increased by providing more forwarding registers in the forwarding tables in each CU, and increasing the size of the dependency table in the IU, thus allowing a higher number of results to be forwarded.

Data-forwarding is initiated by a request from the issue unit to the producer unit. This request contains the register identifier, `regID`, and the consumer unit identifier, `unitID`. Each unit maintains a small table, called the *forwarding table*, that contains pending and already generated register values that can be forwarded on request. Entries in the forwarding table are tuples of the form: ⟨`regID`, `regValue`, `valid`⟩, which specify the register, its value and the validity of its value. When the instruction is received its destination `regID` is added to the table while the value and the validity bit are

[2]The memory unit contains an internal adder for calculating the effective memory addresses.

left unset. These are set after the result is calculated and received from the FU. When the unit receives the request for forwarding, it queries the forwarding table to find the value of the register. Non-valid entries correspond to the results that are pending. If the value is not valid, then this signal will wait for the value to become valid in order to forward it. When the result is received from the FU, the forwarding table is updated with a (valid) value for the result register. During this process care is taken to ensure that there is only one entry per register in the table. The forwarding signals for the awaiting forwarding request are next generated. If the same unit is both the producer and the consumer in a particular compound, then the forwarding signal is not generated but the operand value is directly updated.

The data-forwarding signal contains the following information:

1. The destination unit identifier, `unitID`, which is used to recognise that the destination is reached.

2. The register identifier, `regID`, which determines which of the operand values is forwarded, as both instruction operands can be obtained through data-forwarding.

3. The register value `regValue`.

4. The `ticket`, which specifies the routing path through the network, and is generated according to network topology (as discussed in the following section).

5.6.1 The network topology

The network in network-based architecture can have any topology. The ones considered in this work are *binary hypercube* and *binary de Bruijn graph*, both belonging to the group of so-called *graphs on alphabets* [58, 81], widely used in multi-computer interconnection networks. Hypercubes are sometimes categorised as a separate class, *hypercube graphs*, because of their importance. Such a choice was made because these topologies contain a large number of vertices while having a small diameter, and support oblivious routing algorithms. Efficient routing protocols in these graph topologies have the following desirable properties:

1. It is not necessary to explicitly store the topology of the network in every vertex.

2. The shortest path routing can be performed.

3. Randomised routing algorithms can be efficiently implemented.

A very common routing strategy is the *dimension-order routing* in which a packet is routed along dimension 0 until it reaches its correct position, then along dimension 1, and so on. In the design of efficient routing algorithms, randomisation plays an important role. A major theoretical breakthrough was the probabilistic permutation routing on the binary hypercube proposed by Valiant and Brebner [130]. The two phase algorithm consists of a randomised (Phase 1) and a deterministic routing (Phase 2). In Phase 1, the packet is routed to an independent, randomly selected node in the network. The role of this phase is to decrease the difference between average and worst-case performance. In Phase 2, the packet follows the shortest path to its destination.

5.6.1.1 Hypercubes

Binary hypercubes [58, 81] represent an important class of graphs whose diameter grows slowly with the number of vertices. The binary hypercube graphs are defined on the basis of binary alphabets as described next.

Definition 5.6.1. *Let n be a positive integer. A binary hypercube, denoted by $H(n)$, is a graph consisting of 2^n vertices labelled with n-bit integers from 0 to 2^n-1. Let $(x_{n-1}, x_{n-2}, \ldots, x_0)$ be the label of vertex x from $H(n)$. Vertex x is adjacent to vertices:*

$$(x_{n-1}, x_{n-2}, \ldots, x_{i+1}, \overline{x}_i, x_{i-1}, \ldots, x_0), \forall\, 0 \leq i \leq n-1,$$

where $\overline{x} = 0$, if $x = 1$, and $\overline{x} = 1$, if $x = 0$.

This defines a hypercube with the number of nodes equal to a power of two. For the purposes of this architectural design it was more convenient to define a hypercube with any number of nodes. Therefore, Definition 5.6.1 was modified to meet this requirement.

Definition 5.6.2. *Let N be a positive integer. A* **partial binary hypercube**, *denoted by $PH(N)$, is a graph consisting of N vertices, labelled with n-bit integers from 0 to $N-1$, where n is a positive integer such that $2^{n-1} \leq N \leq 2^n$. Let $(x_{n-1}, x_{n-2}, \ldots, x_0)$ be the label of vertex x from $PH(N)$. Vertex x is adjacent to vertices:*

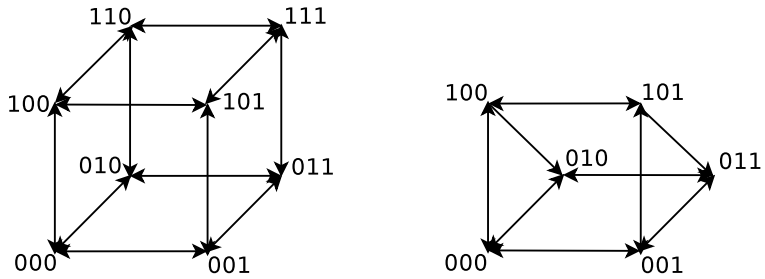
$$x' = (x_{n-1}, x_{n-2}, \ldots, x_{i+1}, \overline{x}_i, x_{i-1}, \ldots, x_0),\ if\ x' \leq N,$$

and

$$x'' = (0, x_{n-2}, \ldots, x_{i+1}, \overline{x}_i, x_{i-1}, \ldots, x_0),\ s.t.\ x'' \neq x \text{ and } x'' \neq x',\ otherwise,$$

for all $0 \leq i \leq n-1$.

The partial hypercube $PH(6)$, shown in Figure 5.14(b), has six vertices labelled with three-bit integers from $0 = (000)_2$ to $5 = (101)_2$. The vertex labelled $(100)_2$, in $H(3)$ connects to vertices $(101)_2$, $(110)_2$ and $(000)_2$. However, vertex $(110)_2$ is not in $PH(6)$ as $(110)_2 > 5$, thus $(100)_2$ connects to $(010)_2$, *i.e.*, node $(010)_2$ acts as both itself and node $(110)_2$. This definition renders $H(3)$ and $PH(6)$ equivalent from the point of view of connectivity and routing.



(a) H(3) (b) PH(6)

Figure 5.14: Binary hypercube H(3) and partial binary hypercube PH(6).

Most hypercube routing algorithms are *oblivious*. A *deterministic oblivious* routing algorithm is one in which the path taken by a packet through the network is a function of its origin and destination. A *randomised oblivious* algorithm is one in which each packet independently chooses a path according to a probability distribution which is a function of its origin and destination.

Deterministic source oblivious routing in a hypercube requires only the knowledge of the source and the destination vertices. Current address and the destination address are compared and if they match, the destination is reached. Otherwise the message is sent along its first dimension where a mismatch occurred. This algorithm is called *greedy routing algorithm* and it sends packets from source to destination along the shortest path.

5.6.1.2 De Bruijn graphs

Both directed and undirected De Bruijn graphs [85, 24, 25, 81] have received considerable attention as a graph topology for interconnection networks. They represent an important family of graphs with small diameter and high number of vertices.

Definition 5.6.3. *Let $d \geq 1$ and $n \geq 2$ be two integers, and A an alphabet of size d. A directed de Bruijn graph, denoted by $DB(d,n)$, is a graph consisting of d^n vertices labelled with words of length n from alphabet A, where n represents the diameter and d represents the degree of the graph. Let $(x_{n-1}, x_{n-2}, \ldots, x_0)$ be the label of vertex x from $DB(d,n)$. Vertex x is adjacent to vertices*

$$(x_{n-2}, x_{n-3}, \ldots, x_0, \alpha), \forall \alpha \in A.$$

Definition 5.6.4. *An undirected de Bruijn graph, denoted by $UB(d,n)$, is a graph consisting of d^n vertices labelled with words of length n from the alphabet A. Let $(x_{n-1}, x_{n-2}, \ldots, x_0)$ be the label of vertex x from $UB(d,n)$. Vertex x is adjacent to vertices*

$$\begin{aligned} & (\alpha, x_{n-1}, x_{n-2}, \ldots, x_1) \\ \textit{and} \quad & (x_{n-2}, x_{n-3}, \ldots, x_0, \beta), \quad \alpha, \beta \in A. \end{aligned}$$

This definition can be generalised to any number of vertices. The vertices of these graphs can be numbered with integers modulo N, where N is the number of vertices in the graph. An edge is drawn from vertex x to vertex y if $y \equiv dx + \alpha \ (mod\ N)$, where $0 \leq \alpha \leq d-1$.

Definition 5.6.5. *Let $N \geq 2$ be an integer. A directed binary de Bruijn graph, denoted by $DB(N)$, is a graph consisting of N vertices labelled with integers modulo N. Vertex x from $DB(N)$ is adjacent to vertices:*

$$x' = 2x + \alpha \ (mod\ N), \ \ \alpha \in \{0,1\}.$$

A directed binary de Bruijn graph with 8 vertices is shown in Figure 5.15.

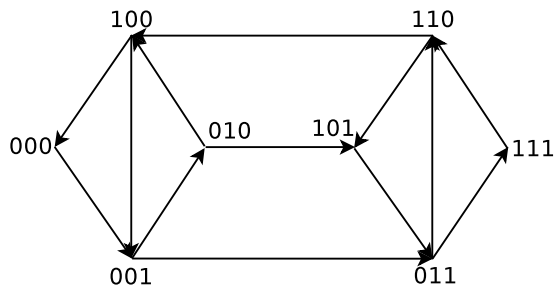


Figure 5.15: Directed binary de Bruijn graph DB(8).

The easiest routing in the network would be one which is *source oblivious*, *i.e.*, at each point of time the next edge to be taken depends only on current vertex and

the destination vertex, but not on the origin. Another routing technique that can be applied in de Bruijn graphs is the so-called *greedy routing.* Greedy routing corresponds to correcting dimensions from left to right, but it does not produce shortest path routes. Valiant-Brebner's [130] routing to random destinations can also be applied on de Bruijn graphs. Aleliunas [16] extended this probabilistic algorithm to dynamic routing on de Bruijn network.

5.6.2 Data-forwarding and randomised routing

For both the hypercube and the de Bruijn graph topologies the Valiant-Brebner [130] routing strategy is employed. This type of routing consists of two phases: the randomisation phase (Phase 1) and the deterministic routing phase (Phase 2). In Phase 1, the packet is sent to an independent, randomly-selected node. In Phase 2, the packet follows the deterministic path to the destination. In the case of a hypercube this path is the shortest one and in the case of the de Bruijn graph the path is specified by the greedy routing strategy. Each data-forwarding signal contains the destination unit identifier, `unitID`, the register identifier, `regID`, the register value, `regValue` and the `ticket`. The routing in both phases of the algorithm is predetermined by the ticket, which specifies the order in which the dimensions of the hypercube are traversed, or the destination node address used for greedy routing in the de Bruijn network. The difference is that the ticket for Phase 1 is randomly generated, while the ticket for Phase 2 is generated according to the destination node. The producer unit generates the ticket for the routing in the Phase 1. Once the forwarded data reaches the randomly-chosen node, a new ticket, that specifies the path to the destination (Phase 2 routing) is generated. There is a slight difference in the generation and interpretation of the ticket, and the way in which the deterministic phase is performed for the two graph topologies and will, therefore, be presented separately.

The initial stage of each of the phases of the routing procedure consists of ticket initialisation. The ticket is initialised to contain information about the path that is taken in that particular phase.

5.6.2.1 Randomised routing in binary hypercubes

In the case of a hypercube the ticket specifies the sequence in which the dimensions of the graph are traversed. In Phase 1 the ticket is initialised to represent the path to the random node. The initialisation procedure for ticket T is as follows [130]:

```
for i = 0 to size(T) do
  T[i] = (rand() % 2 == 0) ?  0 :  i+1;
  pack(T);
end for
```

In Phase 2 the ticket is initialised to represent the shortest path to the destination node. This initialisation procedure for ticket T is as follows:

```
for i = 0 to size(T) do
  T[i] = ((cur_node & (1 << i)) == (dest & (1 << i))) ?  0 :  i+1;
  pack(T);
end for
```

The initialisation part of both phases can be divided into two subroutines: (1) generation of the ticket according to the destination node for that phase (in the Phase 1 this is a random node and in the Phase 2 the destination node); (2) packing of the ticket: given the ticket $T[0 \ldots n-1]$ with r non-zero elements, non-zero elements are assigned to $T[0], \ldots, T[r-1]$, and zero elements are assigned to $T[r], \ldots, T[n-1]$. This facilitates the routing completion detection, since once all the non-zero dimensions of the ticket are traversed the ticket expires, *i.e.*, the destination is reached and the routing stops. If the destination is reached before the ticket expires, then the routing stops regardlessly.

5.6.2.2 Randomised routing in de Bruijn graphs

In the case of a de Bruijn graph, the ticket specifies the destination node, as the greedy routing of correcting the dimensions from left to right is used. Given that the alphabet used to represent a de Bruijn graph is binary, the Phase 1 ticket T initialisation is as follows:

```
for i = 0 to size(T) do
  T[i] = rand()%2;
  de_loop(T, current_node);
  pack_db(T);
end for
```

The following subroutines in the ticket initialisation can be distinguished: (1) generation of the ticket according to the target node (randomly selected one in Phase 1, and the destination node in Phase 2); (2) *de-looping*; and (3) packing of the ticket.

Since the greedy (dimension correcting) routing strategy is used here, it is possible for two consecutive nodes on the route to represent the same node. De-looping is performed in order to remove such redundancy in the path, as shown below:

```
for i=0 to size(T) do
  new_node = current_node << T[i];
  if new_node == current_node then
    T[i] = -1;
  end if
  current_node = new_node;
end for
```

Packing the ticket T is slightly different when compared to hypercube packing: given the ticket $T[0 \ldots n-1]$, the r non-negative elements are assigned to $T[0], \ldots, T[r-1]$, and the negative elements to $T[r], \ldots, T[n-1]$. The packing of the ticket facilitates the routing completion detection, since once all the non-negative elements of the ticket are traversed the ticket expires, the random destination is reached and the routing stops.

5.6.2.3 Routing process

CUs are responsible for most of the communication in the network. A large portion of this communication is data-forwarding, *i.e.*, routing of register values in the network. In accordance with the topology, each CU has a number of inports and outports through which it communicates with other (neighbouring) CUs, as shown in Figure 5.16. Each data-forwarding request received on the inports is queued in the *forwarding queue*. The information that is received in this process contains: (1) the destination unit identifier (`unitID`), (2) the register identifier (`regID`), (3) the register value (`regValue`) and (4) the `ticket`. The ticket specifies the way in which the dimensions of the graph are traversed, *i.e.*, the routing path. Besides the path, the ticket contains a counter that specifies which portion of that path is traversed up to that point. With this information each node on the path can easily calculate the next node to send the information to. The producer unit performs the initialisation of the ticket for Phase 1, at the end of which, the packet is at the randomly-chosen node that performs the initialisation of the ticket for Phase 2. Upon the arrival of a message at a node, there are two possible actions that can be taken: (1) if the message is destined for that node itself, then the routing stops and the node updates the status of the corresponding source operand; (2) otherwise, the next node to send the message to is calculated, the ticket is updated and the data is forwarded. The rate at which packets progress on their route depends on the

load on the outports of nodes along the path, with data being forwarded from one node to another as soon as the connecting port is available.

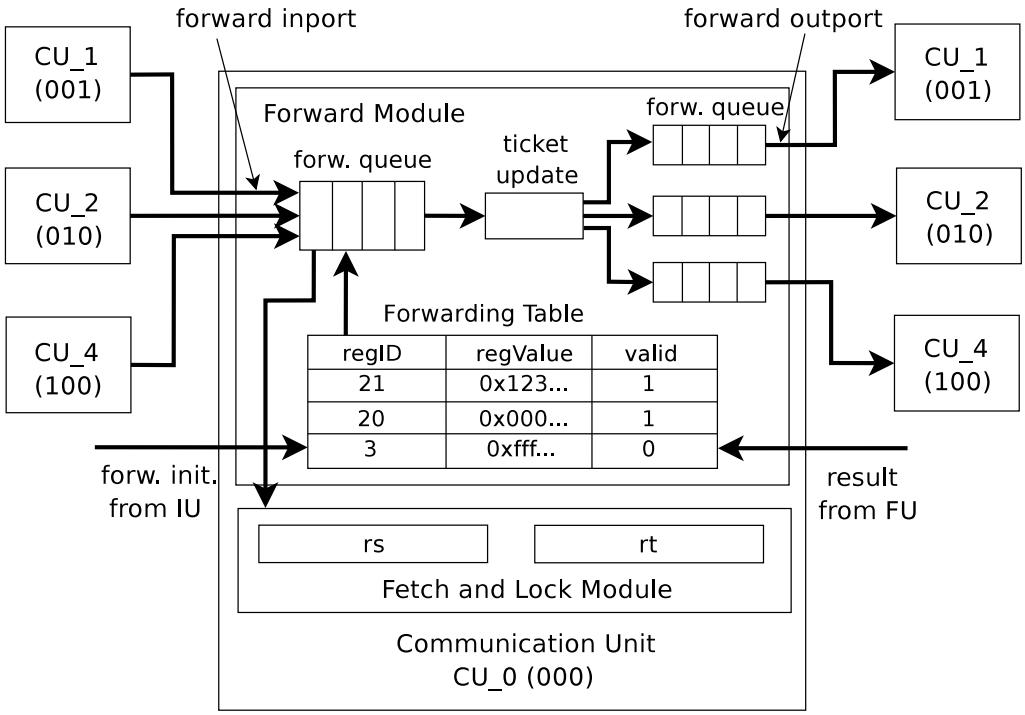


Figure 5.16: Data-forwarding communication in a hypercube network configuration.

5.6.3 Data-forwarding and secret-sharing

The approach of data-forwarding and randomised routing was adopted in order to hide compromising information in the power traces. However, when forwarded, critical register values actually influence the overall data-dependent power consumption. In order to remove the direct contribution of data-forwarding to data-dependent power-consumption, data-forwarding is modified to forward data in the form of *data-shares*. The general technique is as follows: for each register value d, $k-1$ random values, $r_1, r_2, \ldots, r_{k-1}$, are generated and the k shares, $d \oplus r_1 \oplus r_2 \oplus \cdots \oplus r_{k-1}, r_1, r_2, \ldots, r_{k-1}$, are forwarded. This differs from the idea of secret-sharing as given in [36], which involves splitting every relevant bit into several shares and performing operations on shares. This approach has been criticised as being impractical due to performance and architectural overheads [103, 104]. In the network-based architecture, the architectural overhead is minimal as only the number of packets that are routed is increased, which contributes towards additional noise.

5.6.4 On-chip random number generator

The network-based asynchronous architecture is designed for cryptographic devices, such as smart-cards. It assumes the existence of a on-chip source of randomness used for randomising data-forwarding routines. In most designs the source of "random" numbers is provided by a *pseudorandom number generator* (PRNG). PRNGs use deterministic processes that output a series of numbers generated using an initial *seed* state. Since the output is purely a function of the seed data, an attacker who can guess the seed can predict the entire PRNG output. It is crucial, therefore, that the PRNG is seeded by a true random source. A *true random number generator* (TRNG) produces random numbers using an unpredictable source. Most TRNGs measure unpredictable natural processes, such as thermal noise, atmospheric noise or nuclear decay, and use them as a seed source.

Systems that do not have a hardware RNG, usually seed their RNGs using peripherals, such as mouse and keyboard. The most common techniques involve timing user processes. However, these methods are often awkward and slow. On the other hand, systems that implement hardware TRNGs tend to be slow, difficult to implement, require user involvement and often provide unknown amounts of true entropy [71]. It is generally believed that "good random-number generators are hard to design because their security often depends on the particulars of the hardware and software" [121]. An example of a "good random number generator" [71] is the Intel Random Number Generator, which samples thermal white-noise resulting from random electron and material behaviour.

For smart-cards user-interface peripherals are not available for obtaining a random source, so it is often unclear how smart-card RNGs can obtain true random seeds. Existing PRNG for smart-cards (*e.g.* Schlumberger's CryptoFlex card) are considered poor and not to be trusted for key generation. Some designs use a reasonable amount of entropy for seeding, which is available through the card interface. However, secure RNGs for smart-cards are either non-existent or the vendors do not publish their designs.

