Secure Cryptographic Algorithm Implementation on Embedded Platforms

Michael Tunstall

Technical Report
RHUL–MA–2007–5
29 May 2007

Department of Mathematics
Royal Holloway, University of London
Egham, Surrey TW20 0EX, England
http://www.rhul.ac.uk/mathematics/techreports

SECURE CRYPTOGRAPHIC ALGORITHM IMPLEMENTATION ON EMBEDDED PLATFORMS

Michael Tunstall

Royal Holloway, University of London

Thesis submitted to

The University of London

for the degree of

Doctor of Philosophy

2006

Declaration

These doctoral studies were conducted under the supervision of Prof. Chris Mitchell. The equipment for all of the experiments detailed in this thesis was provided *in situ* by Gemalto (formerly Gemplus Card International).

The work presented in this thesis is the result of original research carried out by myself, in collaboration with others, whilst enrolled in the Department of Mathematics as a candidate for the degree of Doctor of Philosophy. This work has not been submitted for any other degree or award in any other university or educational establishment.

Michael Tunstall

October 2006

Abstract

Sensitive systems that are based on smart cards use well-studied and well-developed cryptosystems. Generally these cryptosystems have been subject to rigorous mathematical analysis in an effort to uncover cryptographic weaknesses in the system. The cryptosystems used in smart cards are, therefore, not usually vulnerable to these types of attacks. Since smart cards are small objects that can be easily placed in an environment where physical vulnerabilities can be exploited, adversaries have turned to different avenues of attack.

This thesis describes the current *state-of-the-art* in side channel and fault analysis against smart cards, and the countermeasures necessary to provide a secure implementation. Both attack techniques need to be taken into consideration when implementing cryptographic algorithms in smart cards.

In the domain of side-channel analysis a new application of using cache accesses to attack an implementation of AES by observing the power consumption is described, including an unpublished extension.

Several new fault attacks are proposed based on finding collisions between a correct and a fault-induced execution of a secure secret algorithm. Other new fault attacks include reducing the number of rounds of an algorithm to make a differential cryptanalysis trivial, and fixing portions of the random value used in DSA to allow key recovery.

Countermeasures are proposed for all the attacks described. The use of random delays, a simple countermeasure, is improved to render it more secure and loss costly to implement. Several new countermeasures are proposed to counteract the particular fault attacks proposed in this thesis. A new method of calculating a modular exponentiation that is secure against side channel analysis is described, based on ideas which have been proposed previously or are known within the smart card industry. A novel method for protecting RSA against fault attacks is also proposed based on securing the underlying Montgomery multiplication.

The majority of the fault attacks detailed have been implemented against actual chips to demonstrate the feasibility of these attacks. Details of these experiments are given in appendices. The experiments conducted to optimise the performance of random delays are also described in an appendix.

List of Publications

The following papers covering work discussed in this thesis have been published.

[1] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan, *The sorcerer's apprentice guide to fault attacks*, Workshop on Fault Diagnosis an Tolerance in Cryptography, in association with DSN 2004 — The International Conference on Dependable Systems and Networks, June 2004.

[2] D. Naccache, P. Q. Nguyen, M. Tunstall, and C. Whelan, *Experimenting with faults, lattices and the DSA*, Public Key Cryptography — PKC 2005 (S. Vaudenay, ed.), Lecture Notes in Computer Science, vol. 3386, Springer-Verlag, 2005, pp. 16–28.

[3] D. Naccache, M. Tunstall, and C. Whelan, *Computational improvements to differential side channel attacks*, NATO Security through Science Series D: Information and Communication Security, vol. 2, IOS Press, 2006, pp. 26–35.

[4] H. Choukri and M. Tunstall, *Round reduction using faults*, Workshop on Fault Diagnosis and Tolerance in Cryptography 2005 — FDTC 05 (L. Breveglieri and I. Koren, eds.), 2005, pp. 13–24.

[5] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan. *The sorcerer's apprentice guide to fault attacks*, Proceedings of the IEEE **94** (2006), no. 2, 370–382.

[6] H. Choukri and M. Tunstall. *Fault attacks.* In H. Bidgoli, editor, Handbook of Information Security, Wiley, 2006, pp. 230–240.

[7] M. Tunstall, S. Petit, S. Porte. *Smart card security.* In H. Bidgoli, editor, Handbook of Information Security, Wiley, 2006, pp. 326–341.

[8] C. Clavier and M. Tunstall. *Explaining differential fault analysis on DES.* Workshop on Coding and Cryptography, Cork, May 2006.

[9] J. Fournier and M. Tunstall, *Cache based power analysis attacks on AES*, 11^{th} Australasian Conference on Information Security and Privacy — ACISP 2006 (L. M. Batten and R. Safavi-Naini, eds.), Lecture Notes in Computer Science, vol. 4058, Springer-Verlag, 2006, pp. 17–28.

[10] F. Amiel, C. Clavier and M. Tunstall. *Fault analysis of DPA-resistant algorithms*. Fault Diagnosis and Tolerance in Cryptography 2006 — FDTC 06 (L. Breveglieri, I. Koren, D. Naccache and J.-P. Seifert, eds.), Lecture Notes in Computer Science, vol. 4236, Springer-Verlag, 2006, pp. 223–236.

[11] O. Benoit and M. Tunstall. *Efficient use of random delays*, Cryptology ePrint Archive, Report 2006/272, 2006, `http://eprint.iacr.org/`.

[12] K. Markantonakis, K. Mayes, M. Tunstall, D. Sauveron and F. Piper. *Smart card security*. Computational Intelligence in Information Assurance and Security (N. Nedjah, A. Abraham, and L. M. Mourelle, eds.), Studies in Computational Intelligence, vol. 57, Springer-Verlag, 2007, pp. 201-233.

[13] M. Tunstall and O. Benoit. *Efficient use of random delays in embedded software*. Workshop on Information Security Theory and Practices 2007 — Smart Cards, Mobile and Ubiquitous Computing Systems — WISTP 2007 (D. Sauveron, K. Markantonakis, A. Bilas and J.-J. Quisquater, eds.), Lecture Notes in Computer Science, vol. 4462, Springer-Verlag, 2007, pp. 27–38.

Acknowledgements

A debt of gratitude is owed to David Naccache without whom this thesis would never have been written. When I joined Gemplus in 1998, David tried to motivate me to further my education as far as possible. He even tried to tried to appeal to my Anglo-Saxon roots by pointing out that salaries generally increase in a logarithmic fashion, and that extra qualifications shift this curve upwards. David also helped with the progression of this thesis by involving me in several research projects. These were instrumental in starting the creative process that led to other publications.

I would like to thank Nathalie Feyt and Chris Mitchell for the help and encouragement they gave me during my research, particularly when I was unsure whether certain ideas were worth pursuing.

I would like to thank Konstantinos Markantonakis and Keith Mayes from Royal Holloway's Smart Card Centre for their support. In particular, for the position of Research Assistant afforded to me in the last year of my research.

I would like to thank everyone with whom I worked at Gemplus, who taught me an immense amount about cryptography, side channel analysis and fault attacks. Without their help, support and friendship this thesis could not have been written. In particular, I would like to thank: Frederic Amiel, Philippe Anguita, Bruno Baronnet, Olivier Benoit, Julien Brouchier, Eric Brier, Benoit Chevalier-Mames, Hamid Choukri, Christophe Clavier, Nora Dabbous, Jean-Francois Dhem, Jacques Fournier, Laurent Gauteron, Pascal Guterman, Helena Handschuh, Marc Joye, Laurent Bonnet, Nathalie Feyt, Philippe Loubet-Moundi, Pascal Moitrel, Christophe Mourtel, Johan Pascal, David Naccache, Khanh Nguyen, Francis Olivier, Pascal Pailler, Sebastien Petit, Florence Ques, Stephanie Porte, Philippe Proust, Alexei Tchoulkine, Lionel Victor, and Karine Villegas.

I would also like to thank my co-authors, with whom it has been educational and inspirational to work with. They are Frederic Amiel, Hagai Bar-el, Olivier Benoit, Hamid Choukri, Jacques Fournier, Konstantinos Markantonakis, Keith Mayes, Fred

Piper, David Naccache, Khanh Nguyen, Phong Q. Nguyen, Sebastien Petit, Stephanie Porte, Damien Sauveron, and Claire Whelan.

I would like to thank Benoit Chevalier-Mames, Chris Mitchell, Will Sirett and John Tunstall for their help in proof reading this thesis and the other articles that I have published. In particular, I would like to thank Chris Mitchell for the sterling effort he put into helping me prepare the final version of this thesis.

I would like to thank Kenny Paterson and Mike Scott for the time they committed to examining this thesis. Their comments and suggestions have made a significant contribution to increasing the quality of this thesis.

Lastly, I would like to thank my family for their support, and for putting up with me while I was writing up this thesis.

Contents

Declaration **2**

Abstract **3**

List of Publications **5**

Acknowledgements **7**

Contents **9**

List of Tables **15**

List of Figures **16**

List of Algorithms **18**

Notation and Abbreviations **20**

1 Introduction **23**

1.1 Summary of Contributions 25

1.2 Structure 26

2 Cryptographic Algorithms **29**

2.1 Data Encryption Standard 29

2.2 Advanced Encryption Standard 31

2.3 RSA 33

2.4 The Digital Signature Standard 36

3 Smart Card Architecture **38**

3.1 What is a Smart Card? 38

3.2 Microprocessors 41

3.2.1 Cryptographic Coprocessors 42

3.2.2 Random Number Generators 42
3.2.3 Cache 44
3.2.4 Anomaly Sensors 45
3.2.5 Chip Features 46
3.3 Summary and Conclusions 47

4 Side Channel Analysis Techniques 48
4.1 Timing Analysis 49
4.2 Simple Power Analysis 49
4.3 Statistical Power Analysis 53
4.3.1 Modelling the Power Consumption 53
4.3.2 Differential Power Analysis 55
4.3.3 Correlation Power Analysis 57
4.3.4 Comparing the Different Methods 58
4.4 Electromagnetic Analysis 60
4.5 Summary and Conclusions 61

5 Fault Injection Techniques 62
5.1 Injection Techniques 63
5.2 The Types of Fault 65
5.2.1 Provisional Faults (Taxonomy) 66
5.2.2 Destructive Faults (Taxonomy) 66
5.3 Fault Models 67
5.4 Combining Faults with Side Channel Attacks 70
5.5 Summary and Conclusions 70

6 Generic Attacks 71
6.1 Simple Power Analysis of Key Manipulation 71
6.2 Cache-Based Power Analysis 73
6.2.1 The First ByteSub Function 74
6.2.2 Finding the Rest of the Key 76
6.2.3 Other Algorithms 82

6.3 Fault Attacks on Key Transfer or NVM 82
6.4 Round Reduction Using Faults 83
6.5 Summary and Conclusions 87

7 Generic Countermeasures **88**
7.1 The Basic Principles 88
7.2 Desynchronisation Techniques 89
7.2.1 Desynchronisation in Software 89
7.2.2 Desynchronisation in Hardware 95
7.3 Execution Randomisation 96
7.4 Data Whitening 96
7.5 Integrity Checks 98
7.6 Summary and Conclusions 99

8 Block Cipher Implementations **100**
8.1 Power Analysis of Block Ciphers 101
8.2 Differential Fault Analysis 101
8.2.1 The Fifteenth Round 102
8.2.2 Faults in Earlier Rounds 106
8.2.3 Triple DES 110
8.2.4 Other Algorithms 112
8.3 Collision Fault Analysis 113
8.3.1 Attacking DES 114
8.3.2 DPA-Resistant Algorithms 115
8.3.3 Other Algorithms 119
8.4 Changing S-Box Values 120
8.4.1 Modifying Known S-Box Values of DES 120
8.4.2 Modifying Unknown S-Box Values of DES 122
8.4.3 Other Algorithms 125
8.5 Countermeasures 125
8.6 Summary and Conclusions 127

9 Implementations of RSA **129**

9.1 Simple Power Analysis of RSA 130

9.2 Statistical Power Analysis of RSA 131

9.3 Fault Attack on RSA Signature with CRT 132

9.4 Fault Attack on RSA Signature Without CRT 132

9.5 Faults in the Modulus 136

9.5.1 Forging a Signature 136

9.5.2 Recovering the Key 138

9.6 Exponentiation Algorithms 139

9.6.1 Side Channel Resistant Algorithms 141

9.6.2 Fault Resistant Algorithms 146

9.6.3 Attacking Side Channel Resistant Algorithms 147

9.7 Secure Exponentiation Using CRT 147

9.7.1 Side Channel Resistant Algorithms 148

9.7.2 Fault Resistant Algorithms 148

9.8 Secure Montgomery Multiplication 150

9.8.1 Side Channel Resistant Algorithms 151

9.8.2 Fault Resistant Algorithms 153

9.8.3 Modified Montgomery Algorithm with Redundancy 154

9.8.4 RSA Computation using CRT with Redundancy 156

9.9 Comparing the Fault Protection Methods 159

9.10 Countermeasures 161

9.11 Summary and Conclusions 162

10 Implementations of DSA **164**

10.1 Power Attacks on DSA 164

10.2 Faults in the Private Key 165

10.3 Faults in the Nonce 166

10.4 Countermeasures 170

10.5 Summary and Conclusions 171

11 Conclusion **172**

11.1 Summary of Contributions 172

A Optimising the Differential Calculation **175**

A.1 The Global Sum 176

A.2 Formation of Waveform Equivalence Classes 177

A.3 Combining Waveforms 179

A.4 Chosen Plaintext Differential Power Analysis 182

A.5 Optimisation Results 183

A.6 Remarks 184

B Finding Glitch Parameters **186**

C Attacking a Key Transfer **189**

D Round Reduction of AES **192**

D.1 The Possible Effects 193

D.2 Experimental Results 194

E Efficient Use of Random Delays **197**

E.1 Design Criteria 197

E.2 Deriving a Suitable Distribution 198

E.3 Reverse Engineering the Distribution 203

E.3.1 Potential Attack Scenarios 203

E.3.2 Hypothesis Testing 205

E.4 Remarks 206

F Differential Fault Analysis on DES **207**

G CFA of the First XOR of DPA-Resistant AES **210**

H CFA of the Key Masking of DPA-Resistant AES **214**

I S-Box Modification in DPA-Resistant DES **217**

J Security Analysis of Montgomery Multiplication with Redundancy Check **219**

K Implementation of Montgomery Multiplication with Redundancy Check **222**

L Glitching k During DSA Computations **227**

Bibliography **232**

List of Tables

6.1 The Biham–Shamir Attack 82

7.1 Parameter characteristics for tables of 2^9 entries 94

8.1 Frequency of input and output pairs for the first S-box. 104
8.2 The expected number of hypotheses per S-box for one faulty ciphertext block. 105
8.3 The expected number of hypotheses per S-box for two faulty ciphertext blocks. 105
8.4 The expected number of hypotheses per S-box when $L_{15} \neq L'_{15}$. . . . 108
8.5 The fraction of possible differentials per S-box. 112
8.6 The hypotheses generated by attacking a compressed S-box. 121

9.1 The number of faulty signatures required to identify one byte. 136

10.1 Experimental attack percentage success rates: n is the number of bytes reset in k, and d is the number of signatures. 169

A.1 Optimisation results 183

C.1 The Biham–Shamir attack 189

E.1 Parameter Characteristics for Tables of 2^9 Entries 202
E.2 Parameter Characteristics for Tables of 2^{10} Entries 203

F.1 The breakdown of hypotheses per ciphertext block. 209

G.1 Information derived from 72 collisions. 210

H.1 Information derived from 60 collisions. 214

K.1 Speed comparisons. 224

List of Figures

2.1 The DES round function for round n. 30

3.1 Classic shape and dimensions of a smart card. 39
3.2 A front and rear view of a micromodule before it is set into a plastic card. 39
3.3 Communication initialisation between a smart card and a reader. . . 41
3.4 A chip surface with readily identifiable features. 46
3.5 A chip with a shield present and removed. Removing a shield using hydrofluoric acid renders the chip non-functional but allows the layout to be determined. 47

4.1 Data acquisition tools. 49
4.2 The power consumption of an unprotected modular exponentiation implemented using the square and multiply algorithm. 50
4.3 The power consumption of a DES implementation showing the rounds of the algorithm. 51
4.4 The power consumption of a DES implementation showing the round functions. 52
4.5 Overlaid acquisitions of the power consumption produced by the same instruction but with varying data. 54
4.6 A DPA trace. 56
4.7 DPA traces on bit 1 of S-box$_1$ for various guesses (correct guess is 24). 58
4.8 CPA traces on 4 bits of S-box$_1$ for various guesses (correct guess is 24). 59
4.9 Electromagnetic probing of a chip. 61

5.1 Supply voltage glitch fault injection equipment. 63
5.2 White light fault injection equipment. 64
5.3 Laser fault injection equipment. 65

5.4 Single event latch-up — parasitic transistors T1 and T2. 67
5.5 The effect of a glitch on data being manipulated by a chip, as visible in the power consumption. 68
5.6 The effect of a glitch on an instruction, where the instruction during the glitch is affected. 69

6.1 Two power consumption acquisitions showing the difference produce by a conditional bit set. 72
6.2 The effect of a successful round-reducing fault on the I/O and power consumption. 85

7.1 Random delays visible in the power consumption over time. 91
7.2 The cumulative random delay. 92
7.3 The cumulative random delay using a modified distribution. 95
7.4 Unstable internal clock generation, as visible in the power consumption. 96

9.1 The electromagnetic emanations of an RSA implementation. 130
9.2 An example calculation path for the square and multiply algorithm. 131

B.1 A modified Clio reader. 187

D.1 A power consumption waveform that shows the rounds of AES visible as a repeating pattern. 195

E.1 An example of a modified probability function. 199
E.2 The mean and the standard deviation against k for approximately equal values of a and b. 200
E.3 The mean and the standard deviation against b for fixed values of a and k. 201
E.4 The cumulative random delay using a modified distribution. 204

L.1 Experimental set up 229
L.2 I/O and power consumption (beginning of the trace of the command used to generate signatures). 230

List of Algorithms

2.1 Advanced Encryption Standard 31
2.2 The **MixColumn** Function 32
2.3 The **xtime** function 33
2.4 The AES Key Schedule 33
4.1 The Square and Multiply Algorithm 51
6.1 Insecure bitwise permutation function 72
6.2 Secure bitwise permutation function 73
6.3 One Round of the Advanced Encryption Standard 84
7.1 Randomised Data Transfer 97
7.2 Randomising S-Box Values 97
8.1 The First XOR 115
8.2 Masking the Key 119
9.1 Predicting d mod p by counting 138
9.2 The Square and Multiply Algorithm 140
9.3 The (M,M^3) Algorithm 141
9.4 The 2-ary Algorithm 142
9.5 Side Channel Atomic Square and Multiply Algorithm 143
9.6 The Square and Multiply Always Algorithm 144
9.7 Randomised Exponentiation Algorithm 144
9.8 Secure 2-ary Algorithm 146
9.9 Randomised Exponentiation Algorithm using CRT 148
9.10 Verifying the computation of the RSA algorithm using CRT 149
9.11 Infective computation of RSA 150
9.12 Montgomery Multiplication 151
9.13 Time Constant Montgomery Multiplication 152
9.14 Montgomery Multiplication with Redundancy 155
9.15 Modular Multiplication with Redundancy 157

9.16 Montgomery Exponentiation with Redundancy 158
9.17 RSA Computation with Redundancy 159
K.1 Calculate $\mathbf{A} \leftarrow \mathbf{A} + u_i\mathbf{N} \mod (b-1)$ 223

Notation and Abbreviations

$|$ — Divides, i.e. $A|B$ means that A divides B.

$\wedge$ — the logical AND function.

$\neg$ — the logical NOT function.

$\vee$ — the logical OR function.

λ — the Carmichael function, where $\lambda(n)$ is defined for n as the smallest integer m such that $a^m \equiv 1 \pmod{n}$ for every integer a that is coprime to n.

ϕ — Euler's Totient function, where $\phi(x)$ equals the number of positive integers less than x which are coprime to x.

3G — This refers to the third generation mobile standards developed by 3GPP. These standards seek to address the security problems encountered with the GSM standards.

ALU — Arithmetic Logic Unit, the part of a chip's CPU that conducts arithmetic and logical calculations.

BNC — A BNC connector is commonly used to terminate coaxial cable, and is often used on oscilloscopes.

Comp128 — the original authentication algorithm for 2G GSM networks, although this algorithm was never standardised nor officially recommended for use. The details of this algorithm are not officially in the public domain.

Coprime — two numbers are said to be coprime if they share no common divisors.

CDMA — Code Division Multiple Access, a form of multiplexing. This is used in a U.S. standard defined by QUALCOMM for mobile communications.

CRT — Chinese Remainder Theorem.

EEPROM — Electrically Erasable and Programmable Read Only Memory, used to store code and personalisation data such as cryptographic keys.

gcd — Greatest Common Divisor.

GSM — Global System for Mobile Communications, the most common system used for mobile phones.

Hamming distance — the number of bits that differ between two binary strings (of equal length), or, equivalently, the Hamming weight of the XOR of the two strings.

GF — Galois field.

Hamming weight — the number of bits set to 1 in a given bit string.

LFSR — Linear Feedback Shift Register, often used to generate pseudo-random numbers.

NVM — Non-Volatile Memory, such as EEPROM, flash memory or ROM.

OS — Operating System.

Pr — Probability function.

ROM — Read-Only Memory, used in smart cards to contain the operating system that is unchanging from one user to another.

S-box — Used to refer to a substitution table in a cryptographic algorithm; these are a common feature of block ciphers.

Smooth — Describes a number whose factors consist solely of relatively small numbers.

Subscripts — The base of a value is determined by a trailing subscript, which is applied to the whole word preceding the subscript. For example, FE_{16} is 254 expressed in base 16, and $d = (d_x, d_{x-1}, \ldots, d_0)_2$ gives a binary expression for d.

XOR — Exclusive-OR, or a bitwise addition with no carry propagation. This is also denoted by the $\oplus$ symbol.

$\mathbb{Z}$ — The ring of integers under addition and multiplication.

$\mathbb{Z}_p$ — The ring of integers modulo p.

Chapter 1

Introduction

In recent years, smart cards have become one of the most common secure computing devices. Their uses include such diverse applications as: providing a secure wireless communication framework (GSM, CDMA or 3G), banking and identification. The secure microprocessors used in smart cards are also starting to appear in other devices, such as Dongles used to protect software from unauthorised reproduction [103]. The security of cryptographic algorithms implemented on embedded devices is discussed in terms of smart cards in this thesis. However, the attacks and countermeasures are potentially relevant to all devices that make use of cryptographic algorithms on secure microprocessors.

Direct threats to smart card security include invasive attacks (that alter the chip inside the card), analysis of a side channel (data leaked from inside the card), induced faults, as well as more traditional forms of attack. A variety of countermeasures can be implemented in a smart card to prevent successful attacks that are particular to secure microprocessors.

This thesis describes the current *state-of-the-art* in side channel and fault analysis against smart cards. A brief description of these topics is given below.

Side channel Attacks: Side channel analysis is a class of attacks that seek to deduce information in an indirect manner. This is achieved by extracting secret information held inside a device, such as a smart card, via the monitoring of information that leaks naturally during its operation.

The first publicly described attack of this type was a timing attack against

RSA and certain other public key cryptographic primitives [60]. Publication of this initial result was followed by attacks using power consumption [61] and electromagnetic emanations [39] as a side channel to extract information.

Fault Attacks: In recent years fault analysis has come to the foreground as a possible means of attacking devices such as smart cards. This idea was originally proposed in 1997 [23], in the form of a fault attack against RSA.

The vast majority of papers written on this subject involve very specific faults (usually a bit-flip in a specific variable) that are extremely difficult to produce in practice. However, several attacks have been published that have enough flexibility in the type of fault required that they can be realised with current fault injection methods.

Fault attacks have been implemented against chips that could be used in smart cards [8]. This has meant that countermeasures against this type of attack need to be implemented in embedded devices to protect against this class of attack, which have predominantly been implemented against secure devices such as smart cards. However, it has been shown that some of these attacks can also be applied to standard computers [44].

Both attack techniques need to be taken into consideration when implementing cryptographic algorithms in smart cards. The attacks detailed in this thesis are discussed in terms of smart cards, but it is reasonable to assume that other forms of secure portable devices will also be vulnerable to these attacks. Attacks on embedded platforms, and the necessary countermeasures, that require multi-threaded systems [14] or large memories [44] will not be considered.

The attacks and countermeasures are discussed in terms of software implementations. It is assumed at all times that the underlying hardware is vulnerable to side channel and fault attacks when an algorithm is implemented in a naïve manner. Hardware countermeasures are mentioned where necessary, but it is not always possible to include such functionality in a chip, and little detail is therefore given regarding hardware aspects.

1.1 Summary of Contributions

The following summarises the contributions discussed in this thesis.

- An implementation of a key transfer attack specified in Section 6.1 is described in Appendix C.
- A novel way of applying cache-based side channel analysis to an implementation of AES on smart cards is given in Section 6.2.
- An implementation of a fault attack that reduces the number of rounds of a block cipher is discussed in Section 6.4, and the implementation details are given in Appendices B and D. These were implemented on a Microchip Silvercard.
- Random delays can be used to increase the difficulty of implementing an attack. This is described in Section 7.2, along with a method for optimising the performance and efficiency of this countermeasure. Appendix E shows how this optimisation was derived, and how this can be applied to random delays of any length.
- Four novel attacks based on injecting faults into DPA countermeasures are described in Sections 8.3.2, 8.4.1 and 8.4.2. The implementations of these four attacks are discussed in Appendices G, H, and I.
- In Section 9.4 the currently accepted specification of a fault attack on the private exponent of RSA (when calculated without using the Chinese Remainder Theorem) was shown to be false when more than one bit of the private exponent is changed. An alternative fault model is proposed in which the attack is valid.
- A new modular exponentiation algorithm is given in Algorithm 9.8, in Section 9.6.1, although this is based on previously published countermeasures.
- A novel method of securing the Montgomery multiplication algorithm against fault attacks is given in Section 9.8. This includes how the algorithm can be

securely used to calculate a modular exponentiation. An informal analysis of this suggests that this algorithm is robust against a range of fault attacks. A description of an implementation of this algorithm on 32-bit chip and the results of attempting to overcome the countermeasure by injecting faults with a laser are given in Appendix K.

- An application of a fault attack that fixes certain values of the random value used in DSA is described in Section 10.3, and an implementation of this attack is described in Appendix L. This is the first published practical implementation of an attack of this type.

- Methods for optimising the calculation of DPA traces are described in Appendix A.

1.2 Structure

The initial chapters of this thesis concern themselves with the background to side channel and fault attacks. Chapter 2 details the commonly used cryptographic algorithms that are found in most smart cards. Chapter 3 explains what a smart card is, and why there is interest in using smart cards to calculate cryptographic algorithms. Some of the chip features used for performance or security, are also detailed, and will be referred to in later chapters. Chapter 4 describes the side channels that are available when attacking a smart card. Timing and Simple Power Analysis attacks are briefly discussed, and the capabilities of each type of attack are described. A more detailed study is made of statistical power analysis, and a comparison of two methods used to extract information is given. Electromagnetic Power Analysis is also briefly described. Chapter 5 describes the various methods of injecting faults that could be used by an attacker. The effects that these faults could have on the execution of a cryptographic algorithm are also given, and a fault model is proposed.

The following chapters contain generic attacks and countermeasures that need to be taken into account when implementing cryptographic algorithms on smart cards. Chapter 6 details generic attacks that could be applied to any implementation of

a cryptographic algorithm on a smart card. These include manipulating the key, transferring a key from NVM to RAM, and cutting short repeating loops. Accessing S-boxes via a cache is also considered and extended further in this thesis. Chapter 7 details the countermeasures required to prevent the attacks already described, and that should be applied to every embedded implementation of cryptographic algorithms to prevent side channel and fault attacks. This includes an optimisation to the use of random delays to improve the performance and security of systems.

Chapters 8–10 are more concerned with specific algorithms. Chapter 8 details certain attacks that are predominantly applicable to block ciphers. The application of the previously described power attacks is discussed in terms of block ciphers. The first fault attack described is Differential Fault Analysis, followed by Collision Fault Analysis. The modification of S-box elements is then discussed separately, as it draws upon both Differential and Collision Fault Analysis. A list of countermeasures required to protect an implementation of a block cipher is also given.

Chapter 9 contains a similar analysis to that given in Chapter 8, but for the RSA algorithm when implemented using either a simple modular exponentiation or using the Chinese Remainder Theorem (CRT) [58]. The procedure of applying Simple and Statistical Power Analysis to the RSA signature is described. The numerous fault attacks that can be applied to the RSA signature scheme by modifying different variables of the algorithm are described. This is followed by a description of the various modular exponentiation algorithms that can be used to calculate the RSA algorithm. Methods of rendering these algorithms secure against power analysis are then discussed, including a new algorithm, followed by a description of how the algorithm can be defended against fault attacks. This discussion is then repeated for the RSA signature scheme when computed using the CRT. The security of the underlying multiplication algorithm is also addressed by proposing a secure Montgomery multiplication algorithm that is resistant to side channel and fault attacks. Some discussion of the various fault countermeasure techniques is also given and, as previously, a list of countermeasures is given for a secure implementation of the RSA signature scheme on an embedded chip.

Chapter 10 describes the security issues surrounding the implementation of DSA

on an embedded device. This chapter is relatively brief, as a large portion of the attacks and countermeasures are already discussed in Chapter 9, but the use of a random value in the algorithm requires a different type of countermeasure. An attack based on modifying the random value is described. This is followed by a discussion of the possible countermeasures that can be applied to protect the generation of the random value. This is followed by the Conclusion in Chapter 11.

Appendix A describes how the calculation of DPA traces can be optimised. Appendix B details how glitch parameters for a smart card can be determined, using Microchip's Silvercard as an example. Appendix E shows how the use of random delays can be optimised, and how this can be applied to random delays of any length. Appendices C, D, F, G, H, I, and L describe various fault attacks and how they are implemented. Appendix K describes experiments performed to validate the countermeasure proposed for Montgomery multiplication.

Chapter 2

Cryptographic Algorithms

Numerous cryptographic algorithms are in use today. In this chapter some of the most commonly used cryptographic algorithms are detailed. These descriptions cover algorithms that can be used as examples in generic attacks and countermeasures. Later in this thesis a description of how implementations of these algorithms can be made to resist side channel and fault attacks is given. DES and AES are also used as examples of generic attacks before the algorithm specific attacks are discussed.

The Data Encryption Standard is described in Section 2.1, the Advanced Encryption Standard is described in Section 2.2, RSA is detailed in Section 2.3, and a description of the Digital Signature Standard is given in Section 2.4.

2.1 Data Encryption Standard

The Data Encryption Standard (DES) was introduced by NIST in the mid 1970s [78], and was the first openly available cryptography standard. It has since become a worldwide *de facto* standard for numerous purposes. It is only in recent years that it has been practically demonstrated that an exhaustive search of the keyspace is possible, leading to the introduction of triple DES and the development of the Advanced Encryption Standard (AES) (see Section 2.2).

DES can be considered as a transformation of two 32-bit variables (L_0, R_0), i.e. the message block, through sixteen iterations of the Feistel structure, shown

in Figure 2.1, to produce a ciphertext block (L_{16}, R_{16}). The Expansion and P-permutations are bitwise permutations. For clarity of expression, these permutations will not always be considered and the round function will be written as:

$$R_n = S(R_{n-1} \oplus K_n) \oplus L_{n-1}$$

$$L_n = R_{n-1}$$

where S is the S-box function. The eight different S-boxes are applied in each round to sets of six bits. The subkeys K_n $(1 \leq n \leq 16)$ are each 48 bits generated from the 56-bit key, by permuting the bits of the initial key.

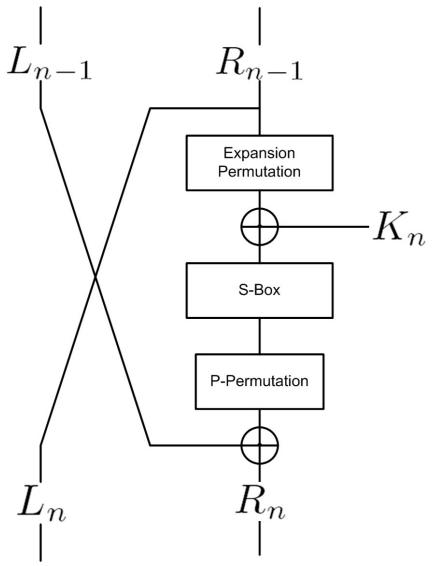


Figure 2.1: The DES round function for round n.

The algorithm also includes an initial and final permutation. These are also bitwise permutations and will be ignored, as they do not contribute to the security of the algorithm.

Triple DES

In order to mitigate the key length problem, a modification to DES was proposed to make an exhaustive key search prohibitively complex. Triple DES is a construction using two different DES keys and is defined in [78]. In the algorithm below these are

labelled K_1 and K_2, and in order to generate a ciphertext block C from a plaintext block M the following calculation is performed:

$$C = \text{DES}(\text{DES}^{-1}(\text{DES}(M, K_1), K_2), K_1)$$

where $\text{DES}(M, K)$ denotes the output of the DES encryption algorithm applied to message block M with key K. Deciphering the ciphertext block C uses the function,

$$M = \text{DES}^{-1}(\text{DES}(\text{DES}^{-1}(C, K_1), K_2), K_1)$$

Another version of triple DES is proposed in [78], in which three different keys are used rather than two.

2.2 Advanced Encryption Standard

The Advanced Encryption Standard (AES) [79] was introduced in 2001 as a replacement to DES. This was because it became apparent that the key used in DES is too short, as an exhaustive search had become feasible. The key used in AES can have a bit length of 128, 192 or 256 bits. For simplicity of presentation, in this thesis only the simplest case of a 128-bit key will be considered.

Algorithm 2.1: Advanced Encryption Standard

Input: M, K
Output: C

$X \leftarrow M \oplus K$
for $i \leftarrow 1$ **to** 10 **do**
  $X \leftarrow$ **ShiftRow**(X)
  $X \leftarrow$ **ByteSub**(X)
  if $i \neq 10$ **then**
    $X \leftarrow$ **MixColumn**(X)
  end
  $K \leftarrow$ **KeySchedule**(K)
  $X \leftarrow X \oplus K$
end
$C \leftarrow X$
return C

The structure of this algorithm is given in Algorithm 2.1, where a message block (M) is enciphered using a key (K) to produce a ciphertext block (C). The

ShiftRow function is a bytewise permutation of the input data. This is followed by the **ByteSub** function that is a substitution table applied to each byte of the input data. This table is an inversion over GF(2^8) followed by a bitwise permutation. The XOR with the subkey is also referred to as the **AddRoundKey** function. The **MixColumn** function is given in Algorithm 2.2, where $\bullet$ represents polynomial multiplication over GF(2^8) modulo the irreducible polynomial $x^8 + x^4 + x^3 + x + 1$.

Algorithm 2.2: The **MixColumn** Function

Input: $X = (x_0, x_1, \ldots, x_{15})_{256}$
Output: $Y = (y_0, y_1, \ldots, y_{15})_{256}$

for $i \leftarrow 0$ **to** 15 **do**
$y_i = 2 \bullet x_i \oplus 3 \bullet x_{(i+4) \bmod 16} \oplus x_{(i+8) \bmod 16} \oplus x_{(i+12) \bmod 16}$
end

return Y

The **xtime** function, as described in Algorithm 2.3, is given in [79] as a function that calculates a polynomial multiplication by the polynomial x over GF(2^8) modulo the polynomial given above. The first part of the function is a left bit shift of the input value, which represents multiplication by the polynomial x. The most significant bit is then tested. If the bit is equal to one then the result of the bit shift needs to be reduced to provide the result modulo the polynomial $x^8+x^4+x^3+x+1$. This is done by subtracting $11B_{16}$ (the hexadecimal representation of the polynomial $x^8 + x^4 + x^3 + x + 1$), which is equivalent to an XOR as no carry is produced in polynomial subtraction. The function described in Algorithm 2.3 is optimised so that the whole calculation can take place in eight bits, i.e. if the most significant bit is equal to one then it will be removed by the XOR and is therefore only required for testing purposes.

The **KeySchedule** is a function that generates a subkey from the previous subkey. The first subkey is the key with no changes, and subsequent subkeys are generated using this function. This function is described in Algorithm 2.4, where z is a constant that varies from one round to another. This function can be inverted, so the key can be generated given any of the subkeys.

A permutation is sometimes also used on the message and key on entry to the

Algorithm 2.3: The **xtime** function

Input: $X = (x_7, x_6, \ldots, x_0)_2$
Output: $Y = \mathbf{xtime}(X)$

$Y \leftarrow (X << 1) \wedge \mathrm{FF}_{16}$
if $x_7 = 1$ **then**
  $Y \leftarrow Y \oplus 1\mathrm{B}_{16}$
end
return Y

Algorithm 2.4: The AES Key Schedule

Input: $X = (x_0, x_1, \ldots, x_{15})_{256}$, z
Output: $X = (x_0, x_1, \ldots, x_{15})_{256}$

for $i \leftarrow 0$ **to** 3 **do**
  $x_{i<<2} \leftarrow x_{i<<2} \oplus \mathbf{ByteSub}(x_{(((i+1)\wedge 3)<<2)+3})$
end
$x_0 \leftarrow x_0 \oplus z$
for $i \leftarrow 0$ **to** 15 **do**
  if $i \neq 0 \bmod 4$ **then**
    $x_i \leftarrow x_i \oplus x_{i+1}$
  end
end
return X

algorithm, to convert the array format to the grid format used in the specification [79]. This is an optional bytewise permutation that changes the way bytes are addressed within an implementation.

2.3 RSA

RSA was first published in 1978 [94], and was the first published example of a public key encryption algorithm. The security of RSA depends on the difficulty of factorising large numbers. This means that RSA keys need to be quite large, because of advances in factorisation algorithms and the constantly increasing processing power available in modern computers.

To generate a key pair for use with RSA, two prime numbers, p and q, typically of equal bit length, are generated; they are then multiplied together to create a value N, the modulus, whose bit length is equal to that desired for the cryptosystem. That is, in order to create a 1024-bit modulus, $2^{511.5} < p, q < 2^{512}$. A public exponent,

e, is chosen that is coprime to both $(p-1)$ and $(q-1)$. A private exponent, d, is generated from the parameters previously calculated, using the formula:

$$ed \equiv 1 \pmod{(p-1)(q-1)}, \text{ or equivalently}$$
$$ed \equiv 1 \pmod{\phi(N)}$$

where ϕ is Euler's Totient function.

In the RSA cryptosystem, to encrypt a message, M, and create ciphertext, C, one calculates:

$$C = M^e \bmod N$$

The value of e is often chosen as 3 or $2^{16}+1$, as these values are small and have a low Hamming weight, which means that the encryption process is fast. However, the value 3 is rarely used, as if the result of an encryption is less than N then the message can be retrieved by taking the integer cube root of the ciphertext [67]. Another example of an attack against RSA encryption when a small exponent is used applies when an attacker can obtain several ciphertexts for the same message computed using different values for the modulus [67]. For example, suppose an attacker obtains the ciphertexts

$$C_1 = M^3 \bmod N_1$$
$$C_2 = M^3 \bmod N_2$$
$$C_3 = M^3 \bmod N_3.$$

An attacker can then use the Chinese Remainder Theorem to compute an integer X that satisfies

$$X \equiv C_1 \pmod{N_1}$$
$$X \equiv C_2 \pmod{N_2}$$
$$X \equiv C_3 \pmod{N_3},$$

where $0 \leq X < N_1N_2N_3$. Since $M^3 < N_1N_2N_3$ (as $M < N_1$, N_2, and N_3) then $X = M^3$, and M can be recovered by taking the integer cube root of X. A similar attack applies if e is any small value.

To decrypt the ciphertext, the same calculation is carried out but using the private exponent, d, which generally has the same bit length as N:

$$M = C^d \bmod N$$

The digital signature scheme involves use of the inverse operations. By convention, this is expressed as:

$$S = M^d \bmod N$$

The generation of a signature, S, uses the private exponent d. The verification therefore uses the public exponent and is expressed as:

$$M \stackrel{?}{=} S^e \bmod N$$

For simplicity the above notation will be used in the attacks described in subsequent sections.

Applying the RSA primitive to the message, as described above, will not yield a secure signature scheme (for reasons beyond the scope of this thesis). To achieve a secure scheme it is necessary to apply the RSA operation to a transformed version of the message, e.g. as can be achieved by hashing the message, adding padding, and/or masking the result.

In the attacks presented in this thesis, it is assumed that the attacker is able to completely manipulate the message being signed. This is clearly a strong attack model, but it allows for a more complete exploration of the attacks that could be applied to the RSA primitive.

Some of the attacks presented in this thesis will not be realistic when a padding scheme is used. However, it is important that the calculation of RSA is secure against all possible attacks. If a given implementation does not use padding or, more realistically, contains a bug that allows an attacker to remove the padding function the implementation should still be able to resist all known side channel and fault attacks.

Using the Chinese Remainder Theorem

The RSA calculation using the private exponent (i.e. where $S = M^d \bmod N$ and $N = p \cdot q$) can be performed using the Chinese Remainder Theorem (CRT) [58]. Initially, the following values are calculated,

$$S_p = (M \bmod p)^{(d \bmod (p-1))} \bmod p$$
$$S_q = (M \bmod q)^{(d \bmod (q-1))} \bmod q$$

which can be combined to form the RSA signature S using the formula $S = aS_p + bS_q \bmod N$, where:

$$\begin{array}{lll} a \equiv 1 & (\bmod\ p) \\ a \equiv 0 & (\bmod\ q) \end{array} \quad \text{and} \quad \begin{array}{lll} b \equiv 0 & (\bmod\ p) \\ b \equiv 1 & (\bmod\ q) \end{array}$$

This can be implemented in the following manner:

$$S = S_q + \left((S_p - S_q)\, q^{-1} \bmod p \right) \cdot q$$

This provides a method of calculating an RSA signature that is approximately four times quicker than a generic modular exponentiation algorithm. This advantage is offset by an increase in the key information that needs to be stored. Rather than storing just the value of d, the values of $(p, q, d \bmod (p-1), d \bmod (q-1), q^{-1} \bmod p)$ need to be precalculated and stored.

2.4 The Digital Signature Standard

The Digital Signature Standard (DSS) is a U.S. standard for generating digital signatures introduced by NIST in 1991 [83]. The algorithm specified in the standard is often referred to as the Digital Signature Algorithm (DSA).

The system parameters for DSA are (p, q, g), where p is a prime (typically of at least 1024 bits), q is a 160-bit prime dividing $p - 1$, and $g \in \mathbb{Z}_p^*$ has order q. The private key is an integer $\alpha \in \mathbb{Z}_q^*$ and the public key is the group element $\beta = g^\alpha \bmod p$.

To sign a message M, the signer generates a random value, referred to as a nonce, $k < q$ and computes:

$$r = g^k \bmod p \bmod q \quad \text{and}$$
$$s = \frac{h(M) + \alpha r}{k} \bmod q$$

where h is a hash function. The signature of M is the pair: (r, s). To check (r, s) the verifier checks whether:

$$r \stackrel{?}{=} \left(g^{wh(M)} \beta^{wr} \bmod p \right) \bmod q$$

where $w = s^{-1} \bmod q$.

Chapter 3

Smart Card Architecture

Smart cards are plastic cards containing an embedded microprocessor that are used as secure devices in a wide range of applications. This chapter describes some of the main features of smart cards. Particular attention is paid to features that have importance to subsequent chapters of this document. Further information on smart card architectures can be found in [91].

This chapter describes the characteristics of smart card architectures. This is not a complete description, but focuses on the features that are important for implementing secure cryptographic algorithms.

Section 3.1 describes what a smart card is, and where the definition of a smart card can be found. The various features of smart card microprocessors are described in Section 3.2.

3.1 What is a Smart Card?

In the late 1980s, the growing number of smart card applications led to the emergence of international standards for this technology. The International Organization for Standardization (ISO) has published standards that define all the basic characteristics of smart cards. The main body of smart card specifications is contained in the ISO/IEC 7816 series of standards.

Physical Properties A smart card, as defined in the ISO/IEC standards, is a rectangular card with round corners. A chip is inserted in the front side, and it can have one (or more) magnetic stripes on the back. The card's form and

contact positions specified in the ISO/IEC 7816-1 [53] and ISO/IEC 7816-2 [54] standards, respectively, and are shown in Figure 3.1.

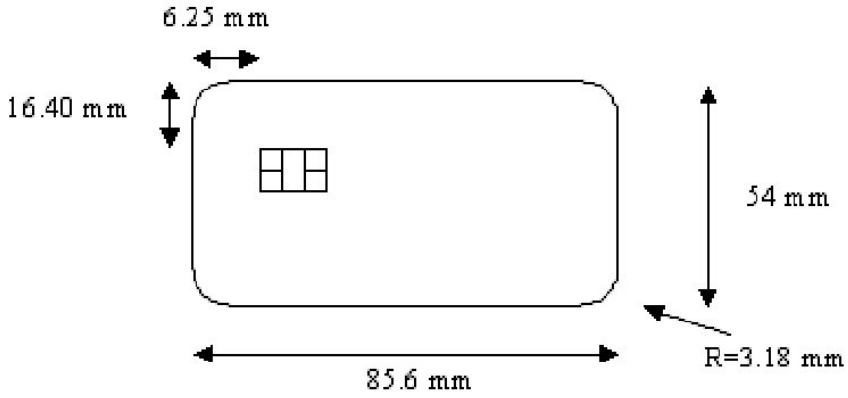


Figure 3.1: Classic shape and dimensions of a smart card.

Electrical Properties The integrated circuit is a 1 to 30 mm^3 silicon chip. This circuit is located in a micromodule and the chip is not visible (see Figure 3.2). In accordance with ISO/IEC 7816-2, the micromodule contains an array of eight contacts, only five of which are connected to the chip.

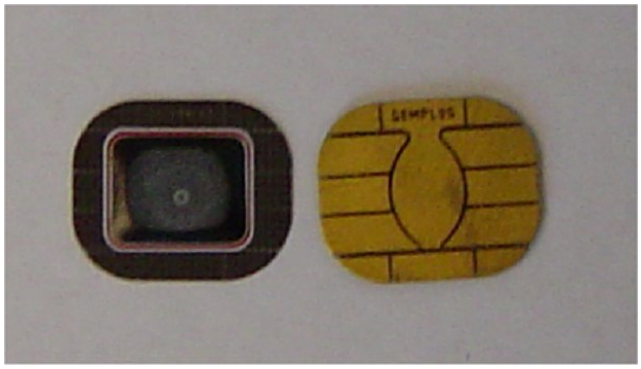


Figure 3.2: A front and rear view of a micromodule before it is set into a plastic card.

The standard voltage supplied to the smart card is 5 volts, with a possible variation of approximately 10%. For GSM applications a voltage range between 3 and 5 volts is required, as cellular phone components function with a

3 volt power supply. Smart cards rely totally on a card reader for their power supply. They do not have their own power supply because of size constraints.

Communication Smart card terminals use a simple protocol to communicate with the microchip embedded in smart cards. Every time the card is inserted into a terminal it is reset. After the voltage supply, the clock, and the reset signal have been applied, the card sends an Answer To Reset (ATR) to the terminal via the I/O pin. This ATR contains a series of bytes that define the parameters and communication protocol that can be used by the card during the session (as specified in the ISO/IEC 7816-3 standard [52]). The terminal has the ability to change the communication protocol, transaction speed or other parameters by sending a Protocol Type Selection (PTS) command to the card. The PTS will specify a set of parameters within the boundaries set by the card in the ATR. After the protocol has been established, the terminal can begin to send commands to the card using the Application Protocol Data Unit format (APDU). These exchanges are depicted in Figure 3.3.

Operating System (OS) At present, smart card Operating Systems fall into one of two classes: either an OS especially designed for a dedicated application (also called a native OS) or a Java Card OS that can host several applications. In the latter case, each application is handled by an applet, and several applets can be loaded onto the Java Card at a given time. The OS chosen is closely related to the chip's characteristics; for example, Java Cards require much more memory to house the virtual machine, applets, and all the functions the applet could call. The Java Card OS corresponds to a proprietary implementation of a standardised API defined by SUN Microsystems [102].

All the implementations discussed in this document will be based on smart cards with a native OS. This is so that the card can be strictly controlled and its behaviour from one execution to another remains constant. Despite this, the attacks and countermeasures described in this thesis still apply to Java Card implementations. However, it is more difficult to determine when a given algorithm is performed, as the CPU spends a large percentage of its

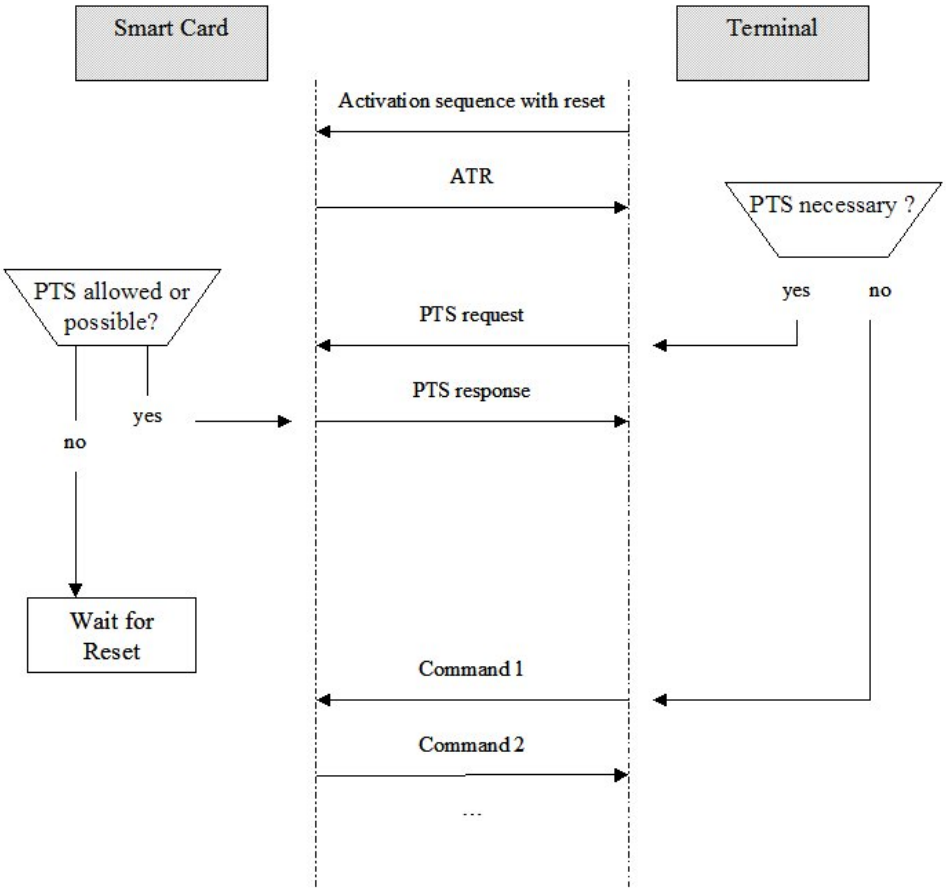


Figure 3.3: Communication initialisation between a smart card and a reader.

time executing the virtual machine. There are also events that occur intermittently, e.g. memory defragmentation, that can desynchronise events from one acquisition to another. This can make things more difficult for an attacker, but can often be overcome, as described in Section 7.2. The specific problems surrounding Java Card implementations and applets are beyond the scope of this thesis.

3.2 Microprocessors

Unlike memory cards and magnetic stripe cards, that are totally passive, the microprocessor embedded in a smart card can manipulate and control all the data present in the card. These processors usually employ well-known architectures.

The 8-bit microprocessors traditionally used in smart cards are based on Complex Instruction Set Computer (CISC) architectures [75]. These are often typically based on by Motorola's 6805 or Intel's 8051 core, with improvements to permit further optimisation of the embedded software. Continuing technological advances, along with the growing need for more sophisticated applications, are currently generating a significant shift in the hardware platforms used in smart cards. More sophisticated smart cards are emerging based on 32-bit Reduced Instruction Set Computer (RISC) architecture chips, containing dedicated peripherals (cryptographic coprocessors, memory managers, large memories, ...) [51, 71]. In this thesis, both types of architecture are analysed with regard to the differences in countermeasure requirements.

3.2.1 Cryptographic Coprocessors

Traditionally smart cards have been based around 8-bit architectures. In order to manipulate large numbers, e.g. to calculate the RSA algorithm described in Section 2.3, dedicated coprocessors can be appended to the CPU. In more modern 32-bit chips [51, 71] a coprocessor is not necessary to compute the RSA algorithm, as efficient software implementations can be achieved.

DES is also often implemented in a coprocessor to help increase performance, and AES implementations should be available in the near future. These coprocessors can increase the smart card's performance, as hardware implementations of secret key algorithms can be expected to require 1 or 2 clock cycles per round of the block cipher. However, the inclusion of coprocessors also increases the size of the chip and the overall power consumption. This means that chips with coprocessors are usually more expensive and are not ideal in environments where the amount of available current is severely limited, e.g. GSM Subscriber Identity Modules.

3.2.2 Random Number Generators

Random number generators are usually included in smart cards, as unpredictable numbers are an important element in many secure protocols. A true random number generator is typically based on a signal generated by an analog device (e.g. a

noisy resistor) which is then treated to remove any bias that may exist, or has been induced, in the bits generated. The correct functioning of all aspects of a smart card chip under varied environmental conditions is important, but is critical for random number generation because the quality of the generated random values can have a profound effect on cryptographic schemes. Random number generators are therefore designed to function correctly in a large range of environmental conditions, including temperature, supply voltage, and so on. However, if an attacker succeeds in modifying the environmental conditions such that the physical source of randomness is affected, the subsequent treatment is included so that an attacker will not be able to determine if the change in conditions had any effect.

Pseudo-random number generators are also often included. These are typically based on Linear Feedback Shift Registers (LFSRs) that are able to generate a new pseudo-random value every clock cycle, but are deterministic over time and are not usually used for critical security functions.

Where random values are required in cryptographic algorithms, a true random number generator is used when the quality of the random value is important, e.g. the generation of the nonce in DSA (see Section 2.4). Where the quality of the random value is less important, a pseudo-random number generator can be used, e.g. to govern the lengths of random delays (see Section 7.2). In some microprocessors only pseudo-random number generators are available. In this case, mechanisms that combine a random seed (that can be inserted into the chip during manufacture) with pseudo-random values can be used to provide random values.

An example of this latter type of random number generator is given in the ANSI X9.17 [3] standard, that uses DES to provide random values based on a random seed and another source of pseudo-random information. This functions by taking a 64-bit pseudo-random input (X), a 64-bit random seed (S) and a DES key (K). X is usually generated by calculating $X = \mathrm{DES}(D, K)$, where D is a the date and/or time, but this information is not available to a smart card and is therefore replaced with values provided by a pseudo-random number generator. To output a random value R the following calculation takes place:

$$R = \mathrm{DES}(X \oplus S, K),$$

and the random seed is updated using:

$$S = \mathrm{DES}(R \oplus X, K).$$

For increased security the DES function can be replaced with triple DES, as the key length used by DES has proven to be too short to entirely resist an exhaustive key search.

3.2.3 Cache

Sophisticated smart cards are emerging based on 32-bit CPUs, containing dedicated mechanisms designed to compensate for time-consuming operations or long data paths. Details of these sophisticated mechanisms are given, for example, in [49, 72]. Two mechanisms that are important in the context of this thesis are pipelining and caching, and are detailed below.

Pipelining: Pipelining is a technique whereby the execution of each instruction is decomposed into elementary and independent steps. Each step is implemented as a separate hardware block that can work in parallel. Typically, a 3-stage pipeline can be decomposed into an Instruction Fetch (IF), an Execute (EX) stage and a Write Back (WB) stage. More sophisticated 5-stage pipelines, such as that discussed in [72], can involve an IF stage, a DC (Decode) stage, an EX stage, a MEM (Memory access) stage and a WB stage. Each stage is designed to be completed within one clock cycle, which means that, even though each instruction takes 5 clock cycles in the case of a 5-stage pipeline, a new instruction can be issued at every clock cycle.

Caching: Smart card architectures include embedded Non-Volatile Memory (NVM) such as EEPROM or Flash to store code or data. Such memory usually has high read latency where, for example, reading one byte involves reading a whole line that takes several clock cycles. This would mean that the IF stage and the MEM stage would take more than one clock cycle, which would stall

the pipeline. This would considerably reduce the rate at which instructions can be issued. To compensate for these "slow" memories, cache mechanisms are implemented. A cache is a small, fast RAM whose role is to buffer the lines of NVM being fetched. Because of the technology used to implement them, and their small size (leading to faster decode and access times), caches allow a word to be fetched in one clock cycle.

When the data is to be fetched from the NVM, the CPU will first check whether this particular word is already in the cache: if yes (this is a *cache hit*), the word is fetched directly from the cache. If, on the contrary, this particular word is not cached, then this is a *cache miss*. The CPU will then fetch a whole line (e.g. 16 bytes) within which the targeted word can be found. This means that, even if fetching this word takes more than one clock cycle, the other words of this line will already be in the cache when required.

This mechanism considerably increases the instruction issue rate and, therefore, performance. Detailed studies of the performance enhancements of cache mechanisms are given in [49]. In order to keep power consumption low, smart card CPUs usually only have one level of cache, with a granularity in the order of 8 to 16 bytes.

3.2.4 Anomaly Sensors

There are usually a number of different types of anomaly detectors present in smart cards. These are used to detect unusual events in the voltage and clock supplied to the card, and the environmental conditions (e.g. the temperature). These enable a smart card to detect when it is exposed to conditions that are outside the parameters within which it is known to function correctly. When unusual conditions are detected, the chip will cease to function until the effect has been removed (i.e. initiate a reset or an infinite loop when the sensor is activated). However, these measures cannot be relied upon to defend against all fault attacks. For example, sensors that detect intense light (such as laser light) will only protect a small area of the chip's surface; an attacker could potentially find an area were a fault can be injected and remain undetected.

3.2.5 Chip Features

The surface of the chip used in a smart card can be uncovered by exposing the micromodule to fuming nitric acid* that will remove the resin visible in Figure 3.2. Once the chip has been revealed the easiest form of analysis is to simply look at it. The various different blocks can often be identified, as shown in Figure 3.4.

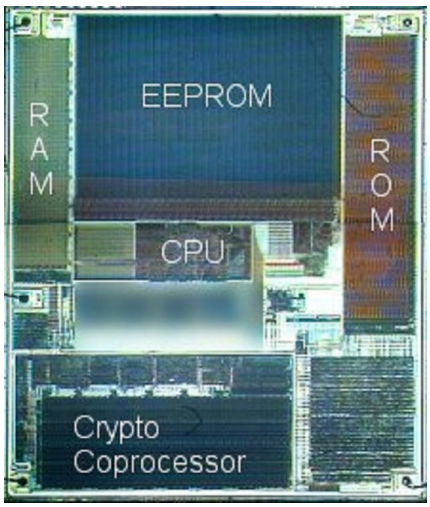


Figure 3.4: A chip surface with readily identifiable features.

Reverse engineering can target the internal design to understand how a given chip or block functions. A malicious attacker can use such information to improve their knowledge of chip design and find potential weaknesses in the chip, which may allow them to compromise the chip's integrity.

In modern smart cards, various features used to inhibit reverse engineering are implemented using glue logic: important blocks are laid out in a randomised fashion that makes reverse engineering difficult. This technique increases the size of the block, and is therefore not used in the design of large blocks such as the ROM and EEPROM.

A common technique to prevent this sort of identification and targeting is to overlay the chip with another metal layer that prevents the chip's features being identified. This can be removed using hydrofluoric acid that eats through the metal

*Fuming nitric acid is nitric acid with a concentration of over 86%.

layer; this reaction is then stopped using acetone before further damage is done and the chip surface can be analysed. The chip becomes non-functional but the layout of the chip can be determined, so that other chips of the same family can be attacked. The result of such a process is shown in Figure 3.5.

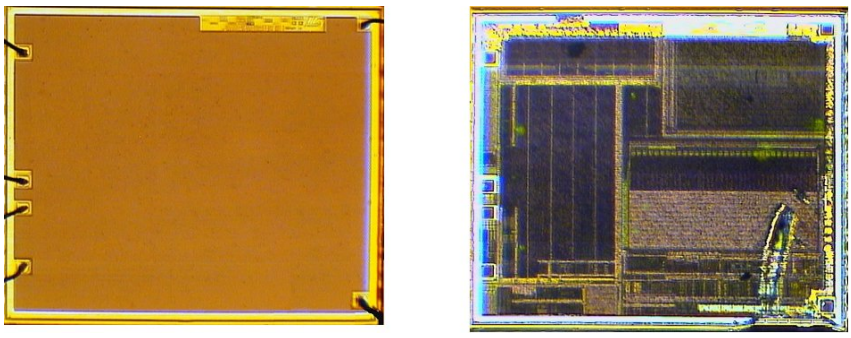


Figure 3.5: A chip with a shield present and removed. Removing a shield using hydrofluoric acid renders the chip non-functional but allows the layout to be determined.

Discovering the layout and functioning of a chip is particularly important when using a laser as a fault injection mechanism. Different areas of a chip can be targeted through the metal layer once the layout of a chip is known.

3.3 Summary and Conclusions

This chapter describes some of the features that are present in smart cards. The physical characteristics of a smart card are defined in the ISO/IEC 7816 series of standards, some features of which are described above. Common security features of smart cards are also described. The features described in this chapter will be referred to in later chapters where appropriate.

Chapter 4

Side Channel Analysis Techniques

Sensitive systems that are based on smart cards use protocols and algorithms that have usually been subjected to rigorous analysis by the cryptographic community. Attackers have therefore sought other means to circumvent the security of protocols and algorithms used in smart card based systems. As smart cards are small devices, they can easily be put in a situation where their behaviour can be observed and analysed. The simplest form of analysis of this type is intercepting the communication between a smart card and its reader. Tools are readily available on the Internet [7] that allow the commands to and from a smart card to be logged. This problem is relatively easy to solve using secure session key generation to encipher subsequent communication, as proposed in [43], but highlights the ease of man-in-the-middle attacks against smart cards.

More complex attacks can be realised by monitoring information that leaks naturally when a smart card processes information. These are referred to as side channel attacks. In this chapter, the types of side channel analysis that are potentially possible against cryptographic algorithms implemented on embedded platforms are described. Particular emphasis is placed on the different versions of Statistical Power Analysis, to demonstrate the difference in the quality of the results.

Timing analysis is briefly discussed in Section 4.1. The possible uses of Simple Power Analysis is detailed in Section 4.2. A detailed description of Differential Power Analysis is given in Section 4.3. Correlation Power Analysis is then described and

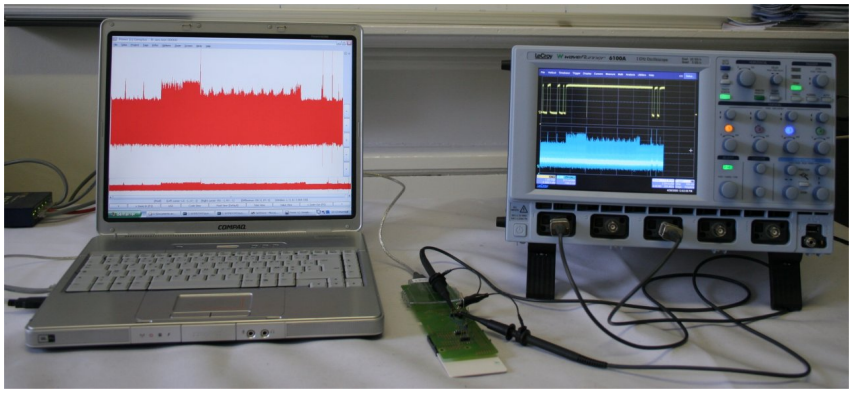


Figure 4.1: Data acquisition tools.

compared with Differential Power Analysis. Electromagnetic Power Analysis is also briefly described in Section 4.4.

4.1 Timing Analysis

The first published side channel attack [60] targeted public key cryptographic algorithms. Timing analysis involves observing the amount of time it takes to perform certain operations. By monitoring the length of time taken by a command, deductions can be made about the data being manipulated within a smart card. This can either be measured by acquiring the I/O during one command using an oscilloscope, as shown in Figure 4.1, or by using certain proprietary card readers. If an oscilloscope is used, the number of clock cycles taken to perform a given command can be deduced by analysing the yellow trace visible on the oscilloscope shown in Figure 4.1. This aspect of side channel analysis will not be covered in depth, as the countermeasures are trivial, but will be mentioned where relevant.

4.2 Simple Power Analysis

The most basic form of power analysis is known as Simple Power Analysis (SPA). This involves observing the power consumption of the card via an oscilloscope, and making deductions about what calculations are being performed based on these observations. Information regarding power consumption can be acquired using the

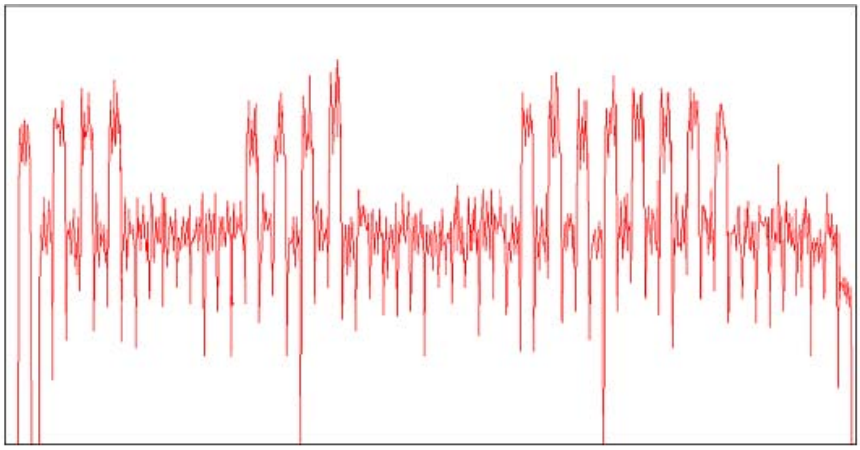


Figure 4.2: The power consumption of an unprotected modular exponentiation implemented using the square and multiply algorithm.

equipment shown in Figure 4.1, where the power consumption is given by the blue trace on the oscilloscope. A widely known attack involves observing the power consumption during the execution of a modular exponentiation.

Examination of the power consumption during a modular exponentiation computation can potentially reveal information about the exponent. An example of the power consumption of a smart card during the execution of a modular exponentiation is shown in Figure 4.2.

Looking closely at the acquired power consumption, a series of events can be seen. There are two types of event at two different power consumption levels, with a short dip in the power consumption between each event. This corresponds well to the simplest algorithm for calculating a modular exponentiation: the square and multiply algorithm. This is where the exponent is read bit-by-bit, a zero results in a squaring operation being performed, and a one results in a squaring operation followed by a multiplication with the original message. This algorithm is detailed in Algorithm 4.1.

Given the ratio of the two features, it can be assumed that the feature with the lower power consumption represents the squaring operation and the higher power

Algorithm 4.1: The Square and Multiply Algorithm

Input: $m, d = (d_0, d_1, d_2, \ldots, d_x)_2, n$
Output: $c = m^d \bmod n$

$c \leftarrow m$
for $i \leftarrow 0$ **to** x **do**
  $c \leftarrow c^2 \bmod n$
  if $d_i = 1$ **then**
    $c \leftarrow c \cdot m \bmod n$
  end
end
return c

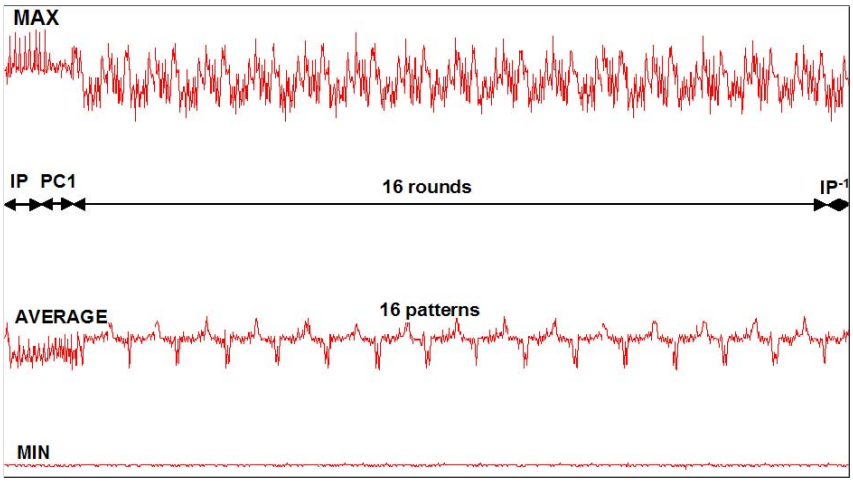


Figure 4.3: The power consumption of a DES implementation showing the rounds of the algorithm.

consumption represents the multiplication. From this, the beginning of the exponent can be read from the power consumption, in this case the exponent used is F00F000FF00_{16}. Further details of this attack and the relevant countermeasures are described in Chapter 9.

In a more general sense, the performance of individual functions are also visible in the power consumption. Figure 4.3 shows the power consumption of a smart card during the execution of DES. The rounds of the DES algorithm are clearly visible as a pattern that repeats 16 times.

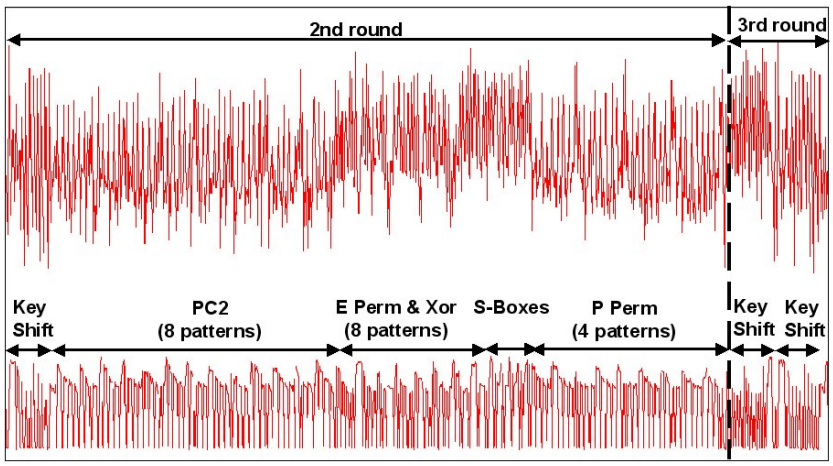


Figure 4.4: The power consumption of a DES implementation showing the round functions.

This analysis can be further improved by closely inspecting the power consumption during one round. The functions executed in the round can be seen. This is shown in Figure 4.4 where the functions in the second round of a DES implementation are evident from the power consumption trace. This may not be immediately apparent, as close inspection of the trace's features is necessary to identify the individual functions.

This is important, as the statistical attacks described in Section 4.3 target specific functions in cryptographic algorithms. A thorough simple power analysis of an algorithm implementation can greatly increase the speed of statistical attacks, as the area that needs to be analysed can be defined, therefore reducing the amount of data that needs to be acquired.

The same is true for fault injection techniques, detailed in later chapters, as it is often necessary to target specific events. If arbitrary functions can be identified using the power consumption, the point in time at which an attacker wishes to inject a fault can be discovered. This can greatly decrease the time required to conduct a successful attack, as less time is wasted injecting faults into areas of the computation that will not produce the desired result.

It should be noted that all the examples given in this section have been taken

from chips that display the differences in an obvious manner. Modern smart cards rarely display the functions being executed as clearly as in the examples given.

4.3 Statistical Power Analysis

Statistical Power Analysis is a more powerful form of power analysis attack (it can be further sub-categorised into Differential Power Analysis (DPA) and Correlation Power Analysis (CPA)). In this type of attack a smart card command is executed numerous times with different inputs, and the corresponding power consumption is captured for each input. A statistical analysis is then performed on the data gathered to infer information about the secret information held inside the card. These techniques require many more data acquisitions than with Simple Power Analysis, and a certain amount of treatment *a posteriori*.

Statistical power attacks are based on the relationship between the power consumption and the Hamming weight of data being manipulated at a given point in time. The differences in power consumption are potentially extremely small, and cannot be interpreted individually, as the information will be lost in the noise incurred during the acquisition process. The small differences produced can be seen in Figure 4.5, where traces were taken using a chip where the acquisition noise is exceptionally low.

The first implementation of this type of attack is known as Differential Power Analysis (DPA) [61], and works by amplifying the differences in the power consumption. This was followed by Correlation Power Analysis [26], that uses the correlation between the power consumption and the Hamming weight of the machine word being manipulated at a given point in time.

4.3.1 Modelling the Power Consumption

Two main models have been proposed to explain how the power consumption changes as data is being processed.