In the case of the network-based architecture an on-chip source of randomness (*e.g.* a PRNG) that is not seeded properly would result in every generated sequence of (pseudo)random numbers, used in randomised routing, being the same for each run of the code. This would result in no variation in execution timings, except for those that are side-effects of asynchronous execution. Therefore, it is critical that the PRNG

is properly seeded from a truly random source. Most criticism that PRNG's receive is connected to the reliability and randomness of secret key generation. There is no need for on-chip key generation in the case of the network-based processor. The source of randomness is only required for generating random trials during randomised data-forwarding. These sequences are not reused, as in cases where keys are generated, so the security issues of an on-chip PRNG are not as crucial. If the PRNG is implemented using a TRNG for seeding, this would ensure that cryptographic devices using the network-based design have randomised execution.

As stated earlier, the aim of this thesis is to evaluate the validity of network-based architecture for introducing non-determinism in the execution. The point is not to give a detailed specification of every building block of the architecture. The question of providing true sources of randomness in smart-cards is outside the scope of this work.

5.7 An example

This section provides a step-by-step description of how compounded instructions are executed in the network-based architecture. The program fragment is used to illustrate the issue, operand fetch-and-lock and execution stages, and also the forwarding of results between instructions. The output shown in Figure 5.17 is extracted from the simulation of the AES encryption algorithm on the network configuration with four units and the hypercube topology.

Figure 5.17 shows executed instructions in program order, the units that the instructions were issued to, the compounding bits and the producer units for each of the operands. The sample instruction sequence consists of an addition (`addi`) executed in the arithmetic unit (AU); a load operation (`lb`) executed in the memory unit (MU), and a left bit shift (`sll`) executed in the logic unit (LU). Each unit is assigned an `unitID`. The network configuration has four units connected as a hypercube $H(2)$, as shown in Figure 5.18. The fourth unit, the multiplier (MULT), is not shown in Figure 5.17, as the execution output is restricted only to operations that concern the three given instructions. Each of the rows represent operations that are taking place concurrently in the functional units. The height of the row and the duration of time the specified operations take are not related, but the rows correspond to the virtual time units in the architectural simulator (see Section 6.2.1) in which the simulated time is discrete.

Prior to the issue of the first instruction the CUs of the AU, LU and MU are all free and ready to accept the next instruction. Therefore, the IU is able to issue the instruc-

instruction	unitID	rsf	rtf	producer 1	producer 2
(1) addi $2, $20, 1	AU (0)	1	n/a	AU (0)	n/a
(2) lb $20, 0 ($2)	MU (3)	1	n/a	AU (0)	n/a
(3) sll $2, $20, 16	LU (1)	1	n/a	MU (3)	n/a

IU	RBA	AU (0)	LU (1)	MU (3)
* issue inst.(1) to AU * init. forw. to AU <r=20,u=0> * add <r=2,u=0> to dep. table				
* issue inst.(2) to MU * init. forw. to AU <r=2,u=3> * add <r=20,u=3> to dep. table		* addi $2, $20, 1 * resLock <r=2>		
* issue inst.(3) to LU * init. forw. to MU <r=20,u=1> * add <r=2,u=1> to dep. table	* lockStat[2]=NR =>lockStat[2]=R	* received forw. req. <r=20, u=0> (forward to itself)		* lb $20, 0 ($2) * resLock <r=20>
	* lockStat[20]=NR =>lockStat[20]=R	* regLock <r=2>	* sll $2, $20, 16 * resLock <r=2>	
	* lockStat[2] = R =>add to queue resLock<2>	* received forw. req. <r=2, u=3>		* regLock <r=20> * received forw. req. <r=20, u=1>
	* update <r=2> =>lockStat[2]=NR * grant resLock<2> =>lockStat[2]=R	* regLock ack =>reg is locked =>transfer to FU =>CU is free		
* AU's CU is free for the next instruction		* FU evaluate * FU write-back * FU write result to forward. table	* regLock <r=2>	
	* update <r=20> =>lockStat[20]=NR	* forw. r=2 to MU * route to LU		* regLock ack =>reg is locked
			* forw. r=2 to MU * route to MU	
	* update <r=2> =>lockStat[2]=NR		* regLock ack =>reg is locked	* received forw. r=2 =>transfer to FU =>CU is free
* MU's CU is free for the next instruction				* FU evaluate * FU write-back * FU write result to forward. table
				* forw. r=20 to LU * route to MULT * MULT route to AU
		* forw. r=20 to LU * update ticket * route to LU		
			* received frw. r=20 =>transfer to FU =>CU is free	
* LU's CU is free for the next instruction			* FU evaluate * FU write-back * FU write result to forwad. table	

Figure 5.17: A sample execution of compounded instructions.

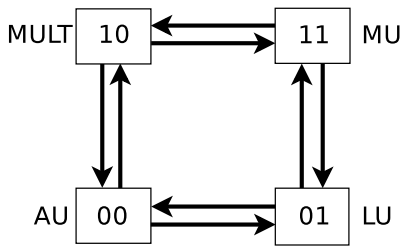


Figure 5.18: Hypercube H(2) organisation of functional units.

tions one after the other, in-order, although there are dependencies between instructions. The operations connected to execution of each of the instructions are coloured in order to make execution of each instruction easier to follow.

At the issue of an instruction the forwarding signals to the producers are initiated and the information about the resulting register and the unit the instruction is issued to are added to the dependency table. Now, each CU will send the `reserveLock` requests to the RBA, which grants it if the `lockStatus` of the register is `NOT_RESERVED` (abbreviated to `NR` in Figure 5.17). For example, the AU requests for the lock to be reserved for register 2 (specified in Figure 5.17 with `resLock`⟨`r=2`⟩). Since the `lockStatus` of this register in the RBA is `NOT_RESERVED`, the lock is reserved and the status of the register becomes `RESERVED` (abbreviated to `R` in Figure 5.17). In the next event the AU receives the forward initialisation for register 20. This register value should be forwarded to unit with `unitID` = 0, which is the AU. This means (not shown in the figure) that the value of register 20 is most recently generated in the AU. The operand value is therefore available from the same unit, so both operand fetch and data-forwarding are avoided. Next, the register lock request for register 2 is sent to the RB (`regLock`⟨`r=2`⟩). Once the register is locked in the RB, the CU can transfer control to the FU which evaluates the instruction. At the same time, the RB sends the `update` signal to the RBA to update the `lockStatus` for register 2. After the instruction is evaluated, the AU responds to the forwarding request previously received, to forward the value of the register 2 to the MU. The routing procedure specified for Phase 1 (see Section 5.6.2.1) generates the path AU→LU→MU (00→01→11). Since the Phase 1 terminates at the destination node, the routing is completed and the MU can update its operand. Both phases of the routing procedure can be seen when routing value of register 20 from MU to LU. Phase 1 routing (routing to a random node) follows the path MU→MULT→AU (11→10→00), whereas Phase 2 routing (shortest path to the destination) specifies the path AU→LU (00→01).

Figure 5.17 does not show the execution of subsequent instructions. The availability of the CU of a particular unit, enables the IU to issue the next instruction of the same type to that unit. The points at which the CUs become available are noted in the figure to hint at further operation of the architecture.

5.8 Features

The architecture model described in this chapter offers a number of interesting features. The network-based asynchronous model is a scalar architecture that features a fast in-order single issue unit, a network of functional units capable of executing concurrently and specialised for forwarding results between the units, and an out-of-order write-back stage. Having a fast single issue unit models a processor capable of issuing more than one instruction at a time without the additional hardware cost of super-scalar designs (*e.g.* [118]). This architectural model aims at exploiting fine-grain instruction level parallelism of both instructions with and without data-dependencies in order to achieve non-deterministic execution.

In VLIW and super-scalar architectures a fast issue rate is achieved through the availability of streams of independent instructions, and by issuing multiple independent instructions. The code for VLIW architectures is prior to issue analysed and scheduled by a compiler, which enables these architectures to issue more than one instruction at a time. As a consequence, the control unit of a VLIW architecture is largely simplified. The super-scalar architectures, in contrast, require significant hardware control for dynamic scheduling, when the compiler is unable to provide independent instructions. The network-based architecture (similarly to VLIW architectures) does not perform dynamic scheduling, and (similarly to super-scalar architectures) prevents hazard at run-time. Instructions are issued at a rapid rate, regardless of the data-dependencies, but at the expense of hardware necessary to perform dynamic instruction compounding during the issue stage. The dependencies between instructions are recognised and used to forward register values between them, which has a number of beneficial effects. The data-forwarding facilitates some of the problems that are introduced by data-dependencies, such as RAW hazards caused by true data-dependencies. Data-forwarding also reduces the register bank response times which has a positive effect on the performance. Further, the randomised data-forwarding protocol introduces interleavings in the execution, which means that the order in which instructions are executed is non-deterministic and is different for each execution of the

algorithm. This in turn leads to decorrelation of power consumption profiles. Considering that the architecture is organised as a network in order to enable randomised routing of register values, this is the key feature of this design.

The instruction compounds are formed at run-time and consist of a number of instructions from the same or different execution blocks. This allows the issue unit to initiate forwarding operations in the producer units and avoid complicated synchronisation and control mechanisms for data-forwarding seen elsewhere. The instructions are issued in-order and as soon as there are available units, while the availability of operands is not an issue. This is largely due to the operand fetch-and-lock stages being moved to the CUs.

The distributed organisation of control, characterised primarily by the distribution of the control over the fetch-and-lock stage among the communication units, and with the novel approach to data-forwarding, enables instructions, both with and without data-dependencies, to operate in parallel. Although the extent of parallel execution of instructions with data-dependencies is subject to the availability of results, mutual exclusion is not enforced. More importantly, the issue stage is not stalled due to these instructions.

The drawback of the traditional implementation of data-forwarding, despite the fact that it can reduce an instruction's execution time, is that it requires that the operand fetch stage is synchronised with the write-back stage. This synchronisation inevitably slows down the faster stage, *i.e.*, the fetch stage, when two dependent instructions are fetched and issued one after the other. If the instructions are not scheduled one after the other, or there is more then one instruction waiting for the result of the first instruction, then the write-back stage of the first instruction could be held up unnecessarily. In the network-based asynchronous architecture data-forwarding, operand fetch stage and the write-back are not synchronised, as explained in Section 5.5.4. In fact, the consumer will not issue the operand-fetch for the register that is obtained through data-forwarding and the write-back stage is completely uncorrelated to the data-forwarding, as the latter is realised through communication of units. Although, the dependencies between consecutive instructions do mean that the consumer would wait for the result to be ready and forwarded, this is still faster than writing to the register bank and reading the result from it.

Out-of-order write-back schemes avoid the need to reorder write-back events when results are ready to be committed. Reordering these events to maintain in-order write-back introduces synchronisations, which reduce the benefits of asynchronous (average-

case) execution. In addition, synchronisations are not desired in the architecture that aims at exploiting non-deterministic execution. In particular, out-of-order write-back is beneficial from the point of power analysis as write-back operations are often targeted in the attacks. The use of both out-of-order execution and out-of-order write-back helps to desynchronise the fragile points in the power traces.

In the network-based asynchronous architecture the fast issue rate, concurrent execution of instructions and randomised data-forwarding are fundamental for achieving non-deterministic execution. Direct synchronisation of different operations along the datapath is not required, as the communication is purely data-driven. In comparison to the MAP architecture [116] which also perceives the processor as a network of functional units, the control in this architectural design is further distributed. Instructions are issued at the faster rate and data-forwarding is implemented without synchronisation overheads. These novelties in the design introduce performance improvements and further exploit non-deterministic execution, that is beneficial for preventing side-channel attacks.

Power consumption is an important factor when designing computing systems. In simple hardware systems, such as smart-cards, the power consumption is important because the power supply of these systems is usually limited. Current synchronous architectures extensively use clock-gating (a technique which isolates inactive circuits from the clock) to save power. Although this technique can improve power consumption, some of its main drawbacks are complicated circuit design and timing analysis [59]. Furthermore, from the security point of view, clock-gating could compromise the security of the device. Namely, electromagnetic analysis is able of pinpointing the regions of the device that are inactive and use that information in the analysis. Thus, on the one hand clock-gating is useful technique, yet on the other hand it could compromise the security of the device.

Asynchronous designs have the ability to reduce power consumption simply by not having a clock and therefore avoiding the clock-related problems of synchronous design. The event-driven nature of asynchronous circuits naturally avoids any unnecessary circuit activity. An asynchronous system activates only those parts of the circuit which are required and so does not dissipate power in the rest of the circuit that is not being used. Similarly to synchronous designs with clock-gating, this characteristic of asynchronous designs could be a potentially exploited by electromagnetic analysis.

In the design of the network-based architecture the functional units that are not executing the instructions could be used in data-forwarding and thus do not appear as inac-

tive. Due to randomised routing, it is not possible to predetermine which any given unit will be used by data-forwarding. In addition, due to the nature of execution, the issue of instructions to functional units changes from one run to the other, as they complete in different times and became available for new instructions in a non-deterministic order. This is a useful characteristic when considering the electromagnetic analysis and its ability to extract the locality information.

5.9 Summary

This chapter presented a novel architectural solution for preventing power analysis. The network-based asynchronous design is an architectural approach for decorrelating the power consumption measurements from the secret values handled in the device by exploiting non-deterministic execution. This processor organisation is a network of functional units that can perform data-forwarding in a randomised fashion and in which the control and the state are distributed. Data-forwarding is used to both improve performance and achieve non-deterministic instruction execution by introducing instruction interleavings through the randomised routing of register values in the network. This design features a fast in-order issue unit, a network of functional units capable of concurrent non-deterministic execution and forwarding of results, and out-of-order completion and write-back. Its datapath is characterised by instructions that run as fast as their data requirements are fulfilled, may overtake each other and compete for resources.

The next chapter presents the evaluation framework and the results of the security and performance evaluations. It presents the stochastic event-driven asynchronous simulation platform geared towards obtaining power and execution performance figures for the execution of algorithms under different architectural configurations. Further, the next chapter presents the security and performance evaluation results and compares different architectural configurations.

Chapter 6

Evaluation

6.1 Introduction

Chapter 5 presented the design of a network-based asynchronous architecture for exploiting non-deterministic execution targeted at secure cryptographic devices. The goal is to render power-based side-channel attacks on the proposed architecture more difficult, or even infeasible. The aim of this chapter is to provide quantitative measures in order to evaluate the proposed approach through simulation and to observe the influence of different architectural configurations.

Section 6.2 presents the simulation platform for investigating the effectiveness of randomised and non-deterministic execution for thwarting side-channel attacks. The evaluation framework is parametrised and can be configured to cover a range of architectural configurations in which different parameters, including the network topology and component delays, can be varied. The architecture was modelled at the instruction-set level and simulated in a stochastic event-driven simulator built on top of the AVT simulation platform [120]. It executes assembly-level instructions compiled from source code using the SUIF compiler [128]. In order to investigate the correlation of the power traces to the source code, the simulator was equipped with power and execution profiling features.

Next, the security and performance evaluation results of the network-based asynchronous architecture are presented. These are divided into two sections: (1) results based on performing differential power analysis attacks on power traces obtained from simulating the execution of AES and DES cryptographic algorithms on different architectural configurations (Section 6.3); and (2) results based on measuring the performance improvements of the network-based approach (Section 6.4).

6.2 Evaluation framework

6.2.1 Asynchronous event-driven simulator

The architecture was modelled at the instruction-set level and simulated in a stochastic event-driven simulator developed on top of the AVT simulation environment [120]. It executes assembly-level instructions compiled from the source programs using the SUIF compiler [128]. The AVT simulator (AVTSIM) is an event-driven sequential simulator for distributed asynchronous systems [120]. AVTSIM models a distributed system consisting of processors connected by a network. Components (functional blocks) of the architectural datapath are simulated as processors (also called entities) of the AVT environment. Each entity has a number of inport and outport unidirectional connections to other entities of the architecture. The number and type of these connections depend on the particular entity and its functionality. For example, the entity that simulates the issue unit (IU) has N (the number of functional units (FU)) outport connections used to transfer instructions to the communication units (CU), an inport from the fetch and branch unit (FBU) used to transfer instructions into the instruction buffer (IB), and another N connections to CUs used to initiate data-forwarding operations (as shown in Figure 5.2 and Section 5.5.2).

The AVT simulation algorithm views the system under simulation as a collection of *logical processes* (LP). The simulation is realised through communication and cooperation of LPs, each of which simulates a part, in space and time, of the simulation model. The simulation models in AVTSIM are described in terms of LPs (entities), inputs and outputs, implemented by the classes `Entity`, `InPort` and `OutPort`, respectively. Entities can have an arbitrary number of inputs and outputs. In the simulator for the network-based architecture[1] inports and outports are simulated using `AsyncInPort` and `AsyncOutPort` classes. These classes are used to simulate the asynchronous communication and asynchronous channels based on bundled data (single-rail) encoding and the four-phase handshaking protocol (see Section 4.4 for description of encoding schemes and handshaking protocols). This encoding scheme was chosen because it is known not to be secure against side-channel analysis [136, 103], and one of the objectives of this research is to investigate to what extent the non-determinism introduced by the architecture improves resistance to power analysis. The choice of unprotected implementation is, therefore, appropriate from this perspective. Each inport connec-

[1]The simulator for the network-based architecture is in the rest of this chapter referred to as the *simulator*, and is not to be confused with AVT simulator, *AVTSIM*.

tion in the simulator is implemented with the `AsyncInPort` class that contains class members:

```
InPortTrace<Type> data;
InPortTrace<bool> req;
OutPortTrace<bool> ack;
```

Each outport connection is implemented with the `AsyncOutPort` class that contains class members:

```
OutPortTrace<Type> data;
OutPortTrace<bool> req;
InPortTrace<bool> ack;
```

where `InPortTrace` and `OutPortTrace` are classes inherited from `InPort` and `OutPort` classes, respectively, and are used to intercept and record all the signal value changes as a part of the execution profiling mechanism.

In the architectural simulator, each functional block is modelled as a separate LP. The LPs communicate with each other by a mechanism (part of AVTSIM) that is used to synchronise activities, receive and propagate events. Each LP maintains a local simulation time clock, which indicates the time-stamp of the most recent event processed, some local state, and a list of time-stamped events that have not yet been processed.

Entities react when they perceive a change at one or more input ports. An event-message carrying the same value as before is not considered to be an event. The incoming events are processed in time-stamped order. The incoming event may cause the local state to be updated, and outport events to be generated and propagated. The time-stamp of the generated outport events must be greater than or equal to the simulation time when the inport event was received and processed. The number of generated outport events depends on the inport event and the entity's functionality. For example, when an instructions is received in the CU, the CU might generate `grantRead` and `reserveLock` requests (see Section 5.5.4.1 for details) at the outports connected to the register bank arbiter (RBA) or in case of a `store` instruction, for which the source operands are forwarded (*i.e.* in the case of an instruction that does not have a destination operand and the source operands are not obtained from the register file), it does not generate any events, but waits for the operand value(s) to be forwarded. The delays on the output ports depend on the events and on the entity, as will be explained in Section 6.2.2. Message delay, as defined in [120], is the amount of simulated real time required to send an event-message from one processor to another. In the architectural

simulator included in the message delay is the figure for the delay of the functional block that had generated the event. Further details are given in Section 6.2.2.

At the highest level the architectural model is described in a structural form, detailing the interconnections between datapath components. The AVT provides the platform for exchanging messages between entities and time-keeping. The AVTSIM maintains the global time and associates a time reference with each event. The simulator progresses with time and executes those events that have the same time-stamp. The simulation finishes when the event list in the AVTSIM becomes empty. The simulated time is measured in so-called virtual time units (VTU) [120], which do not exactly correspond to real-world time units, but are interpreted for the particular simulation model. It should be noted that comparisons were made for a common underlying simulation model with different architectural configurations and, thus, a particular meaning to VTUs was not given in the architectural simulator.

The functionality of the architectural components was modelled at the micro-operation and register transfer level, providing explicit description of the interaction between components of the datapath, but the architecture was not modelled at the gate level. The functionality of an entity is described in method `evaluate()` which is called each time at least one value on the inports of the entity changes. During the current call of `evaluate()`, the entity can generate none, one or more events on its outports, and is considered busy until the time-stamp of the last output generated. The handshake based communication between entities ensures that the handshake that has been started will have no messages lost, but some events could be lost if (due to the busy state of the entity) the initial requests of the handshake is not perceived. This is undesirable characteristic of the AVTSIM especially when simulating a processor architecture. A solution to this problem in the architectural simulator required restriction on the length of message delays, providing more balanced message delays. In this way, it is ensured that the time period during which the entity is considered busy is not longer than the delays of incoming events. Although this solution is ad-hoc and somewhat reduces the possibility for exploring the variety of delays, it is considered acceptable in the current simulation model.

6.2.2 Parametric model

The description of the architecture is parametrised in order to include different types of functional units and different connectivities between them. The following can be configured:

- Network topology
- Network size (*i.e.* number of functional units)
- Number of shares used in data-forwarding
- Component delays
- Power consumption figures for various components and buses
- Size of the register bank
- Size of the data memory

Although the size of the register bank and the data memory can be specified in the configuration file, they are not as freely varied as the network topology and size, because the former depend on the requirements of the compiled programs that are run on the architecture. However, these parameters are configurable and the architectural configuration can be easily adjusted to accommodate the requirements of the code that runs on it.

The instruction set (listed in Appendix B) and the implemented functionality of each functional unit type are hard-coded. The number of source and destination operands is set to a maximum number of two, while the second destination operand is used only in the case of PC-related instructions, in which the first operand is the PC register (`pcReg`) and the second operand is the link register (`linkReg`) (the purpose of the `linkReg` is explained in Section 5.5.5.2). All other instructions have none or one destination operand.

The completion time of a functional unit in an asynchronous architecture depends on both static and dynamic factors [118]. The type of the functional unit determines the static factor, whereas the type of functional unit operation determines a range of delays, which are used to model dynamic factors such as input data. The overall delay of the functional unit is, therefore, specified to be a value in a range bounded by a minimum and a maximum value, as shown in Figure 6.1. Some of these values are based on SPICE simulations from a prototype of a micronet datapath [21]. Unfortunately,

more accurate values were not available for this implementation. In the experimentation, two delay models were used: one where delays are fixed (non-variable), and one where the delays were chosen to be a random value from the specified range, as given in Figure 6.1. The random model attempts to reflect delay values with reasonable variances, so that operations are not considered to complete in fixed times. To reduce the number of power readings required to successfully perform the attacks, and still give an accurate evaluation, power simulations were most extensively run for configurations with non-variable delays. In addition, this delay model provides the measure of non-determinism that is due to execution rather than the data. The random delay model is somewhat optimistic as the actual delays of operations would be data-dependent rather than random. In addition, when performing the timing analysis evaluation this type of model would be inadequate. However, this model has not taken a significant part in the experimentation so its inaccuracy is less crucial. The impact of a true data-dependent delay model on security of the architecture would be more present in the evaluation of the security against timing analysis which is not part of this research. From the standpoint of power analysis, data-dependent delays could be useful in evaluation but are not crucial, as the strength of the attack is in ability to align operations that deal with sensitive data. In the delay model with fixed delays this process is even easier. The separate issue is the vulnerability of asynchronous logic to timing analysis which is not part of this research.

The delays specified in the Figure 6.1 imply that some operations are more expensive than others. This configuration presents arithmetic and logic units which are relatively faster than the memory unit, since a memory operation may include an addition, and the actual process of loading from, and storing into, memory is relatively slow [118]. Similarly, multiplications are considered more costly than arithmetic or logic operations. The fetch and branch unit and the issue unit are considered faster then any other component in order to model a fast single-issue architecture in which functional units are kept busy.

All significant operations in the architectural simulator, such as computing results and sending event-messages, are assigned costs. The assigned cost is given in the form of a delay and power. When an event is created a delay is assigned to it. This delay specifies the time after which the AVT scheduler will process the generated event. The power profiling features are not part of the AVTSIM environment and are implemented separately. The architecture was implemented to support two power models: Hamming weight model and the transition count model, which are explained in Section 6.3.1.

COMPONENT TYPE	COMPONENT DELAY	
	MIN	MAX
ARITHMETIC UNIT (AU)	2	4
LOGIC UNIT (LU)	2	4
MULTIPLIER UNIT (MULT)	4	8
MEMORY UNIT (MU)	10	20
REGISTER BANK (RB)	2	4
OTHER (FBU, IU, RBA, MA)	1	2

Figure 6.1: Delay distribution for different architectural components in virtual time units (VTUs).

The assigned power cost depends on the power model implemented. For example, the generation of an event in the configuration with the Hamming weight power model, consists of the following operations:

```
/*place the data on the bus and consider it available after the delay */
outport->data.presentAfterDelay(data, delay);

/*record the power assigned with this operation*/
power_logger->logPowerEvent(Hamming_Weight(data)*base_power,
   start_time, start_time + delay, entity_name, inst_tag);

/*trigger the request signal after the delay */
outport->req.triggerAfterDelay(1, delay);

/*record the power associated with this operation*/
power_logger->logPowerEvent(base_power, start_time, start_time
   + delay, entity_name, inst_tag);
```

The simulator also implements the execution profiling features, and records every change of the signals on the wires. This allows for all events specific to a particular instruction to be easily extracted. Each time an event is placed on the output port, the change on the bus (wire) is recorded in the event logger:

```
/*record the change on the wire at start_time */
event_logger->logEvent(start_time, entity_name, inst_tag, colour,
   data);
```

6.2.3 SUIF compiler

The Stanford University Intermediate Format (SUIF) [128] is a compiler development framework, which provides the necessary infrastructure for performing a range of compiler optimisations, from high-level transformations to dataflow optimisations. The optimisation transformations can be performed progressively and interchangeably over multiple passes, each producing the same output format. This enables the passes to be reordered by running them in a different order. The SUIF toolkit contains a variety of compiler passes, which can be inserted freely at any point during the compilation. An ANSI C front end is available to translate source programs into SUIF intermediate representation, and after passing a number of SUIF passes, the system can directly produce MIPS code. The assembly-level code run on the architectural simulator is SUIF-generated MIPS code compiled from C implementations.

6.2.4 Power profiling

To investigate the correlation of the power traces to the source code the simulation was equipped with power and execution profiling features. Power dissipation in CMOS circuits largely depends on the switching activity caused by changes in input data [49], and the power profiling was implemented accordingly. However, the power profiles represent the worst-case scenario where all the noise produced by the processor has been removed. Due to the independent data-driven operation of each subcircuit, asynchronous architectures have random timing variations, which make signal averaging in the differential power analysis problematic. To reduce the number of power readings required to successfully perform the attacks, and still give an accurate evaluation, power simulations were most extensively run for configurations with non-variable delays. In this way a smaller number of power traces were needed for (in some cases) successful application of power-based attacks. In addition, non-variable delay configurations provide a measure of the introduced non-determinism. The architectural simulator is implemented to support both variable and non-variable delays, and the results of both types of simulations are given in Section 6.3.

6.3 Security evaluation

The security evaluation process is given in Figure 6.2. This process starts with the C implementation of the cryptographic algorithm, which is then compiled into MIPS code that is run on the architecture simulator. The outputs of the architectural simulation are the execution and power profiles that used to generate the power traces used in power analysis. If successful, the analysis should yield the values of the secret key. The following section gives further details about the experimental setup.

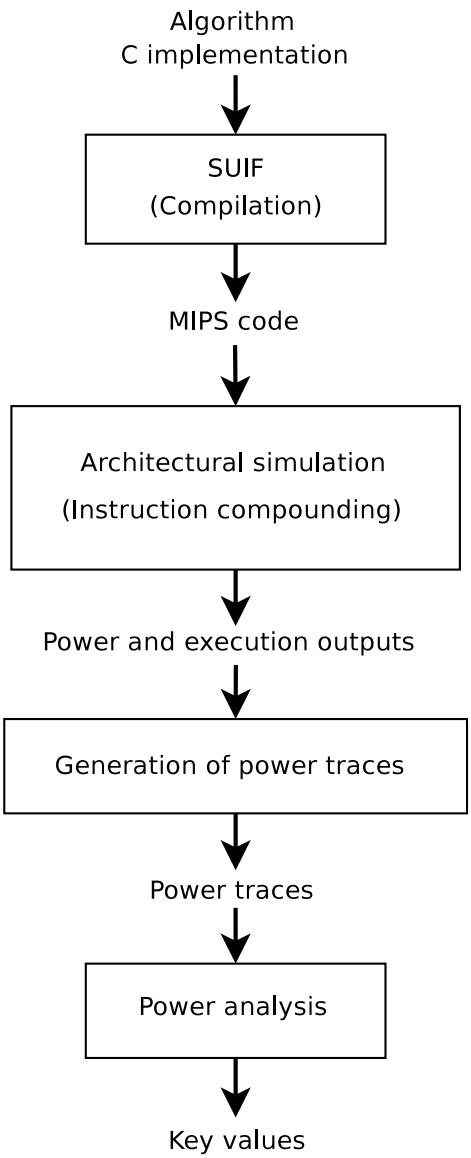


Figure 6.2: Security evaluation process.

6.3.1 Experimental setup

The goal of the research presented in this thesis is to investigate the impact to the security of non-determinism introduced in the execution of the network-based asynchronous architecture. In order to investigate the amount of non-determinism that different architectural configurations yield, differential power analysis attacks were performed on power traces obtained from simulating a number of different configurations in which the following parameters were varied:

1. ***Network topology and size:*** The architecture is implemented to support two network topologies: binary hypercubes and binary de Bruijn networks, as explained in Section 5.6.1. The definition of binary hypercube is extended to support any number of nodes, and is termed *partial binary hypercube*. The vertices of a partial binary hypercube of size N, $PH(N)$, are labelled with n-bit integers from 0 to $N-1$, where $2^{n-1} \leq N \leq 2^n$. A vertex $x = (x_{n-1}, x_{n-2}, \ldots, x_0)$ in $PH(N)$ is adjacent to vertices:

   $$x' = (x_{n-1}, x_{n-2}, \ldots, x_{i+1}, \overline{x_i}, x_{i-1}, \ldots, x_0),\ if\ x' \leq N,$$

   and

   $$x'' = (0, x_{n-2}, \ldots, x_{i+1}, \overline{x_i}, x_{i-1}, \ldots, x_0),\ s.t.\ x'' \neq x\ and\ x'' \neq x',\ otherwise,$$

   for $0 \leq i \leq n-1$.

   The vertices of a binary de Bruijn network of size N, $DB(N)$, are labelled with integers modulo N. An edge is drawn from vertex x to vertex y, if

   $$y \equiv 2x + \alpha\ (mod\ N),\ \alpha \in \{0,1\}.$$

   This choice of network topologies was made because: (1) routing in these topologies is oblivious, *i.e.*, the path taken by a packet through the network is a function of its origin and destination; (2) randomised routing protocols can be efficiently implemented in these topologies.

   In the experiments the size of the network was varied and the distribution of different functional units in each of these variations is shown in Figure 6.3.

2. ***Cryptographic algorithm / side-channel attack:*** Detailed power simulations of AES and DES cryptographic algorithms were performed. The power traces obtained from simulating AES were subjected to a variation of the *covariance attack* [35], that is based on observing the covariance between the plaintext bits

#FU	#AU	#LU	#MULT	#MU
4	1	1	1	1
6	2	2	1	1
7	2	2	1	2

Figure 6.3: Distribution of functional units (FU) among arithmetic (AU), logic (LU), multiplier (MULT) and memory (MU) units.

and the power values of the initial key addition at every sample point for each key bit. The power traces obtained from simulating DES were subject to the DPA attack [78], that is based on observing the differential power trace of the guessed key bits of the two distributions of power curves specified by the guessed key bits.

3. ***Power model:*** The architecture is implemented to support two power models [97]:

   (a) *Hamming weight model*, in which the power consumed is proportional to the number of 1s written onto the bus.

   (b) *Transition count model*, in which the power consumed is proportional to the number of bits that change on the bus.

   These two power models have been observed by Messerges et al. in [97] and resemble the two types of data bus information leakage. The power profiling in this implementation takes into account the power consumed by data buses, the register bank and the memory, according to one of these two power models. The power leakage from the computational logic is taken into account, but it is considered fixed for each functional unit and is not data-dependent. The importance of the leakage from the computational logic is not diminished in this way, but rather, on this level of architectural implementation, observing this type of leakage would not be possible to simulate and any other model is likely to be unrealistic. The leakage from the computational logic is data-dependent and simulating it would require a gate-level implementation of the architecture. Furthermore, this type of experimental evaluation of resistance to power analysis would not possible at that level as it would be extremely time consuming. Designing logic gates that do not leak information during computation is outside the scope of this research which focused on the architectural aspects.

4. ***Data-forwarding mode:*** Forwarding critical register values through the network induces power consumption directly dependent on the sensitive data. An idea to remove this direct contribution was to split data into shares. Data-forwarding in the form of k data-shares consists of generating $k-1$ random values: $r_1, \ldots, r_{k-1}$, and forwarding k values: $d \oplus r_1 \oplus \cdots \oplus r_{k-1}, r_1, \ldots, r_{k-1}$, where d is the register value. This provides blinding of critical register values that are written onto the forwarding buses multiple times during data-forwarding. In addition, this process increases the network activity (congestion), which results in further timing variations and increased noise in the power traces. In the simulations the number of shares was varied to characterise configurations with data-sharing, in which the number of shares is $k = 2$, and configurations without data-sharing, in which number of shares is $k = 1$.

The experiments consisted of varying these parameters over the following sic architectural configurations:

PIPE: A linear single-pipeline asynchronous architecture. Data-forwarding is not implemented in this configuration, but the result of the previous instruction is reused. This is simulated with the configuration of the network-based architecture in which the number of functional units is equal to one.

PH*N***:** A configuration of the network-based asynchronous architecture in which the network is a partial hypercube of size N, $PH(N)$. The number of shares in data-forwarding is equal to one, *i.e.*, data-forwarding does not support data-sharing.

PHS*N***:** A configuration of the network-based asynchronous architecture in which the network is a partial hypercube of size N, $PH(N)$. The number of shares in data-forwarding is equal to two.

DB*N***:** A configuration of the network-based asynchronous architecture in which the network is a de Bruijn network of size N, $DB(N)$. The number of shares in data-forwarding is equal to one, *i.e.* data-forwarding does not support data-sharing.

DBS*N***:** A configuration of the network-based asynchronous architecture in which the network is a de Bruijn network of size N, $DB(N)$. The number of shares in data-forwarding is equal to two.

ASYNC*N***:** An asynchronous architecture with N functional units which supports concurrent execution in the same manner as network-based asynchronous archi-

tecture but in which data-forwarding is turned off, and the network effectively does not exist. Similar to the PIPE configuration, the result of the previous instruction in each of the functional units can be reused.

6.3.2 Covariance attack on AES

The covariance attack [35] was performed on the runs of AES. The covariance of the two variates is defined as follows:

Definition 6.3.1. *Given two sets of variates $X = \{x_i\}$ and $Y = \{y_j\}$ the covariance, $\sigma_{ij} \equiv cov(x_i, y_j)$, is defined as:*

$$cov(x_i, y_j) \equiv \langle (x_i - \mu_X)(y_j - \mu_Y) \rangle,$$

where $\langle \cdot \rangle$ represents the mean operation and $\mu_X = \langle x_i \rangle$ and $\mu_Y = \langle y_j \rangle$ are means of X and Y, respectively.

The covariance of two variates provides a measure of how strongly these variates are correlated. Thus, the covariance of the two independent variates is equal to zero. If $cov(x_i, y_j) > 0$, then Y tends to increase as X increases, and if $cov(x_i, y_j) < 0$, then Y tends to decrease as X increases.

The covariance attack performed on the runs of AES can be summarised in the following three steps:

Step 1 – Obtain the power samples of the initial key addition for several encryptions with the same key and different plaintexts.

Step 2 – For each bit i in the plaintext input, calculate the covariance between the bit and power samples of the runs at every sample point.

Step 3 – For each bit i, investigate the covariance plot for all the sample points. It should be flat except for a few strong peaks. The first significant peak should correspond to the `write` of the plaintext to the register bank. The peak at another important sample point should correspond to the `write` to the register bank of the result of the initial key addition. If these peaks have the same sign, then the i-th key bit is equal to zero, otherwise it is equal to one.

The covariance attack is based on observing the differences in the covariance values derived from the power traces for the initial key addition at several reference points,

for each key (plaintext) bit. The covariance plot for the i-th plaintext bit measures the covariance between the values of the i-th plaintext bit and the power values at each sample point. The two critical sample points, that are observed in the plots, correspond to the covariance between the plaintext bit value and the power value before and after the initial key addition, *i.e.*, before and after the plaintext bit b_i was XORed with the key bit k_i. The signs of the two peaks determine whether the value of the plaintext bit was flipped during the XOR operation, thus the plot for the i-th plaintext bit reveals the value of the i-th key bit. Namely, if the two peaks have different signs, then the value of the plaintext bits was flipped during the initial key addition operation, and the value of the corresponding key bit is equal to one. If the two peaks have the same sign, then the value of the plaintext bit was not flipped, and the key bit is equal to zero. This attack is based on the assumption that the contributions to the power mean when zero and one bits are written on the bus (register bank) are different. Otherwise, the attack would not be successful [35].

6.3.2.1 Results

The covariance attack was first performed on the single-pipeline asynchronous architecture, PIPE, with the Hamming weight power model and non-variable delays. The covariance plots revealed the secret key with as few as 200 power profiles. A sample covariance plot obtained from this experiment is shown in Figure 6.4. The covariance plot is as expected, and specified in the attack description, mostly flat except for a few strong peaks and correspond to the initial key addition.

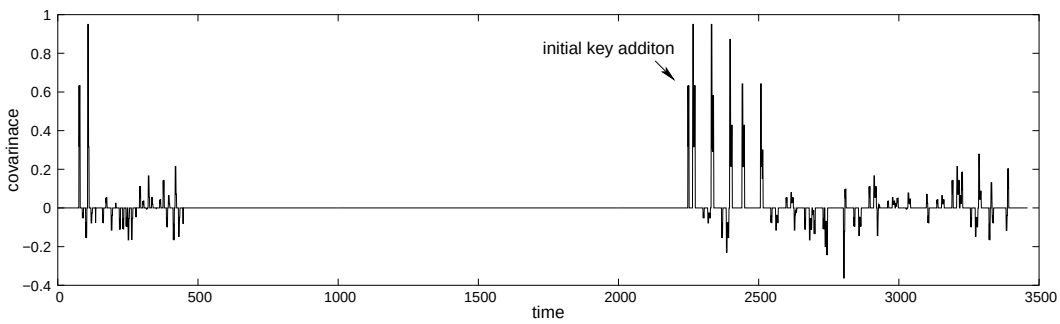
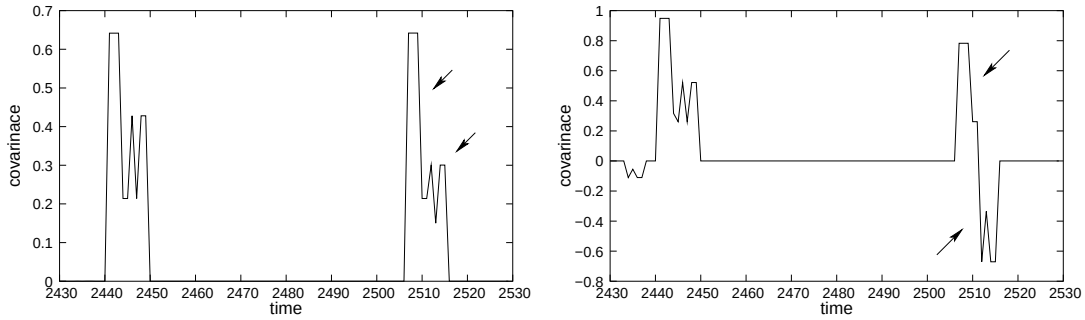


Figure 6.4: A sample covariance plot for the PIPE configuration with the Hamming weight power model and non-variable delays, derived from 200 power profiles.