The Hamming Weight Model: The simplest model, initially proposed in [61, 69], involves an affine relationship between the power consumption and the

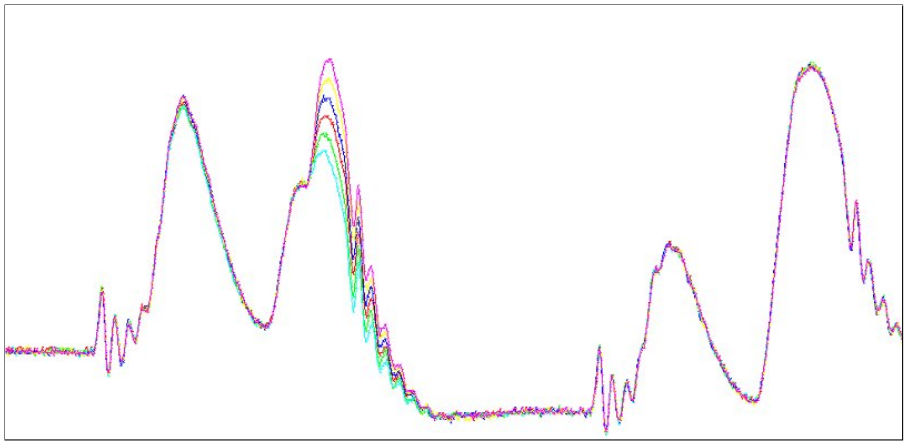


Figure 4.5: Overlaid acquisitions of the power consumption produced by the same instruction but with varying data.

Hamming weight of the data being manipulated at a given point in time. This is largely caused by the bus on the microprocessor, that consumes a large amount of power compared to any other single feature on the chip. This can be expressed as,

$$Y = aH(X) + b$$

where Y is the power consumption, X is the value of the data being manipulated, and H is a function that calculates the Hamming weight. The variable b includes the acquisition noise and the variation from one clock cycle to another, as the commands being executed by the CPU change.

This can be justified by observing that the tracks of the bus will either be zero (and consume no power) or one (and will therefore consume power). As each track should consume the same amount of power, the power is therefore proportional to the number of tracks with current flowing in them. As all the information being processed by the chip is carried across the bus, this relationship can be exploited when secret information is sent across the bus.

The Hamming Distance Model: Another model was proposed in [26] that generalises the Hamming weight model. This model is a similar model but with

an addition of another variable, P. This model is:

$$Y = aH(P \oplus X) + b$$

The notation is the same as described in the previous model, and the value P represents some previous state. The justification for this model is that the Hamming weight model is only valid if the bus is set to zero between each value sent. If this value is not set to zero then there is a transition from state P to state X, so the change in value of the bus can be modelled by an XOR between the two values.

A command fetched by the CPU will often consist of several opcodes: usually an instruction followed by the data being manipulated. In this case, X is the data being manipulated and P is the instruction opcode. If the data is being sent over a separate data bus, P will either be previous data or zero, depending on how the chip manages values on the data bus, i.e. if the bus is set to zero when it is not in use or if it maintains its previous state.

Both models are used in Statistical Power Analysis. In Differential Power Analysis it is not necessary to consider the Hamming Distance Model, as this will only change the sign of the peak produced. The Hamming Distance model becomes more important when conducting Correlation Power Analysis, where a more complete model of the chip is required.

4.3.2 Differential Power Analysis

Differential Power Analysis (DPA) can be performed on any algorithm in which an intermediate operation of the form $\beta = S(\alpha \oplus K)$ is calculated, where α is known and K is the key (or some segment of the key). The function S is a non-linear function, usually a substitution table (referred to as an S-box), which produces an intermediate output value β.

The process of performing the attack initially involves running the target device N times with N random plaintext values P_i, where $1 \leq i \leq N$. The encryption of the plaintext P_i under the key K to produce the corresponding ciphertext C_i

will result in a power consumption waveforms w_i for every i $(1 \leq i \leq N)$. These waveforms are captured with an oscilloscope, and sent to a computer for analysis and processing.

To find a given partial key (K_j), the output produced from the S-box (ℓ_j) when given K_j and all $P_{i,j}$ will be used to categorise the power consumption waveforms. A single output bit b from ℓ_j is used for this categorisation. For all possible hypotheses, i.e. that K_j is an integer in the interval $[0, 2^m - 1]$, and each partial message $P_{i,j}$, b will classify whether waveform w_i is a member of one of two possible sets. The first set S_0 will contain all the waveforms where b is equal to zero, and the second set S_1 will contain all the remaining waveforms, i.e. where the output bit b is equal to one.

For each hypothesis, a differential trace Δ_n is calculated by finding the average of each set and then subtracting the resulting values from each other, where all operations on waveforms are ordinary functions conducted in a pointwise fashion.

$$\Delta_n = \frac{\sum_{w_i \in S_0} w_i}{|S_0|} - \frac{\sum_{w_i \in S_1} w_i}{|S_1|}$$

The DPA waveform with the highest peak will validate a hypothesis for K_j, i.e. $K_j = n$ corresponds to the Δ_n featuring a maximum amplitude. For a single DPA waveform this involves $N - 2$ additions (as there will be an average of $\frac{N}{2}$ elements in each set, each requiring $\frac{N}{2} - 1$ additions to compute) to create the differential Δ_n. Therefore, for all hypotheses, the total number of operations to calculate all DPA waveforms for one S-box is $2^m \times (N - 2)$. Some optimisations to the process of generating DPA traces are described in Appendix A.

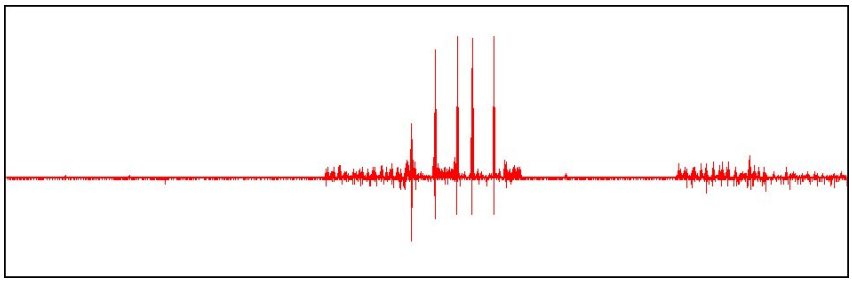


Figure 4.6: A DPA trace.

An example of a DPA trace is shown in Figure 4.6. This is a DPA trace generated with a correct key hypothesis to predict one bit of the output of the first S-box of DES. The first peak corresponds to the output of the S-box, the four subsequent peaks correspond to the same bit being manipulated in the P-permutation. The size of a peak is considered to be its absolute value.

4.3.3 Correlation Power Analysis

Another version of Statistical Power Analysis called Correlation Power Analysis (CPA) was proposed in [26], where the equation for generating differential waveforms is replaced with the formula for Pearson's correlation coefficient. In this case the power consumption is measured as before, represented below by $aH + b$ (where H represents the Hamming weight used in the models given in Section 4.3.1). Rather than analysing one bit, an attacker seeks to predict the Hamming weight of the computer word being manipulated at a given point in time. These predictions are given by H' leading to the following formula to calculate the coefficient.

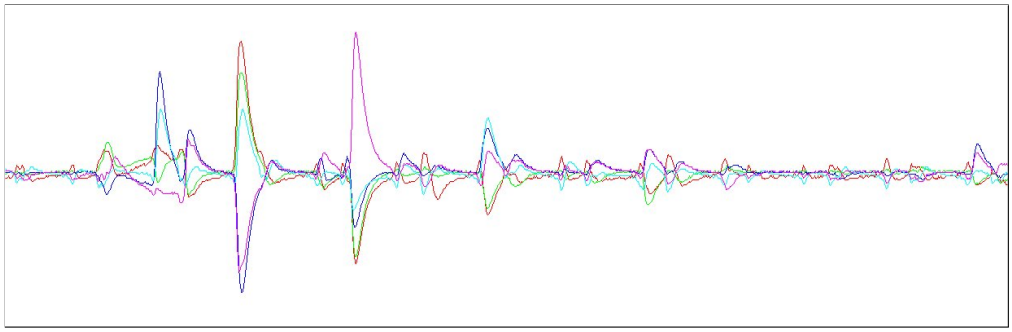
$$\rho = \frac{\mathrm{cov}(aH + b, H')}{\sigma_{(aH+b)}\sigma_{H'}}$$

where ρ is the correlation between $aH + b$ and H'. In practice, this will be calculated using a series of power consumption traces (w_i) and a corresponding list of predictions for the Hamming weight at a given point (H_i). This gives the following calculation:

$$\rho = \frac{N\sum_{i=1}^{N} W_i H_i - \sum_{i=1}^{N} W_i \sum_{i=1}^{N} H_i}{\sqrt{N\sum_{i=1}^{N} W_i^2 - (\sum_{i=1}^{N} W_i)^2}\sqrt{N\sum_{i=1}^{N} H_i^2 - (\sum_{i=1}^{N} H_i)^2}}$$

where N acquisitions have been taken. As previously, the calculation is applied to the series of acquisitions in a pointwise fashion.

The series of data represented by H_i is predicted using one of the models proposed in Section 4.3.1. When the Hamming distance model is used, it is necessary to calculate the correlation coefficient for every possible value for the unknown previous state P. The highest correlation coefficient should give the value of P. When this is used to attack an algorithm, the highest correlation will generally give the



Key:	**Red**	Key hypothesis 42
	Cyan	Key hypothesis 22
	Blue	Key hypothesis 24
	Magenta	Key hypothesis 43
	Green	Key hypothesis 19

Figure 4.7: DPA traces on bit 1 of S-box$_1$ for various guesses (correct guess is 24).

value for a portion of the key and the previous state at the same time. This subject is given a more rigorous treatment in [26].

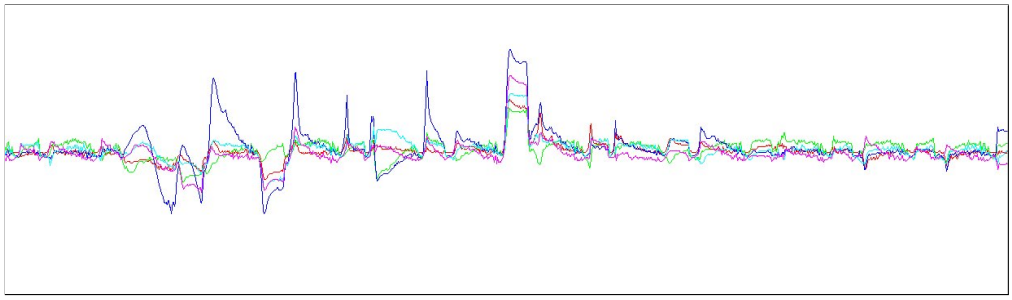
4.3.4 Comparing the Different Methods

It is possible to use DPA to derive a key. However, there are two main problems with DPA. These are:

1. DPA peaks also appear for wrong hypotheses. These are referred to as "ghost peaks" [26]. This can cause problems when trying to differentiate the various possible hypotheses. An example of this phenomenon can be seen in Figure 4.7, where there are various peaks, all corresponding to different key hypotheses for six bits of a DES subkey.

2. The DPA for an incorrect guess can also be larger than for the correct guess. This can be seen in Figure 4.7, where the magenta waveform represents a false hypothesis but has the most significant DPA peak.

A worked example of how this is possible is given in [26] and is repeated here. Consider the leftmost bit of the output of the fifth S-box of DES when the input data (M) varies from 0 to 63 and is combined with two different subkeys,



Key:		
	Red	Key hypothesis 63
	Cyan	Key hypothesis 10
	Blue	Key hypothesis 24
	Magenta	Key hypothesis 05
	Green	Key hypothesis 15

Figure 4.8: CPA traces on 4 bits of S-box$_1$ for various guesses (correct guess is 24).

(MSB(S-box $\ell_5(M \oplus 0)$) and MSB(S-box $\ell_5(M \oplus 36_{16})$)). The series of bits produced by the varying messages can be predicted. These are listed below, followed by the XOR between the two sets of data.

```
11011010100101100010010110010011101010010110110101010010001011 01
10011010110101100010010111010010101011010110100101010010001110 01
01000000010000000000000001000001000001000000010000000000000010 100
```

The third line contains eight bits set to one. This means that there are only eight values that will distinguish between the two possible key values. When these values are used to produce a DPA trace, the false key key value would produce a DPA peak whose height is 56/64 that of the correct value. Given that a certain amount of noise cannot be avoided, this could cause problems.

Another potential problem arises as DPA makes the assumption that all the data not under direct consideration is uniformly distributed. This is not always the case, as the structure of the S-boxes will give some bias to the Hamming weight of the data not being considered. This can potentially change the resulting value by reducing the DPA peak for a correct hypothesis or by supporting an incorrect hypothesis.

CPA removes these problems by considering the whole machine word being manipulated at a given point in time. This can also necessitate making some assumptions about the implementation details [26]. Figure 4.8 shows the CPA waveforms for the same data used to generate the DPA waveforms shown in Figure 4.7. A significant distinction can be seen between the correct hypothesis and the incorrect ones. This demonstrates that CPA can provide more reliable results for an attacker, as it is based on a more complete model of the chip. This is discussed in more depth in [26].

4.4 Electromagnetic Analysis

Electromagnetic analysis [39] is only different to power analysis in the way in which data is acquired. After this, all the *a posteriori* treatment remains the same. The attacks previously described for Simple Power Analysis and Statistical Power Analysis will function in the same manner for electromagnetic acquisitions.

However, the measurement of different types of event is possible via electromagnetic analysis, as the probe is very small when compared to the size of the chip. An example electromagnetic probe is shown in Figure 4.9, where the probe is shown above an opened smart card chip. The probe is of an equivalent size to the chip's features, which can be seen as blocks on the chip's surface. This means that the probe can be placed just above a given feature to try and get a strong signal from this part of the chip, e.g. the bus between two areas of the chip, while excluding noise from other areas of the chip.

This technique can also be used to overcome certain hardware countermeasures such as current scramblers. Unfortunately, it is a much more complicated attack to realise, as the chip needs to be open, as shown in Figure 4.9. If the chip is not open the signal is usually not strong enough for any information to be deduced. The tools required to capture this information are also more complex, as the power consumption can be measured by simply reading the potential difference over a resistor in series with a smart card. To measure the electromagnetic field involves building a suitable probe (the probe in Figure 4.9 is handmade) and the use of

Figure 4.9: Electromagnetic probing of a chip.

amplifiers, so that an oscilloscope can detect the signal.

4.5 Summary and Conclusions

This chapter provides some of the basic side channel attack techniques that will be referred to in later chapter. The techniques of Timing and Simple Power Analysis are described. The two predominant versions of Statistical Power Analysis are also described. It is shown that Correlation Power Analysis allows for a better distinction between a correct hypothesis and an incorrect hypothesis than Differential Power Analysis.

Chapter 5

Fault Injection Techniques

One of the first examples of fault injection in microprocessors was accidental. It was noticed that radioactive particles produced by elements naturally present in packaging material [66] caused faults in chips. Specifically, these faults were caused by Uranium-235, Uranium-238 and Thorium-230 residues present in the packaging decaying to Lead-206 and releasing α particles. These particles created a charge in sensitive chip areas causing bits to flip. Whilst these elements were only present in two or three parts per million, this concentration was sufficient to change the chip's behaviour. Subsequent research included studying and simulating the effects of cosmic rays on semiconductors [112]. Cosmic rays are very weak at ground level because of the Earth's atmosphere, but their effect becomes more pronounced in the upper atmosphere and outer space.

This has provoked a great deal of research by organisations such as NASA and Boeing, and most of the work on fault resistance was motivated by this vulnerability to charged particles in space and the upper atmosphere. Considerable engineering endeavours were devoted to the "hardening" of electronic devices designed to operate in harsh environments. Various other fault induction methods have since been discovered, but all have similar effects on chips. For example, a laser can be used to imitate the effect of charged particles [45]. The different faults that can be produced have been characterised to enable the design of suitable protection mechanisms.

This chapter describes the different fault effects that an attacker could apply to a smart card. This includes both transient and permanent faults, although transient faults are more likely to be of interest to an attacker. The aim of this chapter is to

define a model for the effects that could potentially be produced by a fault.

Section 5.1 details the types of fault injection methods that are potentially available to an attacker. Section 5.2 describes the types of faults that can be provoked at the silicon level of a chip. In Section 5.3, the types of effect that an attacker would expect to be able to provoke in an algorithm are described. A brief description of how side channel analysis can aid in injecting faults is given in Section 5.4. Some of the work in this chapter has been published in [10, 11], and some work remains unpublished [81].

5.1 Injection Techniques

Several ways of inducing faults in smart cards have been discussed in the literature. The most common fault injection techniques are:

Variations in Supply Voltage: [5, 22] during execution may cause a processor to misinterpret or skip instructions. An example of how glitch parameters can be determined is given in Appendix B.

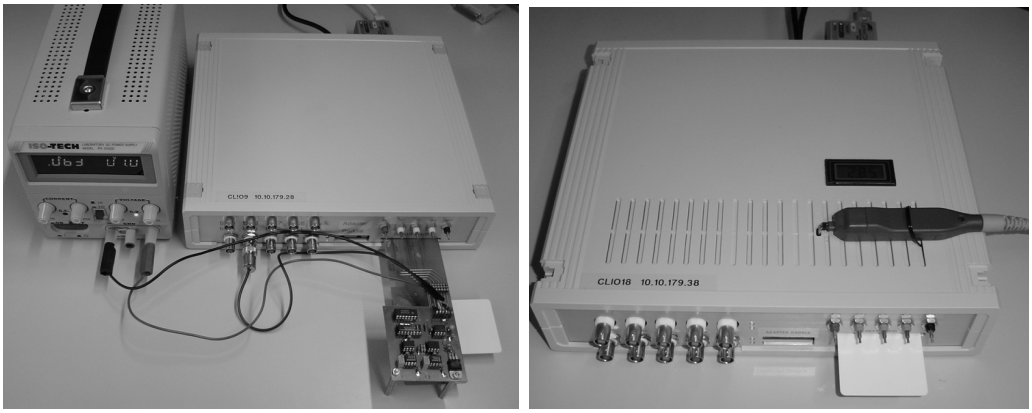


Figure 5.1: Supply voltage glitch fault injection equipment.

Variations in the External Clock: [5, 6, 64] may cause data to be misread (the circuit tries to read a value from the data bus before the memory has time to latch out the correct value) or an instruction miss (the circuit starts executing

instruction $n+1$ before the microprocessor has finished executing instruction n).

Temperature: [23, 44] Circuit manufacturers define upper and lower temperature thresholds within which their circuits will function correctly. An attacker, for example, can vary the temperature using an alcoholic cooler until the temperature drops below the threshold for the chip under attack. When conducting temperature attacks on smart cards, two effects can be obtained: the random modification of RAM cells due to heating, and the exploitation of the fact that read and write temperature thresholds do not coincide in most Non-Volatile Memories (NVMs). By tuning the chip's temperature to a value where write operations work but read operations do not, or the other way around, a number of attacks can be mounted.

White Light: [5] All electric circuits are sensitive to light because of the photoelectric effect. The current induced by photons can be used to induce faults if a circuit is exposed to intense light for a brief period of time. This can be used as an inexpensive means of fault induction [100]. An example of equipment for injecting faults using this method is shown in Figure 5.2.

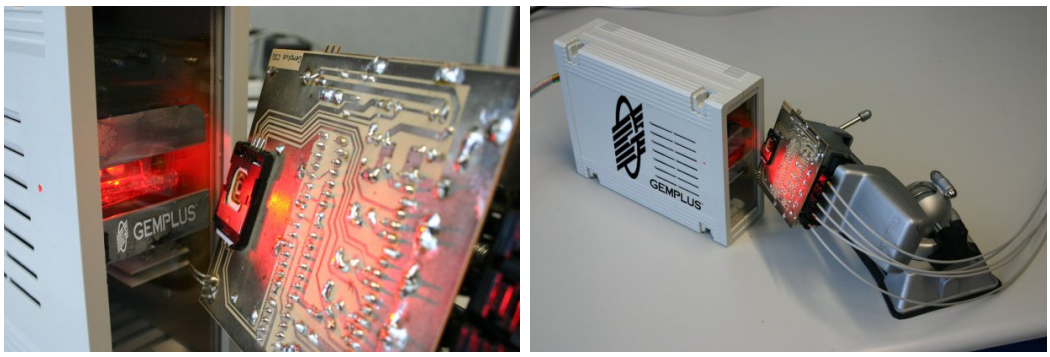


Figure 5.2: White light fault injection equipment.

Laser Light: [37, 45, 90] A wide variety of faults can be produced with laser light, and can be used to simulate faults induced by particle accelerators. The effect produced is similar to white light, but the advantage of a laser over white light

is the directionality that allows a small circuit area to be precisely targeted. An example of laser fault injection equipment is shown in Figure 5.3.

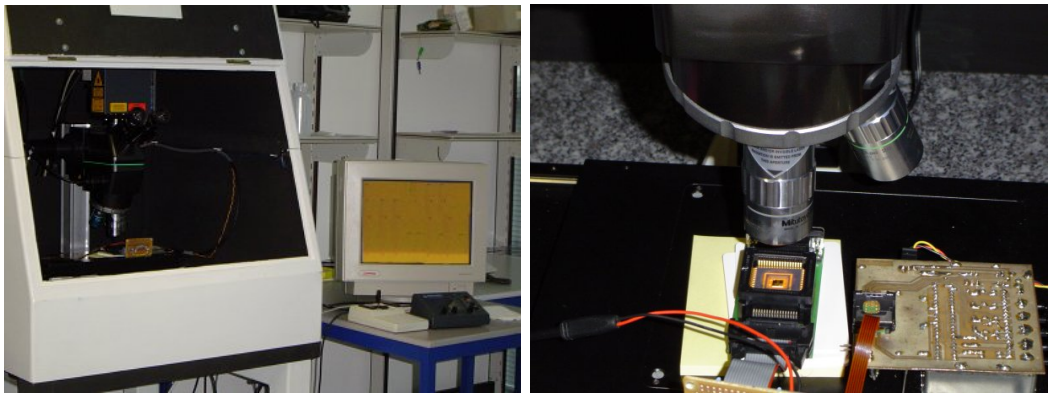


Figure 5.3: Laser fault injection equipment.

X-rays and ion beams: [13] can also be used as fault sources (although these are less common). These have the advantage of allowing the implementation of fault attacks without necessarily de-packaging the chip. However, the equipment required is extremely expensive.

Electromagnetic flux: [95] has also been shown to be able to change values in RAM, as eddy currents can be made strong enough to affect microprocessors.

5.2 The Types of Fault

Electronic circuits can be subjected to two classes of fault: provisional (transient) and destructive (permanent) faults. In a provisional fault, silicon is locally ionised so as to induce a current that, when strong enough, is falsely interpreted by the circuit as an internal signal. As ionisation ceases, so does the induced current (and therefore the resulting faulty signal), and the chip recovers its normal behaviour. Conversely, destructive faults, created by deliberately inflicted defects in the chip's structure, have a permanent effect. Once induced, such faults will affect the chip's behaviour permanently.

5.2.1 Provisional Faults (Taxonomy)

Provisional faults have reversible effects and the circuit will recover its original behaviour after the system is reset or when the stimulus ceases.

Single Event Upsets: (SEUs) are flips in a cell's logical state to a complementary state. The transition can be temporary, if the fault is produced in a dynamic system, or permanent if it appears in a static system. SEUs were first noticed during a space mission in 1975 [84, 88] and stimulated research into the mechanisms by which faults could be created in chips. SEUs can also manifest themselves as a variation in an analogue signal such as the supply voltage or the clock signal.

Multiple Event Upsets: (MEUs) are a generalisation of SEUs. The fault consists of several SEUs occurring simultaneously. A high integration density can provide conditions favourable to the genesis of MEUs.

Dose Rate Faults: [63] are caused by several particles whose individual effect is negligible but whose cumulative effect generates a sufficient disturbance for a fault to appear.

5.2.2 Destructive Faults (Taxonomy)

Destructive faults provoke a permanent fault in a circuit and the chip will not return to its original behaviour. There are four different types of destructive faults, classified in the following manner:

Single Event Burnout faults: (SEBs) are caused by a parasitic thyristor* being formed in the MOS power transistors [65, 101]. This can cause thermal runaway in the circuit causing its destruction i.e. the current flowing through the affected portion of the circuit will heat the circuit damaging it.

*A thyristor is a device similar to a diode, with an extra terminal that is used to turn it on. Once activated, the thyristor will remain on as long as there is a significant current flowing through it. If the current falls to zero the device switches off.

Single Event Snap Back faults: (SESs) [62] are caused by the self-sustained current by the parasitic bipolar transistor in MOS transistor channel N. This type of fault is not likely to occur in devices with a low supply voltage.

Single Event Latch-up faults: (SELs) [1, 37] are propagated in an electronic circuit by the creation of a self-sustained current with the releasing of PNPN parasitic bipolar transistors in CMOS technology. This can potentially destroy the circuit. This effect is shown in Figure 5.4.

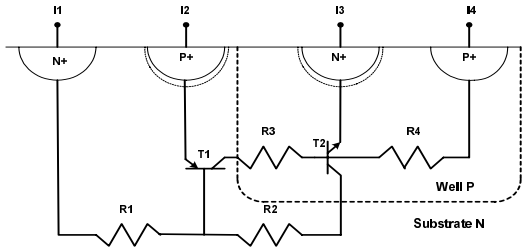


Figure 5.4: Single event latch-up — parasitic transistors T1 and T2.

Total Dose Rate faults: [27] are caused by a progressive degradation of the electronic circuit subsequent to exposure to an environment that can cause defects in the circuit [93].

When using fault injection as an attack method, provisional faults are the method of choice. These allow for faults under a variety of experimental conditions to be attempted until the desired effect is achieved, and also leaves the chip in a functional state after the attack has been completed. A destructive fault would (usually) render the target unusable and will necessitate the manufacturing of a clone.

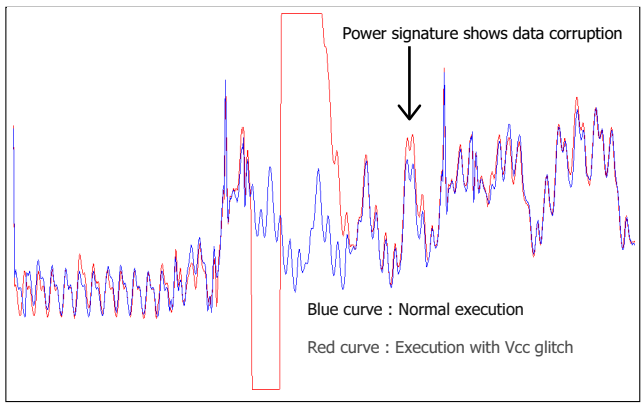
5.3 Fault Models

The fault injection methods described above have many different effects on silicon. They can be modelled in ways that depend on the type of fault injection that has been used. The following list indicates the possible effects that can be created by these methods:

Resetting Data: an attacker could force the data to the blank state, i.e. reset a given byte, or bytes, of data back to `00` or $\texttt{FF}_{16}$, depending on the logical representation.

Data randomisation: an attacker could change the data to a random value. However, the adversary does not control the random value, and the new value of the data is unknown to the adversary. Such an effect can be seen in the power consumption, as shown in Figure 5.5, although it is not clear if the data is randomised or reset.

Modifying opcodes: an attacker could change the instructions executed by the chip's CPU, as described in [5]. This will often have the same effect as the previous two types of attack. Additional effects could include removal of functions or the breaking of loops. The previous two models are algorithm dependent, whereas the changing of opcodes is implementation dependent. Such an effect can been seen in the power consumption as shown in Figure 5.6.

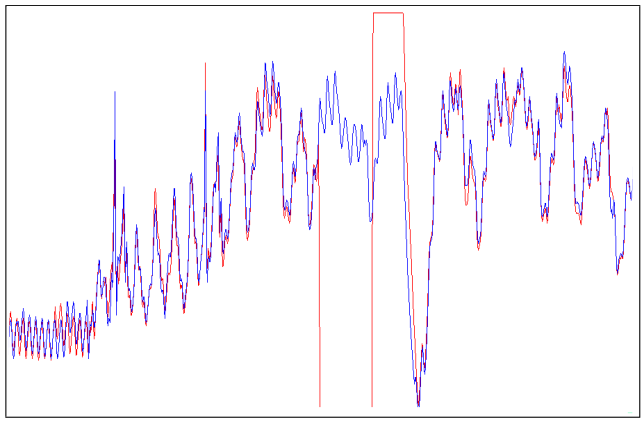


The blue trace represents a normal execution. The red trace shows the same code being executed but with a fault injected that modifies the data being manipulated.

Figure 5.5: The effect of a glitch on data being manipulated by a chip, as visible in the power consumption.

These three types of attack cover everything that an attacker could hope to do to an implementation of an algorithm. It is not usually possible for an attacker to create all of these possible faults in any particular implementation. Nevertheless,

it is important that algorithms are able to tolerate all types of fault, as the fault injection methods that may be realisable on a given platform are unpredictable. While an attacker might only ever have a subset of the above effects available, if that effect is not taken into account then it may have catastrophic consequences for the implementation.



The blue trace represents a normal execution. The red trace shows the same code being executed but with a fault injected that modifies the code being executed during the time that the glitch is taking place.

Figure 5.6: The effect of a glitch on an instruction, where the instruction during the glitch is affected.

These effects can be demonstrated by a supply voltage glitch. In Figure 5.5 the power consumption of a normal execution and a execution where a glitch has been applied can be seen. After the glitch is applied the chip continues to execute the correct code, but the data it is manipulating has changed. This can be seen by the difference in the power consumption at the point indicated in the image.

A different effect can be seen in Figure 5.6. The blue trace represents the power consumption of a normal execution, and the red trace represents the power consumption in the case where a glitch has been used to inject a fault in the chip. After the introduction of the glitch, the chip continues to execute its program as if nothing had happened. A successful glitch will have changed the instruction being executed, where the glitch is applied.

5.4 Combining Faults with Side Channel Attacks

The use of Simple Power Analysis, as described in Section 4.2, is an important aid in conducting fault attacks. SPA can be used to determine at what point in a command a given function has taken place. For example, Figure 4.4 shows how individual functions within a round can be identified in secret key algorithms. This allows an attacker to target small functions within a cryptographic command with a high degree of accuracy.

5.5 Summary and Conclusions

This chapter described the types of fault that could be applied by an attacker. In the following chapters only the transient faults described in Section 5.2.1 are considered. It is potentially possible to base an attack on a permanent fault (see Section 5.2.2), but achieving an implementation will be difficult. In theory, transient faults can be applied at different points during a process until an attack succeeds. This is not possible with permanent faults because of their destructive nature.

A model for the different fault effects that are possible within a smart card is proposed in Section 5.3. Some of the attacks described in this thesis will make use of this model.

Chapter 6

Generic Attacks

This chapter covers some of the generic attacks that are possible against cryptographic algorithms. The attacks are presented in the context of block ciphers, but some of these attacks are also potentially applicable to public key algorithms.

The first attack detailed in this chapter is an insecure method of key manipulation, that can leak information by Simple Power Analysis. In Section 6.2, an attack involving analysing cache hits and misses during the execution of an AES implementation to derive information on the key being used is described. Section 6.3 describes a fault attack that changes the transfer of a key from one type of memory to another, progressively changing more and more of the key so that an exhaustive key search becomes trivial. Section 6.4 details how the number rounds of an algorithm can be reduced, and a simple cryptanalysis technique that can be applied to the resulting algorithm. Some of the work in this chapter has been published in [31, 38].

6.1 Simple Power Analysis of Key Manipulation

It is shown in Section 4.2, that functions and repeating loops are visible in the power consumption of an algorithm implementation. By performing a detailed analysis of one of more power consumption traces, differences of one or more clock cycles can potentially leak secret information. For example, a compiler could implement the bitwise permutations used in DES in an insecure manner. An example of an algorithm that could be produced is given in Algorithm 6.1, where each bit in buffer

X is tested to determine whether a bit in buffer Y should be set.

Algorithm 6.1: Insecure bitwise permutation function

Input: $X = (x_0, x_1, \ldots, x_n)_2$, $P[\cdot]$ containing the indexes of the permutation
Output: $Y = (y_0, y_1, \ldots, y_n)_2$

$Y \leftarrow 0$
for $i \leftarrow 0$ **to** $n - 1$ **do**
    if $x_{P[i]} = 1$ **then** $y_i \leftarrow 1$
end
return Y

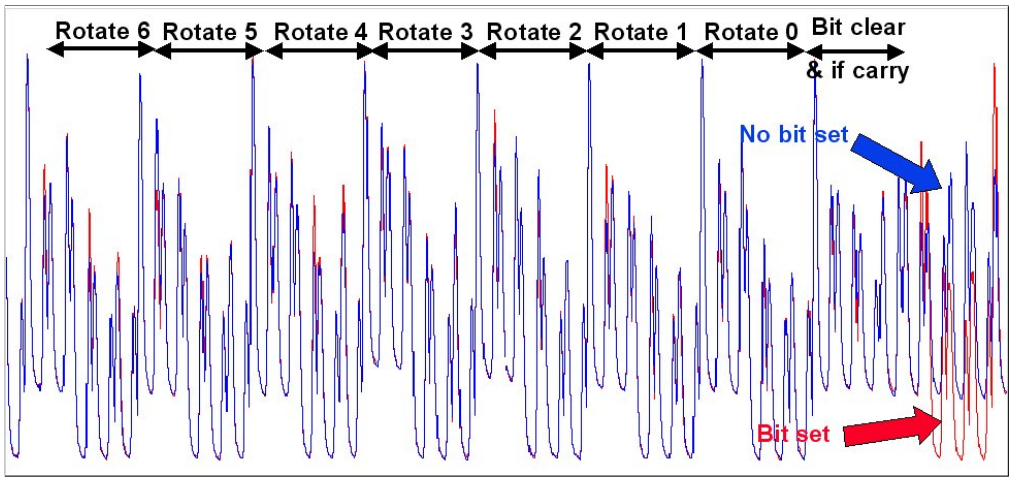


Figure 6.1: Two power consumption acquisitions showing the difference produce by a conditional bit set.

This is not a secure implementation because of the conditional test present in the algorithm. The setting of one bit in buffer Y will take a certain amount of time when the tested bit in buffer X is equal to 1. If a power consumption trace for this function is compared to a trace taken during the manipulation of a known key, for example all zeros, the point where the two traces differ will reveal the first manipulated bit that is not equal to the known key. An example of such a difference is shown in Figure 6.1, where a bit in a register is rotated into the carry and is then tested. The result of this test will determine whether a bit is set in the output buffer or not. In the case where the known key is all zeros, the power consumption will diverge where the manipulated bit of the unknown key is equal to one. In

this case, all previous bits were therefore equal to 0. The power consumption trace corresponding to the unknown key can then be shifted to take into account the time difference caused by this bit, and the process can then be repeated for the rest of the key.

This problem can be avoided by implementing a bitwise permutation, as shown in Algorithm 6.2, where each bit is taken separately for buffer X and assigned to buffer Y. This is usually more time consuming than Algorithm 6.1, as each bit needs to be taken from the relevant machine word and placed in the target machine word.

Algorithm 6.2: Secure bitwise permutation function

Input: $X = (x_0, x_1, \ldots, x_n)_2$, $P[\cdot]$ containing the indexes of the permutation
Output: $Y = (y_0, y_1, \ldots, y_n)_2$

for $i \leftarrow 0$ **to** $n - 1$ **do**
    $y_i \leftarrow x_{P[i]}$
end
return Y

6.2 Cache-Based Power Analysis

From the description of the cache mechanism given in Section 3.2.3, it can be seen that the pipeline is not stalled and normal execution occurs in the case of a cache hit. In the case of a cache miss, the pipeline flow is stalled and the NVM is accessed. When data read from memory provokes a cache miss then:

1. The instruction takes more cycles than a cache hit.

2. The power consumed by the execution is significantly higher than in the case of a cache hit, because NVM accesses will typically consume more power than normal.

With these observations, a power analysis attack based on the distinctive signatures of cache hits and cache misses can be constructed. An example of a hardware simulation of this side channel is described in [15], where a cache hit and a cache miss is simulated. To simplify the discussion below, the cache line size is set to 16

bytes for the architecture under attack, i.e. each cache miss will mean that 16 bytes are loaded into the cache. Cache line sizes may vary, which will modify the details of the attack but the principles will remain the same.