Figure 6.5(a) shows the portion of covariance plot for the first key bit that corresponds to the initial key addition. The same sign of the two marked peaks suggests

that the value of the first key bit is equal to zero. Similarly, Figure 6.5(b) shows the same portion of the covariance plot for the fourth key bit. The different signs of the two peaks suggest that the value of the fourth key bit is equal to one. This experiment demonstrates that the *single-pipeline architectural configuration with the Hamming weight power model and non-variable delays is unprotected against the covariance attack, which was easily mounted using only 200 power samples.*



(a) 1st key bit - value 0.

(b) 4th key bit - value 1.

Figure 6.5: The covariance attack on the PIPE configuration with the Hamming weight power model and non-variable delays.

In the PIPE configuration the instructions were executed one at a time on a single-pipeline architecture. The instruction issue and completion are in-order. In order to investigate whether concurrent execution on its own yields any security improvements, the covariance attacks were performed on configurations of the network-based architecture in which data-forwarding is turned off, namely ASYNC4 and ASYNC6 configurations, with the Hamming weight power model and non-variable delays. In these configurations the network effectively does not exist: the instructions operate concurrently, but communicate only through the register bank. The instructions are issued in-order to multiple units that operate at their own pace, but again, due to fixed delays, the non-determinism that is found in asynchronous architectures executing in this manner is not present. The goal was to investigate whether parallel execution can contribute enough noise to make the attack more difficult to apply. Figures 6.6 and 6.8 show sample covariance plots for configurations ASYNC4 and ASYNC6, respectively. These plots are mostly flat with a number of peaks that correspond to the operations when the value of the plaintext bit was manipulated, *i.e.* initial key addition. Performing the attack on these configurations has shown that 300 power samples were sufficient

to read the secret key from the covariance plots. Figures 6.7 and 6.9 demonstrate the ease with which the key bit values can be extracted from the covariance plots for each of the configurations. Considering that the AES was broken with a low number (300) of power samples in both cases, these results lead to conclusion that the *parallelised (non-randomised) execution on it own does not contribute sufficient noise to provide any security improvements*.

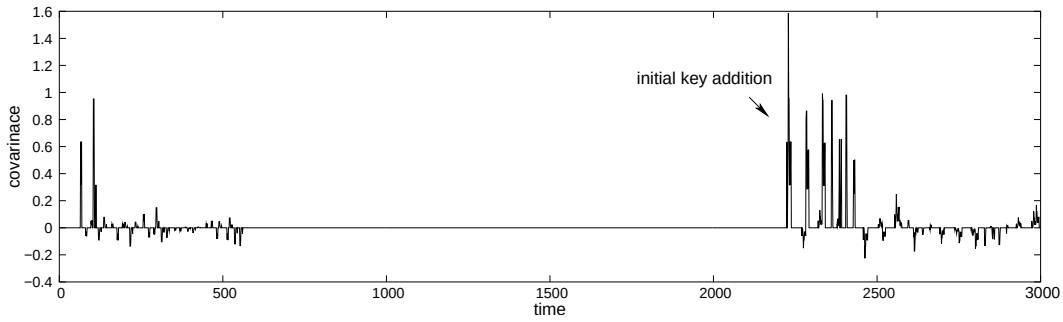
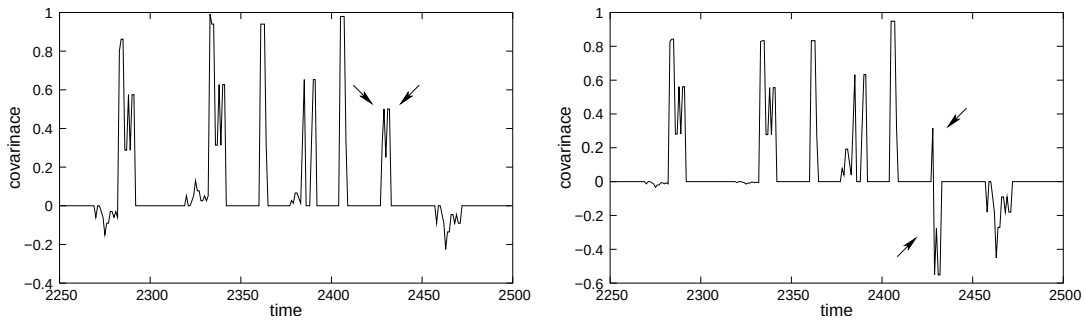


Figure 6.6: A sample covariance plot for the ASYNC4 configuration with the Hamming weight power model and non-variable delays, derived from 300 power profiles.



(a) 1st key bit - value 0.

(b) 3rd key bit - value 1.

Figure 6.7: The covariance attack on the ASYNC4 configuration with the Hamming weight power model and non-variable delays.

Next, the covariance attack was performed on three configurations of the network-based architecture in which data is not forwarded in the form of data-shares and the network topology is a partial hypercube, *i.e.*, PH4, PH6 and PH7 configurations with the Hamming weight power model and non-variable delays. Configuration PH4 specifies a network-based architecture with four functional units: arithmetic, logic, multiplier and memory unit. The delays are non-variable, which on one hand reduces the number

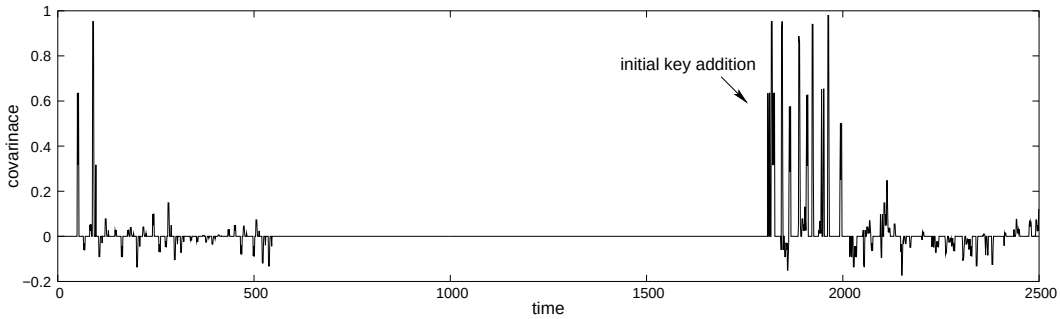
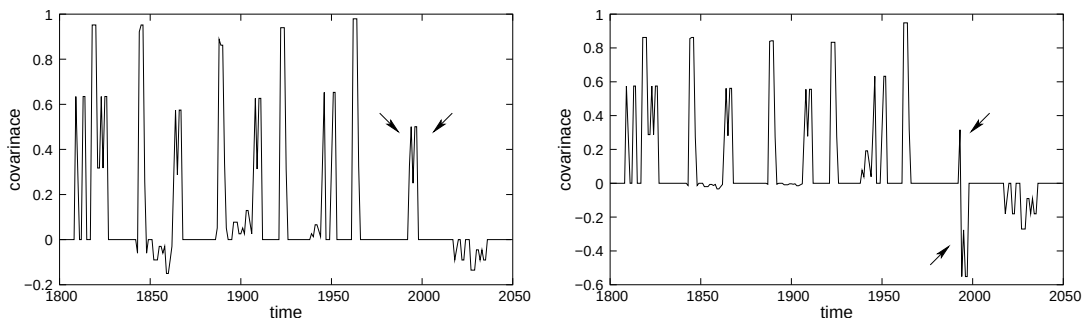


Figure 6.8: A sample covariance plot for the ASYNC6 configuration with the Hamming weight power model and non-variable delays, derived from 300 power profiles.



(a) 1st key bit - value 0.

(b) 3rd key bit - value 1.

Figure 6.9: The covariance attack on the ASYNC6 configuration with the Hamming weight power model and non-variable delays.

of necessary readings to successfully perform the attack, and on the other hand, demonstrated the direct contribution of randomised data-forwarding to non-deterministic execution. In the covariance attack on the PH4 configuration, 5000 power profiles were needed to derive covariance plots from which the values of the secret key could be read. A sample covariance plot obtained from this experiment is shown in Figure 6.10. The portion of the covariance plot that corresponds to the initial key addition can be easily observed, while the rest of the plot is mostly flat. Figures 6.11(a) and 6.11(b) show portions of the covariance plots for the first and fourth key bit, respectively, that correspond to the initial key addition. The signs of the peaks revealed that the value of the first bit is equal to zero, and the value of the fourth bit is equal to one.

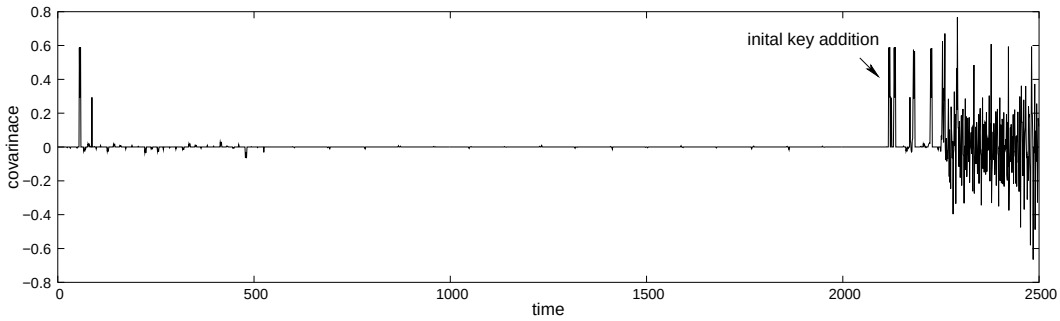
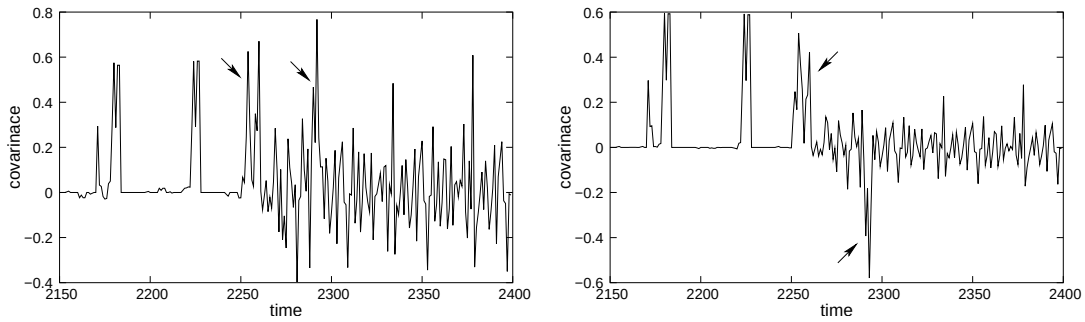


Figure 6.10: A sample covariance plot for the PH4 configuration with the Hamming weight power model and non-variable delays, derived from 5000 power profiles.



(a) 1st key bit - value 0.

(b) 4th key bit - value 1.

Figure 6.11: The covariance attack on the PH4 configuration with the Hamming weight power model and non-variable delays.

As the majority of operations in the AES implementation were arithmetic and logic operations, instruction-level parallelism was expected to be exploited to a higher ex-

tent with an increase in arithmetic and logic units. Thus, the attack was performed on configurations PH6 and PH7 (PH6: 2 AUs, 2 LUs , 1 MULT and 1 MU; PH7: 2 AUs, 2 LUs, 1 MULT and 2 MUs) with the Hamming weight power model and non-variable delays. For configurations PH6 and PH7, 25000 and 50000 power profiles, respectively, were needed to derive covariance plots from which the values of the key bits could be read. Although the covariance plots were considerably noisy, the portion of the traces that corresponded to the initial key addition could be pin-pointed, as shown in Figures 6.12 and 6.15. However, it was difficult to distinguish the covariance plots for different key bit values, as the two peaks (or groups of peaks) that correspond to the two critical operations in the attack were smeared across the covariance plots. By carefully comparing covariance plots for several key bits, the plots for the bits of value zero were distinguished from the ones for the bits of value one, which was then used to discover other key bits. Figures 6.13 and 6.16, which represent the comparisons of covariance plots for the first key bit (value 0) and the fourth key bit (value 1), respectively, demonstrate this approach.

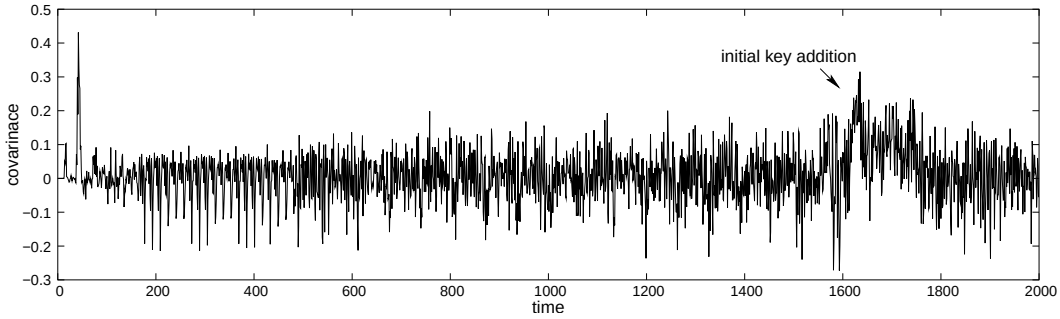


Figure 6.12: A sample covariance plot for the PH6 configuration with the Hamming weight power model and non-variable delays, derived from 25000 power profiles.

Figure 6.14 shows the comparison of covariance plots for the first and the fourth key bits derived from 5000 power samples from the PH6 configuration. No significant difference between the two plots can be noticed from this comparison. This examples demonstrate what is meant by “observing” the covariance plots and shows that the covariance plots derived from the same number of power samples, as those for configuration PH4, in this case did not provide any key bit information.

To measure the contribution of data-forwarding to data-dependent power consumption, attacks were performed on configurations where data is forwarded in the form of two data-shares, *i.e.*, configurations PHS4, PHS6 and PHS7 with the Hamming

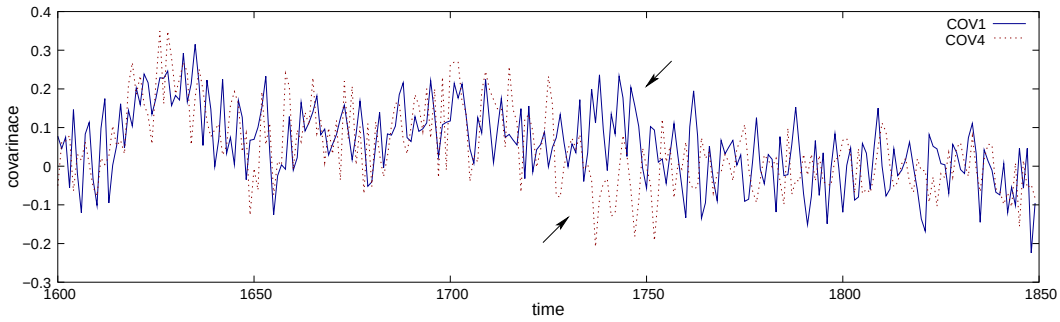


Figure 6.13: The covariance attack on the PH6 configuration with the Hamming weight power model and non-variable delays. COV1 and COV4 are covariance plots for the 1st (value 0) and the 4th key bit (value 1).

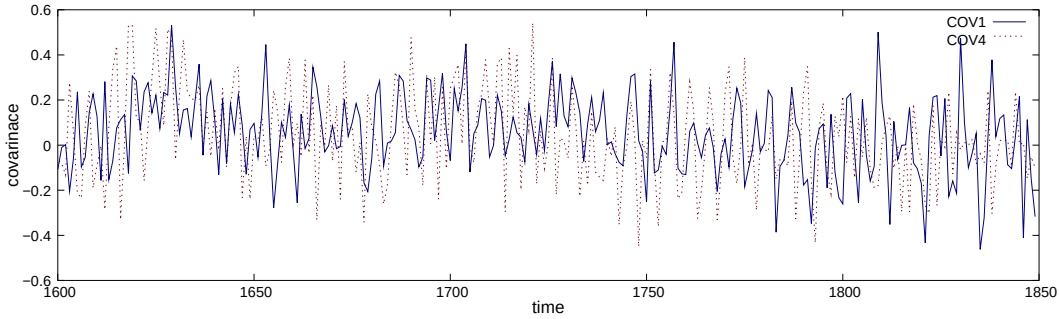


Figure 6.14: The covariance attack on the PH6 configuration with the Hamming weight power model and non-variable delays using 5000 power samples. COV1 and COV4 are covariance plots for the 1st (value 0) and the 4th key bit (value 1).

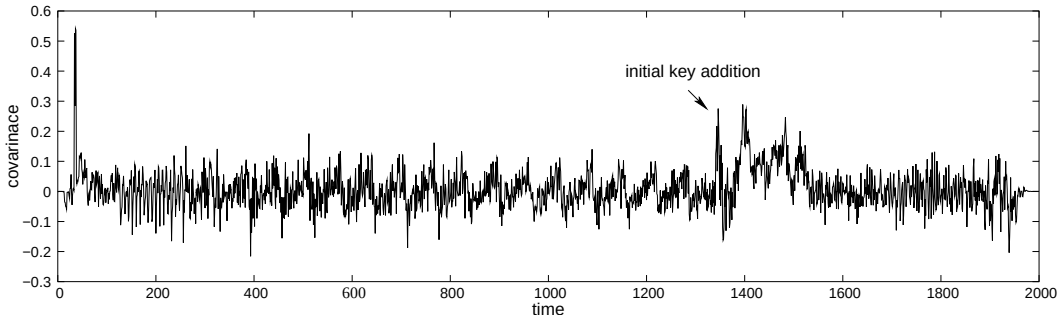


Figure 6.15: A sample covariance plot for the PH7 configuration with the Hamming weight power model and non-variable delays, derived from 50000 power profiles.

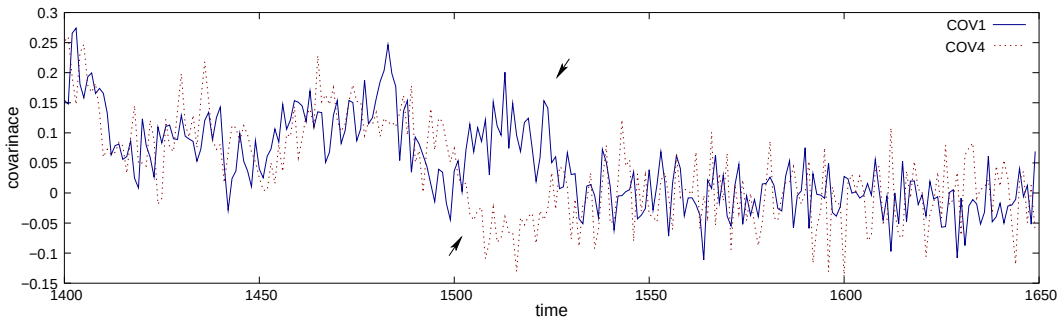


Figure 6.16: The covariance attack on the PH7 configuration with the Hamming weight power model and non-variable delays. COV1 and COV4 are covariance plots for the 1st (value 0) and the 4th key bit (value 1).

weight power model and non-variable delays. The enhancement to the architecture to support data-forwarding with two data-shares consisted of generating a random value r and forwarding two messages: $d \oplus r$ and r, where d is the data value. This increases the number of messages in the network and introduces additional non-determinism as the progress of each packet in the network depends on the number of messages that are routed. The number of power samples required to successfully perform the attack increased to 35000, 60000 and 75000 for PHS4, PHS6 and PHS7 configurations, respectively. Figures 6.17, 6.18 and 6.19 show the comparisons of covariance plots for different key bits for each of these configurations. These results show that *although parallel execution and randomised data-forwarding hide the information leaked in the form of data-dependent power consumption, data-forwarding contributes its own data-dependent leakage*. As demonstrated, *by splitting the data into shares, this leakage can be reduced and the attack threshold further increased.*

The covariance attack was also performed on the de Bruijn network configurations, *i.e.*, DB and DBS configurations, with the Hamming weight power model and non-variable delays. The sample covariance plot for the DB4 configuration shown Figure 6.20 was generated from 35000 power profiles. When compared to the plot for PH4 configuration (Figure 6.10) that was obtained from 5000 power profiles, it was observed that this plot is much noisier. Further observation was that the difference between the covariance plots for different key bits is in this case finer, and it was considerably harder to distinguish the covariance plots for different key bits. The peaks that correspond to critical operations are now further smeared across the covariance plot and were not as easy to spot.

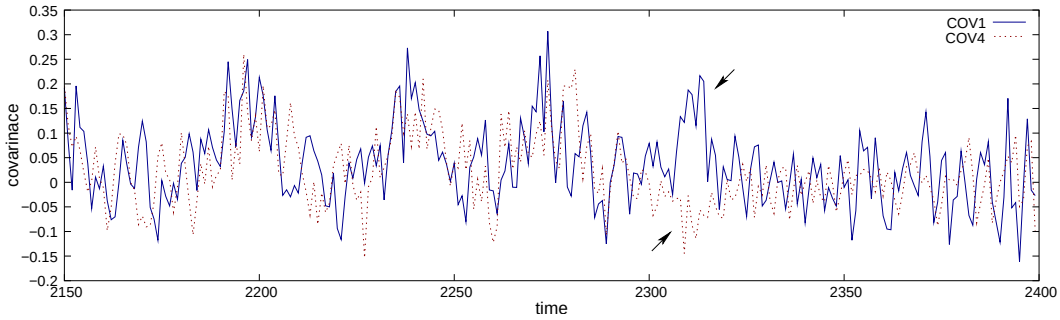


Figure 6.17: The covariance attack on the PHS4 configuration with the Hamming weight power model and non-variable delays derived from 35000 power profiles. COV1 and COV4 are covariance plots for the 1st (value 0) and the 4th key bit (value 1).

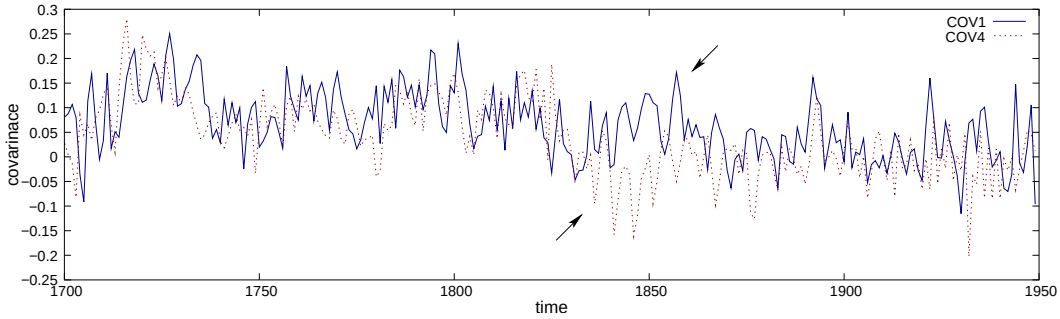


Figure 6.18: The covariance attack on the PHS6 configuration with the Hamming weight power model and non-variable delays, derived from 60000 power profiles. COV1 and COV4 are covariance plots for the 1st (value 0) and the 4th key bit (value 1).

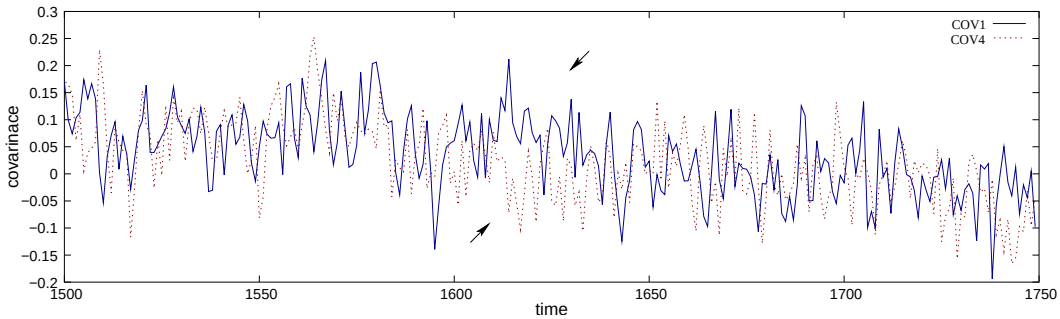


Figure 6.19: The covariance attack on the PHS7 configuration with the Hamming weight power model and non-variable delays, derived from 75000 power profiles. COV1 and COV4 are covariance plots for the 1st (value 0) and the 4th key bit (value 1).

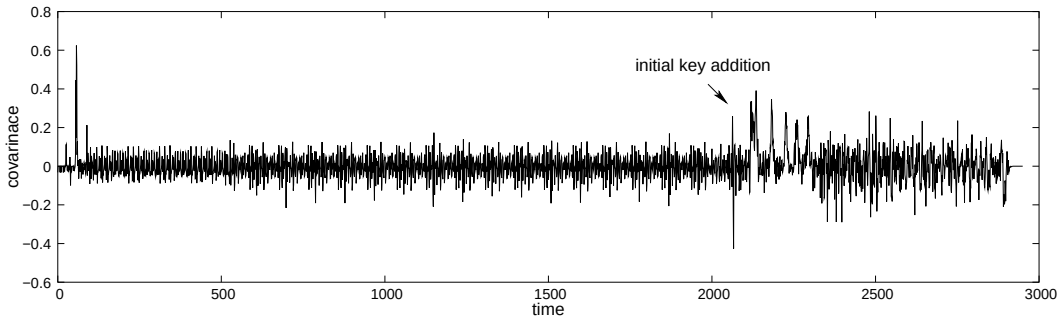


Figure 6.20: A sample covariance plot for the DB4 configuration with the Hamming weight power model and non-variable delays, derived from 35000 power profiles.

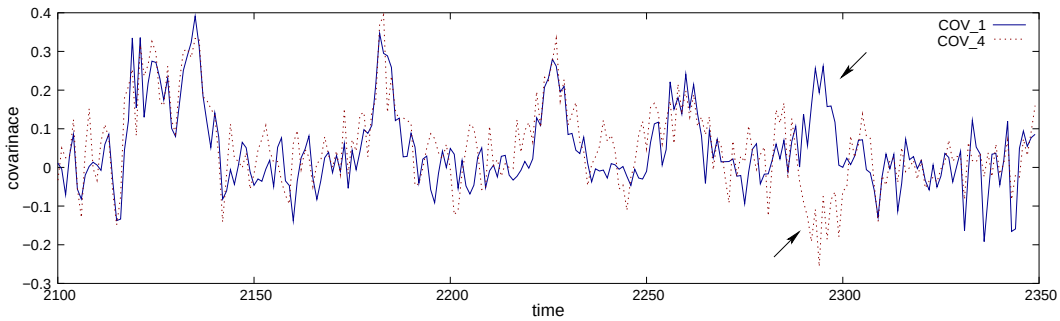


Figure 6.21: The covariance attack on the configuration DB4 with the Hamming weight power model and non-variable delays. COV1 and COV4 are covariance plots for the 1st (value 0) and the 4th key bit (value 1).

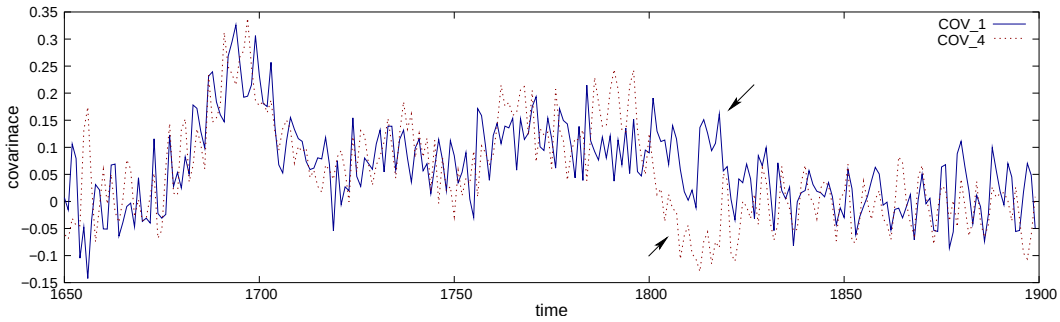


Figure 6.22: The covariance attack on the configuration DB6 with the Hamming weight power model and non-variable delays, derived from 85000 power profiles. COV1 and COV4 are covariance plots for the 1st (value 0) and the 4th key bit (value 1).

Figure 6.21 shows the comparison of covariance plots for the first (value 0) and the fourth (value 1) key bit, obtained from the covariance attack on configuration DB4. The covariance plots were derived from 35000 simulations and contained sufficient information for the plots for different key bit values to be distinguished. Similarly, Figure 6.22 compares covariance plots for configuration DB6 obtained from 85000 power simulations. The success of this attack was dependent on the ability to determine the exact portion of the power trace in which the targeted operations took place. Although the portion that corresponds to the initial key addition can be easily observed in the plots, as shown in Figure 6.20, it is extremely noisy and peaks that correspond to critical operations can be confused with other noise. In these experiments, this was facilitated by the execution profiling feature that the simulation environment provided. In their absence, the precise knowledge of the algorithm source code and the parameters of the architecture, or a more sophisticated attack methodology would be necessary.

The complete results of the covariance attacks performed on different de Bruijn network configurations are given in Figure 6.23. Due to the fact that these experiments are extremely time-consuming[2], final figures for DBS6 and DBS7 configurations were not obtained, but the ones given suggest that data-sharing improves the resistance to the covariance attack, and confirms the benefit of using data-sharing in data-forwarding. A simple comparison of these results and those obtained for hypercube configurations (Figure 6.25), leads to the conclusion that *architectural configurations with de Bruijn network configurations provide more non-determinism when compared to hypercube network configurations.*

CONFIGURATION	NUMBER OF POWER SAMPLES	
	K=1	K=2
DB4	35000	50000
DB6	85000	> 90000
DB7	120000	> 100000

Figure 6.23: Number of power samples necessary to attack de Bruijn network configurations with the Hamming weight power model and non-variable delays.

[2]The fastest simulation of a single run of AES on the DB7 configuration on a 2.4 GHz Pentium 4 machine with 1GB of RAM, takes approximately 1 minute. The fastest simulation of a single run of DES on the DB7 configuration and the same machine, takes approximately 3 minutes. The statistical post-processing of the obtained data from multiple runs of these algorithms takes between 2 to 48 hours depending on the complexity of the attack, and it requires some manual processing connected to guessing the key bits from the obtained covariance/differential traces.

The next set of experiments consisted of performing the covariance attack on the power traces obtained from running AES on PIPE and PH configurations with the *transition count* power model and non-variable delays. The summary of these results is given in Figure 6.24. In case of the PIPE configuration, *most* of the key bits were readable from as little as 200 power plots. The attack on PH4 configuration yielded the first 16 bits from 30000 power profiles, but the rest of the guessed key bits were essentially random, if readable at all. The covariance plots from PH6 and PH7 configurations yielded essentially random key bit values, from 50000 and 100000 power profiles. These figures suggest that the *the architectural implementations with the transition count power model show considerably better security characteristics when compared to those with the Hamming weight power model.*

CONFIGURATION	NUMBER OF POWER SAMPLES
PIPE	200
PH4	> 30000
PH6	> 50000
PH7	> 100000

Figure 6.24: Number of power samples used in the attacks on PIPE and PH configurations with the transition count power model and non-variable delays.

The summary of the results of the covariance attacks on AES running on architectural configurations in which the network topology and size, power model and data-forwarding modes are varied, is given in Figure 6.25.

The architecture was implemented to support both variable and non-variable delays. The security evaluations are most extensively performed for configurations with non-variable delays, because the number of power samples required to successively perform the power analysis attacks is much reduced. To illustrate this fact, Figure 6.26 summarised the results of the performed covariance attacks on PH configurations with *variable delays*. For configuration PH4 with variable delays, 30000 power samples were required to extract secret key values from the power plots. In the case of PH6 and PH7 configurations 45000 and 100000, respectively, were not sufficient to correctly extract all the key bits from respective power samples. Figure 6.26 also shows the comparison of the number of samples necessary to attack the configurations with non-variable and variable delays. As expected, the number of samples is much reduced for non-variable delays (from 30000 to 5000 power samples for PH4 config-

CONFIGURATION	POWER MODEL		
	HAMMING WEIGHT		TRANSITION
	K=1	K=2	COUNT (K=1)
PIPE	200	–	200
ASYNC4	300	–	–
ASYNC6	300	–	–
PH4	5000	35000	$>$ 30000
PH6	25000	60000	$>$ 50000
PH7	50000	75000	$>$ 100000
DB4	35000	50000	–
DB6	85000	$>$ 90000	–
DB7	120000	$>$ 100000	–

Figure 6.25: Number of power samples used to perform the covariance attack on AES run on different architectural configurations with non-variable delays.

uration). Since the power simulations were considerably time consuming, the final figures for configurations PH6 and PH7 were not obtained. However, the presented figures testify that variability of delays in asynchronous architectures contributes towards complicating the application of the power analysis attacks. In some cases the variability of delays could be exploited to mount timing attacks [136], as the delays are data-dependent. However, when combined with non-deterministic execution in the network-based architecture, variable delays should work as additional noise and the extraction of information using the timing channel would not be as easy to achieve in this case.

CONFIGURATION	NUMBER OF POWER SAMPLES	
	NON-VARIABLE DELAYS	VARIABLE DELAYS
PH4	5000	30000
PH6	25000	$>$ 45000
PH7	50000	$>$ 10000

Figure 6.26: Number of power samples necessary to attack hypercube network configurations with the Hamming weight power model.

6.3.3 Differential power analysis of DES

This section reports on the results of performing DPA attacks on power samples obtained from running DES on the hypercube network configurations with the Hamming weight power model and non-variable delays. The DPA attack is performed on the first round of DES and therefore uses known plaintexts. In each of the 16 rounds of DES 8 *S*-box lookups are performed. The 6-bit input to each of the *S*-boxes is generated using the 6 bits of the round key and 6 bits of the intermediate result (register *R*). Therefore, a bit in the 4-bit output of an S-box is influenced by only 6 bits of the round key. As explained in Section 3.5.2, this fact can be used to discover the 48 bits of the first round key by repeatedly applying the attack on the outputs of the 8 *S*-boxes. The remaining 8 bits of the DES key can be discovered using exhaustive search. The steps of the attack can be summarised as follows (see Section 3.5.2 for further details of DES):

Step 1 – Obtain the power samples of the first rounds of DES for several encryptions with the same key and different plaintexts.

Step 2 – For each of the 64 (2^6) possible guesses for the 6 bits of the round key, partition the power traces according to the value of the bit that is output from the *S*-box (defined by the selection function) and calculate the differential power trace of the subpopulations for each partition.

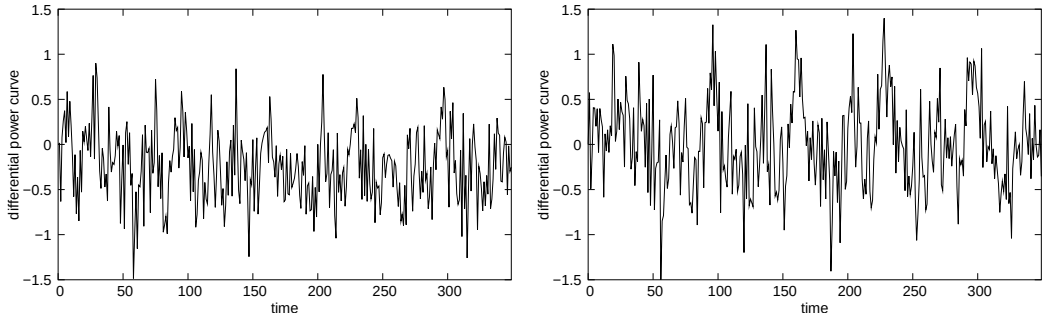
Step 3 – The differential power trace for the correctly guessed 6 bits of the key should show the largest biases, therefore the correct 6 bits correspond to the differential power trace with the largest peaks.

By defining the selection function to be equal to the value of the first bit output from an S-box, the 6 bits of the key that have influenced this bit value can be obtained. By repeating Steps 2 and 3 for the remaining 7 *S*-boxes the entire 48 bits of the first round key can be discovered.

6.3.3.1 Results

The differential power plots obtained from running DES on the PIPE configuration revealed 42 correct key bits from 750 power samples. No correct guesses were made from differential traces PH4 and PH6 configurations, derived from 40000 and 35000 power samples, respectively.

Figure 6.27 shows differential power plots for a correct and an incorrect key guess. The power simulations of the DES implementation in the architecture were extremely



(a) Differential trace of the correct guess. (b) Differential trace of the incorrect guess.

Figure 6.27: The DPA attack on 35000 power profiles obtained from running DES on PH6 configuration with the Hamming weight power model and non-variable delays.

time consuming which is the reason why further security evaluation is not carried out. The figures suggest that the DES algorithm run on the same architectural configuration is more resistant to differential power analysis attack when compared to the AES algorithm. However, one should bear in mind that the side-channel attacks are based not on the algorithmic properties of the particular cryptographic algorithm but on the particular implementation and the physical characteristic of the underlying device that the algorithm is running on (as discussed in Chapter 3). Therefore, this only means that the particular implementation is more resistant to power analysis. In addition, the point of this research is not to compare the resistance of particular implementation of cryptographic algorithms to power analysis, but to compare the physical characteristics that different architectural configurations provide. Therefore, the following conclusion can be drawn: *the implementation of DES run on the different configurations of the network-based architecture shows better resistance to the DPA attack when compared to the implementation of AES used in the evaluation.*

CONFIGURATION	NUMBER OF POWER SAMPLES
PIPE	750
PH4	> 40000
PH6	> 35000

Figure 6.28: Number of power samples used to perform the DPA attack on DES run on the PIPE and PH configurations of the architecture with the Hamming weight power model and non-variable delays.

The results of performing the DPA attack on DES running on hypercube configurations with the Hamming weight power model and non-variable delays are summarised in Figure 6.28.

6.4 Performance evaluation

This section reports on the dynamics of computation of different architectural configurations in which the network topology and network size are varied. The performance evaluation results are obtained from running AES on different configurations of the architecture.

Figure 6.29 shows the relative execution times of hypercube architectural configurations without and with data-sharing, *i.e.*, PH and PHS configurations. The Figure 6.29 demonstrates that the performance improves as the network size increases, and the different distribution of units between arithmetic, logic and memory units, give slightly different results[3]. The overhead of using data-sharing is small, between 5% and 11%, as shown in Figure 6.30.

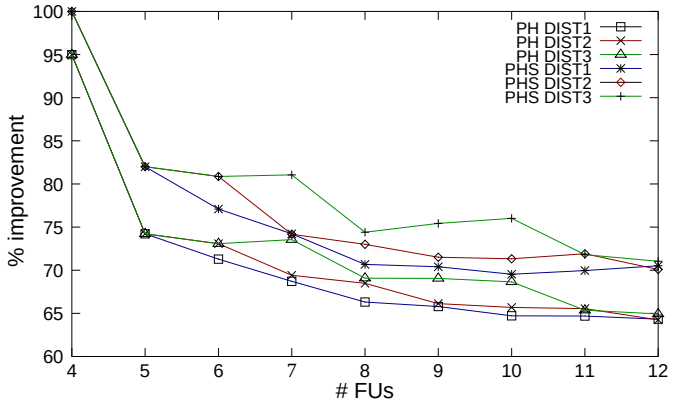


Figure 6.29: Relative execution times of PH and PHS configurations. DIST1, DIST2 and DIST3 represent different distribution of units.

The architecture was designed as an asynchronous network with the aim of decorrelating the power consumption measurements by exploiting the inherent non-determinism of instructions executing in parallel over a network that uses randomised routing. The idea was to introduce different instruction interleavings and in that way decorrelate power consumption measurements. Figure 6.31 shows how the execution times change

[3]Given the small number of multiplications in the code, there is less scope for exploiting ILP with greater number of multiplier units.