In the following sections several attacks using this side channel are described that could be applied to AES implementations, where we assume that a message block $M = (m_1, m_2, \ldots, m_{16})_{256}$ is being enciphered with a key $K = (k_1, k_2, \ldots, k_{16})_{256}$. This is followed by a brief discussion of how this type of side channel attack could be applied to other algorithms.

6.2.1 The First ByteSub Function

This attack is implemented against the AES algorithm as detailed in Section 2.2. The first step of the attack targets the **ByteSub** function of the first round. Just before entering this function the input data is XORed with the key. The resulting 16 bytes enter the **ByteSub** function, which is usually implemented as a look-up table with 256 entries.

Information on the key used can be derived from the cache access events during the table look-up, which depend on the order in which the look-up table is loaded into the cache. It is assumed that, for each acquisition, the cache has been flushed, which can easily be provoked by resetting the smart card under observation.

The attack presented in this section extends the work published in [15, 85]; the major difference here is that this attack relies purely on the observation of the side channel described in Section 6.2. The attacks described in [15, 85] also require the cache to be modified, which is not possible on a smart card. Less information is generated in the version described here, but the attack is more powerful. This is because an attacker only needs to manipulate the message blocks being enciphered, and observe the cache access pattern generated.

The first byte of the message block is set to a fixed value, m_1, and different values for the second byte of the message block, m_2, are tried until a cache hit occurs. In the case under study there are sixteen possible values that can be tried for m_2, as only the upper nibble needs to be modified to test each of the possible cache lines that could have been initialised by m_1.

When a cache hit is observed, we know that $m_1 \oplus k_1 \approx m_2 \oplus k_2$ (i.e. $m_1 \oplus k_1$ is approximately equal to $m_2 \oplus k_2$) as $m_1 \oplus k_1$ and $m_2 \oplus k_2$ both access the same cache line when the **ByteSub** function is called. In the case under study, only the high nibbles of the two expressions will be equal, i.e. $(m_1 \oplus k_1) \wedge \mathtt{F0}_{16} = (m_2 \oplus k_2) \wedge \mathtt{F0}_{16}$. This can be rearranged to place the known values on the left hand side of the equation i.e. to obtain,

$$(m_1 \oplus m_2) \wedge \mathtt{F0}_{16} = (k_1 \oplus k_2) \wedge \mathtt{F0}_{16}.$$

Thus, once a cache hit is observed, an attacker is able to calculate $(k_1 \oplus k_2) \wedge \mathtt{F0}_{16}$, i.e. the XOR of the upper nibbles of k_1 and k_2.

Once $(k_1 \oplus k_2) \wedge \mathtt{F0}_{16}$ has been found, $(k_2 \oplus k_3) \wedge \mathtt{F0}_{16}$ can be found using the same method, by choosing m_1 and m_2 so that a cache hit is always generated between the first two look-ups, and testing the 16 different values for the upper nibble of m_3. It is important to have a cache hit between the first two look-ups, as otherwise it is not known which cache line corresponds to the observed cache hit, and some information is lost. If this process is repeated for each subsequent key byte, the high nibble of each byte will be known as a function of the high nibble of the first byte. With at most 240 acquisitions the complexity of an exhaustive search to find the key of an AES implementation can be reduced from 2^{128} to 2^{68}.

In practice, this will only be true if the implementation is known. The **ByteSub** function can be implemented before or after the **ShiftRow** function, as the **ShiftRow** function is a bytewise permutation. A permutation is also sometimes used on the message block and key on entry to the algorithm to convert the array format to the grid format used in the specification. This is an optional bytewise permutation that will change addressing during the algorithm. Both permutations will change the order in which the data is treated by the **ByteSub** function.

In the following sections it will be assumed that the implementation details are known, as the added complexity arising from these permutations is negligible. The grid permutation will be ignored and the **ShiftRow** function will be assumed to take place after the **ByteSub** function.

6.2.2 Finding the Rest of the Key

The first step described in Section 6.2.1 reduces the size of the keyspace to 2^{68}, and is theoretically trivial. There are several ways to continue the attack to derive the rest of the key using the same side channel. These methods are described below.

The Second ByteSub Function with Precalculated Subkeys

The second **ByteSub** function (i.e. the **ByteSub** of the second round of AES) can be used to determine the rest of the key in a manner similar to that described in [85]. Plaintexts are chosen such that there are no cache misses in the first **ByteSub** function, except for the first table look-up. The plaintext bits that are XORed with the unknown bits of the key (i.e. the first byte and the lower nibbles of the rest of the plaintext) are randomised for each acquisition.

It is assumed that all of the subkeys are stored in NVM, i.e. the subkeys are precalculated and fetched from NVM as required, rather than being calculated *on-the-fly*, and the **ByteSub** calls in the key schedule will therefore not affect the cache. If the first look-up in the second **ByteSub** function is a cache hit, then information on the unknown key bits can be derived. In this case the following relationship is known:

$$(2 \bullet S(m_1 \oplus k_1) \oplus 3 \bullet S(m_6 \oplus k_6) \oplus S(m_{11} \oplus k_{11}) \\ \oplus S(m_{16} \oplus k_{16}) \oplus k_1 \oplus S(k_8) \oplus 1) \wedge \mathtt{F0}_{16} = (k_1 \oplus k_2) \wedge \mathtt{F0}_{16},$$

where the function S represents the look-up table used in the **ByteSub** function and $\bullet$ represents polynomial multiplication over GF(2^8). All the indices for key bytes are for the AES key, i.e. the key schedule for the second subkey is included. The first byte of the second subkey will be equal to $k_1 \oplus S(k_8) \oplus 1$. The rest of the left hand side of the equation is produced by the operation of the first round on the message block.

The value of $(k_1 \oplus k_2) \wedge \mathtt{F0}_{16}$ is known from the first part of the attack, as described in Section 6.2.1. The value of k_1 is unknown but, given k_1, the high nibbles of k_6, k_{11}, k_{16} and k_8 can be derived. This means that there are 24 unknown

bits in the equation. One in 16 of the 2^{24} possible combinations for the left hand side of the equation will be equal to $(k_1 \oplus k_2) \wedge \texttt{F0}_{16}$. One message block that produces a cache hit in the second **ByteSub** function will therefore reduce the unknown bits in the equation from 2^{24} to 2^{20}.

A second cache hit with a different plaintext can then be analysed, the correct key values will be in the intersection of the two sets of 2^{20} values produced. With 6 evaluations of the above equation all the unknown bits can be derived. This corresponds to 96 acquisitions, as the cache hit occurs with a probability of 1/16, given that the input is mostly random. This reduces the number of unknown key bits from 68 to 44. The cache misses could also be used, as they would reduce the keyspace by 15/16, but given the small amount of acquisitions required this should not be necessary.

Any acquisition with two successive cache hits can then be used to derive information on a further 5 key bytes. If the second look up in the second **ByteSub** function is also a cache hit, the following equation holds:

$$(2 \bullet S(m_2 \oplus k_2) \oplus 3 \bullet S(m_7 \oplus k_7) \oplus S(m_{12} \oplus k_{12}) \\ \oplus S(m_{13} \oplus k_{13}) \oplus k_2 \oplus k_1 \oplus S(k_8) \oplus 1) \wedge \texttt{F0}_{16} = (k_1 \oplus k_2) \wedge \texttt{F0}_{16},$$

using the same conventions as before. In this case the second byte of the second subkey is given by $k_2 \oplus k_1 \oplus S(k_8) \oplus 1$.

In this equation there are 20 unknown bits, as the values of k_1 and k_8 are provided by the evaluation of the previous step. As previously, each evaluation of this equation reduces the number of unknown values by a factor of 16. It is expected that 5 such equations need to be evaluated, taking the intersection as before, to provide one value for all of the key bytes in the equation. This event occurs with a probability of 1/256 so the acquisition phase will be lengthier than the previous step. A total of 1280 acquisitions should be required.

If all of the key bytes in the above equations are derived, the key can then be found by an exhaustive search of the remaining unknown key bits. This will be a search in a keyspace of size 2^{24} (i.e. 9 complete key bytes are given by the formulae above; for the remaining six the high nibble is known, leaving 24 unknown bits),

which can easily be performed on a PC. This is fortunate, as continuing the attack for three successive cache hits would be difficult, as the probability of seeing such an event is 1/4096, which would make the attack excessively time consuming.

The last set of equation evaluations are time consuming, which means that it can be advantageous to acquire less data and let the exhaustive search complete the key search. If, for example, an attacker takes 768 acquisitions, the expected size of the remaining exhaustive key search would be around 2^{32}. Another means of speeding up the evaluation of the possible key values for the second equation would be to use the event of a cache hit followed by a cache miss, which occurs with a probability of 15/256, each of which will reduce the size of the unknown keyspace by a factor of 15/16.

The Second ByteSub Function without Precalculated Subkeys

The attack in the previous section assumes that the subkeys used in AES are generated and then stored in NVM for later use. These keys are then accessed when they are required. In Section 2.2 the definition of AES states that the subkeys are generally generated *on-the-fly*, as the memory requirements are generally too costly to store all of the subkeys in NVM.

The generation of a subkey from the previous subkey involves four accesses to the look-up table used in the **ByteSub** function. If X is defined as the value present in memory that is used for the first table look-up in the second round, i.e.

$$X = (2 \bullet S(m_1 \oplus k_1) \oplus 3 \bullet S(m_6 \oplus k_6) \oplus S(m_{11} \oplus k_{11}) \\ \oplus S(m_{16} \oplus k_{16}) \oplus k_1 \oplus S(k_8) \oplus 1) \wedge \mathtt{F0}_{16},$$

then:

$$X = \begin{cases} (k_1 \oplus k_2) \wedge \mathtt{F0}_{16} & \text{with probability } = \frac{1}{5} \\ k_8 \wedge \mathtt{F0}_{16} & \text{with probability } = \frac{1}{5} \\ k_{12} \wedge \mathtt{F0}_{16} & \text{with probability } = \frac{1}{5} \\ k_{16} \wedge \mathtt{F0}_{16} & \text{with probability } = \frac{1}{5} \\ k_4 \wedge \mathtt{F0}_{16} & \text{with probability } = \frac{1}{5} \end{cases}$$

as the lines accessed by the key schedule will also be present in the cache. This assumes that $(k_1 \oplus k_2) \wedge \mathtt{F0}_{16}$, $k_8 \wedge \mathtt{F0}_{16}$, $k_{12} \wedge \mathtt{F0}_{16}$, $k_{16} \wedge \mathtt{F0}_{16}$, and $k_4 \wedge \mathtt{F0}_{16}$ are all

pairwise distinct, and is therefore a simplification of the actual attack. Some of the nibbles may be equal, although this can easily be seen by analysing the information gleaned in the first phase of the attack (see Section 6.2.1). This will be visible when the XOR difference is zero for the four most significant bits.

In the case where subkeys are precalculated, each evaluation of a cache hit at the beginning of the second round reduces the size of the keyspace by a factor of 16. In the above case, the size of the keyspace will be reduced by 5/16, as there are more possibilities for a cache hit. An attacker would therefore need to generate 15 cache hits with different message blocks to be sure of deriving all the 24 unknown bits. This process would take an expected number of 48 different message blocks. This is more efficient than the previous attack as, although more evaluations are required, the cache hit has a higher chance of occurring because of the key schedule.

To derive the rest of the key the attack is continued, as described in the previous section, by analysing two consecutive cache hits in the second round, i.e. where

$$X \equiv (2 \bullet S(m_2 \oplus k_2) \oplus 3 \bullet S(m_7 \oplus k_7) \oplus S(m_{12} \oplus k_{12}) \\ \oplus S(m_{13} \oplus k_{13}) \oplus k_2 \oplus k_1 \oplus S(k_8) \oplus 1) \wedge \mathtt{F0}_{16}$$

and X is distributed as before. This event occurs with probability 25/256. In this case, there are 20 unknown bits, and it will require 12 evaluations of cache hits to derive all the unknown bits. This should require a further 123 acquisitions, giving a total of 171 acquisitions for the second phase.

Again this leaves a key search of 2^{24} hypotheses that can easily be exhausted on a PC. Thus attacking a smart card that generates its subkeys *on-the-fly* is much more efficient than if they are all precalculated.

The xtime Function

Another method that could be used to reduce the key search space is to focus on the **xtime** function. The **xtime** function is a polynomial multiplication by two over GF(2^8), and is used in the **MixColumn** function. The **xtime** function is typically implemented as a bit shift followed by a conditional XOR (as detailed in Section 2.2). This is difficult to implement securely in smart cards, as there is a danger that the

result of the conditional test will be visible in the power consumption, since the two branches will take different amounts of time to complete. Even if this is implemented so that the calculation always takes the same amount of time, there is still a risk of a partitioning attack [92] (a brief description of this attack is given in Section 7.1).

In smart cards, a possible replacement for this function is a look-up table of 256 bytes that will avoid the need for any conditional testing. This protects the implementation against Simple Power Analysis, but the table will be in Non-Volatile Memory, so will be accessed via a cache, as with the look-up table used in the **ByteSub** function. The pattern of cache hits and misses can therefore be analysed in a similar way to the first phase of the attack described in Section 6.2.1. The first look-up to the **xtime** table will be a cache miss. If this is followed by a cache hit then:

$$S(m_1 \oplus k_1) \wedge \mathrm{F0}_{16} = S(m_6 \oplus k_6) \wedge \mathrm{F0}_{16}$$

where, as previously, the S represents the look-up table in the **ByteSub** function. The right hand side of the equation uses $m_6 \oplus k_6$ rather than $m_4 \oplus k_4$ because of the **ShiftRow** function. In this equation there are 2^{12} possible combinations of values for the variables, given that the high nibble of k_6 is known as a function of the high nibble of k_1, from the first part of the attack described in Section 6.2.1. Searching through all the combinations will give 2^8 possible values for the pair (k_1, k_6). Because S is non-linear, another cache hit can be found with a different message block which will provide a different set of 2^8 values. The correct key will be in the intersection between the two sets of possible values. After three cache hits with three different message blocks have been found, there should only be one hypothesis for both k_1 and k_6. Each cache hit will occur with a probability of 1/16, so 48 acquisitions should be sufficient to find the values of k_1 and k_6.

The next cache access is the first **xtime** function call for the next output byte. The values for m_1 and m_6 can be fixed so that a cache hit is always generated between the first two **xtime** look-ups. If a cache hit occurs for the next **xtime** look-up then:

$$S(m_6 \oplus k_6) \wedge \mathrm{F0}_{16} = S(m_2 \oplus k_2) \wedge \mathrm{F0}_{16}$$

In this case, k_6 is known and the high nibble of k_2 is known. This is because k_1 has already been determined, and therefore the high nibble of all the key bytes are known. The four unknown bits of k_2 in the equation can be exhausted for the value of m_2 that provokes a cache hit. One cache hit of this nature would be enough to determine the four unknown bits. This process can be continued using the following equations:

$$S(m_2 \oplus k_2) \wedge \mathrm{F0}_{16} = S(m_7 \oplus k_7) \wedge \mathrm{F0}_{16}$$
$$S(m_7 \oplus k_7) \wedge \mathrm{F0}_{16} = S(m_3 \oplus k_3) \wedge \mathrm{F0}_{16}$$
$$S(m_3 \oplus k_3) \wedge \mathrm{F0}_{16} = S(m_8 \oplus k_8) \wedge \mathrm{F0}_{16}$$
$$S(m_8 \oplus k_8) \wedge \mathrm{F0}_{16} = S(m_4 \oplus k_4) \wedge \mathrm{F0}_{16}$$
$$S(m_4 \oplus k_4) \wedge \mathrm{F0}_{16} = S(m_5 \oplus k_5) \wedge \mathrm{F0}_{16}$$

This process can determine the first 8 bytes of the key with a total of 192 acquisitions, leaving an exhaustive search amongst 2^{32} possible keys. An exhaustive search of size 2^{32} is somewhat time consuming, so further analysis would be advantageous. The next four cache accesses can be analysed requiring a further 64 acquisitions (for a total of 256 acquisitions), giving the following equations.

$$s(p_6 \oplus k_6) \wedge \mathrm{F0}_{16} = s(p_{11} \oplus k_{11}) \wedge \mathrm{F0}_{16}$$
$$s(p_7 \oplus k_7) \wedge \mathrm{F0}_{16} = s(p_{12} \oplus k_{12}) \wedge \mathrm{F0}_{16}$$
$$s(p_8 \oplus k_8) \wedge \mathrm{F0}_{16} = s(p_9 \oplus k_9) \wedge \mathrm{F0}_{16}$$
$$s(p_5 \oplus k_5) \wedge \mathrm{F0}_{16} = s(p_{10} \oplus k_{10}) \wedge \mathrm{F0}_{16}$$

There will be no need to compare $s(p_{11} \oplus k_{11})$ with $s(p_7 \oplus k_7)$, as if a cache hit is generated between $s(p_6 \oplus k_6)$ and $s(p_{11} \oplus k_{11})$ a cache hit will also be generated with $s(p_7 \oplus k_7)$, due to the selected message. This reduces the number of unknown key bits to 16. An exhaustive search amongst 2^{16} keys is trivial, so no further acquisitions are required to derive the key.

6.2.3 Other Algorithms

The above attack technique has been presented exclusively against AES. Previous work of this type has included attacks on DES [87, 104]. More generally, this type of attack can be applied to numerous algorithms that access Non-Volatile Memories via a cache.

6.3 Fault Attacks on Key Transfer or NVM

In this scenario [18], a fault is injected during the transfer of secret data from one type of memory to another. The attack is applicable to any algorithm; the example given here will be for a DES key being transferred from NVM to RAM. If parts of the key can be forced to some fixed value (for example, one byte at a time) it becomes possible to discover the key.

Suppose that a message block M is encrypted with DES to obtain a faultless ciphertext block C_0. This process is repeated and, during the key transfer from NVM to RAM, one key byte is changed to a fixed known value (00 in this example). The resulting ciphertext block, C_1, is recorded and the process is repeated by forcing two bytes to a fixed value, then three bytes, and so on. This continues until the whole key apart from one byte has been set, byte-by-byte, to the fixed value.

This procedure is shown in Table 6.1, where C_i represents the ciphertext block of an unknown key with i bytes set to a fixed value. Once this data has been collected it can be used to derive the DES key.

Table 6.1: The Biham–Shamir Attack

Input	DES Key	Output
$M \rightarrow$	$K_0 =$ XX XX XX XX XX XX XX XX	$\rightarrow C_0$
$M \rightarrow$	$K_1 =$ XX XX XX XX XX XX XX 00	$\rightarrow C_1$
$M \rightarrow$	$K_2 =$ XX XX XX XX XX XX 00 00	$\rightarrow C_2$
$M \rightarrow$	$K_3 =$ XX XX XX XX XX 00 00 00	$\rightarrow C_3$
$M \rightarrow$	$K_4 =$ XX XX XX XX 00 00 00 00	$\rightarrow C_4$
$M \rightarrow$	$K_5 =$ XX XX XX 00 00 00 00 00	$\rightarrow C_5$
$M \rightarrow$	$K_6 =$ XX XX 00 00 00 00 00 00	$\rightarrow C_6$
$M \rightarrow$	$K_7 =$ XX 00 00 00 00 00 00 00	$\rightarrow C_7$

Let K_n represent the original DES key with n bytes replaced with known values.

To find K_7 the 128 different possible values for the first byte of the DES key are tried until one produces the ciphertext block C_7*. After this K_6 can be found by searching through the 128 different possible values for the second byte, as the first byte will be known. Finding the entire key will require searching through a total of $128 \times 8 = 1024$ different keys. This attack can also be used when known data is manipulated by an unknown algorithm. The pre-requisite for such an attack is the ability to re-key the device (running the unknown algorithm) with arbitrary keys. In this case the exhaustive search phase can be performed using the attacked device itself. An example of an implementation of this attack is given in Appendix C.

6.4 Round Reduction Using Faults

Block ciphers, such as DES and AES, are based on a function that is computed iteratively as a series of rounds to provide a high level of security. This function is referred to as the round function. In [6] the idea was put forward to implement a fault attack that would effectively reduce the number of rounds of a block cipher to enable the key to be derived.

The location where a fault can be injected to reduce the number of rounds to one can potentially be determined by observing the power consumption, as described in Section 4.2. If a card is available for which it is possible to change the key, a variety of faults can be induced. The expected output after one round can then be searched for within the data acquired, i.e. the outputs can be checked for the ciphertext block produced by one round of an algorithm. This will indicate the case when the algorithm has been reduced to one round.

If the key cannot be changed, the I/O can be acquired each time a glitch is applied. This can be used to signal when an attack has been successful, as the time between the command and the response will shorten when a large section of code has been bypassed. An example of such an approach is shown in Figure 6.2, where the shortening of the command can be seen in the I/O and is confirmed by the power consumption. This shortening of the command time is only significant when

*Although one byte is unknown, only 128 different values are possible as the least significant bit is a parity bit.

the status returned by a smart card implies that everything has executed correctly and sixteen bytes have been returned. Otherwise, it could be confused with a reset provoked by an overly aggressive attack.

Once the means has been found of launching such a round-reducing attack has been found, the implementation can be attacked several times with different message blocks to acquire the data needed to derive the AES key being used in the smart card under attack. An implementation of AES that has been reduced to one round will consist of the functions described in Algorithm 6.3.

Algorithm 6.3: One Round of the Advanced Encryption Standard

Input: M, K
Output: C

$X \leftarrow M \oplus K$
$X \leftarrow \mathbf{ShiftRow}(X)$
$X \leftarrow \mathbf{ByteSub}(X)$
$X \leftarrow \mathbf{MixColumn}(X)$
$K \leftarrow \mathbf{KeySchedule}(K)$
$X \leftarrow X \oplus K$
$C \leftarrow X$

return C

These functions are described in Section 2.2. The **MixColumn** function is not necessarily included, as the test that determines the end of the loop calling the round function can be placed before or after this function. An assembly implementation could have a jump at the end of the loop, with the round counter being tested before the **MixColumn** function is called. In a compiled programming language, C for example, this test would generally appear after the **MixColumn** as a programmer would use a **for** or a **do-while** loop.

If we present two message blocks (M_1 and M_2) to this algorithm to produce two corrupt ciphertext blocks (C_1 and C_2 respectively), then the data acquired can be compared in the following fashion:

$$\mathbf{ByteSub}\,(M_1 \oplus K) \oplus \mathbf{ByteSub}\,(M_2 \oplus K) = \mathbf{MixColumn}^{-1}\,(C_1 \oplus C_2)$$

where K is the first subkey. The **ShiftRow** function is not taken into account as it

The acquisitions from top to bottom: The I/O trace of a normal AES execution and its power consumption, where the rounds can be seen. This is followed by the I/O trace of an AES execution where the number of rounds have been reduced to one and the power consumption shows the reduction in the number of rounds.

Figure 6.2: The effect of a successful round-reducing fault on the I/O and power consumption.

is a bytewise permutation. The last XOR is ignored as the effect of this function is removed by XORing the two corrupt ciphertext blocks together.

The right hand side of the equation equals the XOR of the AES calculations after the **ByteSub** function. As M_1 and M_2 are known, this equation can be evaluated for all the possible values of the first subkey in a bytewise fashion.

With two ciphertext blocks this will usually lead to two different hypotheses for each byte of the first subkey. There is no calculation involved in the generation of the first subkey, so these hypotheses apply directly to the key. This can be seen by calculating a table of differentials, as used in [16] for DES. This leads to an exhaustive search amongst 2^{16} possible keys, as:

$$\left(\frac{\Sigma(\text{Non zero differentials})}{\#(\text{Non zero differentials})}\right)^{16} = 2^{16}$$

If three ciphertext blocks are available, then the number of keys that are included in an exhaustive search is much reduced. This is because three different comparisons can be made using the formula described above. Each one provides approximately two different hypotheses for each byte of the key.

In practice, the data acquired is likely to be noisy, as there will be some faults that will produce a corrupt ciphertext block and change the I/O but will not have the desired properties. For this reason it is best to compare each ciphertext block that comes from a command that returned 16 bytes that are not equal to the correct ciphertext block with all the others that fulfil the same criteria. This optimises the attack, as faulty ciphertext blocks without the desired properties can be managed.

This procedure does not slow down the attack because if one, or both, ciphertext blocks do not have the required properties then the calculated Hamming distance will be meaningless, as deriving a list of hypotheses will not be possible. The probability that random data will produce at least one hypothesis for every byte of the key is:

$$\left(\frac{\#(\text{Non zero differentials})}{256^2}\right)^{16} = 3.14 \times 10^{-3}$$

Even with a large amount of acquired data the time needed to search for the

key remains reasonable. A description of an implementation of this attack is given in Appendix D.

This attack can be directly applied to other block ciphers. The main difference is in the manner in which the data acquired is exploited. In the case of the DES, hypotheses on the key can be derived by inspection, because of the structure of the Feistel network. This will give a keyspace of 2^{24} to be searched as a result of one corrupt ciphertext block. The size of the search can be further reduced by examining other corrupt ciphertext blocks to derive the first subkey. Reducing subsequent DES executions to two rounds can then give direct information on the eight bits not present in the first subkey. As stated in [6], this can also provide a method of attacking the DES without knowledge of the message blocks being enciphered.

An equivalent attack can be implemented if the number of rounds is reduced by one. It will then be possible to derive hypotheses on the last subkey rather than the first. This is because it will be possible to compare the effect of the last round of the calculation by comparing the actual ciphertext block with a ciphertext block that was calculated with the same algorithm but with one less round.

6.5 Summary and Conclusions

This chapter provides a description of some of the possible generic attacks. These attacks include manipulating secret data (e.g. a key) in a non-deterministic manner, observing cache access patterns, progressively setting parts of a key to a fixed value, and reducing the number of rounds of an algorithm. The attack described in Section 6.1 is well-known within the smart card industry. The attack described in Section 6.2 is novel, and is part of the contribution of this thesis. The attacks presented in Sections 6.3 and 6.4 have been proposed previously in the literature, and implementations of these attacks are described in Appendices C and D, and are presented as part of this thesis.

Chapter 7

Generic Countermeasures

A variety of countermeasures can be deployed to defend against the attacks already detailed in this thesis. In this chapter we consider generic countermeasures, i.e. those that are applied to implementations of cryptographic algorithms to prevent side channel and fault attacks, but do not necessarily provide protection against attacks that are specific to one algorithm. Details for protecting individual algorithms are given in subsequent chapters.

Section 7.1 details basic countermeasures that can be applied to stop trivial timing and Simple Power Analysis attacks. A description of the use of random delays to hinder an attacker is given in Section 7.2. This description includes a novel optimisation of the use of such delays, to improve the performance and security of systems using random delays. Section 7.3 explains the use of execution randomisation to further inhibit an attacker, a measure that complements the random delays already defined. Section 7.4 describes the use of data whitening to mask the data being manipulated, to prevent Statistical Power Analysis. Section 7.5 details some of the fault countermeasures that can be used to prevent faults from changing the computation of an algorithm. Some of the work described in this chapter has previously been published in [10, 11, 12].

7.1 The Basic Principles

There are two basic principles that should be followed when implementing cryptographic algorithms on embedded platforms to avoid the simplest attacks. These

are:

Constant Time: The various forms of timing analysis, described in Section 4.1, can be avoided by making sure that commands are always executed in the same number of clock cycles. Any variation in the length of the time taken to execute a command should not depend on secret information.

Constant Execution: As described above, the time taken by an algorithm should remain constant, so that no deductions regarding secret information can be made. This extends to individual processes being executed by a smart card. If a process takes different lengths of time depending on some secret information, and the difference in time is made up by a dummy function, then there is a good chance that this will be visible in the power consumption (as detailed in Section 4.2). It is therefore important that an algorithm is written so that the same code is executed for all the possible entry values. Even small differences can produce attacks that completely compromise the security of an algorithm. For example, the Partitioning attack [92] detects information from the carry after the addition of two 8-bit variables in the Comp128 algorithm. In the case of cache accesses, as described in Section 3.2.3, constant execution can be achieved by using commands that avoid using the cache [71].

7.2 Desynchronisation Techniques

The use of random delays in embedded software is often proposed as a generic countermeasure against side channel analysis, such as Simple Power Analysis (SPA) and Statistical Power Analysis. Statistical analysis is meaningless in the presence of desynchronisation; an attacker must resynchronise acquisitions at the area of interest before being able to interpret what is happening at a given point in time.

7.2.1 Desynchronisation in Software

A specific case of this countermeasure is considered in [34], where a CPU inserts delays of 1 clock cycle at random points in time. An attack against these small

random delays can be conducted, based on evaluating the integral of a window of acquisitions rather than in a pointwise fashion (as described in Section 4.3).

This attack is not applicable in the case of software random delays, as the information is spread over too many clock cycles. In general, a software random delay will consist of a dummy loop where a random value is generated and then decremented until the random value reaches zero before executing any further code. An example of code that could be used in the 8051 assembly language is as follows:

```
   mov   a, RND
   mov   r0, a
Delay_Label:
   djnz  r0, Delay_Label
```

This adds very little extracode to the overall program as this moves the random value held in the register `RND` to another register (via the accumulator), and then decrements this value and loops until the accumulator contains zero. Generally, random number generators in embedded devices employ a noisy resistor or a clock generator with a bad ring*, and are tested using a suite of tests (see e.g. [58]). The value placed into the accumulator can therefore be considered to be uniformly distributed across all the admissible values. In the case of the 8051 assembly language, designed for an 8-bit architecture, this would be expected to be an integer from the interval $[0, 255]$.

These loops are inserted to prevent Statistical Power Analysis and fault attacks. An example of power consumption acquisitions that include a random delay are shown in Figure 7.1. All three acquisitions are synchronous at the left hand side of the figure. This is followed by a random delay, visible as a result of the repeating pattern of the loop, after which the acquisitions are no longer synchronous.

To conduct a Statistical Power Analysis, an attacker needs to exploit the relationship between the data being manipulated at a given point in an algorithm and the power consumption. In order to do this, the acquisitions need to be synchronised

*A clock generator with a bad ring will have non-deterministic transitions between zero and one. This is a hinderance for circuit design, but can be used to generate random values.

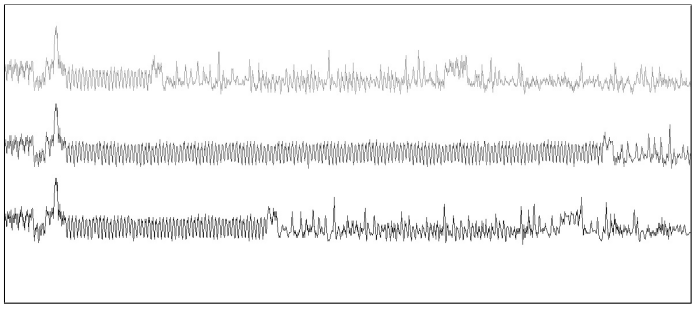


Figure 7.1: Random delays visible in the power consumption over time.

a posteriori at the point that an attacker wishes to analyse. As the size of the random delay increases, an attacker is obliged to acquire more information to be sure of acquiring the point of interest. The amount of work required to resynchronise the acquisitions also increases, as pattern-matching software will have to search in a larger window to find the same point in each acquisition.

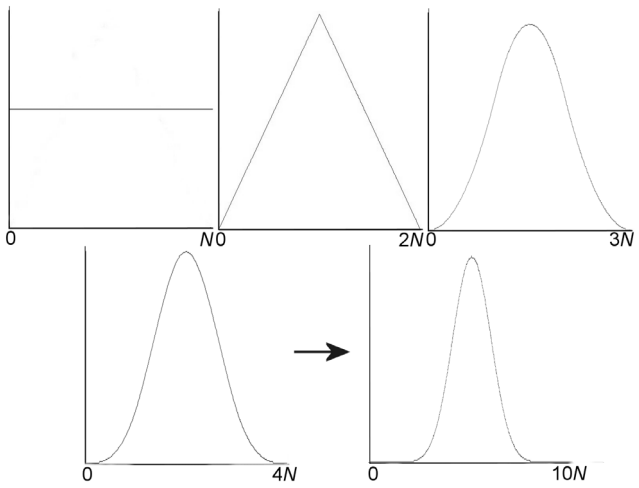
An attacker wishing to conduct a fault attack needs to synchronise the fault injection with the event that they wish to affect in real time. There are certain features that can be synchronised with automatically, such as I/O events or an EEPROM write command, but this will only remove a certain amount of the desynchronisation effect. Because of the ease of synchronising with events of this type it is considered prudent to include a random delay after such events, with the delay length taken from a comparatively large interval. This measure is designed to hinder an attacker that can synchronise in real time. In the presence of such measures, an attacker is obliged to inject a fault where the event is likely to occur, and then repeatedly inject the fault until the event coincides with the targeted point. This renders an attack more complex, as an attacker cannot be sure where the fault has been injected, and a means of detecting that the desired fault has occurred needs to be implemented (e.g. analysing the power consumption *a posteriori*). A notable exception to this is the first published fault attack, that attacks the RSA when it is calculated using the Chinese Remainder Theorem [23] (see Section 9.3). In this case, the target is so large that the use of any form of random delay is unlikely to hinder the attack.

In general, a smart card operating system will typically incorporate a few large

random delays placed at strategic points to prevent fault injection attacks. Secret key cryptographic algorithms will typically have smaller random delays placed between every subfunction of every round function (the use of random delays in public key algorithms is somewhat similar). More random delays are typically included in cryptographic algorithms to hinder attempts to conduct Statistical Power Analysis. In the presence of such delays, if an attacker synchronises a set of power acquisitions for a given function, then the next function will still be desynchronised, and will require further work to synchronise.

The use of random delays is not a particularly robust countermeasure, as it cannot prevent an attack from taking place. However, it can render an attack so time consuming that an attack is no longer practical.

As random delays are rarely used in just one place, an attacker is likely to be dealing with the cumulative delay of several random delays. The cumulative delay is therefore distributed with a cumulative uniform distribution. This distribution is shown in Figure 7.2 for some small numbers of random delays, where the y-axis is normalised to show how the form of the distribution changes.



The cumulative random delay generated by a sequence of random delays generated from uniformly distributed random variables. The number of random delays considered are 1, 2, 3, 4, and 10, from top left to bottom right.

Figure 7.2: The cumulative random delay.

As can be seen, as the number of random delays increases, the distribution rapidly becomes binomial in nature, i.e. it approximates to a discrete normal distribution. It is also interesting to note that after 10 such random delays there is a tail on either side of the binomial, where the probability of a delay of this length occurring is negligible.

Modifying the Distribution of Software Random Delays

A modification to the distribution of individual random delays can be incorporated to increase the standard deviation and reduce the mean of the cumulative distribution. To use the modified distribution in a constrained environment, for example a smart card, the delays will need to be expressed as a table, where the number of entries for a given value represents the probability of that value being chosen. A uniformly distributed random number can then be used to select an entry from this table.

A function (chosen arbitrarily, see Appendix E for more details) that can be used to govern the number of entries for each value of x, where x takes integer values in the interval $[0, N]$, is:

$$y = \lceil ak^x + bk^{N-x} \rceil$$

where a and b represent the values of the probability function when x takes 0 and N respectively, and y governs the number of entries in a table implementing the function for each value of x. The sum of y for all values of x will therefore give the total number of table entries. The value of k governs the slope of the curve and can take values in the interval $(0, 1)$. Different values for a and b are used so that a bias can be introduced into the sum distribution to lower the mean delay. The two elements ak^x and bk^{N-x} are both close to zero when x is approximately equal to $N/2$ for the majority of values of k that will be of interest. The ceiling function is therefore used to provide a minimum number of entries in the table for each value. This is important, as if values are removed from the distribution it will decrease the number of values that the random delay can take, and therefore reduce the desired randomising effect.

To provide a table that can be efficiently implemented, the number of entries in the table should be a power of two. A random number generator can be used to provide a random word, and the relevant number of bits can then be masked off and used to read the value at the corresponding index of the table to dictate the length of each random delay. If the number of entries is not equal to a power of two, any values generated between the number of entries and the next power of two will have to be discarded. This testing of values will slow down the process, and have a potentially undesirable effect on the distribution of the random delays, as suitable random numbers will only be generated with a certain probability.

Parameters that were found to naturally generate a table of 2^9 entries are shown in Table 7.1. It would also be possible to choose some parameters and then modify the table so that it has 2^9 entries, but this seemed overly complicated, as the desired number of entries can be made to occur naturally. The percentage changes shown are in comparison to the mean and standard deviation of a uniformly distributed random delay. The change in the mean and the standard deviation is not dependent on the number of random delays that are added together, i.e. it is not dependent on the number of random delays that occur.