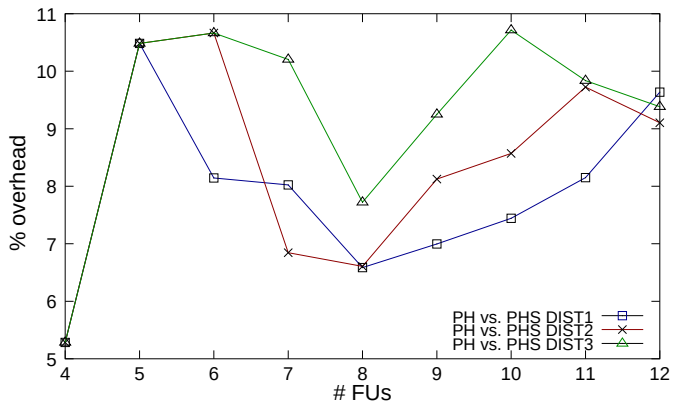
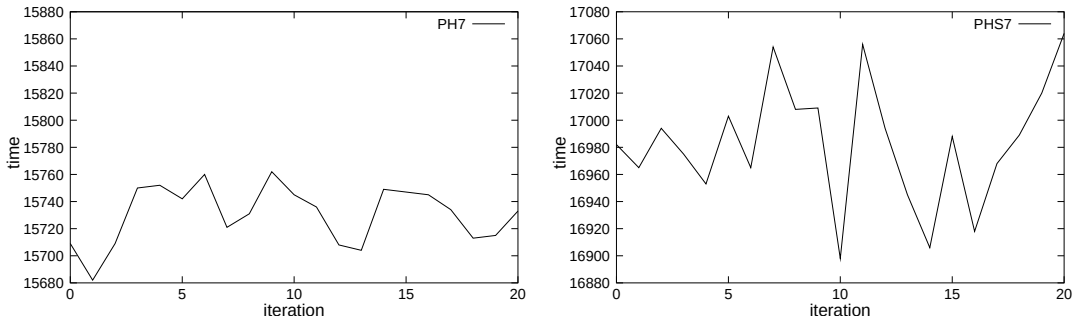


Figure 6.30: Performance overheads of data-sharing for PH and PHS configurations. DIST1, DIST2 and DIST3 represent different distribution of units.

for the successive runs of the same code on the same (hypercube) network configuration. The figure shows that the execution times for 20 runs of AES on the PH7 configuration varied between 15680 and 15760 VTUs, *i.e.*, the observed variation in execution time is 0.5%. On the other hand, the execution times for 20 runs of AES on the PHS7 configuration varied between 16880 and 17060 VTUs, *i.e.*, the observed variation in the execution times is equal to 1.1%. These results confirm that, *due to non-determinism in the execution, different runs of the same code on the same architectural configuration have different execution times*. In PHS configurations data-sharing contributes towards decorrelating power consumption measurements from the data in the following way. Firstly, the increase in the number of forwarded register values through the network introduces the higher variation in execution times. These in turn are beneficial for removing time references in the power traces. The existence of time references in the power traces is crucial for successful application of power-based attacks. Further, by forwarding register values in the form of data-shares the direct correlation between the data and the measured power consumption is removed. These two effects of data-sharing work together towards further decorrelating the power consumption measurements as the results in Section 6.3 have confirmed.

Figures 6.32 and 6.33 show relative execution times and overheads of data-sharing for de Bruijn network configurations without and with data-sharing, *i.e.*, DB and DBS configurations. Figure 6.32 shows that for small network sizes, performance improves with the increase in number of functional units. However, as the Figure 6.32 shows, for the networks size greater than 8 this trend is not followed. The performance overheads of data-sharing in the case of de Bruijn network configurations was between 7% and



(a) PH7 configuration. (b) PHS7 configuration.

Figure 6.31: Variations in execution times of successive runs of the same algorithm for PH7 and PHS7 configurations.

40% (Figure 6.33). Clearly this overhead is higher than the overhead of data-sharing for PHS configurations. This higher overhead is the result of routing in de Bruijn networks. Namely, in a de Bruijn network the path lengths between nodes are on average longer than the path lengths in a hypercube network of the same size. This increases the duration of routing, which appears as if a higher number of packets are being routed, as each packet remains in the network for a longer period of time. This causes forwarding queues to be more populated and, therefore, the routing to be slowed down. On the other hand, despite the observed performance overhead, DB and DBS configurations show better security properties as shown in Section 6.3.

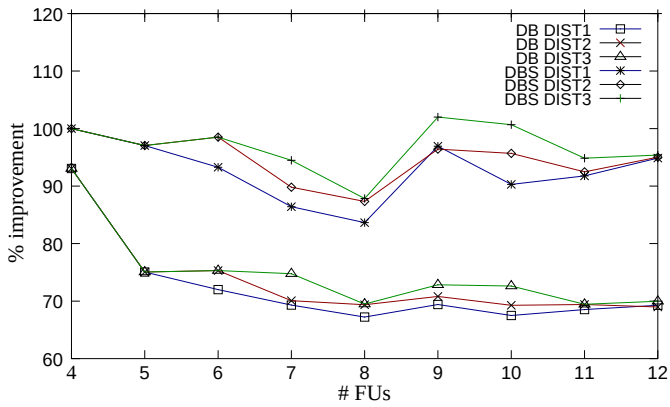


Figure 6.32: Relative execution times of DB and DBS configurations. DIST1, DIST2 and DIST3 represent different distribution of units.

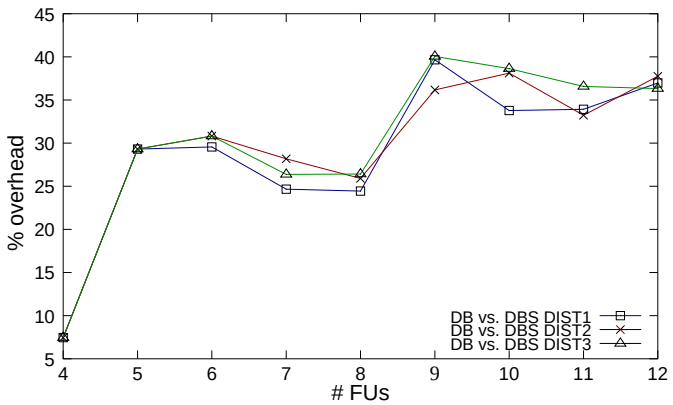
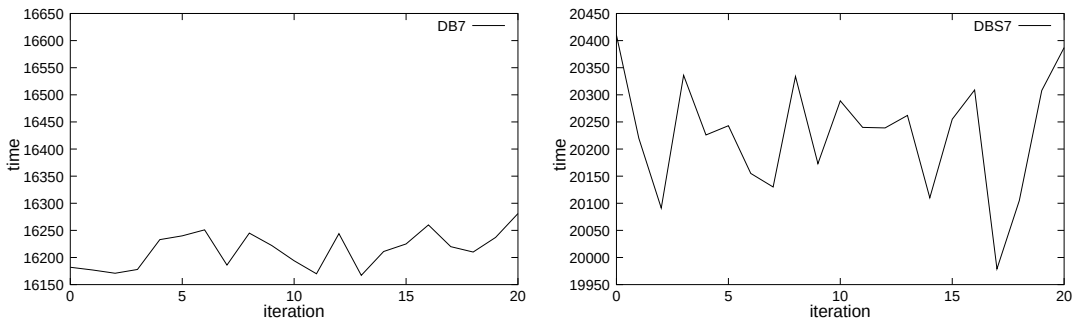


Figure 6.33: Performance overheads of data-sharing for DB and DBS configurations. DIST1, DIST2 and DIST3 represent different distributions of units.

Figure 6.34 shows the variation in execution times for successive runs of the same code on the same de Bruijn network configuration. The execution time for 20 runs on the DB7 configuration varied between 16160 and 16280 VTUs, *i.e.*, the observed variation is the execution times is 0.75%. On the other hand, the execution time for 20 runs on the DBS7 configuration varied between 20000 and 20450 VTUs, *i.e.*, the variation in the execution times is 2.25%. This shows that de Bruijn network configurations provide more variation in execution times when compared to hypercube network configurations, which contributes to its security characteristics, albeit with worse performance characteristics.



(a) DB7 configuration. (b) DBS7 configuration.

Figure 6.34: Variations in execution times of successive runs of the same algorithm for DB7 and DBS7 configurations.

Figure 6.35 compares five different configurations of the architecture as the number of functional units changes. Figure 6.35 clearly shows that data-forwarding does improve the performance (plot ASYNC is above all other plots). When comparing configurations with data-forwarding, the best performance show PH configurations and the worst DBS configurations. For the small number of units (≤ 8) the performance progressively improves for all five architectural configurations. This figure shows that *the network-based organisation, besides providing considerable security improvements, also shows performance improvements*.

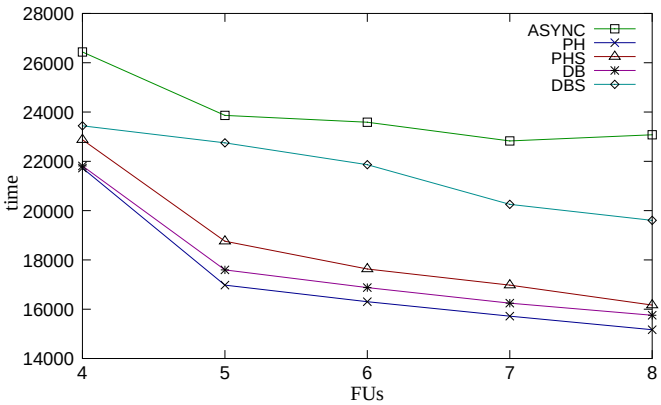


Figure 6.35: Performance comparisons of five architectural configurations.

6.5 Summary

In this chapter the results of the security and performance evaluations for different configurations of the network-based architecture were presented. The results were obtained from running implementations of AES and DES cryptographic algorithms. Since the obtained results depend both on the architectural platform and the algorithm implementation, and since different implementations of the same cryptographic algorithm can show different security features, the security features of AES and DES when run on the architecture were not compared. The goal of this research is not to show whether these algorithms are resistant to the power analysis, but to use them as benchmarks for measuring the introduced non-determinism in the execution of the network-based architecture. The two algorithms were selected for the security evaluation due to their historical and practical importance.

In the current experimentation different configurations of the architecture in which different parameters are varied, were compared. The parametric model was based on

the delay and power figures that were available at the time of the experimentation. However, any further investigation of the security and performance features of the proposed architecture would require more detailed data-dependent delay and power information.

Although the precise figures for a few of the security evaluation experiments were not obtained, the following conclusions can be drawn:

- De Bruijn network configurations provide more scope for non-deterministic execution than hypercube configurations. This is due to the fact that the path lengths in de Bruijn networks are longer than those of a partial hypercube, and provide more variation in the execution times and, therefore, more randomness in the execution.

- The increase in the network size provides more scope for non-deterministic execution, but it also has a positive effect on performance as shown in Figures 6.29 and 6.32.

- Data-forwarding in the form of data-shares improves resistance to power analysis attacks. Not only is the direct contribution to the power of routing sensitive register values removed in this way, but also further variations in execution times are introduced, as shown in Figures 6.31 and 6.34. However, routing more register values introduces performance overheads, as shown in Figures 6.29 and 6.32. This suggests that the trade-off between the desired level of security and performance might have to be made.

- If the power model of an architecture is characterised with the transition count power model, then the architecture is more resistant to power analysis than the one with the Hamming weight power model.

- Non-deterministic execution and forwarding register values through the network of functional units does not produce a performance overhead as shown in Figure 6.35. However, the performance improvements do not necessarily improve with the increase of the network size, *i.e.*, number of functional units.

These results confirm that the network-based architecture is indeed more robust to differential power analysis than the simple pipelined alternatives. By implementing the architecture as a network both higher security and performance improvements are gained, which should make this approach attractive for cryptographic devices.

Chapter 7

Conclusions and Future Work

7.1 Summary

With the introduction of side-channel analysis and the development of different side-channel attack strategies the challenge of improving the security of cryptographic devices has become increasingly important. While most practical implementations seem ad-hoc, even theoretically sound solutions show weaknesses when implemented. As the question of achieving tightly secure cryptographic systems (in theory and practice) remains open, one of the approaches to consider is to increase the cost of performing side-channel attacks.

One of the proposed ideas toward increasing the cost of performing side-channel attacks was to introduce randomness and non-determinism in the execution [80, 78, 36, 91]. Some of the investigations into this idea [91, 92, 66] have demonstrated that the paradigm of randomising the execution can help make the side-channel attacks more difficult to apply. These solutions harness the algorithm's instruction level parallelism (ILP) in order to achieve non-deterministic execution and obscure the compromising information in the power traces.

This thesis presented the *network-based asynchronous architecture* in which execution is *non-deterministic*. The network-based architecture presents a novel architectural idea in which functional units in the processor datapath are connected as an asynchronous network rather than as a linear pipeline. The architecture is a network of functional units that operate concurrently and communicate register values through the network. The non-deterministic execution is achieved by: (1) exploiting fine-grain instruction level parallelism, where concurrent execution of both instruction with and without data-dependencies is utilised; and (2) performing data-forwarding using ran-

domised routing of register values through the network. The investigations performed in this thesis focused on exploring architectural principles which could be beneficial for efficiently and effectively decorrelating power consumption measurements, rather than on providing a detailed and complete low-level architectural description.

In [91] non-deterministic execution is achieved by exploiting instruction level parallelism in the algorithm. Namely, independent instructions are grouped together and issued in random order, thus removing the correlation between features in the DPA profile and the algorithm source code. This approach is further combined with random register renaming [92] and the addition of random operations to the algorithm [66] in order to increase the noise in the device's power profile. In the network-based architecture a different approach is taken in order to achieve a similar goal. The instructions are issued *in-order* to multiple functional units which operate concurrently. The dependencies between instructions, that would normally cause synchronisations or stalls in the issue stage, are exploited in order to enable both concurrent and randomised execution. The instructions execute at their own pace, and may overtake one another, compete for resources, and complete different execution stages *out-of-order* and, thus, render the execution non-deterministic.

In most implementations data-dependencies impose serialisation in the execution of instructions and, thus, restrain the scope for exploiting ILP. For example, read-after-write (RAW) dependencies require the completion of the instruction before its result can be used by a dependent instruction. In hardware, the data-forwarding mechanism was conceived to avoid the penalty of having to write the result to the register bank, and some time later read the result from it. Thus, the ILP can be exploited by issuing instructions without data-dependences in parallel and by forwarding register values between instructions with data-dependencies. This general paradigm was used in the design of the network-based architecture, but exploiting fine-grain parallelism even between instructions with data dependencies is also achieved.

In the micronet architecture [116] the instruction issue rate is limited by data-dependencies and the availability of functional units. In the network-based architecture, stalls in the instruction issue stage due to data-dependencies are not present, and the issue rate is effectively limited only by the availability of functional units. Although the instructions are issued one at a time, the issue unit operates at a faster speed than the rest of the components, which emulates multiple instruction issue.

The instructions are issued in-order rapidly to the functional units of the appropriate type. The instruction issue initiates a series of micro-operations in the respective com-

munication units (that are part of the target functional units) that deal with the given instruction. The processes of identifying data-dependencies (instruction compounding) and initialising data-forwarding operations is also performed in the issue unit. A small table is used to store the information about a number of previously issued instructions and determine dependencies between instructions. If no dependencies are found, then the instruction is able to progress at its own pace without synchronising with the execution stages of other instructions. If data-dependencies are found, then the issue unit initiates data-forwarding operation(s) to the producer unit(s). The mutual exclusion of instructions with data-dependencies is not enforced, but rather, they are issued successively, and allow them to execute concurrently subject to availability of operands. This leads to efficient utilisation of functional units, further distribution of both state and control, and minimisation of high-level synchronisations and stalls that are usually caused by data-dependencies.

Although the performance gains of data-forwarding are evident (as shown in Section 6.4), they are not the primary goal of this design. The focus is on the variation in execution times and power signatures that the network-based design introduces. Namely, the data is routed according to a randomised routing protocol, that depends on the implemented network topology. In general, the routing consists of two parts: (1) the data is routed to a *randomly* chosen unit, and (2) the data follows the deterministic route to its destination (the consumer unit). The randomised routing introduces variations in instruction execution times. *Asynchronous execution and randomised routing introduce different execution times even when the same instruction with the same input data is observed.*

Therefore, the order in which different micro-operations are performed changes from one run to the other, allowing the results of instructions to be available out of program order. In-order write-back to the register bank was not enforced as it would be a performance overhead and could introduce synchronisations. *Out-of-order instruction completion and write-back is beneficial from the security point of view, as writes to the register bank are often targeted in attacks*. An example of this was seen in the case of covariance attack performed on the runs of AES (see Section 6.3.2).

The security evaluation results (presented in Section 6.3) have shown that, when compared to simple pipelined configurations, the resistance to power analysis of algorithms running on the networked configurations is considerably improved. The level of improvement depends on a number of parameters: network topology and the amount of randomness that it provides, the size of the network, the data-forwarding imple-

mentation, and the algorithm implementation. Both AES and DES were simulated and subjected to power-based attacks. Although cracking DES was more difficult when compared to AES, the purpose of this research was not to compare these algorithms, or their (unprotected) implementations, but to measure the non-determinism introduced by the network-based architectural approach. The simulated results in the case of both algorithms have confirmed that the introduced non-determinism considerably increases the cost of performing the power analysis attacks.

The evaluation results also included the performance comparisons (Section 6.4) of different architectural configurations, and have shown that the network-based organisation provides performance improvements. The performance gains, similarly to security improvements, depend on the network configuration, number of functional units in the network, the distribution of functional unit types, and the number of data-shares used in the data-forwarding. However, one should remember that performance is not the primary goal of this design, and that tradeoffs between performance and security might be necessary, depending on the desired level of security/performance.

7.2 Future work

Future research could proceed in a number of different directions. A further and more detailed evaluation of both performance and security features can be performed. However, these investigations would require more detailed delay and power consumption figures, which were not available for the current experimentation. Also, comparisons of the presented approach could be made with existing techniques for increasing the cost of side-channel analysis attacks. For example, this approach could be compared with the secret-sharing scheme (see Section 3.5.7.5).

A further exploration of possible improvements to the architectural implementation could also be beneficial. Besides exploring ways for achieving non-deterministic execution, this thesis has presented a novel approach for distributing the control and execution in asynchronous architectures. The non-deterministic execution is achieved through exploiting the fine-grain ILP of instructions both with and without data dependencies. However, the current implementation does not implement any of the conventional techniques for increasing ILP and parallelism in general. One of the issues that can be investigated is the relation of the design of the network-based architecture to performance improving dynamic techniques in microprocessors, such as speculative execution, instruction and data caches, speculative branches and multithreading.

When addressing these, an important issue would also be to investigate the expected overheads in terms of hardware implementation.

Correct operation in the presence of data-dependencies often requires serialisation in the execution of instructions, which in turn limits the scope for exploiting parallelism. This is particularly true in case of RAW dependences, which require the completion of the instruction before its result can be used by a dependent instruction. One of the hardware approaches to this problem is data-forwarding. In software, the effect of true dependencies could be minimised through the use of instruction scheduling. In the case of the network-based architecture instruction scheduling could be also used to remove both resource contention and stalls in the operand fetch and branch stage. Namely, if the result of the producer instruction is already available when forwarding requests are initiated, then the instruction execution could be further improved and stalls reduced. Stalls are also undesirable from the security point of view as they can be easily observed in the power traces. Electromagnetic attacks are even more threatening as they can exploit locality information.

In the network-based asynchronous architecture, if the size of the dependency table in the issue unit is equal to the number of functional units, then all RAW dependences between non-control transfer instructions are covered by data-forwarding. This is because RAW hazards occur only between instructions that are currently in execution. Having the operand fetch stage distributed to the communication units, the number of instructions that are in flight at any point is bounded from above by twice the number of functional units. This can be characterised as the theoretical throughput of the architecture. The actual (average) throughput depends on the particular code. This is another area where scheduling could be used: to improve the throughput of architectures and potentially bring it closer to its theoretical best, which could have a positive effect on non-deterministic execution and performance.

The number of instructions that can be affected by RAW hazards is equal to the number of instructions that are currently in the operand fetch and lock stage and is bounded by the number of functional units. Therefore, appropriately adjusting the size of the dependency table (see Section 5.5.2) eliminates most of the RAW hazards at no additional cost. RAW hazards between control transfer instructions and the rest of the instruction set are not removed this way. Exploring techniques for removing these dependencies could also be one of the future tasks.

7.3 Conclusions

This thesis presented the design and evaluation of the network-based asynchronous architecture. It focused on exploring a particular hardware paradigm for exploiting non-deterministic execution in order to benefit cryptographic devices. These investigations confirmed that the level of introduced non-determinism increases resistance to power analysis, when compared to simple pipelined configurations, as it makes power analysis attacks considerably more difficult to apply. In addition, the non-deterministic execution does not present a performance overhead, although a tradeoff between the desired level of the non-determinism (*i.e.* security) and performance might need to made for a particular implementation.