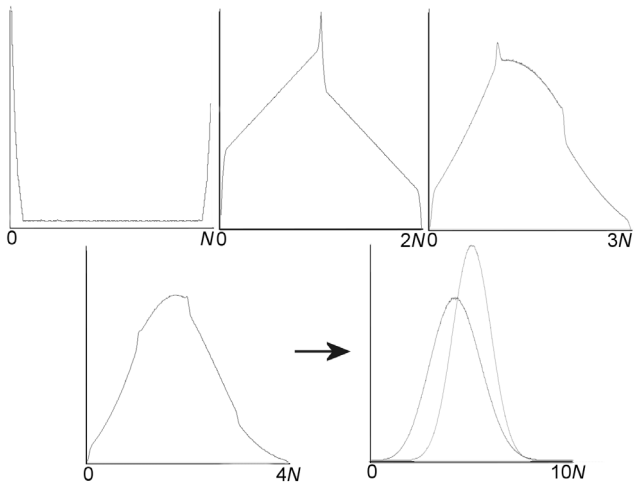
Table 7.1: Parameter characteristics for tables of 2^9 entries

a	b	k	Mean	σ
			% decrease	% increase
22	13	0.88	13.4	34.5
23	12	0.88	16.3	33.5
23	15	0.87	11.6	35.4
24	11	0.88	19.7	32.1
24	14	0.87	13.4	34.9
25	10	0.88	22.6	30.7
26	6	0.89	32.8	23.5
26	9	0.88	25.6	29.0
26	12	0.87	19.7	32.5
32	8	0.86	31.4	25.8

As can be seen, a large change is achieved in the mean and standard deviation. It can also be seen that we cannot achieve the largest standard deviation increase

and mean decrease at the same time. Implementers will have to make a compromise between maximising the standard deviation and minimising the mean.

An example of the effect of using this sort of method to govern the length of random delays on the cumulative distribution is shown in Figure 7.3, where $a = 26$, $b = 12$ and $k = 0.87$. The y-axis is normalised to show the change in the cumulative distribution. The last graph shows the cumulative distribution for 10 random delays taken from the modified distribution plotted alongside the cumulative distribution for 10 random delays generated from uniformly distributed random delays. The differences in the mean and standard deviation can clearly be seen, and the tail present on the left hand side is much smaller for the cumulative distribution of random delays from the modified distribution. Further details of how these parameters were derived are given in Appendix E.



The cumulative random delay generated by a sequence of random delays generated from a biased distribution. The number of random delays considered are 1, 2, 3, 4, and 10, from top left to bottom right. The last graph also shows the distribution of 10 uniformly distributed random delays for comparative purposes.

Figure 7.3: The cumulative random delay using a modified distribution.

7.2.2 Desynchronisation in Hardware

The most effective form of desynchronisation is the use of an unstable internal clock. This changes the acquired power consumption to that shown in Figure 7.4. The two

acquisitions start with the external clock and are therefore synchronous, and after the internal clock is started the two waveforms diverge. This makes it difficult to even achieve a local synchronisation for power attacks, and will also create problems for an attacker wishing to inject a fault.

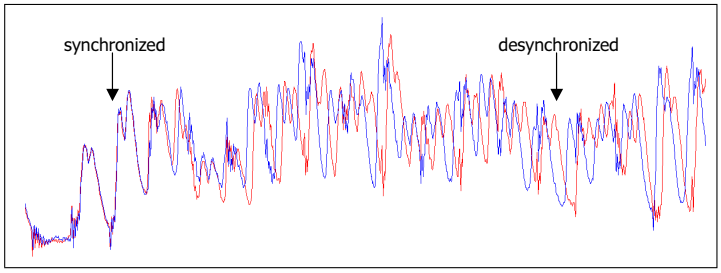


Figure 7.4: Unstable internal clock generation, as visible in the power consumption.

7.3 Execution Randomisation

If the order in which operations in an algorithm are executed is randomised, it becomes difficult to predict what the machine is doing at any given clock cycle. An example of this is given in Algorithm 7.1 for the process of copying 256 bytes from buffer A to buffer B. For most fault attacks this countermeasure will only slow down a determined adversary, as eventually a fault will hit the desired instruction. However, this will thwart attacks that require faults in specific places or in a specific order. In the case of Statistical Power Attacks, it spreads the desired information across a number of clock cycles. This is one of the most effective countermeasures against Statistical Power Analysis, especially when combined with data whitening (see Section 7.4).

7.4 Data Whitening

The most commonly discussed defence against statistical power analysis, described in Section 4.3, is to use data whitening, where the message block being manipulated is masked with a random value. The data is then manipulated in such a way that the value present in memory is always masked with the same random value. This mask

Algorithm 7.1: Randomised Data Transfer

Input: $A = (a_0, a_1, a_2, \ldots, a_{255})_{256}$, four random bytes $\{x, y, z, w\}$ where x is odd.

Output: $B = (b_0, b_1, b_2, \ldots, b_{255})_{256}$ containing A.

for $i \leftarrow 0$ **to** 255 **do**
    $j \leftarrow (x \times (i \oplus w) + y \bmod 256) \oplus z$
    $b_i \leftarrow a_j$
end

return B

is then removed at the end of the algorithm to produce the ciphertext block. The most common form of masking is Boolean masking, where all data manipulated is XORed with a random value. A function will take the masked value and produce a result that is masked with the same random value. This countermeasure was initially proposed in [28], and an example of this sort of implementation is described in [2].

The size of the random value is generally limited, as look-up tables need to be randomised before the execution of the algorithm. This is so that the input and output values of the S-box leak no information. An example of how this can be achieved is given in Algorithm 7.2. As described here, the random value used for masking the input data can be no larger than n, and the random value used for the output value can be no larger that x. As the same S-box is used numerous times, this affects the size of the random mask that can be used in the algorithm.

Algorithm 7.2: Randomising S-Box Values

Input: $S = (s_0, s_1, s_2, \ldots, s_n)_x$ containing the S-box, $\mathbf{R}$ a random value $\in [0, n]$, and r a random value $\in [0, x)$.

Output: $RS = (rs_0, rs_1, rs_2, \ldots, rs_n)_x$ containing the randomised s-box.

for $i \leftarrow 0$ **to** n **do**
    $rs_{(i \oplus \mathbf{R})} \leftarrow s_i \oplus r$
end

return RS

In the case of AES, both $\mathbf{R}$ and r will be one byte, which means that the random mask used during the calculation is likely to be one byte long. There are theoretical attacks against this protection method [68] called higher order Differential Power Analysis, but these are not feasible when this countermeasure is implemented with

the random effects described in Sections 7.2 and 7.3. An implementation of second order DPA under controlled conditions is described in [86].

This measure does not directly apply to public key cryptographic algorithms such as RSA, but the principal can be applied to modular exponentiation. This is covered in Chapter 9.

7.5 Integrity Checks

Various types of countermeasure can be implemented to defend against fault attacks. These are usually based on existing techniques used for integrity purposes. It would normally be expected that anomaly sensors are implemented in a smart card that would detect an attempted fault attack, but (as described in Section 3.2.4) this cannot always be relied upon. Some of the software countermeasures that can be used in embedded software to prevent fault are listed below.

Checksums — These can be used to verify the integrity of data at all times. This is especially important when data is being moved from one memory area to another. For example, checksums can be used to prevent the key transfer attack described in Section 6.3.

Variable redundancy — This refers to the reproduction of a variable in memory, and functions are then performed on each copy of the variable independently. The result will be known to be correct if the two answers are identical. This can be used for round counters to prevent attacks, such as the round reduction attack given in Section 6.4.

Execution redundancy — The same function, part of the function, or its inverse can be executed after a given function. The results of the two executions are then compared to verify that no fault has occurred.

Ratification counters and baits — This involves including small functions in sensitive code that perform a calculation and then verify the result. When an incorrect result is detected, a fault is known to have been injected. A

possible reaction to this would be to decrement a counter, and when this counter reaches zero the smart card would then cease to function.

7.6 Summary and Conclusions

This chapter provides a description of some of the generic countermeasures that are commonly used in embedded implementations of cryptographic algorithms. All of the countermeasures described above are well-known within the smart card industry. However, the modification to the distribution that governs the lengths of random delays (see Section 7.2) is novel, and is part of the contribution of this thesis. Further details of this modification are given in Appendix E.

Chapter 8

Block Cipher Implementations

In most widely implemented block ciphers, a "simple" function is used as a round function and is repeated a set number of times to encrypt a block of data. Typically, a round function will involve permutations, arithmetic operations, substitution functions (S-boxes), and XOR operations. This chapter describes some of the attacks that are particular to block ciphers, and the countermeasures that are required to prevent the realisation of these attacks, using DES and AES as examples.

Section 8.1 details how power analysis can be applied to block ciphers. Differential Fault Analysis (DFA) is discussed in terms of DES in Section 8.2, with a short discussion of how this technique can be applied to other algorithms. Collision Fault Analysis is described in Section 8.3 in terms of DES, and versions of this attack are described that can be applied to DPA-resistant algorithms are given in Section 8.3.2. The modification of S-box elements is then discussed separately in Section 8.4, as this draws upon both Differential and Collision Fault Analysis. Possible countermeasures for implementations of a block cipher are listed in Section 8.5. Some of the work in this chapter has been published in [4].

Differential Fault Analysis is well-known within the smart card industry and descriptions of implementations have been published. The method behind exploiting faults is presented in detail, as little information is presented in the literature [18, 42]. The extension to triple DES is novel, and is part of the contribution of this thesis. Collision Fault Analysis is more recent, although the analysis is somewhat similar to Differential Fault Analysis. The application of DFA and CFA to DPA-resistant

implementations of DES and AES, i.e. that use data whitening and execution randomisation (see Chapter 7), is described.

Implementations of four of the attacks described in this chapter are discussed in Appendices G, H, and I, and are also part of the contribution of this thesis.

8.1 Power Analysis of Block Ciphers

Simple Power Analysis is not typically a problem for implementations of block ciphers, as it is straightforward to implement block ciphers using the principles given in Section 7.1. However, if these principles are not applied, then attacks become possible, e.g. the key manipulation attack given in Section 6.1.

Statistical power analysis, as described in Section 4.3, can be used to break a block cipher. An attacker forms a hypothesis about the result of a given function, e.g. an S-box, where the following computation is performed:

$$I = S(M \oplus K)$$

i.e. a part of the message block M is XORed with part of the key K, and then passed through an S-box (the function S) to produce an intermediate state I. Normally, this value is not visible to an attacker, but can be determined using DPA and related attacks.

An attacker calculates a DPA trace for each possible value for K. If a peak occurs in the DPA trace, then this will validate the hypothesis for K, allowing part of the key to be deduced. This would appear to be analogous to an exhaustive search, but the output of one S-box in DES, for example, depends on six bits of the key. An attacker therefore only needs to test 64 hypotheses for these six bits to determine their value. The process can then be repeated for each S-box to derive the entire key. This was discussed in more detail in Section 4.3.

8.2 Differential Fault Analysis

The first differential fault attacks to be made public were against DES, and were published in [18]; one-bit faults were assumed to occur in random places throughout

an execution of DES. The ciphertext blocks corresponding to faults occurring in the fourteenth and fifteenth round were taken, enabling the derivation of the key through differential cryptanalysis. This was possible as the effect of a one-bit fault in the last three rounds of DES is visible in the ciphertext block when it is compared with a correct ciphertext block. This allowed the key to be recovered using between 50 and 200 different ciphertext blocks. It is claimed in [18] that, if an attacker can be sure of injecting faults towards the end of the algorithm, the same results could be achieved with only ten faulty ciphertext blocks, and that, if a precise fault could be induced, only three faulty ciphertext blocks would be required.

This algorithm was improved upon in [42], using ideas given in [17]. When searching for a key, the number of times a given hypothesis is found is counted. This means that faults from earlier rounds can be taken into account. It is claimed in [42] that faults from the eleventh round onwards can be used to derive information on the key, and that in the ideal situation only two faulty ciphertext blocks are required.

The mechanisms that allow the secret key to be derived from faulty/correct ciphertext block pairs are described in detail, since the literature does not provide detail on how this attack functions [18, 42] beyond noting that the differentials produced can be exploited using differential cryptanalysis.

8.2.1 The Fifteenth Round

The simplest case of a fault attack on DES involves injecting a fault in the fifteenth round, and such an attack is well-known within the smart card industry. The last round of DES, as described in Section 2.1, can be expressed in the following manner:

$$\begin{aligned} R_{16} &= S(R_{15} \oplus K_{16}) \oplus L_{15} \\ &= S(L_{16} \oplus K_{16}) \oplus L_{15} \end{aligned}$$

If a fault occurs during the execution of the fifteenth round, i.e. R_{15} is randomised by a fault to become R'_{15}, then:

$$\begin{aligned} R'_{16} &= S(R'_{15} \oplus K_{16}) \oplus L_{15} \\ &= S(L'_{16} \oplus K_{16}) \oplus L_{15} \end{aligned}$$

and

$$R_{16} \oplus R'_{16} = S(L_{16} \oplus K_{16}) \oplus L_{15} \oplus S(L'_{16} \oplus K_{16}) \oplus L_{15}$$
$$= S(L_{16} \oplus K_{16}) \oplus S(L'_{16} \oplus K_{16})$$

This provides an equation in which only the last subkey, K_{16}, is unknown. All of the other variables are available from the ciphertext block. This equation holds for each S-box in the last round, which means that it is possible to search for key hypotheses in sets of six bits. All 64 possible key values corresponding to the XOR just before each S-box can be used to generate a list of possible key values for these key bits. After this, all the possible combinations of the hypotheses can be searched though, with the extra eight key bits that are not included in the last subkey, to find the entire key.

If R'_{15} is randomised by a fault, then the expected number of hypotheses that are generated can be predicted. For a given input and output difference (XOR), there are a certain number of values that could create the pair of differences. As shown in [16], the number of input and output values that will create a given pair of differences can be set out in tabular form.

Table 8.1 shows part of this table for the first S-box, where columns represent the output difference and the rows represent the input difference. Only the first twenty rows are given as an example. The information this table gives is the number of key hypotheses that will be returned for a given input and output difference over one S-box. Each S-box has its own table, as the structure of each S-box is different.

The first row represents the 64 possible entries that have a difference of zero, which will always produce an output with a difference of zero. This is trivial as the same input values to the S-box will produce the same output values. The second row represents the 64 possible entries that have an input difference of one. This time the output difference can take the values 3, 5, 6, 7, 9, 10, 11, 12, 13, 14, or 15. It is interesting to note that the other 5 output differences cannot occur.

The average number of hypotheses returned by a differential across the k-th S-box can be calculated as the expected number for a random pair (S_{in}, S'_{in}). This is equal to:

Table 8.1: Frequency of input and output pairs for the first S-box.

		$S_{out} \oplus S'_{out}$															
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	0	64	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1	0	0	0	6	0	2	4	4	0	10	12	4	10	6	2	4
	2	0	0	0	8	0	4	4	4	0	6	8	6	12	6	4	2
	3	14	4	2	2	10	6	4	2	6	4	4	0	2	2	2	0
	4	0	0	0	6	0	10	10	6	0	4	6	4	2	8	6	2
	5	4	8	6	2	2	4	4	2	0	4	4	0	12	2	4	6
	6	0	4	2	4	8	2	6	2	8	4	4	2	4	2	0	12
	7	2	4	10	4	0	4	8	4	2	4	8	2	2	2	4	4
S_{in}	8	0	0	0	12	0	8	8	4	0	6	2	8	8	2	2	4
$\oplus$	9	10	2	4	0	2	4	6	0	2	2	8	0	10	0	2	12
S'_{in}	10	0	8	6	2	2	8	6	0	6	4	6	0	4	0	2	10
	11	2	4	0	10	2	2	4	0	2	6	2	6	6	4	2	12
	12	0	0	0	8	0	6	6	0	0	6	6	4	6	6	14	2
	13	6	6	4	8	4	8	2	6	0	6	4	6	0	2	0	2
	14	0	4	8	8	6	6	4	0	6	6	4	0	0	4	0	8
	15	2	0	2	4	4	6	4	2	4	8	2	2	2	6	8	8
	16	0	0	0	0	0	0	2	14	0	6	6	12	4	6	8	6
	17	6	8	2	4	6	4	8	6	4	0	6	6	0	4	0	0
	18	0	8	4	2	6	6	4	6	6	4	2	6	6	0	4	0
	19	2	4	4	6	2	0	4	6	2	0	6	8	4	6	4	6
		⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

$$E_k = \sum_{i=1}^{16} \sum_{j=1}^{64} x_{ij} \frac{x_{ij}}{64^2}$$

where x_{ij} is the value in the table for S-box k ($1 \leq k \leq 8$) corresponding to column i and row j. This equation can be applied to the table corresponding to each S-box, and gives the results shown in Table 8.2.

The expected number of hypotheses for the last subkey will be the product of all eight expected values E_k; this gives an expected number of around 2^{24}. This is just for the last subkey, an actual exhaustive search will need to take into account the eight bits that are not included in the last subkey, giving an overall expected keyspace size of 2^{32}.

If more than one faulty ciphertext block has been acquired, the intersection

Table 8.2: The expected number of hypotheses per S-box for one faulty ciphertext block.

S-box	E_k
1	7.54
2	7.67
3	7.58
4	8.36
5	7.73
6	7.41
7	7.91
8	7.66

between the keyspaces can be used for the final exhaustive search. The expected intersection of two keyspaces generated in the above manner has an expected intersection size as shown in Table 8.3 [33]. As previously, the expected number of hypotheses given for the last subkey will be the product of the expectations per S-box. This gives 2^6 hypotheses for the last subkey, and therefore 2^{14} hypotheses for an exhaustive search of the entire key.

Table 8.3: The expected number of hypotheses per S-box for two faulty ciphertext blocks.

S-box	E_k
1	1.68
2	1.70
3	1.69
4	1.86
5	1.72
6	1.65
7	1.76
8	1.70

A discussion of an implementation of this attack using ciphertext blocks acquired with faults in the fifteenth round is given in Appendix F. The first published description of an implementation of this attack appears in [42].

8.2.2 Faults in Earlier Rounds

The attack described in the previous section corresponds to the ideal situation where a large fault can be injected in the fifteenth round. This fault model will only allow an attack when R_{15} is changed. If R_n is randomised where $n < 15$, then L_{15} will also become randomised and will not cancel as required. A smaller fault is therefore required to minimise the change in L_{15}.

In [18], a one-bit fault model was considered, which is used in this section. The aim of the following analysis is to determine in which rounds a one-bit fault will allow information on the key to be deduced, and how many faulty ciphertext blocks are required to be able to deduce the key.

The Fourteenth Round

In order to determine the effect of a one-bit fault in R_{14}, the effect of a one-bit fault was simulated in a software implementation. A program was written that simulated all the possible on-bit faults in R_{14}, for all the possible values of R_{14} (i.e. $2^{32} \times 32 = 2^{37}$ possible faults). The effect of this fault was observed in R_{15} and the entry to the S-boxes in the sixteenth round by comparing the intermediate state with the expected intermediate state. The change produced by each fault was noted to characterise the effect of a one-bit fault on the subsequent round.

If a one-bit fault is introduced into R_{14}, then the created R'_{15} will have between two and eight bits that are different to R_{15}. This would be expected to modify 3.82 bits in R_{15}, which would modify 5.74 bits across 4.92 different S-boxes. A one-bit fault therefore has a significant impact on the subsequent round, although there will also be a one-bit fault in L'_{15}, as this value is equal to R_{14}.

We also have:

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

This will be equal to $S(L_{16} \oplus K_{16}) \oplus S(L'_{16} \oplus K_{16})$ if $L_{15} = L'_{15}$. If the formula is applied to each S-box to determine six-bit fragments of the key, it will hold for only seven of the eight S-boxes. That is, one of the S-boxes will generate erroneous

hypotheses, as L_{15} will not be equal to L'_{15}, because of the one-bit fault.

This means that it may not be possible to derive the key from one fault, as the keyspace generated will not necessarily contain the last subkey. If the position of the bit that was changed in R_{14} is known, then this problem can be avoided, but this position information will not be available from an analysis of the ciphertext block alone. When several keyspaces are generated, it is also no longer possible to look at the intersection of the hypotheses generated for each S-box. It becomes necessary to use the most frequently generated hypotheses (this algorithm is detailed in [42]). For a one-bit fault in R_{14}, the probability that $L_{15} = L'_{15}$ is 7/8 for each S-box.

Using the simulation described at the beginning of this section, it can be shown that the expected number of bits that would not have a difference of zero on the input of the S-boxes of the fifteenth round is approximately 5.74. If the distribution of the number of bits that are different are observed, the probability that a group of six bits are all equal to zero, given a one-bit fault in R_{14}, is $\Pr(\text{six bits} = 0) = 0.469$.

Therefore, the probability that information can be derived regarding the key for each S-box per faulty ciphertext block is $\Pr((\delta S_{in} \neq 0) \wedge (L_{15} = L'_{15})) = (1 - 0.467) \times \frac{7}{8} = 0.465$, where δS_{in} is the differential on entry to the S-box.

The case where δS_{in} gives weight to every hypothesis, i.e. when $L_{15} \neq L'_{15}$ or $\delta S_{in} = 0$, and the hypotheses derived are uniformly distributed over all the possible key values, will occur with probability $1 - \Pr((\delta S_{in} \neq 0) \wedge (L_{15} = L'_{15})) = 0.535$.

If $\delta S_{in} = 0$ and $L_{15} = L'_{15}$ then all 64 key hypotheses will be valid. When $\delta S_{in} \neq 0$ and $L_{15} \neq L'_{15}$ the analysis will either return no hypotheses (i.e. an impossible differential occurs) or a set of hypotheses. The number of hypotheses returned will be dictated by the table of possible differences (as shown in Table 8.1). However, the probability of a given number of hypotheses being returned is not weighted, as each possibility will occur with the same probability. In this case the expected number of key hypotheses generated is:

$$E_k = \sum_{i=1}^{16} \sum_{j=1}^{64} x_{ij} \frac{1}{(16 \times 64)}$$

although the generated hypotheses will not necessarily include the key. The expected

number of hypotheses generated will be different for each S-box. The value for each S-box is shown in Table 8.4.

Table 8.4: The expected number of hypotheses per S-box when $L_{15} \neq L'_{15}$.

S-box	E_k
1	5.03
2	5.09
3	5.02
4	5.83
5	5.22
6	4.97
7	5.18
8	5.18

Assume that $\delta S_{in} \neq 0$ for every S-box. If $L_{15} \neq L'_{15}$ for the first S-box, the expected number of returned hypotheses is 5.03, and the expected number of weight given to each hypothesis will be $5.03/64 = 0.0786$. This will occur with a probability of 1/8. If $L_{15} = L'_{15}$ then an expected 7.54 hypotheses are generated, which will include the correct key hypothesis. The expected weight given to the correct hypothesis is therefore 1, and the expected weight added to the other key hypotheses is $(7.54 - 1)/(64 - 1) = 0.104$.

In order to distinguish the correct hypothesis from the erroneous hypotheses, the difference in the expected weight should be at least one. Let E_w be the weight given to the correct key hypothesis per evaluation of a correct/faulty ciphertext pair, and E_f be the weight given to the erroneous hypotheses. The number, X, of evaluations that will be required to determine the six key bits used before the first S-box can be calculated as follows:

$$\begin{aligned}
(E_w - E_f)\, X &> 1 \\
\left(\left(\frac{5.03}{64} \times \frac{1}{8} + 1 \times \frac{7}{8}\right) - \left(\frac{5.03}{64} \times \frac{1}{8} + \frac{(7.54-1)}{(64-1)} \times \frac{7}{8}\right)\right) X &> 1 \\
(0.885 - 0.101)X &> 1 \\
0.784X &> 1 \\
X &> 1.28
\end{aligned}$$

where the expected number of hypotheses generated are taken from Tables 8.2 and 8.4. Since X has to be an integer value, we have $X \geq 2$. This applies to each S-box individually, with the same final inequality that $X \geq 2$.

However, in practice δS_{in} will be equal to zero for some of the data acquired. As described above, a one-bit fault will change between two and eight bits in the following round across an expected 4.92 different S-boxes. It would therefore be prudent to evaluate sufficient pairs of faulty/correct ciphertext blocks to have at least two differentials across each S-box.

Other Rounds

An algorithm for exploiting small faults is described in [42]; it is claimed that this technique can be used to derive information on the last subkey from one-bit faults in the eleventh round onwards. A program was written in C to simulate this attack, that could generate faulty/correct ciphertext block pairs with one-bit faults in an given R_i $(0 \leq i \leq 15)$. Another program was written to analyse an arbitrary number of faulty/correct ciphertext block pairs to derive the key. This allowed the attack to be characterised to determine what faults could produce a viable attack. It was possible to recover the key for faults between the eleventh and fifteenth round using less than 1000 faulty/correct ciphertext block pairs, which could potentially be acquired in a reasonable period of time. It was possible to determine the key from faults in the tenth round, but several million faulty/correct ciphertext block pairs were required. This number of pairs of faulty/correct ciphertext blocks would take an excessive amount of time to acquire and it is therefore not likely to be a realistic attack.

Gaining Extra Rounds

The round-specific complexity numbers given above can be transferred to a point two rounds earlier given the trivial observation that, for each fault in R_n there will be an equivalent fault in L_{n-1}. However, L_{n-1} is a static value and is therefore a difficult target for a fault attack. This value is written to memory when R_{n-2} is duplicated at the beginning of round $n-1$. The identification of the point in time

when this duplication takes place should be possible using Simple Power Analysis, as described in Section 4.2. Whether this value can be attacked depends on how the duplication of R_{n-2} is implemented. If the value is duplicated and stored until required, the fault attack is possible; however, if it is used immediately to calculate the round function then the fault attack is no longer possible.

It should therefore be possible to derive key information by using one-bit faults in L_{10}, by targeting the duplication of R_9 in the eighth round. If there is no limit to the number of faulty ciphertext blocks that can be acquired this can be extended to R_8, but this appears unlikely to be a realistic scenario.

8.2.3 Triple DES

Differential Fault Analysis can also be applied to triple DES by independently injecting faults in the second and third instantiations of DES involved in a triple DES computation. A method for attacking triple DES is given in [18], which involves first deriving the last subkey of the last round, and then the penultimate subkey, in order to derive the key used for the third instantiation of DES. This approach then allows faults in the second instantiation to be exploited. This assumes that the fault injection is perfect; a more robust and novel approach would be to use an extended version of the algorithm presented in Section 8.2.1, and this approach is part of the contribution of this thesis.

This approach involves independently injecting faults into the fifteenth round of the second and third instantiations of DES. Let the correct ciphertext block (generated from message block M) be C, a ciphertext block derived as a result of a fault in the fifteenth round of the third instantiation of DES be C', and a ciphertext block derived as a result of a fault in the fifteenth round of the second instantiation of DES be C''.

An attacker can compare C and C' to generate a set of possibilities for the key used in the last instantiation of DES. For each key hypothesis in this set, C and C'' are deciphered, to give a hypothesised state between the second and the third instantiations of DES. These candidate deciphered values can then be compared to yield information on the last subkey of the second instantiation of DES. The set

of candidate keys resulting from this process can then be searched through, using the key used to decipher C and C'' to encipher M. When the key for the second instantiation of DES is found, the key hypothesis used to generate the input and the output of the second instantiation of DES is also validated.

For one attack on the fifteenth round of the second and third instantiation of DES, the total number of hypotheses generated will be $2^{32} \times 2^{32} = 2^{64}$, i.e. the expected size of the set of possibilities produced by combining C and C' as shown in Section 8.2.1, each of which will produce a set of the same expected size.

This is a significant reduction when compared to the exhaustive search of size 2^{112} that is required to attack triple DES by exhaustively searching all the possible key values. This can be improved upon by acquiring more data; if two faulty ciphertext blocks are acquired for both the second and third instantiation of DES; the number of hypotheses generated becomes $2^{14} \times 2^{14} = 2^{28}$, as demonstrated in Section 8.2.1.

This can be further reduced if we take into account the fact that some deciphered values of C and C'' will not produce a valid keyspace. If C and C'' are deciphered with an incorrect key, then the intermediate states produced will be random values. When these values are compared, the relationship between L_{16} and R_{16} in the generated intermediate state will not depend on any bits of the key. The values generated will therefore no longer be distributed with the frequencies given in Table 8.1, but will be uniformly distributed across all the possible combinations.

If an impossible differential (i.e. where the table entry for $(S_{in} \oplus S'_{in}\ ,\ S_{out} \oplus S'_{out})$ is 0 in Table 8.1) is created when analysing the deciphered C and C'', then the key hypothesis used for deciphering C and C'' can be discarded.

The probability that all eight S-boxes produce a valid contribution to the keyspace is the product of the fraction of possible differentials for all of the S-boxes. This gives a probability of 0.125, so the keyspace generated by this attack becomes $2^{32} \times (2^{32} \times 0.125) = 2^{61}$ for one faulty ciphertext block from a fault in the fifteenth round of the second instantiation, and one faulty ciphertext block from a fault in the third instantiation of DES. This can be improved to $2^{14} \times (2^{14} \times 0.125^2) = 2^{22}$, if two faulty ciphertext blocks are produced for both the second and third instantiations

Table 8.5: The fraction of possible differentials per S-box.

S-box	Fraction of Possible Differentials
1	814/1024
2	805/1024
3	816/1024
4	702/1024
5	784/1024
6	824/1024
7	791/1024
8	790/1024

of DES.

This attack can be extended to triple DES with three independent keys, in which case faults will be required in each the fifteenth round of each of the three instantiations of DES. Following the same reasoning as for two-key triple DES leads to a keyspace of size $2^{32} \times \left(2^{32} \times 0.125 \times \left(2^{32} \times 0.125\right)\right) = 2^{90}$ for one faulty ciphertext block from a fault in the fifteenth round of each instantiation of DES. This can be reduced to $2^{14} \times \left(2^{14} \times 0.125^2 \left(2^{14} \times 0.125^2\right)\right) = 2^{30}$ if two faulty ciphertexts are generated by injecting faults into each instantiation of DES.

This is the simplest attack scenario; a more robust version of this attack can be implemented using the algorithm described in Section 8.2.2.

8.2.4 Other Algorithms

Differential Fault Analysis (DFA) is applicable to all block cipher implementations where a predictable fault can be induced. The analysis presented above is based on techniques used in differential cryptanalysis. The discussion in Section 8.2.2 would suggest that DFA should be as efficient across as many rounds as differential cryptanalysis. The first results of differential cryptanalysis of DES, for example, could break a DES where the number of rounds had been reduced to eight. The algorithm given in [42] (described in Section 8.2.2) could be used to conduct a chosen message attack against an implementation of DES where the number of rounds have been reduced to eight. However, the difference between the fault-induced and

correct intermediate state within the algorithm (i.e. the message blocks in terms of differential cryptanalysis) is not necessarily known. It is also a time-consuming task to gain a significant number of faulty ciphertext blocks (which is trivial in the differential cryptanalysis attack model), as a certain amount of effort is required to gain each one.

Since the adoption of AES, several different methods for breaking AES using DFA [22, 29, 36, 40, 89] have been proposed. The structure of AES is more resistant to DFA than DES, and it is only possible to exploit faults in the last three rounds of AES.

8.3 Collision Fault Analysis

The use of collisions to find and exploit a fault occurring at the beginning of an algorithm has appeared in several papers [22, 48]. This method of fault attack involves injecting faults in the early stages of an algorithm. A message block is then found that would naturally give the same ciphertext block, i.e. another message block is found that produces the faulty ciphertext block without any fault occurring during computation. This involves enciphering message blocks, each with a small difference to the original message block, to try to produce the observed faulty ciphertext block. Once a collision has been found, the two message blocks can be used to derive information, where the analysis is similar to DFA. If an attacker is able to encipher and decipher with the same device, conducting such an attack is trivial, as the faulty ciphertext can be deciphered to find the desired message block.

A trivial case of this type of attack is given in [22], where a known bit of the first **AddRoundKey** (see Section 2.2) in AES is assumed to be forced to zero. If the ciphertext block changes then this bit would have been a one; if the ciphertext block remains the same then the bit is a zero. This approach can be used to break an AES implementation with 128 executions with successful fault injections. However, modifying bits in such a manner requires too much precision to be practical.

Another trivial Collision Fault Analysis (CFA) based attack on AES is a bytewise implementation of the attack given in [22]. If a fault is injected so that a byte of the

output becomes zero during the first **AddRoundKey** function, a similar attack can be implemented. All the possible combinations of message block bits corresponding to the modified byte can be tested and the algorithm executed for each value. This process is stopped once a collision is found with the faulty ciphertext block. This will give a message block with an intermediate state where the fault was injected that is naturally equal to zero. This means that the message byte found is equal to zero after being XORed with the corresponding key byte, and the two values are therefore equal. This approach requires 16 faulty ciphertext blocks to be generated, and a search of size 2^8 with the targeted device to find each key byte, giving a total search complexity of $16 \times 2^8 = 2^{12}$.

8.3.1 Attacking DES

In [48] a method of exploiting faults in the early rounds of a DES implementation is described. Faults are injected during the execution of DES to produce a faulty ciphertext block. A message block is then found that will produce this ciphertext block when enciphered with the same key without a fault being applied. This produces a pair of message blocks that are related in exactly the same way as the two ciphertext blocks considered in the DFA applied to DES described in Section 8.2. The only difference here is that the attacker derives hypotheses regarding the first subkey rather than the last subkey, by comparing the two message blocks.

The search for all possible faults to conduct this attack is prohibitively complex, so an attacker is required to assume that all possible faults can be modelled as a change of between one to four bits in the exit of one S-box. This allows collisions to be found in a reasonable amount of time, and ensures that, when collisions are found, information can be determined about the key. This allows an attacker to derive information from faults in the first three rounds of DES. In theory, faults in later rounds would allow information on the first subkey to be derived, but the finding a collision becomes too time-consuming after the third round. The number of message blocks that need to be tested, to cover all the different effects that the fault could have produced, becomes prohibitively large, especially as each message block has to be tested using the device under attack.

8.3.2 DPA-Resistant Algorithms

An implementation of a block cipher on an embedded platform should be resistant to DPA, and must therefore use the data whitening countermeasure (as described in Section 7.4). This requires the implementation of functions present specifically to manage this masking. These functions can be subjected to CFA in the same way that the algorithm can. Two such attacks are described below, and are part of the contribution of this thesis.

Attacking the First XOR of DPA-Resistant AES

At the beginning of the computation of AES, an XOR between the message block and the key is conducted before the first **ByteSub** function. This calculation is described in Algorithm 8.1.

Algorithm 8.1: The First XOR

Input: $M = (m_1, m_2, m_3, \ldots, m_{16})_{256}$ containing the message block,
$K = (k_1, k_2, k_3, \ldots, k_{16})_{256}$ containing the key masked with a random byte $\mathbf{R}$.

Output: $KM = (km_1, km_2, km_3, \ldots, km_{16})_{256}$ also masked with $\mathbf{R}$.

for $i \leftarrow 1$ **to** 16 **do**
    $km_i \leftarrow m_i \oplus k_i$
end

return KM

If the **for** loop in Algorithm 8.1 is attacked, so that the loop only runs to 14 rather than 16, two bytes will not be written to KM, i.e. two bytes will remain untouched. This means that these bytes could be set to 0 in RAM, and therefore the final two output bytes of KM will take the value of the random byte $\mathbf{R}$, as a result of the masking. The effects of this approach depend on the logic used in the chip, i.e. whether empty memory is interpreted by the chip as a zero or a one, and whether the memory has been previously used by other functions. This latter possibility can be minimised by resetting the smart card being attacked before each attempt at injecting a fault.

If the fifteenth and sixteenth byte of the result of the first XOR are set to zero, it will be possible to find a collision where $m_{15} \oplus k_{15} \oplus \mathbf{R} = 0$ and $m_{16} \oplus k_{16} \oplus \mathbf{R} = 0$,

by generating ciphertexts and trying the 2^{16} possible values for m_{15} and m_{16}. If a collision is found then

$$m_{15} \oplus k_{15} \oplus \mathbf{R} = m_{16} \oplus k_{16} \oplus \mathbf{R}$$

$$m_{15} \oplus k_{15} = m_{16} \oplus k_{16}$$

$$m_{15} \oplus m_{16} = k_{15} \oplus k_{16}$$

where the left hand side of the equation is known, i.e. the effect of $\mathbf{R}$ is removed. If the value of the memory becomes $\texttt{FF}_{16}$, for example, it can still be assumed to be 00 and the error included in the value of $\mathbf{R}$. It is therefore not important what values km_{15} and km_{16} become, but they do need to be the same value.