This thesis introduced a general approach for achieving non-deterministic execution in processors. This approach can be complementary to and combined with existing software and/or hardware countermeasures in order to achieve further resistance to side-channel analysis. For instance, it could be combined with the randomised scheduling techniques presented in [91, 92, 66] to further randomise the execution; or it could be implemented using balanced asynchronous logic as a possible way to hide the sensitive information in the power and electromagnetic traces, which is still found to be present in balanced systems [49]. Electromagnetic attacks are of particular concern as they have proven to be able to extract information even when the power channel provides no information. The non-deterministic and distributed manner of operation of the network-based architecture could potentially be beneficial against electromagnetic attacks and their ability to exploit the locality information.

Although asynchronous circuits have been shown to be a suitable platform for exploiting non-deterministic execution, some characteristics of asynchrony can be exploited in side-channel analysis. For example, data-dependent latencies could be a source of information. One of the aims of this approach is to introduce fine-grain execution timing variations through randomised routing, which can help reduce this type of information leakage.

In summary, the investigations performed in this thesis have shown that the introduced level of non-determinism considerably complicates the application of power analysis attacks and suggests that the introduced architectural concepts could be beneficial to cryptographic devices.

Appendix A

Published Paper

A.1 Design and Evaluation of the Network-based Asynchronous Architecture for Cryptographic Devices

TITLE:	Design and Evaluation of the Network-based Asynchronous Architecture for Cryptographic Devices
AUTHORS:	Ljiljana Dilparić and D. K. Arvind
PRESENTED AT:	15th IEEE International Conference on Application-Specific Systems, Architectures, and Processors (ASAP 2004)
PLACE:	Galveston, Texas, USA
DATE:	27-29 September 2004
PUBLISHER:	IEEE Computer Society Press

Appendix B

Instruction Set

Arithmetic instructions

INSTRUCTION	DESCRIPTION
`li $rd, I`	rd.word = I;
`la $rd, I ( $rs )`	rd.word = I + rs.word;
`lai $rd, I1 ( I2 )`	rd.word = I1 + I2;
`move $d, $rs`	rd. word = rs.word;
`add $rd, $rs, $rt`	rd.word = rs.word + rt.word;
`addi $rd, $rs, I`	rd.word = rs.word + I;
`sub $rd, $rs, $rt`	rd.word = rs.word - rt.word;
`subi $rd, $rs, I`	rd.word = rs.word - I;
`seq $rd, $rs, $rt`	rd.word = rs.word == rt.word;
`seqi $rd, $rs, I`	rd.word = rs.word == I;
`sge $rd, $rs, $rt`	rd.word = rs.word >= rt.word;
`sgei $rd, $rs, I`	rd.word = rs.word >= I;
`sgt $rd, $rs, $rt`	rd.word = rs.word > rt.word;
`sgti $rd, $rs, I`	rd.word = rs.word > I;
`sle $rd, $rs, $rt`	rd.word = rs.word <= rt.word;
`slei $rd, $rs, I`	rd.word = rs.word <= I;
`slt $rd, $rs, $rt`	rd.word = rs.word < rt.word;
`slti $rd, $rs, I`	rd.word = rs.word < I;
`sne $rd, $rs, $rt`	rd.word = rs.word != rt.word;
`snei $rd, $rs, I`	rd.word = rs.word != I;

Logic instructions

INSTRUCTION	DESCRIPTION
`and $rd, $rs, $rt`	rd.word = rs.word & rt.word;
`andi $rd, $rs, I`	rd.word = rs.word & I;
`or $rd, $rs, $rt`	rd.word = rs.word | rt.word;
`xor $rd, $rs, $rt`	rd.word = rs.word ˆ rt.word;
`sll $rd, $rs, I`	rd.word = rs.word << I;
`srl $rd, $rs, I`	rd.word = rs.word >> I;

Multiplication instructions

INSTRUCTION	DESCRIPTION
`mul $rd, $rs, $rt`	rd.word = rs.word * rt.word;
`muli $rd, $rs, I`	rd.word = rs.word * I;
`div $rd, $rs, $rt`	rd.word = rs.word / rt.word;
`divi $rd, $rs, I`	rd.word = rs.word / I;
`rem $rd, $rs, $rt`	rd.word = rs.word % rt.word;
`remi $d, $s0, I`	rd.word = rs.word % I;

Memory instructions

INSTRUCTION	DESCRIPTION
`lw $rd, I ( $rs )`	rd.word = readData (I + rs.word, 4);
`lh $rd, I ( $rs )`	rd.word = readData(I + rs.word, 2);
`lb $rd, I($rs)`	rd.word = readData(I + rs.word, 1);
`sw $rs, I($rt)`	writeData(I + rt.word, rs.word, 4);
`sh $rs, I($rt)`	writeData(I + rt.word, rs.word, 2);
`sb $rs, I($rt)`	writeData(I + rt.word, rs.word, 1);
`lwi $d, I1, I2`	rd.word = readData(I1 + I2, 4);
`swi $s0, I1, I2`	eriteData(I1 + I2, rs.word, 4);

Control transfer instructions

INSTRUCTION	DESCRIPTION
`bra L`	pcReg.word = L;
`j L`	pcReg.word = L;
`jr $rs`	pcReg.word = rs.word;
`jal L`	linkReg.word = pcReg.word + 4;
	pcReg.word = L;
`jalr $rs`	linkReg.word = pcReg.word + 4;
	pcReg.word = rs.word;
`beq $rs, $rt, L`	rs.word == rt.word ? pcReg.word = L : {};
`bne $rs, $rt, L`	rs.word != rt.word ? pcReg.word = L : {};
`ble $s0, $s1, L`	rs.word <= rt.word ? pcReg.word = L : {};
`blt $rs, $rt, L`	rs.word < rt.word ? pcReg.word = L : {};
`bge $rs, $rt, L`	rs.word >= rt.word ? pcReg.word = L : {};
`bgt $rs, $rt, L`	rs.word > rt.word ? pcReg.word = L : {};
`beqz $rs, L`	rs.word == 0 ? pcReg.word = L : {};
`bnez $rs, L`	rs.word != 0 ? pcReg.word = L : {};
`blez $rs, L`	rs.word <= 0 ? pcReg.word = L : {};
`bltz $rs, L`	rs.word < 0 ? pcReg.word = L : {};
`bgez $rs, L`	rs.word >= 0 ? pcReg.word = L : {};
`bgtz $rs, L`	rs.word > 0 ? pcReg.word = L : {};

Appendix C

Rijndael and DES Tables

N_r	$N_b = 4$	$N_b = 6$	$N_b = 8$
$N_k = 4$	10	12	14
$N_k = 6$	12	12	14
$N_k = 8$	14	14	14

Figure C.1: Rijndael: Number of rounds as a function of the block and key length.

N_b	C_1	C_2	C_3
4	1	2	3
6	1	2	3
8	1	3	4

Figure C.2: Rijndael: Shift offsets for different block lengths

32	1	2	3	4	5
4	5	6	7	8	9
8	9	10	11	12	13
12	13	14	15	16	17
16	17	18	19	20	21
20	21	22	23	24	25
24	25	26	27	28	29
28	29	30	31	32	1

Figure C.3: DES: E bit-selection table.

57	49	41	33	25	17	9
1	58	50	42	34	26	18
10	2	59	51	43	35	27
19	11	3	60	52	44	36
63	55	47	39	31	23	15
7	62	54	46	38	30	22
14	6	61	53	45	37	29
21	13	5	28	20	12	4

Figure C.4: DES: Key schedule permuted choice 1.

14	17	11	24	1	5
3	28	15	6	21	10
23	19	12	4	26	8
16	7	27	20	13	2
41	52	31	37	47	55
30	40	51	45	33	48
44	49	39	56	34	53
46	42	50	36	29	32

Figure C.5: DES: Key schedule permuted choice 2.

Iteration Number	Number of Left Shifts
1	1
2	1
3	2
4	2
5	2
6	2
7	2
8	2
9	1
10	2
11	2
12	2
13	2
14	2
15	2
16	1

Figure C.6: DES: Key schedule left shift order.

Bibliography

[1] AES Lounge - AES Security. Available from http://www.iaik.tu-graz.ac.at/research/krypto/AES/#rijndael.

[2] First AES Candidate Conference (AES1), August 20-22, 1998, Ventura, California. http://csrc.nist.gov/CryptoToolkit/aes/round1/conf1/aes1conf.htm.

[3] Public Key Cryptography Standards (PKCS) No. 1, RSA Cryptography Standard. ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-1/pkcs-1v2-1.pdf.

[4] Second AES Candidate Conference (AES2), March 22-23, 1999, Rome, Italy. http://csrc.nist.gov/CryptoToolkit/aes/round1/conf2/aes2conf.htm.

[5] Serpent block cipher. http://www.cl.cam.ac.uk/ rja14/serpent.html.

[6] The Advanced Encryption Standard: A Status Report - August 1999. http://csrc.nist.gov/publications/nistbul/html-archive/aug-99.html.

[7] The block cipher Square. http://www.esat.kuleuven.ac.be/∼rijmen/square/.

[8] Digital Signature Standard (DSS). Federal Information Processing Standards Publication 186, National Institute of Standards and Technology, 1994. Available from http://www.itl.nist.gov/fipspubs/fip186.htm.

[9] Advanced Encryption Standard (AES). Federal Information Processing Standards Publication 197, National Institute of Standards and Technology, 2001. Available from http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

[10] Digital Encryption Standard (DES). Federal Information Processing Standards Publication 46-2, National Institute of Standards and Technology, December 1993. Available from http://www.itl.nist.gov/fipspubs/fip46-2.htm.

[11] Security Requirements for Cryptographic Modules. Federal Information Processing Standards Publication 140-2, National Institute of Standards and Technology, January 1994. Available from http://csrc.nist.gov/publications/fips/fips140-2/fips1402.pdf.

[12] D. Agrawal, B. Archambeault, J. Rao, and P. Rohatgi. The EM Side-Channel(s): Attacks and Assessment Methodologies. Technical report, I.B.M. T.J. Watson Research Center, Yorktown Heights, NY. Available from http://www.research.ibm.com/intsec.

[13] D. Agrawal, B. Archambeault, J. Rao, and P. Rohatgi. The EM Side-Channel(s). In B. S. Kaliski, Ç. K. Koç, and C. Paar, editors, *Revised Papers from the 4th International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2002)*, volume 2523-LNCS, pages 29–45. Springer-Verlag, 2002.

[14] M.-L. Akkar, R. Bevan, P. Dischamp, and D. Moyart. Power Analysis, What Is Now Possible... In T. Okamoto, editor, *The Proceedings of the 6th International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT 2000)*, volume 1976-LNCS, pages 489–502. Springer-Verlag, 2000.

[15] M.-L. Akkar and L. Goubin. A Generic Protection against High-Order Differential Power Analysis. In T. Johansson, editor, *Revised Papers from the 10th International Workshop on Fast Software Encryption (FSE 2003)*, volume 2887-LNCS, pages 192–205. Springer-Verlag, 2003.

[16] R. Aleliunas. Randomised Parallel Communication. In *The Proceedings of the ACM-SIGOPS Symposium on Principles of Distributed Computing*, pages 60–72, 1982.

[17] R. Anderson and M. Kuhn. Tamper Resistance - a Cautionary Note. In *The Proceedings of the Second USENIX Workshop on Electronic Commerce*, pages 1–11. USENIX Association, 1996.

[18] R. Anderson and M. Kuhn. Low Cost Attacks on Tamper Resistant Devices. In B. Christianson, B. Crispo, M. Lomas, and M. Roe, editors, *The Proceedings of the 5th International Workshop on Security Protocols (IWSP'97)*, volume 1361-LNCS, pages 125–136. Springer-Verlag, 1997.

[19] D. K. Arvind and V. E. F. Rebello. On the Performance Evaluation of Asynchronous Processor Architectures. In P. Dowd and E. Gelenbe, editors, *The Proceedings of the International Workshop on Modelling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'95)*, pages 100–105. IEEE Computer Society Press, 1995.

[20] D. K. Arvind and V. E. F. Rebello. Static Scheduling of Instructions on Micronet-based Asynchronous Processors. In *The Proceedings of the 2nd International Symposium on Advanced Research on Asynchronous Circuits and Systems (ASYNC'96)*, pages 80–91. IEEE Computer Society Press, 1996.

[21] D. K. Arvind and V. E. F. Rebello. Instruction-Level Parallelism in Asynchronous Processor Architectures. In M. Moonen and F. Catthoor, editors, *The Proceedings of the 3rd International Workshop on Algorithms and Parallel VLSI Architecture*, pages 203–215. Elsevier Science Publisher, August 1994.

[22] W. J. Bainbridge. *Asynchronous System-on-Chip Interconnect*. PhD thesis, University of Manchester, 2000.

[23] M. Bellare and P. Rogaway. The Exact Security of Digital Signatures - How to Sign with RSA and Rabin. In U. M. Maurer, editor, *The Proceedings of International Conference on the Theory and Application of Cryptographic Techniques (EUROCRYPT'96)*, volume 1070-LNCS, pages 399–416. Springer-Verlag, 1996.

[24] J.-C. Bermond, R. W. Daves, and F. Ö. Ergincan. De Bruijn and Kautz Bus Networks. *Networks: An International Journal*, 30:205–218, 1997.

[25] J.-C. Bermond and F. Ö. Ergincan. Bus Interconnection Networks. *DAMATH: Discrete Applied Mathematics and Combinatorial Operations Research and Computer Science*, 68, 1996.

[26] I. Biehl, B. Meyer, and V. Müller. Differential Fault Attacks on Elliptic Curve Cryptosystems. In M. Bellare, editor, *The Proceedings of the 20th Annual International Conference on Advances in Cryptology (CRYPTO 2000)*, volume 1880-LNCS, pages 131–146. Springer-Verlag, 2000.

[27] E. Biham and A. Shamir. Differential Cryptanalysis of DES-like Cryptosystems. *Journal of Cryptology: the journal of the International Association for Cryptologic Research*, 4(1):3–72, 1991.

[28] E. Biham and A. Shamir. Differential Fault Analysis of Secret Key Cryptosystems. In B. S. Kaliski Jr., editor, *The Proceedings of the 17th Annual International Conference on Advances in Cryptology (CRYPTO'97)*, volume 1294-LNCS, pages 513–525. Springer-Verlag, 1997.

[29] E. Biham and A. Shamir. Power Analysis of the Key Scheduling of the AES Candidates. In *Second Advanced Encryption Standard (AES) Candidate Conference*, http://csrc.nist.gov/encryption/aes/round1/conf2/aes2conf.htm, 1999.

[30] D. Boneh, R. DeMillo, and R. Lipton. On the Importance of Checking Cryptographic Protocols for Faults. In W. Fumy, editor, *The Proceedings of the International Conference on the Theory and Application of Cryptographic Techniques (EUROCRYPT'97)*, volume 1233-LNCS, pages 37–51. Springer-Verlag, 1997.

[31] É. Brier and M. Joye. Weierstrass Elliptic Curves and Side-Channel Attacks. In D. Naccache and P. Paillier, editors, *The Proceedings of the 5th International Workshop on Practice and Theory in Public Key Cryptosystems (PKC 2002)*, volume 2274-LNCS, pages 335–345. Springer-Verlag, 2002.

[32] D. Brumley and D. Boneh. Remote Timing Attacks are Practical. In *The Proceedings of the 12th USENIX Security Symposium*. USENIX Association, 2003.

[33] J. A. Brzozowski and C.-J. H. Seger. *Asynchronous Circuits*. Springer-Verlag New York Inc., 1995.

[34] S. M. Burns and A. J. Martin. Syntax-directed Translation of Concurrent Programs into Self-timed Circuits. In J. Allen and F. Thomson Leighton, editors, *The Proceedings of the Fifth MIT Conference on Advanced Research in VLSI, March 1988*, pages 35–50. MIT Press, 1988.

[35] S. Chari, C. Jutla, J. Rao, and P. Rohatgi. A Cautionary Note Regarding Evaluation of AES Candidates on Smart-Cards. In *Second Advanced Encryption Standard (AES) Candidate Conference*, Rome, Italy, 1999.

[36] S. Chari, C. Jutla, J. Rao, and P. Rohatgi. Towards Sound Approaches to Counteract Power-Analysis Attacks. In M. J. Wiener, editor, *The Proceedings of the 19th Annual International Conference on Advances in Cryptology (CRYPTO'99)*, volume 1666-LNCS, pages 398–412. Springer-Verlag, 1999.

[37] S. Chari, C. Jutla, J. Rao, and P. Rohatgi. Template Attacks. In B. S. Kaliski, Ç. K. Koç, and C. Paar, editors, *Revised Papers from the 4th International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2002)*, volume 2523-LNCS, pages 13–28. Springer-Verlag, 2002.

[38] M. Ciet, J.-J. Quisquater, and F. Sica. Preventing Differential Analysis in GLV Elliptic Curve Scalar Multiplication. In B. S. Kaliski, Ç. K. Koç, and C. Paar, editors, *Revised Papers from the 4th International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2002)*, volume 2523-LNCS, pages 540–550. Springer-Verlag, 2002.

[39] C. Clavier, J.-S. Coron, and N. Dabbous. Differential Power Analysis in Presence of Hardware Countermeasures. In Ç. K. Koç and C. Paar, editors, *The Proceedings of the Second International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2000)*, volume 1965-LNCS, pages 252–ff. Springer-Verlag, 2000.

[40] J.-S. Coron. Resistance Against Differential Power Analysis for Elliptic Curve Cryptosystems. In Ç. K. Koç and C. Paar, editors, *The Proceedings of the First International Workshop on Cryptographic Hardware and Embedded Systems (CHES'99)*, volume 1717-LNCS, pages 292–302. Springer-Verlag, 1999.

[41] J.-S. Coron and L. Goubin. On Boolean and Arithmetic Masking against Differential Power Analysis. In Ç. K. Koç and C. Paar, editors, *The Proceedings of the Second International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2000)*, volume 1965-LNCS, pages 231–ff. Springer-Verlag, 2000.

[42] J.-S. Coron, P. Kocher, and D. Naccache. Statistics and Secret Leakage. In Y. Frankel, editor, *The Proceedings of the 4th International Conference on Financial Cryptography (FC 2000)*, volume 1962-LNCS, pages 157–173. Springer-Verlag, 2001.

[43] N. Courtois and J. Pieprzyk. Cryptanalysis of Block Ciphers with Overdefined Systems of Equations. In Y. Zheng, editor, *The Proceedings of the 8th International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT 2002)*, volume 2501-LNCS, pages 267–287. Springer-Verlag, 2002.

[44] J. Daemen, M. Peeters, and G. Van Assche. Bitslice Ciphers and Power Analysis Attacks. In B. Schneier, editor, *The Proceedings of the 7th International Workshop on Fast Software Encryption (FSE 2000)*, volume 1978-LNCS, pages 134–ff. Springer-Verlag, 2000.

[45] J. Daemen and V. Rijmen. AES Proposal: Rijndael. Available from http://csrc.nist.gov/encryption/aes/rijndael/Rijndael.pdf.

[46] J. Daemen and V. Rijmen. Resistance Against Implementation Attacks: A Comparative Study of the AES Proposals. In *Second Advanced Encryption Standard (AES) Candidate Conference*, http://csrc.nist.gov/encryption/aes/round1/conf2/aes2conf.htm, 1999.

[47] J.-F. Dhem, F. Koeune, P. A. Leroux, P. Mestré, J.-J. Quisquater, and J. L. Willems. A Practical Implementation of the Timing Attack. In *The Proceedings of the Third International Conference on Smart Card Research and Advanced Applications (CARDIS 1998)*, volume 1820-LNCS. Springer-Verlag, 1998.

[48] W. Diffie and M. E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, November 1976.

[49] J. J. A. Fournier, S. Moore, H. Li, R. Mullins, and G. Taylor. Security Evaluation of Asynchronous Circuits. In C. Walter, Ç. K. Koç, and C. Par, editors, *The Proceedings of the 5th International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2003)*, volume 2779-LNCS, pages 137–151. Springer-Verlag, 2003.

[50] S. B. Furber and P. Day. Four-Phase Micropipeline Latch Control Circuits. *IEEE Transactions on VLSI Systems*, 4(2):247–253, June 1996.