This reduces the size of the keyspace from 2^{128} to 2^{120}. The attack can be continued by repeating the attack with three bytes of the key being left uninitialised by Algorithm 8.1. The above method can then be used to derive $k_{14} \oplus k_{15}$, requiring the same amount of work as required to determine $k_{15} \oplus k_{16}$. This process can be continued until the whole key is derived. This is similar to the attack described in Section 6.3, but an attacker needs to search through 2^{16} different values to determine the message values that enable information on the key to be derived. This also needs to be done with the smart card under attack as the rest of the key is unknown; the attack is therefore time-consuming.

In DPA-resistant algorithms, it is usual to perform as much as possible of the computation in a random order (see Section 7.3). The loop given in Algorithm 8.1 would therefore take one of the 120, i.e. $\binom{16}{2}$, different orders possible for this process. This is so that the data being manipulated from one execution to another will be manipulated at different points in time.

If the random order countermeasure is implemented, and the last two bytes are not assigned, the algorithm will take these bytes as identical random bytes equal to $\mathbf{R}$. To find a collision, an attacker will have to search through the 120×2^{16} (approximately 2^{23}) possible message blocks. The random value $\mathbf{R}$ involved will be different each time, so if we consider the results from two successful faults, e.g.

$$k_i \oplus m_i \oplus \mathbf{R} = 0,\ k_j \oplus m_j \oplus \mathbf{R} = 0, \text{ and}$$
$$k_j \oplus m'_j \oplus \mathbf{R}' = 0,\ k_l \oplus m'_l \oplus \mathbf{R}' = 0$$

information of three bytes of the key is not immediately visible. However, if they have one index in common (in this case j) it is possible to change the random mask so that they become the same by XORing the two values together, i.e.

$$k_j \oplus m_j \oplus \mathbf{R} \oplus k_j \oplus m'_j \oplus \mathbf{R}' = 0$$

and hence

$$\mathbf{R} \oplus \mathbf{R}' = m_j \oplus m'_j.$$

As m_j and m'_j are known, these values can be used to find $\mathbf{R} \oplus \mathbf{R}'$. This value can then combined with the equation $k_l \oplus m'_l \oplus \mathbf{R}' = 0$ to yield $k_l \oplus m'_l \oplus \mathbf{R} = 0$. This then gives:

$$m_i \oplus \mathbf{R} = k_i$$
$$m_j \oplus \mathbf{R} = k_j$$
$$m'_l \oplus \mathbf{R} = k_l$$

A further pair, e.g.

$$k_x \oplus m''_x \oplus \mathbf{R}'' = 0 \text{ and } k_y \oplus m''_y \oplus \mathbf{R}'' = 0$$

can then be used to derive information on another key byte, if either x or y has the same value as either i, j or k. This process can then be repeated until information is derived on every byte of the key, then each key byte is known XORed with $\mathbf{R}$. This continues until an attacker has seen all of the index values twice. This then leaves an exhaustive search of 2^8 to find the value of $\mathbf{R}$ and therefore the key. An implementation of this attack on an implementation using the random order countermeasure is described in Appendix G.

A more efficient version of this attack would be to generate pairs,

$$k_x \oplus m_x \oplus \mathbf{R_i} = 0 \text{ and } k_y \oplus m_y \oplus \mathbf{R_i} = 0$$

where x and y represent the indexes for a given pair, and $\mathbf{R_i}$ is the random value generated to mask AES during the i-th attack. An attacker can continue to acquire pairs until each index value has been seen once. This will allow each unknown element to be determined using linear algebra. The fact that $\mathbf{R_i}$ varies from one pair to another is not an issue, as enough repetitions will occur while an attacker is waiting until each index had been observed to make the system overdetermined.

In order to be able to find collisions an attacker needs to generate a dictionary of 2^{23} entries. These values depend on the key so they need to be generated with the device under attack. This is difficult for devices, such as smart cards, that use relatively slow communication protocols. To form an idea of the amount of time required to create a dictionary, a smart card with a DPA-resistant AES was timed. It took the smart card approximately 149 milliseconds to create one dictionary entry. The whole dictionary would therefore require around 14.5 days to create with this smart card.

This can be reduced by generating a certain amount of the dictionary and conducting more fault attacks to generate the data required to derive the key. For example, if a dictionary of size 2^{19} was generated, which would take around 21 hours with the test card, faulty ciphertext blocks could be generated until a collision was found. It would be expected that one in 2^4 faulty ciphertext blocks would be present in the dictionary. An attacker would therefore need 16 times more data compared to the attack described above, but, as only a few faulty ciphertext blocks are required to realise the attack, this is an efficient option.

Attacking the Key Masking of DPA-Resistant AES

Before the key can be used by the algorithm in the fashion described in Section 7.4, the random value, $\mathbf{R}$, that is applied to the S-boxes needs to be applied to the key. The key values are usually stored XORed with a random value of the same bit length as the key. This is because this random value is a static value for a card and is stored in EEPROM. This mask needs to be removed, and replaced with $\mathbf{R}$ with

the key value remaining masked. This process is shown in Algorithm 8.2. Again, this happens in a random order rather than as shown.

Algorithm 8.2: Masking the Key

Input: $KR = (kr_1, kr_2, kr_3, \ldots, kr_{16})_{256}$ masked with a random $R = (r_1, r_2, r_3, \ldots, r_{16})_{256}$, $\mathbf{R}$ a random byte.
Output: $K = (k_1, k_2, k_3, \ldots, k_{16})_{256}$ masked with $\mathbf{R}$.

for $i \leftarrow 1$ **to** 16 **do**
    $k_i \leftarrow kr_i \oplus \mathbf{R}$
    $k_i \leftarrow k_i \oplus r_i$
end
return K

A similar attack to that described in Section 8.3.2 can be envisaged against this process. If only one byte is initialised by this loop, the resulting memory could be predominately set to zero. Again, the algorithm will take this value as being equal to $\mathbf{R}$. There are 2^{16} different values for the combination of the initialised key byte and $\mathbf{R}$. However, the random order countermeasure will mean that an attacker will not know which of the 16 key bytes has not been initialised. This means that the dictionary for this attack will contain $16 \times 2^{16} = 2^{20}$ possible ciphertext blocks, that need to be generated before the attack is conducted. This dictionary can be generated on a PC, as the dictionary entries are independent of the key being used.

This attack can be repeated until enough information is derived about the key to enable an exhaustive search to take place. The expected number of faulty ciphertext blocks needed to derive each byte in this way can be calculated using the Coupon Collectors Test given in [58]. In the case of AES this attack requires 50 ciphertext blocks to derive the whole key. An implementation of this attack is described in Appendix H.

8.3.3 Other Algorithms

The above attacks apply to all block cipher implementations. A version of CFA applied to AES appears in [19]. The attacks described in Section 8.3.2 are also generic, and can be applied to other algorithms with very little modification, as the targeted features are common to all DPA-resistant implementations.

8.4 Changing S-Box Values

In algorithms that implement data whitening, as detailed in Section 7.4, S-boxes need to be moved from NVM to RAM. In this section, two attacks are described against this mechanism, whose operation depends on how S-boxes are transferred into RAM. These attacks are described separately, as they draw from both DFA and CFA. An implementation of these attacks is described in Appendix I.

8.4.1 Modifying Known S-Box Values of DES

If the S-box values are created in RAM in a known order, a fault attack can then be constructed around the modification of known S-box values.

The first value of the first S-box is modified by a fault, and the algorithm is executed with a message block for which the corresponding ciphertext block is known. If the ciphertext block is not equal to the known ciphertext block, then this S-box value was used by the algorithm; if it stays the same, then the S-box value was not used anywhere in the algorithm. All 64 values of the first S-box can each be changed in this manner and the algorithm executed. After which, all the S-box entries used from the first S-box will be known for a given message block.

The expected number of S-box entries used per DES execution can be calculated using the solution to the Classical Occupancy Problem, as described in [67], giving a value of 14.3. Therefore, if the attack is repeated for each S-box, a list of approximately 14 different values will be given for the number of entries used in each S-box.

If these values are taken as possible hypotheses for the S-box entries used in the first round, the index values of the S-box entries can be turned into hypotheses on the first subkey. To do this, the index values simply need to be XORed with the relevant message bits. This will produce slightly under 2^{31} hypotheses for the first subkey leading to a total exhaustive search of size 2^{39} to find the entire DES key.

In order to reduce the size of the exhaustive search, the attack can be repeated with a different message block. The intersection of the two keyspaces will contain the first subkey. This provides an expected number of $14.255 \times (14.255/64) = 3.18$

different hypotheses per S-box, which gives 2^{13} hypotheses for the first subkey, leading to a total exhaustive key search of size 2^{21}.

In practice, embedded implementations of DES are unlikely to have the 512 S-box values necessary for DES written separately in memory. These are generally compressed to optimise the amount of memory required by the DES implementation.

One way of achieving this is to store the data in 256 bytes, where the odd numbered S-boxes are stored in the high nibbles and the even numbered S-boxes are stored in the low nibbles. This corresponds to the attack implementation detailed in Appendix I, and all further discussion will assume that this is the case. There are several other ways in which the S-box data could be compressed; however, this is not considered to be something that an attacker needs to know before conducting an attack, as all the possible combinations can be attempted until the correct one is found.

The number of key hypotheses generated by implementing this attack against an implementation of DES using compressed S-boxes is shown in Table 8.6 for different numbers of message blocks used. As can be seen, this is more efficient than modifying one S-box value, as information on two boxes can be gained at once, i.e. fewer faults are required to derive the key.

Table 8.6: The hypotheses generated by attacking a compressed S-box.

Message Blocks	Hypotheses per S-box pair	Hypotheses for the first round key	Total Keyspace
1	25.3	2^{37}	2^{45}
2	10.0	2^{27}	2^{35}
3	3.97	2^{16}	2^{24}
4	1.57	2^{5}	2^{13}

This attack can be further optimised by analysing the faulty ciphertext blocks generated by the modified S-boxes. It should be apparent from the ciphertext block if a faulty S-box value has been used in the fifteenth or sixteenth round. As the aim is to try to derive hypotheses on the first subkey, these ciphertext blocks provide no information. Ciphertext blocks where the faulty S-box is used in the last round

can be considered to be equivalent to the S-box value not being used. If the faulty value is used in the fifteenth round, no information is provided, as the detection of this event is subject to false positives (as described in Section 8.4.2), and a different message block needs to be used to provide information on this S-box value. This allows some faulty ciphertext blocks to be treated in the same way as correct ciphertext blocks, reducing the number of key hypotheses produced.

One possible countermeasure for this attack is to randomise the order in which the S-boxes are randomised. This applies to the order in which the S-boxes are treated, and to the order in which the S-box elements are masked, i.e. the counter is XORed with a random value before being used so that the order in which the S-box elements are treated is unknown.

If just the order in which the S-boxes are treated is randomised, an attack could be envisaged based on searching for S-box elements that never change the ciphertext block when modified. This is because the information about which index value has been changed will be present. If the same S-box element is repeatedly changed, but after numerous executions with the same message block the ciphertext block never changes, it can reasonably be assumed that this index value does not represent a key hypothesis for any part of the first subkey. The expected number of executions required to be sure of this information is 22 (from the Coupon Collectors Test as defined in [58]). The randomisation of the order of treatment would therefore make the attack significantly more complex. 1408 fault injections would be required to treat every S-box value for a given message block, and this would give an expected number of hypotheses of 55.5 per S-box, and a total key search of $2^{54.3}$ possibilities. A total of ten different message blocks would be required to bring the expected key search to $2^{39.5}$, which would lead to a time-consuming, but not infeasible, exhaustive search. However, such a large number of fault injections may be unrealistic if the effect of the fault is not deterministic.

8.4.2 Modifying Unknown S-Box Values of DES

If an S-box element can be modified, but the attacker does not know which element has been modified, the attack described in Section 8.4.1 will not work. Nevertheless,

an attack can still be implemented using Differential Fault Analysis.

If one S-box value is modified and used in round 15, and only in round 15, then the ideas described in [18] will apply. The modification of one S-box look-up in the fifteenth round will, on average, change the entry value for 3.2 different S-boxes in the sixteenth round, providing differentials across these S-boxes for key hypothesis testing.

The advantage of this attack over the attack described in Section 8.4.1 is that the effect of the desired fault can be seen in the ciphertext block. The fault can be detected by calculating the differential of the S-box output in the fifteenth round, which can be achieved by examining the ciphertext block. If only one nibble in this value is not equal to zero, then there is a high probability that the corresponding S-box value was only used in the fifteenth round. The probability that this event occurs is $\left(\frac{63}{64}\right)^{15} \frac{1}{64} = 0.0123$.

This probability is high enough that an attacker can conduct the attack a number of times until the desired event is observed. Information about the key can then be derived and the process repeated.

There is a possibility that the S-box is used in the fourteenth round and that this will yield a value that will be detected as a S-box value used in the fifteenth round. This occurs when the modification in the fourteenth round produces a one-bit fault (all the output bits from an S-box go to different S-boxes). There are four possible values among the 15 possible faults that will produce this effect. Half of these values will modify more than one S-box in the fifteenth round, i.e. they will span two S-boxes as a result of the expansion permutation. This leaves only two possible values from the fifteen possible faults that affect just one S-box. The probability of a false positive is therefore $\frac{2}{15}\left(\frac{63}{64}\right)^{15} \frac{1}{64} = 0.00165$.

The probability of a false positive is relatively high compared to the probability of the event that will enable the attack. That is, approximately 1 in 7 detections will be false positives. However, as described in [42], the false hypotheses introduced by these false positives will not have a major effect on the success of the attack.

As discussed in Section 8.4.1, S-box values are usually stored in a compressed state, so an attacker may be forced to modify several S-box entries at once. If two

S-box entries are modified, the probability of one of the values being used in the fifteenth round is $2\left(\frac{63}{64}\right)^{15}\frac{1}{64}\left(\frac{63}{64}\right)^{16} = 0.0192$.

This is more efficient than modifying one S-box value, as the probability of the S-box value being used in the fifteenth round is higher. This probability will change following the method of S-box compression, but only the case under study is analysed.

The probability of one S-box value being used in the fourteenth round, and causing a false positive then this can be calculated as before. If two modified S-box values are used in the fourteenth round this can also provoke a false positive if two 1-bit errors are caused and these bits are then used in the same S-box without being reproduced by the expansion permutation in the fifteenth round. The probability of such an event was derived by simulating all the possible combinations, giving a value of 89/147456. The overall probability of a false positive is therefore $2\frac{2}{15}\left(\frac{63}{64}\right)^{15}\frac{1}{64}\left(\frac{63}{64}\right)^{16} + \frac{89}{147456}\left(\left(\frac{63}{64}\right)^{15}\frac{1}{64}\right)^2 = 0.00256$.

The probability of a false positive, given that a detection has occurred, is about the same (approximately 2/15), given that the event has been detected for both implementations. The implementation using a compressed S-box will provide results using fewer induced faults, as the desired event occurs with a higher probability.

In the case of an implementation using compressed S-boxes, it would be logical to use the event of both faulty S-box values being used in the fifteenth round. As previously, this can be observed by looking for two nibbles with a non-zero differential in the ciphertext block. This information can be combined with the event of one nibble having a differential in the ciphertext block. The probability of this occurring is $2\left(\frac{63}{64}\right)^{15}\frac{1}{64}\left(\frac{63}{64}\right)^{16} + \left(\left(\frac{63}{64}\right)^{15}\frac{1}{64}\right)^2 = 0.0193$.

As previously, there is a chance of a false positive. In this case the events of one or two modified S-box values being used in the fourteenth round could potentially simulate one or two changed values in the fifteenth round. If one S-box value is changed in the fourteenth round, the probability that two values are modified in the fifteenth round is 1/5. If two values are changed in the fourteenth round, the probability that two values are changed in the fifteenth round is 914609/29491200.

Again, these were derived by simulating all the possible combinations. The probability of a false positive is therefore $2\frac{2}{15}\left(\frac{63}{64}\right)^{15}\frac{1}{64}\left(\frac{63}{64}\right)^{16}+\frac{89}{147456}\left(\left(\frac{63}{64}\right)^{15}\frac{1}{64}\right)^{2}+2\frac{1}{5}\left(\frac{63}{64}\right)^{15}\frac{1}{64}\left(\frac{63}{64}\right)^{16}+\frac{914609}{29491200}\left(\left(\frac{63}{64}\right)^{15}\frac{1}{64}\right)^{2}=0.00640$.

The probability of getting useful information remains approximately the same as when only one modified S-box is considered, but the probability of a false positive is approximately 2.5 times greater. The data acquired will therefore be much more noisy and will increase the amount of time required to conduct the attack. There is therefore little interest in conducting the attack in this manner.

8.4.3 Other Algorithms

The attacks described above have been discussed in terms of DES, but this is a generic attack and can be applied to any DPA-resistant algorithm. The AES, for example, uses the same S-box throughout the algorithm. There are a total of 200 accesses to this table of 256 different elements. The expected number of different S-box elements used is 139 (given by the Classical Occupancy Problem, as described in [67]). These elements could be modified and used to derive information in a similar manner to those described in Section 8.4.1. An attacker would be able to form hypotheses on the first subkey and would also know that these S-box values were used to generate the rest of the subkey. The number of hypotheses could be significantly reduced using this information. In the case where an attacker does not know which value is being modified, it would be expected that a differential fault attack could be formed using the techniques given in [22, 29, 36, 40, 89].

8.5 Countermeasures

There are many different countermeasures that can be used to protect implementations of block ciphers on an embedded platform. They are listed individually here for clarity.

Constant Execution — As described in Section 7.1, it is important to implement block ciphers in a way that the same code is executed for all possible entry

values. This is important to avoid attacks such as those described in Section 6.1. This also applies to functions like the **xtime** function in AES, as detailed in Section 2.2. This will also remove the time side channel, as the use of the same code implies that the algorithm must always have the same execution time and therefore removes any possibility of a timing attack (see Section 4.1).

Variable Randomisation — The data manipulated by a block cipher should be masked by a random value that is generated at the beginning of each execution. This process is described in Section 7.4, and prevents Statistical Power Attacks, as described in Sections 4.3 and 8.1.

Random Delays — These cannot directly prevent an attack, but can complicate an attacker's task, as described in Section 7.2. These should be placed between each subfunction of the round function.

Random Execution — Block ciphers should be implemented so that as much of the computation as possible is conducted in a random order (see Section 7.3). Each subfunction can generally be conducted in a random order, as these functions usually operate on portions of the data or key being manipulated. Also, when variables are transferred from one part of the chip to another, e.g. from NVM to RAM, they need to transferred in a random order to avoid attacks, such as those described in Section 6.3. The use of a checksum attached to each variable is also advisable, in order to be able to verify the integrity of the data once it has been moved.

Round Redundancy — In order to prevent Differential and Collision Fault Analysis (see Sections 8.2 and 8.3) the rounds at the beginning and the end of a block cipher should be repeated to detect faults. In the case of DES it would be advisable to repeat the last six rounds and the first three rounds (if an attacker is able to execute the inverse DES, it is necessary to repeat the first six rounds). This represents slightly over half the algorithm and is therefore more efficient than simply repeating the whole algorithm. Theoretically, attacks in

the area that is not repeated are still possible, but not currently practical.

It may be possible to inject the same fault in the algorithm and then in the redundant code, although this will be extremely difficult in the presence of random delays. For this reason, it may be prudent to use the inverse round function, where possible, in the redundant code to minimise this possibility.

Checksums — If S-boxes need to be constructed in RAM they need to be protected by a checksum to prevent the attacks described in Section 8.4. The simplest method of achieving this would be to XOR all the values together after the table has been created, i.e. after the table has been written to memory. This has the added advantage of removing the data whitening countermeasure (see Section 7.4), as the number of entries in the S-box will be an even number. Nevertheless, this is not adequate to defend against the attacks described above. If the checksum is on 1 byte an attacker could modify several values and have a probability of 1/256 of having a valid checksum. A second checksum calculated in a different manner could remove this problem, as the second checksum can be chosen such that there is no fault that will allow both checksums to remain valid.

Memory Randomisation — All "work" areas of RAM used can be filled with independent random values before the start of the algorithm. The feasibility of the attacks described in Section 8.3 would then rest on the quality of the random values used. If, for example, an LFSR was used to generate these values it may be possible to predict the value of one byte if the previous byte is known. This could mean that the attack described in Section 8.3.2 is still possible with some changes, i.e. the initial state of the LFSR is included as an extra variable to be derived by linear algebra.

8.6 Summary and Conclusions

This chapter provides a description of some of the attacks that can be applied to implementations of block ciphers on embedded platforms. Differential Fault Analysis, described in Section 8.2, is well-known within the smart card industry,

and descriptions of implementations have been published. However, the extension to triple DES described in Section 8.2.3 is novel, and is part of the contribution of this thesis. Collision Fault Analysis, described in Section 8.3, is more recent, although the analysis is somewhat similar to Differential Fault Analysis.

The application of DFA and CFA to DPA countermeasures, described in Sections 8.3.2 and 8.4, is novel, and is part of the contribution of this thesis.

Implementations of four of the attacks described in this chapter are discussed in Appendices G, H, and I, and are also part of the contribution of this thesis.

Chapter 9

Implementations of RSA

The RSA algorithm [94], described in Section 2.3, is one of the most commonly used public key cryptographic algorithms. Implementations of RSA are therefore often included in smart cards, so that the RSA private key can be protected by the smart card's tamper resistant environment. The problems of implementing RSA on a smart card are detailed here, as they are significantly different to the problems encountered when implementing block ciphers.

This chapter will present the attacks that are possible against a naïve RSA implementation. This is followed by details of the various types of exponentiation algorithm that can be implemented on a smart card, and how these can be rendered resistant to side channel and fault attacks. The best method of combining the various types of protection is then discussed.

Section 9.1 describes Simple Power Analysis of RSA, as previously described in Section 4.2. The possible application of Statistical Power Analysis to RSA is discussed in Section 9.2. The different possible fault attacks that can be applied to the calculation of RSA are detailed in Sections 9.3–9.5. The different algorithms for calculating a modular exponentiation, i.e. RSA, are discussed in Section 9.6. The same discussion is applied to the calculation of RSA using the Chinese Remainder Theorem (CRT), with a description of the different side channel and fault resistant algorithms in Section 9.7.

In Section 9.8, the underlying multiplication algorithm that is used to calculate RSA is discussed. The application of side channel and fault attacks to Montgomery multiplication is discussed in Sections 9.8.1 and 9.8.2. The application of

this method to the calculation of RSA with or without CRT is described in Section 9.8.4.

A comparison of the various fault protection methods is given in Section 9.9, followed by a list of all the countermeasures that are required to protect against side channel and fault attacks when implementing RSA in Section 9.10. Some of the work in this chapter has been published in [10, 11], and some work remains unpublished [81].

9.1 Simple Power Analysis of RSA

As described in Section 4.2, an RSA implementation can be vulnerable to Simple Power Analysis if an attacker can distinguish between the two functions that are used to calculate a modular exponentiation. If the square and multiply algorithm is considered (see Algorithm 4.1); the key can be derived if an attacker can distinguish between a squaring operation and a multiplication. This issue was discussed in Section 4.2.

Distinguishing a square from a multiplication can be immediately obvious (as in Figure 4.2), but is often more subtle. The differences can be in execution time or power consumption patterns.

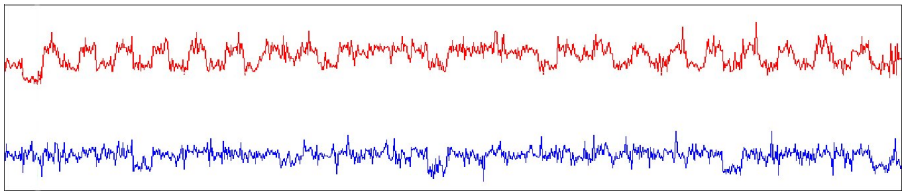


Figure 9.1: The electromagnetic emanations of an RSA implementation.

Figure 9.1 shows the electromagnetic emanations of an RSA implementation for two different keys, (FFA5FF_{16} and 666666_{16}). This gives a different example of a different way in which RSA can leak information, that is less obvious than Figure 4.2. However, the differences are still significant enough that an attack using Simple Power Analysis should be possible.

9.2 Statistical Power Analysis of RSA

The use of Statistical Power Analysis against RSA is not well documented, but it should be possible to determine the bits of the private RSA exponent d in a manner similar to that described in Section 4.3. It should be possible to replace the function in the attack described in Section 8.1 with a series of squares and multiplications from the square and multiply algorithm (see Algorithm 4.1).

An example of the results of the calculation for the first three bits of d is given in Figure 9.2 for a message M, assuming that the most significant bit is $d_x = 1$. A statistical attack could be used to predict the outcome for a given hypothesis for selected bits of d, and generate a DPA or CPA waveform to verify this hypothesis.

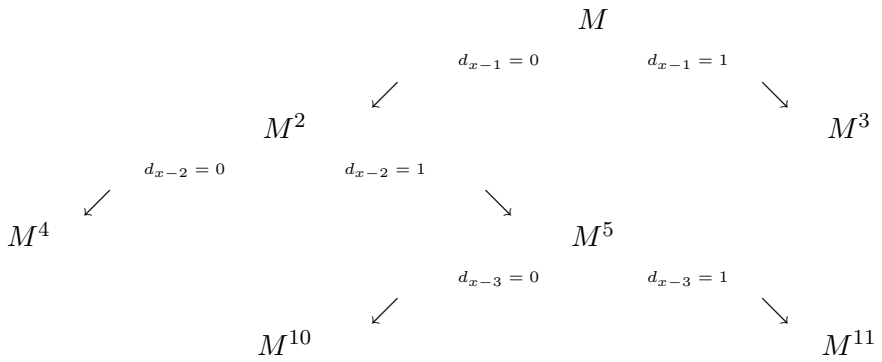


Figure 9.2: An example calculation path for the square and multiply algorithm.

This attack is not necessarily practical, as the word size manipulated by a co-processor may be too large to enable the generation of CPA waveforms with a significant peak. Some results on the use of DPA to derive information on a secret RSA exponent are given in [70], although such an approach is expected to suffer greatly from false positives when used against systems deploying coprocessors that use large word sizes. However, it remains necessary to defend against this attack in case a practical attack becomes possible.

9.3 Fault Attack on RSA Signature with CRT

The first published fault attack [23], proposed an attack focused on an implementation of RSA using the Chinese Remainder Theorem (CRT). The attack allows for a wide range of fault injection methods, as it only requires one fault to be inserted in order to factorise the RSA modulus.

The technique requires an attacker to obtain two signatures for the same message, where one signature is correct and the other is the result of the injection of a fault during the computation of S_p or S_q (see Section 2.3). That is, the attack requires that one of S_p and S_q is computed correctly, and the other is computed incorrectly.

Without loss of generality, suppose that $S' = aS_p + bS'_q \mod N$ is the faulty signature, where S_q is changed to $S'_q \neq S_q$. We then have:

$$\begin{aligned}\Delta &\equiv S - S' \pmod{N} \\ &\equiv (aS_p + bS_q) - (aS_p + bS'_q) \pmod{N} \\ &\equiv b(S_q - S'_q) \pmod{N}.\end{aligned}$$

As $b \equiv 0 \pmod{p}$ and $b \equiv 1 \pmod{q}$, it follows that $\Delta \equiv 0 \pmod{p}$ (but $\Delta \not\equiv 0 \pmod{q}$) meaning that Δ is a multiple of p (but not of q). Hence, we can derive the factors of N by observing that $p = \gcd(\Delta \bmod N, N)$ and $q = N/p$.

In summary, all that is required to break RSA is one correct signature and one faulty one. This attack will be successful regardless of the type or number of faults injected during the process, provided that all faults affect the computation of either S_p or S_q.

Although initially theoretical, this attack (an implementation is described in [8]) stimulated the development of a variety of fault attacks against a wide range of cryptographic algorithms.

9.4 Fault Attack on RSA Signature Without CRT

We now describe a different type of fault attack that involves modification to a single bit of the private exponent. Suppose that, due to a fault, one bit in the

binary representation of the secret key d flips from 1 to 0 or vice versa, and that this faulty bit position is randomly located. Suppose than an attacker arbitrarily chooses a plaintext M and causes the smart card to compute a signature S on M. While the signature is being generated the attacker generates a fault causing one bit of the private exponent to be changed, resulting in a faulty signature S'. Assuming that the i-th bit of d is complemented, then, as described in [9], we have:

$$\frac{S'}{S} \equiv M^{d'-d} \equiv \begin{cases} M^{2^i} \pmod N & \text{if the } i\text{-th bit of } d = 0, \\ \frac{1}{M^{2^i}} \pmod N & \text{if the } i\text{-th bit of } d = 1. \end{cases}$$

where S is the correct signature.

In [56] it was shown that this attack can be significantly improved by raising the formula to the power of the public exponent e, giving:

$$\frac{S'^e}{M} \equiv \begin{cases} (M^e)^{2^i} \pmod N & \text{if the } i\text{-th bit of } d = 0, \\ \frac{1}{(M^e)^{2^i}} \pmod N & \text{if the } i\text{-th bit of } d = 1. \end{cases}$$

In this case it is not necessary to know the correct signature S.

An attacker can compute $\frac{S'^e}{M} \bmod N$ for all the possible one-bit fault errors in d. This requires a total of $\log_2 d$ trials to completely derive d.

This attack has been extended to cover the case where more than one bit is changed [9]. If, for example, two bits (j and k) are changed in the private exponent, then

$$d' = d \pm 2^j \pm 2^k.$$

If

$$S' = M^{d'} \bmod N$$

then this can reveal information on the private key using the following relationships.

$$\frac{S'^e}{M} \equiv \begin{cases} (M^e)^{2^j}(M^e)^{2^k} \pmod N & \text{if the } j\text{-th bit of } d = 0 \\ & \quad \text{and the } k\text{-th bit of } d = 0 \\ (M^e)^{2^j}\frac{1}{(M^e)^{2^k}} \pmod N & \text{if the } j\text{-th bit of } d = 0 \\ & \quad \text{and the } k\text{-th bit of } d = 1 \\ \frac{1}{(M^e)^{2^j}}(M^e)^{2^k} \pmod N & \text{if the } j\text{-th bit of } d = 1 \\ & \quad \text{and the } k\text{-th bit of } d = 0 \\ \frac{1}{(M^e)^{2^j}}\frac{1}{(M^e)^{2^k}} \pmod N & \text{if the } j\text{-th bit of } d = 1 \\ & \quad \text{and the } k\text{-th bit of } d = 1 \end{cases}$$

Unfortunately, the above analysis is flawed, and the attack only works if just one bit is flipped. To see this, rewrite the above expression

$$\frac{S'}{S} \equiv M^{d'-d} \equiv M^{\pm 2^j \pm 2^k}$$

where the $\pm$ depends on how the bit has been changed. It can be seen that another two faults (changing bits l and n) will yield the same value for d' if:

$$\pm 2^j \pm 2^k = \pm 2^l \pm 2^n$$

For example, if the fifth and sixth bits of d are flipped by a fault from 0 to 1 to produce d', and the fifth bit of d flips from 1 to 0 and the seventh bit of d flips from 0 to 1 to produce d'', then:

$$M^{d'} = M^{d+2^6+2^5} = M^{d+2^7-2^5} = M^{d''}$$

since $2^6 + 2^5 = 2^7 - 2^5$. In this case the attacker has ambiguous information about the private exponent d, i.e. the attacker knows that either the fifth and sixth bits are zero, or that the fifth bit is one and the seventh bit is zero, and hence the attack becomes ineffective. This ambiguity worsens when greater numbers of bits are changed by fault injection, as more combinations of faults give equivalent results.

The ambiguities arising in the above version of the attack would be removed if an attacker knows precisely which bits are affected by a fault, although this degree of precision not currently possible (as described in Section 8.3). The effect of a

fault will generally correspond to one of the three models proposed in Section 5.3, where bytes are modified rather than individual bits. However, one of these models will produce a feasible attack. If bits are only changed in one direction, i.e. the data resetting model, it is sufficient to know in which direction the bits change. For example, if a byte of d is set to zero, each of the possible 2^8 values for S' will be unique.

Another method of avoiding this problem is described in [41], and is repeated here in terms of faults in the private exponent used in RSA. Suppose that the i-th byte of d, referred to as $d[i]$, is randomised to become $\widetilde{d[i]}$, then we have:

$$d[i] + \gamma = \widetilde{d[i]}$$

where γ represents the effect of the induced fault. Both $d[i]$ and $\widetilde{d[i]}$ are individual bytes of a larger value, i.e. the private exponent, and therefore $d[i], \widetilde{d[i]} \in \{0, 1, \ldots, 255\}$. The effect of the fault γ has to allow $d[i]$ and $\widetilde{d[i]}$ to take any values in $\{0, 1, \ldots, 255\}$, so $\gamma \in \{-255, -254, \ldots, 255\}$, where $\gamma = 0$ is represents no fault in $d[i]$. However, in practice $\gamma \in \{-d[i], -d[i] + 1, \ldots, 255 - d[i]\}$.

From the two bit fault example, given above, it can be seen that the integer representation of the fault can be determined, as

$$M^{d+2^6+2^5} = M^{d+2^7-2^5} = M^{d+96}.$$

An attacker can therefore determine the integer change produced by a fault but not the exact bits that have changed. The integer value of a fault will, however, restrict the value that $d[i]$ can take. For example, if $\gamma = 96$, then

$$d[i] + 96 = \widetilde{d[i]}.$$

Note that $\widetilde{d[i]}$ is $\in \{0, 1, \ldots, 255\}$, and therefore $d[i]$ is $\in \{0, 1, \ldots, 159\}$ (i.e. the maximum integer that $d[i]$ could take is $255 - 96 = 159$).

In [41] the authors then consider two faults, γ and γ', where they seek to maximise the difference between the two faults. If we define $\gamma < \gamma'$, then if $\gamma' - \gamma = 255$

the byte $d[i]$ can be uniquely determined and will be equal to $-\gamma$. This is because there will only be one value of $d[i]$ that will allow the faults γ and γ' to occur.

In practice, it may not be possible to find two faults with a difference of 255 within a reasonable amount of time, as each fault injection requires a certain amount of effort. In [41] the statistically expected amount of faults required to reduce the possible values of $d[i]$ to n hypotheses are given for $1 \leq n \leq 8$. This information is repeated in Table 9.1.

Table 9.1: The number of faulty signatures required to identify one byte.

Number of hypotheses for $d[i]$	Faulty Signatures Required
1	384
2	213
3	149
4	115
5	94
6	79
7	69
8	60

This process could be repeated for each byte of the private exponent to derive its value, or reduce the number of possibilities to aid an exhaustive search.

9.5 Faults in the Modulus

We next describe two major classes of attack based on injecting faults into the modulus N before it is used to generate or verify a signature.

9.5.1 Forging a Signature

An attack of this type is presented in [97]. This attack involves changing the modulus N to N', where N' is prime and e (the public RSA exponent) is coprime to $\phi(N') = N'-1$. Since N' is prime, finding a d such the $ed \equiv 1 \pmod{N'-1}$ is straightforward. The attacker then constructs a fraudulent signature using this value of d with N', and manipulates the card to use the modified N' when verifying the signature.

Given that e is often chosen to be prime, obtaining a value N' such that N' is coprime to e is trivial. However, obtaining a value for N' that is itself prime is more

complex. For a random change to N, assuming N' is odd, the probability that N' is prime is given in [74] as,

$$\Pr(N' \text{ is prime}) \approx \frac{2}{\ln N'} > \frac{2}{\ln 2^n} \equiv \frac{2}{n \ln 2}$$

if N' is bounded by $2^{n-1} \leq N' < 2^n$. A more complete analysis is given in [97].

This attack was extended in [74] to cover the case where N' is any value that is easy to factorise. For example, when N' is smooth it can be readily factorised. This involves slightly more work than the attack detailed in [97], but remains computationally trivial. The bound for the largest factor in N' is, arbitrarily, set to 2^{10} in [74] but a higher bound is certainly acceptable. This increases the possibility of being able to implement this attack, as the amount of values that N' can take increases. The exact details of this more general attack are beyond the scope of this thesis.

In the case where N can be directly modified this attack is trivial to implement. However, this is not possible in the case of a smart card. An attacker could potentially target the key when it is being transferred from EEPROM to RAM before the RSA algorithm is executed. An example of such an attack is described in Section 6.3, where parts of the key being transferred in RAM are set to 0. Given the fault models given in Section 5.3, if a byte, or bytes, can be randomised, the attack becomes potentially feasible.

The problem arises in knowing what fault has been injected *a priori*, since factorising N' (and hence computing an appropriate d) requires its value to be known. An attacker needs to predict the effect of the fault mechanism at the point in time that it is applied, factorise N', and then generate a signature which will be accepted as valid if the appropriate fault is induced. This signature is then presented when conducting the attack. Such a procedure is likely to be overly complex in the case of smart cards, especially where desynchronisation countermeasures are implemented (described in Sections 7.2 and 7.3).

9.5.2 Recovering the Key

A second class of attack based on injecting faults into the modulus is given in [25], in which the aim is to recover the secret exponent d inducing a target to verify a faulty signature. As previously, a fault is injected to transform N to N', so that N' is uniformly distributed across the integers in the interval $[1, 2^{\log_2 N}]$.

A series of pairs of plaintexts M_i and signatures S_i are gathered, each of which is assumed to have a different fault. So for each pair there will be a corresponding, and unknown, N'_i. These pairs are then used to determine d by determining $d \bmod p_k$ for some small prime powers p_k. When the product $T = \prod_k p_k$ exceeds N and therefore also $\phi(N)$, d can be recovered by using the Chinese Remainder Theorem. The algorithm for deriving $d \bmod p_k$ is shown in Algorithm 9.1 and is taken from [25].

Algorithm 9.1: Predicting $d \bmod p$ by counting

Input: $p = q^f$, a small power of a prime
Output: a prediction for $d_p = d \bmod p$

for $i \leftarrow 0$ **to** $p - 1$ **do** $\texttt{count}[i] \leftarrow 0$

$r \leftarrow 2p + 1$
while r *is not prime* **do**
    $r \leftarrow r + p$
end

foreach (M_i, S_i) **do**
    if $(r \not| M_i) \wedge (r \not| S_i)$ **then**
    end
    if $(p|$ ***order of*** $M_i \bmod r) \wedge (DL(M_i, S_i, r, p)$ ***exists***$)$ **then**
        $\texttt{count}[DL(M_i, S_i, r, p)] + +$
    end
end

return *Maximum from* $(\texttt{count}[0], \ldots, \texttt{count}[p-1])$

The function $DL(M_i, S_i, r, p)$ used in Algorithm 9.1 comes from the discrete logarithm of S_i with respect to the basis $(M_i \bmod r) \bmod p$. This value is either an integer modulo p or does not exist, which means that S_i is not a power of $M_i \bmod r$.

A simulation of this attack is given in [25], in which 25000 faults are required to derive 512 bits of residue information about d. This is enough to recover a 1024 bit key with a small public exponent. While this attack is powerful as it does not

require very specific faults, it is not practical because of the massive amount of information required.

A further attack based on building a dictionary for possible values of N' is also discussed in [25], which is a more efficient method of implementing the attack as more information can be derived. The difficulty in this method is predicting and applying the fault that needs to be injected to successfully implement the attack. The details of this attack are beyond the scope of this thesis.

9.6 Exponentiation Algorithms

Many different algorithms can be used to calculate the modular exponentiation algorithm required for RSA. All the algorithms described in this section assume that the exponent is treated in a left to right manner. Some of the algorithms have a version that treats the exponent in a right to left manner, but these are omitted for brevity. In practice, a large number of algorithms cannot be implemented on smart cards, as the amount of available memory does not usually allow numerous intermediate values to be stored in RAM. However, this problem is starting to disappear as smart cards with large amounts of RAM are appearing on the market [71].

The manipulation of large numbers is usually performed using a coprocessor (see Section 3.2.1), as implementing a multiplication on an eight-bit platform would not give the desired performance level. Within a cryptographic coprocessor there are usually only a limited number of registers available to store values. The transfer of data from RAM to the coprocessor that manipulates the numbers is often an expensive process, as all the information has to travel across a bus that is small in comparison to the size of the number being moved, although this is less of a problem with the more recent 32-bit chips. It is therefore desirable to design the exponentiation algorithm to use as few registers as possible. In the algorithms given in this chapter, the registers are expressed as R followed by a number. This makes it straightforward to see how many registers are required to implement each algorithm. Variables that do not need to be kept within a coprocessor are given arbitrary names.

The simplest algorithm is the square and multiply algorithm [67], given in Algorithm 9.2. It has already been shown in Sections 4.2 and 9.1 that this algorithm is insecure when the difference between a squaring operation and a multiplication can be determined. This algorithm uses two registers and has an expected number of operations of $1.5 \log_2 d$, i.e. each bit of d requires one square and half of the bits of d will require an extra multiplication. This is because d can be assumed to have half its bits set to 1, as it is essentially a random value.

Algorithm 9.2: The Square and Multiply Algorithm

Input: $M, d = (d_x, d_{x-1}, \ldots, d_0)_2, N$
Output: $C = M^d \bmod N$

$R_0 \leftarrow 1$
$R_1 \leftarrow M$
for $i \leftarrow x$ **to** 0 **do**
    $R_0 \leftarrow {R_0}^2 \bmod N$
    if $(d_i = 1)$ **then**
        $R_0 \leftarrow R_0 \cdot R_1 \bmod N$
    end
end
return R_0

This algorithm can be optimised by precalculating $M^t \bmod N$ for some small values of t. One example of this is also given in Algorithm 9.3, which is a specific case of the sliding-window exponentiation given in [67]. In this algorithm the current working value is squared until a bit of d is equal to 1, at which point a multiplication by M or M^3 will take place, depending on whether one or two consecutive bits are set to 1 in the exponent. This algorithm has an expected number of operations equal to $1.333 \log_2 d$ [59], and uses three registers.

Another version of this type of optimisation is given in Algorithm 9.4, which is a specific case of the left to right k-ary exponentiation scheme given in [67]. This is based on the same principle as Algorithm 9.3, but treats the exponent in blocks of two bits. This assumes the exponent has an even number of bits, although the algorithm can be readily adjusted for an odd number of bits by setting d_x to zero. This algorithm has an expected number of operations equal to $1.375 \log_2 d$, and uses three registers.

Algorithm 9.3: The (M,M^3) Algorithm

Input: $M, d = (d_x, d_{x-1}, \ldots, d_0)_2, N$
Output: $C = M^d \bmod N$

$R_0 \leftarrow 1$
$R_1 \leftarrow M$
$R_2 \leftarrow M^3 \bmod N$
$i \leftarrow x$
while $i \geq 0$ **do**
  if $(d_i = 0)$ **then**
    $R_0 \leftarrow {R_0}^2 \bmod N$
    $i \leftarrow i - 1$
  end
  else if $(d_i = 1) \wedge ((d_{i-1} = 0) \vee (i = 0))$ **then**
    $R_0 \leftarrow {R_0}^2 \bmod N$
    $R_0 \leftarrow R_0 \cdot R_1 \bmod N$
    $i \leftarrow i - 1$
  end
  else if $(d_i = 1) \wedge (d_{i-1} = 1)$ **then**
    $R_0 \leftarrow {R_0}^2 \bmod N$
    $R_0 \leftarrow {R_0}^2 \bmod N$
    $R_0 \leftarrow R_0 \cdot R_2 \bmod N$
    $i \leftarrow i - 2$
  end
end
return R_0

Both of these algorithms increase the performance of the algorithm, but cannot be regarded as secure against Simple Power Analysis, as the pattern of squares and multiplications will still vary depending on the bits of d. It is not likely that an attacker will be able to distinguish between a multiplication by M or $M^3 \bmod N$, but an attacker will still be able to derive the majority of the key using SPA. It has also been suggested that a detailed SPA can detect the **if-else** structure, and detect which branch has been taken [60].

9.6.1 Side Channel Resistant Algorithms

The simplest countermeasure against side channel analysis is to try to remove the difference between a square and a multiplication by avoiding the calculation of a square with the cryptographic coprocessor. This means that, to square a value x, the calculation of $x^2 \bmod N$ is replaced with $x \cdot x \bmod N$. This idea is put

Algorithm 9.4: The 2-ary Algorithm

Input: $M, d = (d_x, d_{x-1}, \ldots, d_0)_2, N$
Output: $C = M^d \bmod N$

$R_0 \leftarrow 1$
$R_1 \leftarrow M$
$R_2 \leftarrow M^3 \bmod N$
$i \leftarrow x$
while $i \geq 0$ **do**
  $R_0 \leftarrow {R_0}^2 \bmod N$
  if $(d_i = 0) \wedge (d_{i+1} = 0)$ **then**
    $R_0 \leftarrow {R_0}^2 \bmod N$
  end
  else if $(d_i = 0) \wedge (d_{i+1} = 1)$ **then**
    $R_0 \leftarrow {R_0}^2 \bmod N$
    $R_0 \leftarrow R_0 \cdot R_1 \bmod N$
  end
  else if $(d_i = 1) \wedge (d_{i+1} = 0)$ **then**
    $R_0 \leftarrow R_0 \cdot R_1 \bmod N$
    $R_0 \leftarrow {R_0}^2 \bmod N$
  end
  else if $(d_i = 1) \wedge (d_{i+1} = 1)$ **then**
    $R_0 \leftarrow {R_0}^2 \bmod N$
    $R_0 \leftarrow R_0 \cdot R_2 \bmod N$
  end
  $i \leftarrow i - 2$
end
return R_0

forward in [30], where two instructions are defined as side channel equivalent if they are indistinguishable through side channel analysis, and algorithms are termed side channel atomic if the algorithm can be broken down into indistinguishable blocks. This principle, applied to the square and multiply algorithm, is demonstrated in Algorithm 9.5. This has the same expected number of operations and register usage as the square and multiply algorithm.

This approach will make it harder to distinguish between a square and a multiplication, as the coprocessor makes no distinction between them. However, for the remainder of this section it will be assumed that an attacker is able to make the distinction between a square and a multiplication. The use of an index to find the desired register is also more secure against SPA than a series of **if-else** statements.

Algorithm 9.5: Side Channel Atomic Square and Multiply Algorithm

Input: $M, d = (d_x, d_{x-1}, \ldots, d_0)_2, N$
Output: $C = M^d \bmod N$

$R_0 \leftarrow 1$
$R_1 \leftarrow M$
$i \leftarrow x$
$k \leftarrow 0$
while $i \geq 0$ **do**
    $R_0 \leftarrow R_0 \cdot R_k \bmod N$
    $k \leftarrow k \oplus d_i$
    $i \leftarrow i - \neg k$
end
return R_0

The evaluation of **if-else** commands will take a different amount of time depending on which of the **if-else** statements is true. The evaluation of an index will take a constant amount of time, as this is usually implemented as a variable offset from a fixed point in RAM. In the case where registers are used in a cryptographic coprocessor, the case under consideration, this index could be used to select the relevant register. However, the exact details are totally dependent on the design of the coprocessor and are therefore beyond the scope of this thesis.

A modification to the square and multiply algorithm is proposed in [35] that changes it to always calculate a multiplication, and is detailed in Algorithm 9.6. In this case a dummy register is included that takes the result of the multiplication when $d_i = 0$. This modification increases the number of operations to $2 \log_2 d$ and uses three registers.

This is a big reduction in efficiency compared to the square and multiply algorithm, but is secure against SPA as the only change in execution, that depends on the key, is to which register the result of the multiplication is written.

This provides an algorithm that is secure against Simple Power Analysis but not Statistical Power Analysis. In order to prevent Statistical Power Analysis the data being manipulated needs to be masked at every point in the algorithm. In [55] a method of randomising the calculation is proposed that is analogous to the data whitening countermeasure, as described in Section 7.4. This involves modifying

Algorithm 9.6: The Square and Multiply Always Algorithm

Input: $M, d = (d_x, d_{x-1}, \ldots, d_0)_2, N$
Output: $C = M^d \bmod N$

$R_0 \leftarrow 1$
$R_1 \leftarrow 1$
$R_2 \leftarrow M$
for $i \leftarrow x$ **to** 0 **do**
    $R_1 \leftarrow {R_1}^2 \bmod N$
    $R_{d_i} \leftarrow R_1 \cdot R_2 \bmod N$
end
return R_1

each parameter with a small random value and then removing it at the end of the algorithm. Each random value used has the effect that each multiplication is randomised by a value whose effect is equivalent to a multiplication by 1 mod N and is therefore easily removed at the end. This algorithm is detailed in Algorithm 9.7.

Algorithm 9.7: Randomised Exponentiation Algorithm

Input: $M, d, N,$ small random values r_1, r_2, r_3
Output: $C = M^d \bmod N$

$M' \leftarrow M + r_1 \cdot N$
$d' \leftarrow d + r_2 \cdot \lambda(N)$
$N' \leftarrow r_3 \cdot N$
$C' \leftarrow {M'}^{d'} \bmod N'$
$C \leftarrow C' \bmod N$
return C

The bit length of the random values used is often determined by the algorithm and/or the architecture used. For example, if Montgomery multiplication [73] is used, the most natural choice would be to have a random value whose bit length is the same as the basic data unit (see Section 9.8 for a definition of Montgomery multiplication). In the case of cryptographic coprocessors the bit length of the random values will typically be dictated by the size of the registers available. A coprocessor will typically be designed for a specific modulus bit length, with extra bits available to allow some flexibility in how they are used. The role of these random values is to make the branch prediction described in Section 9.2 prohibitively time-consuming. This should be possible if r_1, r_2, and r_3 are independent random values.

The change in d will make any statistical analysis meaningless, as an attacker will have to guess the mask for each acquisition, and is further complicated by the modifications to M and N.

A novel combination of Algorithms 9.4 and 9.7 can be used to provide a secure modular exponentiation algorithm. This is described in Algorithm 9.8, and is one of the contributions of this thesis. As previously, each input variable is multiplied by a small random value to prevent Statistical Power Analysis. The significant change this allows is in the case where $(d'_i = 0)$ and $(d'_{i+1} = 0)$, as a multiplication by a value equivalent to 1 mod N can be conducted. Such a change could not be used before, as the presence of 1 mod N may be detected by the coprocessor, resulting in no computation taking place. When values are loaded into the coprocessor it would be natural for the values to be reduced modulo N once loaded, as modular multiplication by a value greater than the modulus would be inefficient. A multiplication by one could either be detected by the coprocessor and ignored, or take place in a very short period of time, which may be visible in the power consumption when compared to a multiplication with a value whose bit length is similar to the modulus. As the modulus is increased to $r_3 \cdot N$ the calculation can be multiplied by a value equivalent to 1 mod N.

The other major change in this algorithm applies when $(d'_i = 1)$ and $(d'_{i+1} = 0)$; in this case the multiplication is by M'^2 mod N'. This allows the square function to always take place twice before the multiplication. This means that each loop of the algorithm contains two squares and a multiplication. As with Algorithm 9.4, this assumes that the exponent comprises an even number of bits, and the most significant bit needs to be set to zero if this is not the case. The expected performance of this algorithm is $1.5 \log_2 d'$ operations, and it uses five registers. This returns us to the efficiency of the square and multiply algorithm, but requires more registers. This algorithm will also take the same amount of operations irrespective of the value of d', and should therefore be immune to timing attacks and Simple Power Analysis.

Algorithm 9.8 can be made more efficient by changing the algorithm so that it analyses three bits per loop rather than two. This would require eight registers to

Algorithm 9.8: Secure 2-ary Algorithm

Input: $M, d = (d_x, d_{x-1}, \ldots, d_0)_2, N$, small randoms r_1, r_2, r_3, r_4
Output: $C = M^d \bmod N$

$M' \leftarrow M + r_1 \cdot N$
$d' \leftarrow d + r_2 \cdot \lambda(N)$
$N' \leftarrow r_3 \cdot N$
$R_0 \leftarrow r_4 \cdot N + 1 \bmod N'$
$R_1 \leftarrow M'$
$R_2 \leftarrow M'^2 \bmod N'$
$R_3 \leftarrow M'^3 \bmod N'$
$R_4 \leftarrow 1$
$i \leftarrow \lfloor \log_2 d' \rfloor + 1$
while $i \geq 0$ **do**
    $R_4 \leftarrow {R_4}^2 \bmod N'$
    $R_4 \leftarrow {R_4}^2 \bmod N'$
    $R_4 \leftarrow R_4 \cdot R_{2d'_i + d'_{i-1}} \bmod N'$
    $i \leftarrow i - 2$
end
$R_4 \leftarrow R_4 \bmod N$
return R_4

contain all the precomputed values for M'^i, where $0 \leq i \leq 7$. Each loop would then consist of three squarings and one multiplication. This would therefore require nine registers and have an expected complexity of $1.333 \log_2 d'$ operations. This is probably not practical on a coprocessor because of the amount of registers required, but could potentially be implemented on a 32-bit chip where the calculation is conducted by the CPU. However, the memory requirements are probably still prohibitive, as a certain amount of memory needs to be available for the operating system.

This idea can be further extended to n bits, requiring $2^n + 1$ registers and giving an expected performance of $\left(1 + \frac{1}{n}\right) \log_2 d'$ operations. However, given that analysing three bits in each loop is likely to be prohibitively complex, calculating a modular exponentiation with larger values of n is unlikely to be practical.

9.6.2 Fault Resistant Algorithms

Several different approaches can be used to protect a modular exponentiation against fault attacks. The simplest mechanism for detecting a fault attack is to duplicate data and code, and then check that the results are identical. Because of the nature

of physical perturbations, different executions of the same fault injection are likely to have different consequences. This in turn results in a difference between one execution and its duplicate, which can be detected by the implementation. However, this would typically double the execution time of an algorithm, and is therefore not desirable from a performance perspective.

Another protection technique is to include redundant code as a verification stage. This usually consists of executing the inverse of an algorithm instead of repeating it. This prevents two identical faults from bypassing the verification stage of the countermeasures, which is a potential problem with simply repeating code.

The RSA signature scheme provides a good example of where a verification stage can be efficiently implemented. This would involve calculating $S = M^d \bmod N$ followed by $M' = S^e \bmod N$. The consistency check would be to verify that $M' = M$. This technique offers superior performance to the duplication method, as the public exponent e is usually much smaller than the secret exponent d.

Another alternative is the generic method, described in Section 9.8, that involves applying a checksum that can easily be applied to a modular exponentiation.

9.6.3 Attacking Side Channel Resistant Algorithms

A fault attack is presented in [57] that uses the structure of the square and multiply always algorithm (see Algorithm 9.6) to derive the exponent being used. If a fault is injected in a multiplication, this will change the resulting ciphertext if it used, i.e. the relevant bit of the exponent is equal to one. This can be done for each multiplication in succession, to reveal each bit of the exponent. This is similar in nature to the attack on S-boxes presented in Section 8.4.1. It is therefore not advisable to have dummy functions or registers that are only present to prevent side channel analysis, as this can be detected using a fault attack.

9.7 Secure Exponentiation Using CRT

Signatures are rarely generated using a simple modular exponentiation, as the signature can be generated using the Chinese Remainder Theorem (see Section 2.3) four times more quickly. If this approach is used, then the algorithm must be protected

in a different way to that described in Section 9.6, as the algorithm has different characteristics.

9.7.1 Side Channel Resistant Algorithms

The calculation of RSA using CRT can be protected against Simple Power Analysis in the same way as proposed for RSA in Section 9.6, i.e. the exponentiations in the CRT algorithm can be performed using the Algorithms given in Section 9.6. The rest of the algorithm can then be executed using the constant time and constant execution countermeasures given in Section 7.1.

The algorithm needs to be randomised to protect it against Statistical Power Analysis. This can be achieved by using a slightly more complex version of Algorithm 9.7. Such an approach is described in Algorithm 9.9 [105], in which all the entry variables, except the message, are multiplied by a small random value to mask the actual value being used. These random masks are then removed at the end by taking the result modulo $p \cdot q$.

Algorithm 9.9: Randomised Exponentiation Algorithm using CRT

Input: $M, p, q, d, q^{-1} \bmod p$, small random values r_1, r_2, r_3, r_4
Output: $S = M^d \bmod p \cdot q$

$p' \leftarrow p \cdot r_1$
$d'_p \leftarrow (d \bmod (p-1)) + r_2 \cdot (p-1)$
$q' \leftarrow q \cdot r_3$
$d'_q \leftarrow d \bmod (q-1) + r_4 \cdot (q-1)$
$S'_p \leftarrow (M \bmod p')^{d'_p} \bmod p'$
$S'_q \leftarrow (M \bmod q')^{d'_q} \bmod q'$
$S \leftarrow (S'_q + (((S'_p - S'_q) \cdot q^{-1} \bmod p) \bmod p') \cdot q) \bmod (p \cdot q)$
return S

9.7.2 Fault Resistant Algorithms

Several different methods have been proposed to protect RSA using CRT from fault attacks, other than simply repeating the entire function or its inverse and verifying the result. The first method of protecting the RSA algorithm using the Chinese Remainder Theorem was proposed in [98]. This essentially involved randomising the modular exponentiations by multiplying the primes used in RSA by a small

random prime, and then verifying that the results are coherent. This protects the modular exponentiations, but does not protect the end calculation, when S_p and S_q are combined. The attack described in [23] will therefore still be possible by attacking the recombination process. An implementation of the original attack and an attack on the recombination are described in [8]; further countermeasures are also proposed to secure the entire calculation, as shown in Algorithm 9.10.

Algorithm 9.10: Verifying the computation of the RSA algorithm using CRT

Input: $M, p, q, d, q^{-1} \bmod p$, a small random prime r
Output: $S = M^d \bmod (p \cdot q)$ or error

$p' \leftarrow p \cdot r$
$d'_p \leftarrow d \bmod ((p-1) \cdot (r-1))$
$S'_p \leftarrow (M \bmod p')^{d'_p} \bmod p'$
if $\neg(p' \pmod p \equiv 0) \wedge (d'_p \pmod{p-1} \equiv d_p)$ **then**
    return error
end

$q' \leftarrow q \cdot r$
$d'_q \leftarrow (d \bmod (q-1)) \cdot (r-1)$
$S'_q \leftarrow (M \bmod q')^{d'_q} \bmod q'$
if $\neg(q' \pmod q \equiv 0) \wedge (d'_q \pmod{q-1} \equiv d_q)$ **then**
    return error
end

$S_p \leftarrow S'_p \bmod p$
$S_q \leftarrow S'_q \bmod q$
$S \leftarrow S_q + ((S_p - S_q) \cdot q^{-1} \bmod p) \cdot q$
if $\neg((S \bmod p = S_p) \wedge (S \bmod q = S_q))$ **then**
    return error
end

$S_{pr} \leftarrow S'_p \bmod r$
$d_{pr} \leftarrow d'_p \bmod r - 1$
$S_{qr} \leftarrow S'_q \bmod r$
$d_{qr} \leftarrow d'_q \bmod r - 1$

if $S_{pr}^{d_{qr}} \equiv S_{qr}^{d_{pr}} \pmod r$ **then**
    return S
else
    return error
end

In [111] the concept of infective computing is presented, where if a fault is produced in one exponentiation it would have an effect on a second exponentiation.

This approach was proposed in an effort to avoid having decisional testing for error detection. An attack on this system was presented in [110]. Another algorithm based on the same idea was proposed in [20], and some attacks were presented in [106], although not all of the attacks seem to be valid [21].

A further algorithm is given in [32], as detailed in Algorithm 9.11, that builds on the algorithms proposed in [20, 111]. This method functions by generating three values c_1, c_2 and k that are used to detect an error; when no error is present then $c_1 = c_2 = k = 1$. The last step is to raise the output to the power of k that will change the output so that no key information can be derived from an incorrect result.

Algorithm 9.11: Infective computation of RSA

Input: M, p, q, d, small random coprime values r_1, r_2, a small random value r_3

Output: $S = M^d \bmod p \cdot q$

$p' \leftarrow p \cdot r_1$
$d_p \leftarrow d \bmod (p-1)$
$q' \leftarrow q \cdot r_2$
$d_q \leftarrow d \bmod (q-1)$
$i_{q'} = {q'}^{-1} \bmod p'$
$S'_p \leftarrow M^{d_p} \bmod p'$
$S'_q \leftarrow M^{d_q} \bmod q'$
$s_1 \leftarrow (M^{d_p \bmod \phi(r_1)}) \bmod r_1$
$s_2 \leftarrow (M^{d_q \bmod \phi(r_2)}) \bmod r_2$
$S' \leftarrow S'_q + q' \cdot i_{q'} \cdot (S'_p - S'_q) \bmod p'$
$c_1 \leftarrow (S' - s_1 + 1) \bmod r_1$
$c_2 \leftarrow (S' - s_2 + 1) \bmod r_2$
$k \leftarrow \lfloor (r_3 c_1 + (2^{\log_2 r_3} - r_3) \cdot c_2) / 2^{\log_2 r_3} \rfloor$
$S \leftarrow {S'}^k \bmod (p \cdot q)$

return S

9.8 Secure Montgomery Multiplication

The basic function that is used to calculate the modular exponentiation for RSA is modular multiplication. One of the most common methods of calculating a modular multiplication is using Montgomery multiplication [73] because of its efficiency,

especially as it can be parallelised in hardware. A description of Montgomery multiplication is given in Algorithm 9.12 [67]. Here b is the size of the basic data unit, usually a machine word, and z is the number of words in the representation of N, X and Y.

Algorithm 9.12: Montgomery Multiplication

Input: $X = (x_{z-1}, \ldots, x_1, x_0)_b$, $Y = (y_{z-1}, \ldots, y_1, y_0)_b$, $N = (n_{z-1}, \ldots, n_1, n_0)_b$, $R = b^z$ with $gcd(N, b) = 1$, and $N' = -N^{-1} \mod b$

Output: $A = XYR^{-1} \mod N$

$A \leftarrow 0$ (Notation $A = (a_z, a_{z-1}, \ldots, a_1, a_0)_b$.)

for $i = 0$ **to** $z - 1$ **do**
  $u_i \leftarrow (a_0 + x_i y_0)N' \mod b$
  $A \leftarrow (A + x_i Y + u_i N)/b$
end

if $A \geq N$ **then** $A \leftarrow A - N$

return A

Montgomery multiplication does not return the simple product of X and Y modulo N. The algorithm actually returns $XYR^{-1} \mod N$, where $R^{-1} \mod N$ is introduced by the algorithm ($R = b^z$). In order to use Montgomery multiplication X and Y need to be converted to their Montgomery representation, i.e. $\widetilde{X} \leftarrow XR \mod N$ and $\widetilde{Y} \leftarrow YR \mod N$. Then, when $\widetilde{X}$ and $\widetilde{Y}$ are multiplied together using Montgomery multiplication, the result is $XYR \mod N$. This is useful when calculating numerous multiplications, e.g. an exponentiation, as the input can be changed to the Montgomery representation and the result of each Montgomery multiplication will also be in the Montgomery representation.

9.8.1 Side Channel Resistant Algorithms

The Montgomery multiplication is potentially vulnerable to a side channel attack, since the last step of the algorithm involves a conditional subtraction [60]. This can potentially be detected by observing the power consumption, where the presence, or absence, of the last subtraction may be visible. Attacks based on this reduction have been applied to RSA, as the time taken to calculate an exponentiation $M^d \mod q$ is proportional to how close M is to q.

As M increases the number of extra reductions in a modular exponentiation increases, that need to be performed as part of an RSA. At exact multiples of p or q the number of extra reductions drops dramatically, providing information on the factors of N.

A simple modification can be made to the algorithm to remove this side channel, namely to always calculate the extra reduction, as shown in Algorithm 9.13. In the case where $A < M$, the value of B will not contain any useful information, but an attacker will not know which value has been returned.

Algorithm 9.13: Time Constant Montgomery Multiplication

Input: $X = (x_{z-1}, \ldots, x_1, x_0)_b$, $Y = (y_{z-1}, \ldots, y_1, y_0)_b$,
$N = (n_{z-1}, \ldots, n_1, m_0)_b$, $R = b^z$ with $\gcd(N, b) = 1$, and
$N' = -N^{-1} \bmod b$

Output: $A = xyR^{-1} \mod m$

$A \leftarrow 0$ (Notation $A = (a_z, a_{z-1}, \ldots, a_1, a_0)_b$.)

for $i = 0$ **to** $z - 1$ **do**
  $u_i \leftarrow (a_0 + x_i y_0)N' \mod b$
  $A \leftarrow (A + x_i Y + u_i N)/b$
end

$B \leftarrow A - N$

if $A \geq N$ **then**
  return B
else
  return A
end

However, this is potentially vulnerable to a fault attack, as described in Section 9.6.3, where a fault is injected into the last subtraction. If this changes the result then the reduction has been used. A similar attack to that proposed in [57] can therefore be derived from this observation.

In [46, 107, 108] it is pointed out that the final modular subtraction can be omitted, as even without this subtraction, the output of a Montgomery multiplication will be less than $2N$. Furthermore, if both input values, i.e. X and Y, are less than $2N$, then the output of the Montgomery multiplication will also be less than $2N$. A series of Montgomery multiplications can therefore take place without any need for a conditional subtraction at each stage. The final Montgomery multiplication by

$R^{-1} \bmod N$ will produce a result less than N (as shown in [107, 108]). This removes the side channel without making the algorithm vulnerable to a fault attack. In the following sections the final conditional subtraction will be included for completeness but is not taken into account when considering the security of Montgomery multiplication.

9.8.2 Fault Resistant Algorithms

A method of securing modular arithmetic against fault attacks is described in [109], based on applying a checksum to the input and output variables. This is potentially a more efficient solution than the fault resistant algorithms described in Section 9.7.2, as less computation is involved in deriving the checksum than the other techniques described there.

One of the most natural checksum functions is the modulo operation with respect to a modulus D. This enables each function being calculated to be verified after calculation. This is similar to the idea behind Algorithm 9.10, but each modification of a variable can be checked. The check function is of the form

$$f(A) = A \bmod D$$

where $D > 1$ and is coprime to $2b$, where b is the word size of the processor. If a step of the algorithm requires the computation of $A \otimes B \bmod N$, for some function $\otimes$, then the check function needs to satisfy.

$$f(A \otimes B) = f(A) \otimes f(B)$$

This typically means that we need to ensure that D divides N, and needs to be adjusted so that it takes into account the modular reductions. If there are Q subtractions of N, then the check function can be written as.

$$f(A \otimes B) = f(A) \otimes f(B) - f(Q) \times f(N) \bmod D$$

which is valid for modular arithmetic functions such as addition, multiplication and exponentiation; i.e. this could be used to verify all the steps of the calculation of RSA

using CRT. This method is not secure against all fault attacks, as the checksum is valid throughout the calculation of $A \otimes B$. In the case of a multiplication, if the loop terminates early the checksum will be valid, as the number of modular reductions counted will be correct for the reduced number of iterations.

We next consider a specific example of the check function defined in [109], for the case where the modulus is set to $b-1$. This is more convenient for calculations, and will allow for more efficient code. This check function is defined as:

$$f(X) = \sum_{i=0}^{k} x_i \bmod (b-1) = X \bmod (b-1),$$

where $X = (x_k, \ldots, x_1, x_0)_b$. This checksum function has the desired property that any change to the data units of X should result in a different value of $f(X)$. It also has the following properties:

- $f(X) + f(Y) = f(X+Y) \bmod (b-1)$,
- $f(X) \cdot f(Y) = f(X \cdot Y) \bmod (b-1)$,
- $f(X/b) = f(X) \bmod (b-1)$, if b divides X.

The security provided by a checksum depends on its size. It is assumed that the value of b is reasonably large, e.g. a 32-bit word. Smart cards have typically been based around eight-bit chips, but modern smart cards are starting to use 32-bit technologies [51, 71], so this is not an unreasonable assumption.

9.8.3 Modified Montgomery Algorithm with Redundancy

Using the checksum function $f(X) = X \bmod (b-1)$, a modified Montgomery algorithm with redundancy is proposed in Algorithm 9.14, where the variables in **bold** have been introduced as checksums for redundancy purposes.

One trivial way of adding redundancy is to include a redundant instruction for each instruction of the algorithm. In the proposed algorithm, redundant code for the generation of u_i is not required, as the same value of u_i can be used in the modular multiplication and the redundant code. This avoids having an extra calculation for the check function, thereby increasing the efficiency of the algorithm.

Algorithm 9.14: Montgomery Multiplication with Redundancy

Input: $X = (x_{z-1}, \ldots, x_1, x_0)_b$, $\mathbf{X} = X \bmod (b-1)$, $Y = (y_{z-1}, \ldots, y_1, y_0)_b$, $\mathbf{Y} = Y \bmod (b-1)$, $N = (n_{z-1}, \ldots, n_1, m_0)_b$, $\mathbf{N} = N \bmod (b-1)$, $R = b^z$ with $\gcd(N, b) = 1$, and $N' = -N^{-1} \bmod b$

Output: $A = XYR^{-1} \bmod N$ and $\mathbf{A}$

$A \leftarrow 0$ (Notation $A = (a_n, a_{n-1}, \ldots, a_1, a_0)_b$.)
$\mathbf{A} \leftarrow 0$ (Notation $\mathbf{A} = (\mathbf{a_0})_b$.)

for $i = 0$ **to** $z - 1$ **do**
    $u_i \leftarrow (a_0 + x_i y_0) N' \bmod b$
    $A \leftarrow (A + x_i Y + u_i N)/b$
    $\mathbf{A} \leftarrow \mathbf{A} + u_i \mathbf{N} \bmod (b-1)$
end

$\mathbf{A} \leftarrow \mathbf{A} + \mathbf{XY} \bmod (b-1)$

if $A \geq N$ **then** $\mathbf{A} \leftarrow \mathbf{A} - \mathbf{N} \bmod (b-1)$
if $A \geq N$ **then** $A \leftarrow A - N$

return A,$\mathbf{A}$

The conditional statement at the end of the algorithm could be a potential vulnerability. With side channel analysis the decision may be detected, leading to information leakage about what has been calculated. This can be avoided by always calculating $A - N$ and deciding which value to return depending on the carry bit generated by $A - N$. Detecting this will require a much more subtle side channel analysis than a naïve implementation, and such an attack could be impossible, depending on the implementation. A more serious problem could be created by a fault attack, in which a fault changes the decision taken by the **if** statement, so that the algorithm returns the valid pair $A - N$, $\mathbf{A} - \mathbf{N}$ instead of A, $\mathbf{A}$ or *vice versa*. It is for this reason that steps five and six are independent **if** statements, as two faults would be required to change both to the false value. However, these steps can be omitted as described in Section 9.8.1.

Redundancy Verification: During the computation, the integrity of the intermediate value A can be verified by checking the relation $\mathbf{A} \equiv A \pmod{(b-1)}$ where $\mathbf{A}$ is the checksum corresponding to A. Such checks can be implemented at any place in the code. These checks can be part of the method used to decide whether or not an answer is given. All the algorithms described in this, and subsequent, sections

assume that the calling function will test the output after the function has returned its result and checksum. If a random fault occurs in the generation of **A**, then the checksum will still be valid with a probability of $1/(b-1)$, which is the best that can be achieved with a checksum of size $b-1$.

Security Analysis: Algorithm 9.14 can be shown to provide a robust defence against fault attacks. The probability of a fault remaining undetected has a probability of $1/(b-1)$ for the majority of the possible faults, where the possible faults are dictated by the three models presented in Section 5.3. The probability of a fault remaining undetected is $2/(b-1)$ for the specific case where an iteration of the **for** loop is omitted. The only major problem is that if X and Y are known, and $(x_j Y + u_j N) \equiv 0 \pmod{(b-1)}$, then it is not possible to detect a fault that causes $A \leftarrow (A + x_i Y + u_i N)/b$ to be omitted during computation. This is covered in more detail in Appendix J.

9.8.4 RSA Computation using CRT with Redundancy

In this section an implementation of the RSA signature generation using CRT with the redundancy technique given above is described. This involves defining secure implementations of the basic building blocks for the algorithm, and then showing how these can be used to give a secure implementation of the RSA signature scheme using CRT.

Modular Multiplication with Redundancy

To build an RSA implementation using Algorithm 9.14, a modular multiplication algorithm with redundancy is required to convert values to their Montgomery representation. This algorithm is a trivial case of adding redundancy to the conventional modular multiplication algorithm, as given in Algorithm 9.15.

Note that the **if** statements at the end of the algorithm involve the same test, but are repeated. This is to a prevent a fault attack in which an attacker targets the evaluation of the **if** function. In this algorithm an attacker would have to target both **if** functions for the checksum to remain valid.