[51] S. B. Furber, P. Day, J. D. Garside, N. C. Paver, and J. V. Woods. A Micropipelined ARM. In T. Yanagawa and P. A. Ivey, editors, *The Proceedings of the IFIP International Conference on Very Large Scale Integration (VLSI'93)*, pages 5.4.1–5.4.10. North Holland, September 1993.

[52] S. B. Furber, J. D. Garside, P. A. Riocreux, S. Temple, P. Day, J. Liu, and N. C. Paver. AMULET2e: An Asynchronous Embedded Controller. In *The Proceedings of the IEEE*, volume 87, pages 243–256, February 1999.

[53] K. Gandolfi, C. Mourtel, and F. Olivier. Electromagnetic Analysis: Concrete Results. In Ç. K. Koç, D. Naccache, and C. Paar, editors, *The Proceedings of the Third International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2001)*, volume 2162-LNCS, pages 251–261. Springer-Verlag, 2001.

[54] C. H. Gebotys and R. J. Gebotys. Secure Elliptic Curve Implementations: An Analysis of Resistance to Power-Attacks in a DSP Processor. In B. S. Kaliski, Ç. K. Koç, and C. Paar, editors, *Revised Papers from the 4th International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2002)*, volume 2523-LNCS, pages 114–128. Springer-Verlag, 2002.

[55] D. M. Gordon. A survey of fast exponentiation methods. *Journal of Algorithms*, 27(1):129–146, 1998.

[56] L. Goubin. A Refined Power-Analysis Attack on Elliptic Curve Cryptosystems. In Y. Desmedt, editor, *The Proceedings of the 6th International Workshop on Theory and Practice in Public Key Cryptography (PKC 2003)*, volume 2567-LNCS, pages 199–211. Springer-Verlag, 2003.

[57] L. Goubin and J. Patarin. DES and Differential Power Analysis - The ”Duplication” Method. In Ç. K. Koç and C. Paar, editors, *The Proceedings of the First International Workshop on Cryptographic Hardware and Embedded Systems (CHES’99)*, volume 1717-LNCS, pages 158–172. Springer-Verlag, 1999.

[58] M. D. Grammatikakis, D. F. Hsu, M. Kraetzl, and J. F. Sibeyn. Packet Routing in Fixed-Connection Networks: A Survey. *Journal of Parallel and Distributed Computing*, 54(2):77–132, 1998.

[59] W. J. Grundman, D. Dobberpuhl, R. L. Allmon, and N. L. Rethman. Designing High Performance CMOS Microprocessors Using Full Custom Techniques. In *The Proceedings of the 34th Design Automation Conference (DAC’97)*, pages 722–727. ACM/IEEE, June 1997.

[60] J. C. Ha and S. J. Moon. Randomized Signed-Scalar Multiplication of ECC to Resist Power Attacks. In B. S. Kaliski, Ç. K. Koç, and C. Paar, editors, *Revised Papers from the 4th International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2002)*, volume 2523-LNCS, pages 551–563. Springer-Verlag, 2002.

[61] M. A. Hasan. Power Analysis Attacks and Algorithmic Approaches for their Countermeasures for Koblitz Curve Cryptosystems. In Ç. K. Koç and C. Paar, editors, *The Proceedings of the Second International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2000)*, volume 1965-LNCS, pages 93–ff. Springer-Verlag, 2000.

[62] S. Hauck. Asynchronous Design Methodologies: An Overview. In *The Proceedings of the IEEE*, volume 83, pages 69–93, January 1995.

[63] M. Hellman. A Cryptanalytic Time-Memory Tradeoff. *IEEE Transactions on Information Theory*, 26(4):401–406, 1980.

[64] H. Hulgaard, S. M. Burns, and G. Borriello. Testing Asynchronous Circuits: A Survey. Technical Report TR-FR-35, Department of Computer Science and Engineering, University of Washington, Seattle, WA, USA, March 1994.

[65] H. Hulgard. *Timing Analysis and Verification of Timed Asynchronous Circuits*. PhD thesis, University of Washington, Seattle, WA, USA, 1995.

[66] J. Irwin, D. Page, and N. P. Smart. Instruction Stream Mutation for Non-Deterministic Processors. In M. Shulte, S. Bhattacharyya, N. Burgess, and R. Schreiber, editors, *The Proceedings of the 13th IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP 2002)*, pages 286–295. IEEE Computer Society Press, 2002.

[67] T. Izu and T. Takagi. A Fast Parallel Elliptic Curve Multiplication Resistant against Side Channel Attacks. In D. Naccache and P. Paillier, editors, *The Proceedings of the 5th International Workshop on Practice and Theory in Public Key Cryptosystems (PKC 2002)*, volume 2274-LNCS, pages 280–296. Springer-Verlag, 2002.

[68] M. Joye and J.-J. Quisquater. Attacks on Systems Using Chinese Remaindering. Technical Report CG-1996/9, Université Catholique de Louvain, Crypto Group, Belgium, 1996. Available from http://www.dice.ucl.ac.be/crypto/techreports.html.

[69] M. Joye and J.-J. Quisquater. Hessian Elliptic Curves and Side-Channel Analysis. In Ç. K. Koç, D. Naccache, and C. Paar, editors, *The Proceedings of the*

Third International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2001), volume 2162-LNCS, pages 402–ff. Springer-Verlag, 2001.

[70] M. Joye and C. Tymen. Protecting against Differential Analysis for Elliptic Curve Cryptography - An Algebraic Approach. In Ç. K. Koç, D. Naccache, and C. Paar, editors, *The Proceedings of the Third International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2001)*, volume 2162-LNCS, pages 377–ff. Springer-Verlag, 2001.

[71] B. Jun and P. Kocher. Analysis of the Intel Random Number Generator. White Paper, Cryptography Research Inc., April 1999.

[72] J. Kelsey, B. Schneier, D. Wagner, and C. Hall. Side Channel Cryptanalysis of Product Ciphers. *Journal of Computer Security*, 8(2-3):141–158, 2000.

[73] A. Khoche and E. Brunvand. Testing Self-timed Circuits Using Partial Scan. In *The Proceedings of the 2nd Working Conference on Asynchronous Design Methodologies*, pages 160–169. IEEE Computer Society Press, 1995.

[74] N. Koblitz. Elliptic Curve Cryptosystems. *Mathematics and Computation*, 48:203–209, 1987.

[75] N. Koblitz. CM-Curves with Good Cryptographic Properties. In J. Feigenbaum, editor, *The Proceedings of the 11th Annual International Conference on Advances in Cryptology (CRYPTO'91)*, volume 0576-LNCS, pages 279–287. Springer-Verlag, 1991.

[76] P. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS and Other Systems. In N. Koblitz, editor, *The Proceedings of the 16th Annual International Conference on Advances in Cryptology (CRYPTO'96)*, volume 1109-LNCS, pages 104–113. Springer-Verlag, 1996.

[77] P. Kocher, J. Jaffe, and B. Jun. Introduction to Differential Power Analysis and Related Attacks. Technical report, Cryptography Research Inc., 1998. Available from http://www.cryptography.com/dpa/technical.

[78] P. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. In M. J. Wiener, editor, *The Proceedings of the 19th Annual International Conference on Advances in Cryptology (CRYPTO'99)*, volume 1666-LNCS, pages 388–397. Springer-Verlag, 1999.

[79] F. Koeune and J.-J. Quisquater. A timing attack against Rijndael. Technical Report CG-1999/1, Université Catholique de Louvain, 1999. Available from htttp://www.dice.ucl.ac.be/crypto.

[80] O. Kömmerling and M. Kuhn. Design Principles of Tamper-Resistant Smartcard Processors. In *The Proceedings of the 1st USENIX Workshop on Smartcard Technology*, pages 9–20. USENIX Association, 1999.

[81] G. Kotsis. Interconnection Topologies for Parallel Processing Systems. In *The Proceedings of the 3rd Workshop on Parallel Systems and Algorithms (PASA'93)*, June, 1993.

[82] A. Lenstra. *Memo on RSA Signature Generation in the Presence of Faults*, September, 1996. Available from http://cm.bell-labs.com/who/akl/rsa.doc.

[83] P.-Y. Liardet and N. P. Smart. Preventing SPA/DPA in ECC Systems Using a Jacobi Form. In Ç. K. Koç, D. Naccache, and C. Paar, editors, *The Proceedings of the Third International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2001)*, volume 2162-LNCS, pages 391–ff. Springer-Verlag, 2001.

[84] R. Lidl and H. Niederreiter. *Introduction to Finite Fields and Their Applications*. Cambridge University Press, 1986.

[85] Z. Liu and T.-Y. Sung. Routing and Transmitting Problems in de Bruijn Networks. *IEEE Transactions on Computers*, 45(9):1056–1062, 1996.

[86] J. Markoff. Secure Digital Transactions Just Got a Little Less Secure. *New York Times*, page A1, 11 December 1995.

[87] A. J. Martin. Programming VLSI: From Communicating Processes to Delay-insensitive Circuits. Technical Report Caltech-CR-TR-89-1, Department of Computer Science, California Institute of Technology, 1989.

[88] A. J. Martin. The Limitations to Delay-Insensitivity in Asynchronous Circuits. In *The Proceedings of the 6th MIT Conference on Advanced Research in VLSI*, pages 263–278. MIT Press, 1990.

[89] A. J. Martin, A. Lines, R. Manohar, R. Southworth M. Nyström, P. Penzes, and U. Cummings. The Design of an Asynchronous MIPS R3000 Microprocessor. In *The Proceedings of the 17th Conference on Advanced Research in VLSI*, pages 164–181. IEEE Computer Society Press, 1997.

[90] M. Matsui. Linear Cryptanalysis Method for DES Cipher. In T. Helleseth, editor, *The Proceedings of the Workshop on the Theory and Application of Cryptographic Techniques (EUROCRYPT'93)*, volume 765-LNCS, pages 386–397. Springer-Verlag, 1994.

[91] D. May, H. L. Muller, and N. P. Smart. Non-Deterministic Processor. In V. Varadharajan and Y. Mu, editors, *The Proceedings of the 6th Australasian Conference on Information Security and Privacy (ACISP 2001)*, volume 2119-LNCS, pages 115–129. Springer-Verlag, 2001.

[92] D. May, H. L. Muller, and N. P. Smart. Random Register Renaming to Foil DPA. In Ç. K. Koç, D. Naccache, and C. Paar, editors, *The Proceedings of the Third International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2001)*, volume 2162-LNCS, pages 28–38. Springer-Verlag, 2001.

[93] C. Mead and L. Conway. *Introduction to VLSI Systems*, chapter 9. Addison-Wesley, 1980.

[94] T. S. Messerges. Power Analysis Attack Countermeasures and Their Weaknesses. Communications, Electromagnetics, Propagation, and Signal Processing Workshop (CEPS 2000).

[95] T. S. Messerges. Using Second-Order Power Analysis to Attack DPA Resistant Software. In Ç. K. Koç and C. Paar, editors, *The Proceedings of the Second International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2000)*, volume 1965-LNCS, pages 238–ff. Springer-Verlag, 2000.

[96] T. S. Messerges. Securing the AES Finalists Against Power Analysis Attacks. In B. Schneier, editor, *The Proceedings of the 7th International Workshop on Fast Software Encryption (FSE 2000)*, volume 1978-LNCS, pages 150–ff. Springer-Verlag, 2001.

[97] T. S. Messerges, E. Dabbish, and R. Sloan. Investigations of Power Analysis Attacks on Smartcards. In *USENIX Workshop on Smartcard Technology*, pages 151–161. USENIX Association, 1999.

[98] T. S. Messerges, E. Dabbish, and R. Sloan. Power Analysis Attacks of Modular Exponentiation in Smartcards. In Ç. K. Koç and C. Paar, editors, *The Proceedings of the First International Workshop on Cryptographic Hardware and Embedded Systems (CHES'99)*, volume 1717-LNCS, pages 144–157. Springer-Verlag, 1999.

[99] V. S. Miller. Use of Elliptic Curves in Cryptography. In H. C. Williams, editor, *The Proceedings of Annual International Conference on Advances in Cryptology (CRYPTO'85)*, volume 218-LNCS, pages 417–426. Springer-Verlag, 1986.

[100] B. Möller. Securing Elliptic Curve Point Multiplication against Side-Channel Attacks. In G. I. Davida and Y. Frenkel, editors, *The Proceedings of the 4th International Conference on Information Security (ISC 2001)*, volume 2200-LNCS, pages 324–334. Springer-Verlag, 2001.

[101] P. L. Montgomery. Speeding the Pollard and Elliptic Curve Methods for Factorization. *Mathematics and Computation*, 48:243–264, 1987.

[102] S. Moore, R. Anderson, P. Cunningham, R. Mullins, and G. Taylor. Improving Smart Card Security using Self-timed Circuits. In *The Proceedings of the Eight International Symposium on Asynchronous Circuits and Systems (ASYNC'02)*, pages 211–218. IEEE Computer Society Press, 2002.

[103] S. Moore, R. Anderson, R. Mullins, G. Taylor, and J. Fournier. Balanced Self-Checking Asynchronous Logic for Smart Card Applications. *Microprocessors and Microsystems*, 2003.

[104] J. A. Muir. Techniques of Side Channel Cryptanalysis. Master's thesis, University of Waterloo, 2001.

[105] R. D. Mullins. *Dynamic Instruction Scheduling and Data Forwarding in Asynchronous Superscalar Processors*. PhD thesis, University of Edinburgh, 2001.

[106] K. Okeya, H. Kurumatani, and K. Sakurai. Elliptic Curves with the Montgomery Form and their Cryptographic Applications. In H. Imai and Y. Zheng, editors, *The Proceedings of the Third International Workshop on Practice and Theory in Public Key Cryptography (PKC 2000)*, volume 1751-LNCS, pages 238–257. Springer-Verlag, 2000.

[107] K. Okeya and K. Sakurai. Power Analysis Breaks Elliptic Curve Cryptosystems Even Secure against the Timing Attack. In B. Roy and E. Okamoto, editors, *The Proceedings of the First International Conference in Cryptology in India (INDOCRYPT 2000)*, volume 1977-LNCS, pages 178–ff. Springer-Verlag, 2000.

[108] K. Okeya and T. Takagi. A More Flexible Countermeasure against Side Channel Attacks Using Window Method. In C. Walter, Ç. K. Koç, and C. Par, editors, *The Proceedings of the 5th International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2003)*, volume 2779-LNCS, pages 397–410. Springer-Verlag, 2003.

[109] K. Okeya and T. Takagi. The With-w NAF Method Provides Small Memory and Fast Elliptic Scalar Multiplications Secure against Side Channel Attacks. In M. Joye, editor, *Topics in Cryptology - CT-RSA 2003: The Cryptographers' Track at the RSA Conference 2003, San Francisco, CA, USA, April 13-17, 2003*, volume 2612-LNCS, pages 328–342. Springer-Verlag, 2003.

[110] E. Oswald and M. Aigner. Randomized Addition-Subtraction Chains as a Countermeasure against Power Attacks. In Ç. K. Koç, D. Naccache, and C. Paar, editors, *The Proceedings of the Third International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2001)*, volume 2162-LNCS, pages 39–ff. Springer-Verlag, 2001.

[111] N. C. Paver, P. Day, S. B. Furber, J. D. Garside, and J. V. Woods. Register Locking in an Asynchronous Microprocessor. In *The Proceedings of the IEEE International Conference on Computer Design: VLSI in Computer & Processors (ICCD'92)*, pages 351–355. IEEE Computer Society Press, 1992.

[112] A. M. G. Peeters. *Single-Rail Handshake Circuits*. PhD thesis, Technische Universiteit Eindhoven, Eindhoven, The Netherlands, 1996.

[113] J.-J. Quisquater and D. Samyde. ElectroMagnetic Analysis (EMA): Measures and Countermeasures for Smart Cards. In I. Attali and T. P. Jensen, editors, *The Proceedings of the International Conference on Research in Smart Cards (E-smart 2001)*, volume 2140-LNCS, pages 200–210. Springer-Verlag, 2001.

[114] B. Rahardjo. Asynchronous Tools. Available from http://www.cs.man.ac.uk/async/tools/index.html.

[115] J. Rao and P. Rohatgi. EMpowering Side-Channel Attacks. Technical Report 2001/037, I.B.M. T.J. Watson Research Center, Yorktown Heights, NY, 2001.

[116] V. E. F. Rebello. *On the Distribution of Control in Asynchronous Processor Architectures*. PhD thesis, University of Edinburgh, 1996.

[117] R. L. Rivest, A. Shamir, and L. M. Adelman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. Technical Report MIT/LCS/TM-82, Massachusetts Institute of Technology, 1977.

[118] S. S. Salazar. *Instruction Scheduling in Micronet-based Asynchronous ILP Processor*. PhD thesis, University of Edinburgh, 2002.

[119] W. Schindler. A Timing Attack against RSA with the Chinese Reminder Theorem. In Ç. K. Koç, D. Naccache, and C. Paar, editors, *The Proceedings of the Second International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2000)*, volume 1965-LNCS, pages 109–124. Springer-Verlag, 2000.

[120] J. Schneiders. *Area Virtual Time*. PhD thesis, University of Edinburgh, 2004.

[121] B. Schneier. Security Pitfalls in Cryptography. White Paper, Counterpane Internet Security, Inc., 1998.

[122] C. L. Seitz. System Timing. In C. A. Mead and L. A. Conway, editors, *Introduction to VLSI Systems*, chapter 7. Addison-Wesley, 1980.

[123] A. Shamir. How to Check Modular Exponentiation. Rump session talk at Eurocrypt 1997.

[124] A. Shamir. Protecting Smart Cards from Passive Power Analysis with Detached Power Supplies. In Ç. K. Koç, D. Naccache, and C. Paar, editors, *The Proceedings of the Second International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2000)*, volume 1965-LNCS, pages 71–ff. Springer-Verlag, 2000.

[125] C. E. Shannon. Communication Theory of Secrecy Systems. *Bell System Technical Journal*, 28(4):656–715, 1949.

[126] S. Skorobogatov and R. Anderson. Optical Fault Induction Attacks. In B. S. Kaliski, Ç. K. Koç, and C. Paar, editors, *Revised Papers from the 4th International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2002)*, volume 2523-LNCS, pages 2–12. Springer-Verlag, 2002.

[127] D. Sokolov, J. Murphy, A. Bystrov, and A. Yakovlev. Improving the Security of Dual-Rail Circuits. In M. Joye and J.-J. Quisquater, editors, *The Proceedings of the 6th International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2004)*, volume 3156-LNCS, pages 282–297. Springer-Verlag, 2004.

[128] Stanford SUIF Compiler Group. The SUIF Compiler System. http://suif.stanford.edu.

[129] I. E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, June 1989.

[130] L. G. Valiant and G. J. Brebner. Universal Schemes for Parallel Computation. In *The Proceedings of the 13th ACM Symposium on the Theory of Computing*, pages 263–277. ACM Press, 1981.

[131] C. H. K. van Berkel. *Handshake Circuits : An Asynchronous Architecture for VLSI Programming*. Cambridge University Press, 1993.

[132] C. H. K. van Berkel, M. B. Josephs, and S. M. Nowick. Scanning the Technology : Applications of Asynchronous Circuits. *Proceedings of the IEEE. Special Issue in Asynchronous Circuits*, 87(2):223–233, February 1999.

[133] C. D. Walter. Breaking the Liardet-Smart Randomized Exponentiation Algorithm. In *The Proceedings of the 5th Smart Card Research and Advanced Application Conference (CARDIS 2002)*, pages 59–68. USENIX Association, Berkley, CA, USA, 2002.

[134] P. Wayner. Code Breaker Cracks Smart Cards' Digital Safe. *New York Times*, page D1, 22 June 1998.

[135] N. H. E. Weste and K. Eshraghian. *Principles of CMOS VLSI Design - A Systems Perspective*, pages 231–238. Addison-Wesley, 2nd edition, 1993.

[136] Z. C. Yu, S. B. Furber, and L. A. Plana. An Investigation into the Security of Self-timed Circuits. In *The Proceedings of the Ninth International Symposium on Asynchronous Circuits and Systems (ASYNC'03)*, pages 201–210. IEEE Computer Society Press, 2003.