This algorithm is secure against a single fault attack, as there is no common

Algorithm 9.15: Modular Multiplication with Redundancy

Input: $X = (x_{z-1}, \ldots, x_1, x_0)_b$, $\mathbf{X} = X \bmod (b-1)$, $Y = (y_{z-1}, \ldots, y_1, y_0)_b$, $\mathbf{Y} = Y \bmod (b-1)$, $N = (n_{z-1}, \ldots, n_1, m_0)_b$, and $\mathbf{N} = N \bmod (b-1)$

Output: $A = XY \bmod N$ and $\mathbf{A}$

$A \leftarrow 0$
$\mathbf{A} \leftarrow 0$ ($A = (a_{z-1}, \ldots, a_1, a_0)_b$ $\mathbf{A} = (\mathbf{a}_0)_b$)
$\mathbf{q} \leftarrow 0$ ($\mathbf{q} = (\mathbf{q}_0)_b$)

if $Y \geq N$ **then** $\mathbf{Y} \leftarrow \mathbf{Y} - \mathbf{N} \mod (b-1)$
if $Y \geq N$ **then** $Y \leftarrow Y - N$

$\mathbf{A} \leftarrow \mathbf{XY} \mod (b-1)$

for $i = z-1$ **to** 0 **do**
    $A \leftarrow Ab + x_i Y \mod N$
    $\mathbf{q} \leftarrow x_i Y \text{ div } N$
    $\mathbf{A} \leftarrow \mathbf{qN} + \mathbf{A} \bmod (b-1)$
end

return A,$\mathbf{A}$

variable between the computation of the output and its redundancy. While the performance of this algorithm is not good, it is acceptable, as it is only used once for each exponentiation, i.e. to convert the message to its Montgomery representation. Another method of calculating this modular multiplication is given in [109].

Montgomery Exponentiation Algorithm with Redundancy

A Montgomery exponentiation algorithm with redundancy is given in Algorithm 9.16. In this algorithm, the modulus N is assumed to be greater than $R/2$, which is the standard case for RSA moduli. $Hw(e)$ is used to denote the Hamming weight of e, $MulModR(X, \mathbf{X}, Y, \mathbf{Y}, N, \mathbf{N})$ to denote Algorithm 9.15 (which computes XY mod N with redundancy). The function $MontR^*(X, \mathbf{X}, Y, \mathbf{Y}, N, \mathbf{N}, R, N', \mathbf{e})$ is used to denote Algorithm 9.14. The additional parameter $\mathbf{e}$ is included to provide a trace mechanism; each execution of $MontR^*$ will also include the instruction: $\mathbf{e} \leftarrow \mathbf{e} - 1$. At the beginning of the algorithm $\mathbf{e}$ is set to the number of $MontR^*$ calls that will take place during the algorithm, and this will be decremented down to zero as the algorithm executes. If an incorrect number of multiplications has taken place, then $\mathbf{A} + \mathbf{e} \not\equiv A \pmod{(b-1)}$ and the checksum returned will be incorrect.

Algorithm 9.16: Montgomery Exponentiation with Redundancy

Input: $N = (n_{z-1}, \ldots, n_1, n_0)_b$, $\mathbf{N} = N \bmod (b-1)$, integer $1 \leq X < N$, $\mathbf{X} = X \bmod (b-1)$, $e = (e_{k-1}, \ldots, e_1, e_0)_2$, $\mathbf{e} = Hw(e) + k + 1$, $R = b^z$ with $\gcd(N, b) = 1$, and $N' = -N^{-1} \bmod b$

Output: $A = X^e \bmod N$ and $\mathbf{A}$

$A \leftarrow R \bmod N$
$\mathbf{A} \leftarrow b - 1 - \mathbf{N} \bmod (b-1)$

$(\widetilde{r}, \widetilde{\mathbf{r}}) \leftarrow MulModR(R, 1, R, 1, N, \mathbf{N})$
$(\widetilde{X}, \widetilde{\mathbf{X}}) \leftarrow MontR^*(X, \mathbf{X}, \widetilde{r}, \widetilde{\mathbf{r}}, R, N', \mathbf{e})$

for $i = k-1$ **to** 0 **do**
  $(A, \mathbf{A}) \leftarrow MontR^*(A, \mathbf{A}, A, \mathbf{A}, R, N', \mathbf{e})$
  if $e_i = 1$ **then** $(A, \mathbf{A}) \leftarrow MontR^*(A, \mathbf{A}, \widetilde{X}, \widetilde{\mathbf{X}}, R, N', \mathbf{e})$
end

$(A, \mathbf{A}) \leftarrow MontR^*(A, \mathbf{A}, 1, 1, R, N', \mathbf{e})$

return A,$\mathbf{A} + \mathbf{e}$

As Algorithm 9.16 consists of a sequence of calls to Algorithms 9.14 and 9.15, any data randomisation attack or data reset attack would be detected by these two algorithms. Opcode modification attacks against Algorithms 9.14 and 9.15 would also be detected by these algorithms (see Appendix J). Opcode modification attack on the jumps to the subroutines are protected with the trace mechanism $\mathbf{e} \leftarrow \mathbf{e} - 1$, as described above.

Algorithm 9.16 is a square and multiply algorithm with a checksum calculation added, and is not secure against side channel attacks. To address this, other exponentiation techniques can be used (see Section 9.6.1). The redundancy techniques above can be readily applied to side channel resistant complex algorithms, but for clarity the simplest case is given in Algorithm 9.16.

RSA Computation with Redundancy

Using the algorithms described above, an RSA computation using the CRT technique with redundancy is proposed in Algorithm 9.17. Here $ModExpR(x^y \bmod m)$ denotes Algorithm 9.16 that computes $x^y \bmod m$. As there are no common variables in the computation of S and $\mathbf{S}$, this algorithm is secure against a single fault injection attack, assuming that Algorithm 9.16 is secure. The performance of this

algorithm is proportional to the performance of the Montgomery multiplication algorithm, as other overheads are negligible in comparison to the basic multiplication algorithm.

Algorithm 9.17: RSA Computation with Redundancy

Input: $M, \mathbf{M} = M \bmod (b-1), p, \mathbf{p} = p \bmod (b-1), q, \mathbf{q} = q \bmod (b-1), d, v = q^{-1} \bmod p, \mathbf{v} = v \bmod (b-1), (N = p \cdot q)$

Output: S and $\mathbf{S}$

$(S_p, \mathbf{S}_p) \leftarrow ModExpR(M^d \bmod p)$
$(S_q, \mathbf{S}_q) \leftarrow ModExpR(M^d \bmod q)$
$S \leftarrow S_q + (S_p - S_q) \cdot v \cdot q$
$\mathbf{S} \leftarrow \mathbf{S}_q + (\mathbf{S}_p - \mathbf{S}_q) \cdot \mathbf{v} \cdot \mathbf{q}$

return S, $\mathbf{S}$

9.9 Comparing the Fault Protection Methods

Several methods for protecting the calculation of an RSA against fault attacks have been described. In each case we have considered an attacker that can inject one fault into the algorithm. The security of these algorithms also needs to be evaluated against an attacker that can implement multiple faults, as given sufficient equipment, this should be possible.

Nevertheless, this type of fault injection is very difficult to achieve, as an attacker needs to be absolutely sure about the effect of the fault and where it needs to be produced. When a single fault attack is attempted, an attacker will seek a method of detecting when a fault has occurred in the result or a side channel. The attacker can then continue until the fault has occurred and been detected. In the case of two faults, both faults need to be injected at the right points in time in order to have the desired effect. Until both faults are in the correct positions an attacker will have no indication that one of the faults is correctly placed, unless this is visible in a side channel. The use of random delays in the implementation of the algorithm (as described in Section 7.2) can render this process exceedingly difficult.

In the case of RSA implementations where the modular exponentiation is calculated directly as $M^d \bmod N$, the only available countermeasure in order to verify

that the calculation has been completed correctly is to repeat the algorithm or calculate its inverse. This is potentially vulnerable to two faults: one to modify the algorithm and one to change the evaluation of the redundant function.

In the generation of an RSA signature using CRT (see Algorithm 9.10), a fault could be induced during the execution of the CRT and then a precisely positioned second fault used to change the result of the verification stage. However, it may be necessary to fault several verifications before the attack can be completed. Nevertheless, an attacker only needs to be accurate with the faults used on the verification stages, as the first fault can occur in numerous places and still provoke the desired effect.

Algorithms 9.16 and 9.17 can be made secure against this sort of attack, because it is possible to verify the relationship $\mathbf{A} \equiv A \pmod{(b-1)}$ at numerous points during the algorithm. An attacker would therefore need to identify and then inject a fault at each one of these tests, rendering an attack much more difficult, especially in the presence of random delays. This would not significantly alter the algorithm's performance, as the calculation of a checksum for a given variable is inexpensive when compared to a single multiplication. If, for example, this condition was verified after each multiplication, it would be difficult to inject the hundreds of faults required to fault each test.

It is expected that an implementation based on this countermeasure will be more efficient than implementations based on infective computing [20, 32, 111], as demonstrated in Algorithm 9.11, even if the checksum is verified after every multiplication in the calculation. It has also yet to be proven that these algorithms are secure against fault attacks, as two of the three algorithms have been subject to theoretical attacks [106, 110], although mistakes have been found in some of the attacks [21]. Modular exponentiation algorithms based on infective computing should be resistant to multiple faults, but the evaluation of these algorithms is beyond the scope of this thesis.

The real advantage of using Algorithms 9.16 and 9.17 is that they also provide integrity for the input variables, as each variable can be stored with its checksum. Any modification of the variable before calculation will be detected in the same

way as a dynamic fault, which protects against the attacks described in [25, 74, 97]. However, this protection does not extend to the exponents used, so other mechanisms need to be employed to protect against the modification of these variables. These other mechanisms are needed to prevent attacks based on transient faults in the private exponent [47, 9]. In Algorithm 9.16 the value of $\mathbf{e}$ could be stored with the exponent, and would therefore act as a checksum. An attacker would therefore have to modify the exponent so that $\mathbf{e}$ remains valid.

9.10 Countermeasures

There are many different countermeasures that need to be applied to an implementation of RSA in order to achieve a secure solution. These are listed individually here for clarity.

Variable Randomisation — To defend against Statistical Power Analysis the randomisation given in Algorithm 9.7 needs to be used. This masks the variables being used with a small random value. This is analogous to the principles of data whitening, as described in Section 7.4.

Constant Execution — The most important countermeasure for RSA is that an attacker should only be able to see a constant pattern of squaring operations and multiplications in the power consumption. Algorithms 9.5 and 9.8 prevent this type of attack. Algorithm 9.5 will leak the Hamming weight of the key, but this will be masked in a real implementation, as this would be combined with Algorithm 9.7.

Random Delays — These are not directly necessary but can complicate an attack, as described in Section 7.2, and should be placed at the beginning of each loop in the exponentiation.

Random Execution — When variables are transferred from one part of the chip to another, e.g. from NVM to RAM, they need to be transferred in a random order so as to avoid attacks such as those described in Section 6.3. Attaching

a checksum to each variable is also advisable, in order to be able to verify the integrity of the data once it has been moved.

Fault Resistance — Several methods for providing fault resistance have been described in this chapter. In the case where a coprocessor is used, the Algorithm described in Algorithm 9.11 should be implemented, as this provides more protection against multiple faults. In the case where the RSA algorithm is being implemented on a 32-bit chip, the algorithm proposed in Section 9.8.4 can be used. This has the added advantage of naturally providing integrity, in the form of a checksum, to the input variables when they are moved into RAM before computation.

However, if the modular exponentiation is being calculated directly, care needs to be taken to include the randomisation described in Algorithm 9.7. Otherwise, if an attacker can control the input into an implementation of an RSA signature scheme, a fault attack could be based around the observation that, when X and Y are known and $(x_j Y + u_j N) \equiv 0 \pmod{(b-1)}$, the algorithm cannot detect an opcode modification attack which skips instruction $A \leftarrow (A + x_i Y + u_i N)/b$ in Algorithm 9.14.

9.11 Summary and Conclusions

This chapter provides a description of some of the attacks that can be applied to implementations of RSA on embedded platforms.

The fault attack described in Section 9.4 is shown to apply to one-bit faults in the private key used in RSA, and that the version described in [9, 56] cannot be applied to more than one bit. This observation is novel and is part of the contribution of this thesis. An alternative fault model is proposed in which the attack is valid, and another attack described [41] is detailed that allows an attacker to derive information if more than one bit is modified.

The different methods for computing RSA are described. The algorithms that can be used to calculate a modular exponentiation are described. The secure algorithm described in Section 9.6.1 (see Algorithm 9.8) is novel, and part of the

contribution of this thesis. The algorithms that can be used to compute RSA using the Chinese Remainder Theorem are also described.

The modular multiplication used to compute RSA is often conducted using Montgomery multiplication. The security issues implied by the use of this algorithm are described. A novel method of securing Montgomery multiplication against fault attacks is proposed in Section 9.8, and is part of the contribution of this thesis. An analysis of the security of this algorithm is given in Appendix J. The application of this method to provide a fault resistant implementation of RSA is also described. An implementation of this method is described in Appendix K, and is part of the contribution of this thesis.

Chapter 10

Implementations of DSA

The Digital Signature Algorithm (DSA) is used as an alternative to RSA as a signature scheme, and is described in Section 2.4. This chapter discusses the attacks that are applicable to DSA when implemented on an embedded platform.

This chapter is relatively brief, as a large portion of the attacks and countermeasures have already been discussed in Chapter 9; however, use by the algorithm of a random value adds new vulnerabilities and creates the need for corresponding countermeasures. The application of power analysis to DSA is briefly discussed in Section 10.1. A fault attack that relies on modifying the secret key is given in Section 10.2, and an attack based on modifying the random value used in DSA is described in Section 10.3. This is followed by a discussion of the countermeasures that can be applied to protect the generation of the random value in Section 10.4. Some of the work in this chapter has previously been published in [76].

10.1 Power Attacks on DSA

Applying power analysis to an implementation of DSA is similar in nature to the attacks on RSA (see Sections 9.1 and 9.2). The target of a power attack against DSA will typically be the value of the random value k generated during the calculation of a signature. If this value is known, the private key can be calculated from the public information.

A statistical power analysis of DSA will function in the same manner as described for RSA in Section 9.2, as the first calculation conducted with k is the computation

of $g^k \bmod p$, i.e. a modular exponentiation. The Simple Power Analysis of DSA will be similar in that an attacker will seek to determine the bits of k by analysing this modular exponentiation.

10.2 Faults in the Private Key

An attack is described in [9] in which the value of the private key used in DSA (α) is altered during the calculation of s. As in the other fault attacks we have described, it is assumed that the i-th bit is flipped, i.e.

$$\alpha' = \begin{cases} \alpha + 2^i & \text{if the } i\text{-th bit of } \alpha = 0, \\ \alpha - 2^i & \text{if the } i\text{-th bit of } \alpha = 1. \end{cases}$$

which, in turn, gives,

$$\begin{aligned} r &= \left(g^k \bmod p\right) \bmod q, \text{ and} \\ s' &= \frac{h(m) + r\alpha'}{k} \bmod q. \end{aligned}$$

Using the values for r and $u' = s'^{-1} \bmod q$ an attacker can calculate:

$$\begin{aligned} T &= g^{u'h(m) \bmod q} y^{u'(h(m)+\alpha r) \bmod q} \\ &= \left(g^{u'(h(m)+\alpha r) \bmod q} \bmod p\right) \bmod q \end{aligned}$$

If we let $R_i = \left(g^{u'r2^i \bmod q} \bmod p\right) \bmod q$ for all possible values of i, then we have

$$\begin{aligned} TR_i &= \left(g^{u'(h(m)+r(\alpha+2^i)) \bmod q} \bmod p\right) \bmod q \\ \frac{T}{R_i} &= \left(g^{u'(h(m)+r(\alpha-2^i)) \bmod q} \bmod p\right) \bmod q \end{aligned}$$

which can be simplified, to give,

$$(r \bmod p) \bmod q \equiv \begin{cases} TR_i & \text{if the } i\text{-th bit of } s = 0 \\ \frac{T}{R_i} & \text{if the } i\text{-th bit of } s = 1 \end{cases}$$

where TR_i and T/R_i can be precalculated for all possible values of i.

This attack has the same complexity as the attack described for RSA in Section 9.4. In [9] it is suggested that this attack will be applicable to faults on more

than one bit. However, faults on multiple bits will not necessarily be uniquely determined, following the reasoning presented in Section 9.4. In order to apply a multiple bit attack, an attacker needs to know exactly which bits are being affected by a fault, or find a fault that will only set bits to a fixed value.

As described in Section 9.4, another possible attack would be to analyse the integer value of the fault induced on a byte of the private key. This attack is described in detail in [41].

10.3 Faults in the Nonce

A fault applied to a nonce k can allow the lattice attacks of [50, 82] on El Gamal type signature schemes to be applied to DSA. If a few bits of the corresponding k are known for many DSA signatures, such attacks can recover the private key. The resetting data fault model given in Section 5.3 allows an attacker to set an arbitrary amount of data to its uninitialised state, which is ideal for the attacks proposed in [50, 82]. The attack described in this section was published in [76], and was developed and characterised by Phong Q. Nguyen. It is included here to provide the theory behind the implementation described in Appendix L.

Roughly speaking, lattice attacks focus on the linear part of DSA, that is, they exploit the congruence $s \leftarrow \frac{\text{SHA-1}(m)+\alpha r}{k} \bmod q$ used in the signature generation. When no information on k is available this reveals nothing, but if partial information is available each congruence discloses some information about the private key α. If a sufficient number of signatures are collected, this can be used to recover α. If ℓ bits of k are known for each of a number of signatures, we expect that about $160/\ell$ signatures will suffice to recover α.

The attack is described below. For a rational number z and $m \geq 1$, we denote by $\lfloor z \rceil_m$ the unique integer a, $0 \leq a \leq m-1$, such that $a \equiv z \pmod{m}$ (provided that the denominator of z is coprime with regard to m). The operator $|.|_q$ is defined as $|z|_q = \min_{b \in \mathbb{Z}} |z - bq|$ for any real z.

Assume that we know the ℓ least significant bits of a nonce $k \in \{0, \ldots, q-1\}$ which will be used to generate a DSA signature (for an analysis of the case where

the attacker knows other bits, e.g. the most significant bits or bits in the middle of k, see [82]). That is, we are given an integer a such that $0 \leq a \leq 2^\ell - 1$ and $k - a = 2^\ell b$ for some integer $b \geq 0$. Given a message m (whose SHA-1 hash is h) signed using the nonce k, the congruence

$$\alpha r \equiv sk - h \pmod{q},$$

can be rewritten for $s \neq 0$ as:

$$\alpha r 2^{-\ell} s^{-1} \equiv (a - s^{-1}h)2^{-\ell} + b \pmod{q}.$$

If we define t and u as

$$t = \left\lfloor 2^{-\ell} r s^{-1} \right\rfloor_q,$$
$$u = \left\lfloor 2^{-\ell}(a - s^{-1}h) \right\rfloor_q$$

then both values can be readily computed by the attacker from the publicly known information, as b is in the range $0 \leq b \leq q/2^\ell$. This gives

$$0 \leq \lfloor \alpha t - u \rfloor_q < \frac{q}{2^\ell}.$$

which can be rewritten as

$$|\alpha t - u - q/2^{\ell+1}|_q \leq \frac{q}{2^{\ell+1}}.$$

The attacker therefore knows an integer t and a rational number $v = u + q/2^{\ell+1}$ such that:

$$|\alpha t - v|_q \leq \frac{q}{2^{\ell+1}}.$$

This gives an approximation to $\alpha t \bmod q$. If this is repeated for many signatures, an attacker can create d DSA signatures (r_i, s_i), with corresponding message hashes h_i (where $1 \leq i \leq d$) where the ℓ least significant bits of the corresponding nonce k_i are known. The attacker can then calculate integers t_i and rational numbers v_i such that:

$$|\alpha t_i - v_i|_q \leq \frac{q}{2^{\ell+1}}.$$

Using this information to recover the DSA private key α is very similar to the so-called hidden number problem introduced in [24]. In [24, 82], the problem is solved by transforming it into a lattice closest vector problem. More precisely, consider the $(d+1)$-dimensional lattice L spanned by the rows of the following matrix:

$$\begin{pmatrix} q & 0 & \cdots & 0 & 0 \\ 0 & q & \ddots & \vdots & \vdots \\ \vdots & \ddots & \ddots & 0 & \vdots \\ 0 & \dots & 0 & q & 0 \\ t_1 & \dots & \dots & t_d & 1/2^{\ell+1} \end{pmatrix}.$$

The inequality $|v_i - \alpha t_i|_q \leq q/2^{\ell+1}$ implies the existence of an integer c_i such that:

$$|v_i - \alpha t_i - qc_i| \leq q/2^{\ell+1}.$$

Note that the row vector $\vec{c} = (\alpha t_1 + qc_1, \ldots, \alpha t_d + qc_d, \alpha/2^{\ell+1})$ belongs to L, since it can be obtained by multiplying the last row vector by α and then subtracting appropriate multiples of the first d row vectors. The last coordinate of this vector discloses the hidden number α. The hidden vector, $\vec{c}$, is very close to the (publicly known) row vector $\vec{v} = (v_1, \ldots, v_d, 0)$. By trying to find the closest vector to $\vec{v}$ in the lattice L, one can thus hope to find the hidden vector $\vec{c}$ and therefore the private key α. The article [82] presents provable attacks of this kind, and explains how the attack can be extended to bits at other positions.

In the case of bits being reset to an uninitialised state, the previously mentioned lattice and the target vector $\vec{v}$ can be constructed, and an attempt can be made to solve the closest vector problem with respect to $\vec{v}$. This can be achieved using an embedding technique that heuristically reduces the lattice closest vector problem to the shortest vector problem, as described in [82]. For each close vector candidate, a candidate y for α can be derived from its last coordinate that can be evaluated using the public key to see whether the candidate satisfies $\beta = g^y \bmod p$.

To estimate more precisely the efficiency of the lattice attack, the success rates were computed by running the attack 100 times with different parameters. These results are given in Table 10.1. The number of signatures is small, and the lattice dimension is therefore also relatively small, which makes the running time of the lattice attack negligible. For example, on an Apple PowerBook G4, the lattice attack takes about 1 second for 25 signatures, and 20 seconds for 38 signatures.

Table 10.1: Experimental attack percentage success rates: n is the number of bytes reset in k, and d is the number of signatures.

	Number d of Signatures															
$n \downarrow$	2	3	4	5	6	7	8	10	11	12	22	23	24	25	26	27
1											0	10	39	63	87	100
2								0	69	100						
3					0	69	100									
4				0	100											
5		0	2	100												
6		0	100													
7	0	96	100													
10	6	100														
11	100															

Table 10.1 shows how many signatures are required to make the lattice attack work, depending on the number of least significant bytes reset in k. Naturally, there will be a trade-off between the fault injection and the lattice reduction: when generating signatures with nonces with more reset bytes, the lattice phase of the attack will require less signatures. When only one signature is available, the lattice attack cannot work because there is not enough information in the single available congruence. However, if a signature is generated using a value of k for which a large proportion of its bytes are set to zero, it might be possible to compute k by exhaustive search (using the congruence $r \leftarrow \left(g^k \bmod p\right) \bmod q$), and then recover α. From Table 10.1, we see that, when two signatures are available, the lattice attack starts working when 11 bytes are reset in each k. When only one byte is reset in k, the lattice attack starts working (with non-negligible probability) with 23 signatures.

It should be stressed that the lattice attack does not tolerate mistakes. For instance, 27 signatures with a single byte reset in k are enough to conduct the attack,

but the attack will not work if one of the signatures is not generated using a value for k where no bytes were reset. It is therefore important that the signatures input to the lattice attack satisfy the assumption about the number of bytes that have been reset. Hence, if one is able to obtain many signatures such that the corresponding value of k is expected (but not necessarily all the time) to have a certain number of reset bytes, then one should not input all the signatures to the lattice attack. Instead, one should pick at random a certain number of signatures from the whole set of available signatures, and launch the lattice attack on this subset of signatures: Table 10.1 can be used to select the minimal number of signatures that will make the lattice attack successful. This leads to a combination of exhaustive search and lattice reduction. An implementation of this attack is described in Appendix L.

10.4 Countermeasures

The countermeasures that can be used to protect implementations of DSA are equivalent to the countermeasures used for RSA. The discussion of modular exponentiation methods is directly applicable to the treatment of the random value k, and the same conclusions regarding the countermeasures can be drawn. The arguments concerning the use of fault resistant Montgomery multiplication described in Section 9.8 are also directly applicable to DSA. The same list of countermeasures can therefore be used. However, it is also necessary to protect the generation of the value k against fault attacks, to prevent the attack described in Section 10.3. Unlike the case of fault attacks on RSA signatures using CRT [23] (see Section 9.3), the faulty DSA signatures generated by this attack will pass a verification process, so it is necessary to protect the generation of k itself. Some possible methods of achieving this are as follows.

Checksums — The checksum function described in Section 9.8 can be applied to k once it is generated. This will protect subsequent manipulations of k but cannot protect the generation of k.

Execution Randomisation — As described in Section 7.2, if the order in which an algorithm is processed is randomised it becomes difficult to predict what

the machine is doing at any given clock cycle. This can be applied to the generation of k, and will be very effective in preventing the attack described in Section 10.3. This is because the attack is only possible when the bits set to a fixed value are known for every signature, as the fault attack does not allow for noisy data.

Repeated refreshments — k can be protected by generating several nonces and XORing them with each other, separating each nonce generation from the previous by a random delay. This forces an attacker to inject multiple faults at randomly shifting points in time in order to reset specific bits of k.

Testing the Random Value — It may also be possible to provide real time testing of the random numbers being generated by the smart card, such as that proposed in FIPS140-2 [80]. However, even if this is practical, it may be of limited use, as successfully attacking k requires very few signatures. Consequently, the attack may well be complete before the modification is detected.

10.5 Summary and Conclusions

This chapter provides a description of some of the attacks that can be applied to an implementation of DSA. The attack described in Section 10.2 is described in [9]; the observation that this attack only applies to one-bit faults is novel, and is part of the contribution of this thesis. The attack described in Section 10.3 was developed and characterised by Phong Q. Nguyen. However, an implementation of this attack is presented in Appendix L, and is part of the contribution of this thesis. To the best of the author's knowledge, this is the first practical implementation of this attack.

Chapter 11

Conclusion

This thesis describes the current *state-of-the-art* in side channel and fault analysis that can be applied to embedded devices, such as smart cards. The necessary countermeasures are discussed in a general sense. The attacks specific to block ciphers, RSA, and DSA are then discussed, with a list of countermeasures for each algorithm that a programmer would need to implement to achieve a secure implementation.

As described in Chapter 2, this thesis only discusses the implementation issues on embedded platforms. There are other issues concerning the implementation of cryptographic algorithms, but these issues are beyond the scope of this thesis. This thesis is concerned with how to securely implement cryptographic algorithms on embedded platforms, rather than how they are to be used.

11.1 Summary of Contributions

The following summarises the contributions discussed in this thesis.

- An implementation of a key transfer attack specified in Section 6.1 is described in Appendix C.
- A novel way of applying cache-based side channel analysis to an implementation of AES on smart cards is given in Section 6.2.
- An implementation of a fault attack that reduces the number of rounds of a block cipher is discussed in Section 6.4, and the implementation details

are given in Appendices B and D. These were implemented on a Microchip Silvercard.

- Random delays can be used to increase the difficulty of implementing an attack. This is described in Section 7.2, along with a method for optimising the performance and efficiency of this countermeasure. Appendix E shows how this optimisation was derived, and how this can be applied to random delays of any length.

- Four novel attacks based on injecting faults into DPA countermeasures are described in Sections 8.3.2, 8.4.1 and 8.4.2. The implementations of these four attacks are discussed in Appendices G, H, and I.

- In Section 9.4 the currently accepted specification of a fault attack on the private exponent of RSA (when calculated without using the Chinese Remainder Theorem) was shown to be false when more than one bit of the private exponent is changed. An alternative fault model is proposed in which the attack is valid.

- A new modular exponentiation algorithm is given in Algorithm 9.8, in Section 9.6.1, although this is based on previously published countermeasures.

- A novel method of securing the Montgomery multiplication algorithm against fault attacks is given in Section 9.8. This includes how the algorithm can be securely used to calculate a modular exponentiation. An informal analysis of this suggests that this algorithm is robust against a range of fault attacks. A description of an implementation of this algorithm on 32-bit chip and the results of attempting to overcome the countermeasure by injecting fault with a laser are given in Appendix K.

- An application of a fault attack that fixes certain values of the random value used in DSA is described in Section 10.3, and an implementation of this attack is described in Appendix L. This is the first published practical implementation of an attack of this type.

- Methods for optimising the calculation of DPA traces are described in Appendix A.

Appendix A

Optimising the Differential Calculation

This appendix is concerned with optimising the calculation required to generate DPA waveforms, as described in Section 4.3.2. The work described in this section has been published in [77].

The method typically used to conduct Differential Power Analysis is repeated here for clarity. DPA can be performed on any algorithm in which an intermediate operation of the form $\beta = S(\alpha \oplus K)$ is calculated, where α is known and K is the key (or some segment of the key). The function S is a non-linear function, usually a substitution table (referred to as an S-box), which produces an intermediate output value β.

The process of performing the attack initially involves running the target device N times with N random plaintext values P_i, where $1 \leq i \leq N$. The encryption of the plaintext P_i under the key K to produce the corresponding ciphertext C_i will result in a power consumption waveforms w_i for every i ($1 \leq i \leq N$). These waveforms are captured with an oscilloscope, and sent to a computer for analysis and processing.

To find a given partial key (K_j), the output produced from the S-box (ℓ_j) when given K_j and all $P_{i,j}$ will be used to categorise the power consumption waveforms. A single output bit b from ℓ_j is used for this categorisation. For all possible hypotheses, i.e. K_j is an integer in the interval $[0, 2^m - 1]$, and each partial message $P_{i,j}$, b will classify whether waveform w_i is a member of one of two possible sets. The first set

S_0 will contain all the waveforms where b is equal to zero, and the second set S_1 will contain all the remaining waveforms, i.e. where the output bit b is equal to one.

For each hypothesis, a differential trace Δ_n is calculated by finding the average of each set and then subtracting the resulting values from each other, where all operations on waveforms are computed in a pointwise fashion.

$$\Delta_n = \frac{\sum_{w_i \in S_0} w_i}{|S_0|} - \frac{\sum_{w_i \in S_1} w_i}{|S_1|}$$

The DPA waveform with the highest peak will validate a hypothesis for K_j, i.e. $K_j = n$ corresponds to the Δ_n featuring a maximum amplitude. For a single DPA waveform this involves $N - 2$ additions (as there will be an average of $\frac{N}{2}$ elements in each set) to create the differential Δ_n. Therefore, for all hypotheses, the total number of operations to calculate all DPA waveforms for one S-box is $2^m \times (N - 2)$.

A.1 The Global Sum

The simplest optimisation for calculating DPA traces involves the calculation of the global sum G. This is the summation of all the power consumption waveforms that have been acquired,

$$G = \sum_{i=1}^{N} w_i.$$

The calculation of the DPA waveform now only involves the summation of a single set.

$$\Delta_n = \frac{G - \sum_{w_i \in S_{\text{least}}} w_i}{N - |S_{\text{least}}|} - \frac{\sum_{w_i \in S_{\text{least}}} w_i}{|S_{\text{least}}|}$$

where S_{least} represents the set with the least number of elements, i.e.

$$S_{\text{least}} = \begin{cases} S_0 & \text{when } |S_0| \leq |S_1| \\ S_1 & \text{when } |S_0| > |S_1| \end{cases}$$

The expected number of additions required to generate S_{least} is:

$$\begin{aligned}\mathrm{E}(X) &= \sum_{i=0}^{N} i \Pr\left[X = i\right] \\ &= \sum_{i=0}^{\frac{N}{2}} i \binom{N}{i} \left(\frac{1}{2}\right)^{N} + \sum_{i=\frac{N}{2}+1}^{N} (N-i) \binom{N}{i} \left(\frac{1}{2}\right)^{N-i}\end{aligned}$$

If, for example, 1000 acquisitions were taken, this would result in an expected number of additions per hypothesis of 487. This is an improvement over the case where a set is chosen arbitrarily, when the expected number of additions would be 499 (i.e. $\frac{N}{2} - 1$).

The cost of precalculating G for a single hypothesis is obviously not worthwhile. Note, however, that separate pre-computation of G is not mandatory. The trick here consists in computing a first hypothesis, just as in the original version of DPA, and then summing the two resulting average waveforms to get G, thereby allowing the complexity of the next $2^m - 1$ hypotheses calculations to be reduced at no extra cost.

In the remainder of this analysis, S_{least} will no longer be used. This is because of the optimisations described in the following sections, where the change in which the raw data is distributed does not allow this optimisation to be used. In subsequent sections the differential trace Δ_n will be calculated by subtracting the average of the waveforms in either S_0 or S_1 from G, where the choice between S_0 or S_1 is completely arbitrary.

A.2 Formation of Waveform Equivalence Classes

The input to each S-box will consist of partial bits $P_{i,j}$ of the plaintext P_i and partial bits K_j of the key K. If we consider $P_{i,j}$, there are only 2^m possible values which can enter a given S-box. Therefore, a number of the waveforms will have the same value for $P_{i,j}$, and thus can be treated in the same manner (i.e. they can be divided into a different set of equivalence classes*). W_n is defined for ℓ_j as:

*Note the waveforms form an equivalence class for each ℓ_j, i.e. for ℓ_0 the representative waveforms will be formed in a particular way, for ℓ_1 they will be formed in a different way, etc.

$$W_n = \sum_{i=1}^{N} \chi_i w_i \quad \text{where} \quad \chi_i = \begin{cases} 1 & \text{if } P_{i,j} = n \\ 0 & \text{otherwise} \end{cases}$$

This will produce 2^m representative waveforms from the N acquisitions. The partitioning of the power traces according to the key hypothesis will result in $N/2^m$ waveforms in each set. Therefore, creating each representative waveform will require $\frac{N}{2^m} - 1$ additions, and this will be repeated for each of the 2^m possible entry values.

The differential trace can therefore be calculated as,

$$\Delta_n = \frac{\sum_{W_i \in S_0} W_i}{|S_0|} - \frac{\sum_{W_i \in S_1} W_i}{|S_1|}$$

where S_0 and S_1 will now contain exactly 2^m elements[†]. The pre-calculation involves the formation of the representative waveforms W_n and the global sum G. G can be constructed as a function of W_n:

$$G = \sum_{n=1}^{2^m} W_n.$$

The generation of G will require a further $2^m - 1$ additions. Generating a single DPA waveform will require $2^m - 1$ additions. For all hypotheses, a total of $2^m (2^m - 1)$ additions will be required to calculate all the DPA waveforms for one S-box.

Therefore, the total number of operations that will be incurred in generating all DPA waveforms is:

$$\begin{aligned} \text{Total Calculations} &= 2^m \left(\frac{N}{2^m} - 1 \right) + (2^m - 1) + 2^m (2^m - 1) \\ &= N - 1 + 2^m (2^m - 1) \end{aligned}$$

where for waveforms W_1 to W_{2^m}, approximately $\frac{N}{2^m} - 1$ additions will be required to make up each W_n. An additional $2^m - 1$ summations will be required to form the global sum G, and $2^m - 1$ additions will be required to generate each DPA waveform, for the 2^m key hypotheses.

[†]If the case arises where a value for W_n does not occur, this may create a bias. This is the only situation where $|S_0| \neq 2^m$ or $|S_1| \neq 2^m$.

A.3 Combining Waveforms

Pre-calculating certain waveform combinations enables groups of waveforms, that occur in the same set for one hypothesis, to be recycled in subsequent hypothesis testing. Since we are dealing with 2^m representative waveforms, pre-computing all possible $2^m!$ waveform combinations is infeasible. Therefore it is proposed that the waveforms are partitioned into groups of size x, and the different possible combinations for each group are pre-computed.

For example, for $x = 2$, adjacent waveforms are summed. We define each pair as the value $W_{2n,2n+1}$, where n takes the integer values in the interval $[0, 2^{m-1}]$ (i.e. $[0, 2^m/2]$).

$$W_{2n,2n+1} = W_{2n} + W_{2n+1}$$

The use of the combined waveforms $W_{2n,2n+1}$ in the evaluation of the set S_0, results in three possible scenarios for each pair:

1. The pair occurs, i.e. $W_{2n} + W_{2n+1}$ appears in S_0. The probability of this occurring is $\frac{1}{4}$. In this case, one additional summation must be performed.

2. The pair does not appear in S_0, i.e. the pair is in G. The probability of this happening is also $\frac{1}{4}$. In this case, no action is performed.

3. The two waveforms W_{2n} and W_{2n+1} appear in two separate sets. There is a higher probability of this happening, i.e. $\frac{1}{2}$. In this case, a single addition is required.

The expected number of additions per Δ_n (assuming the global set already exists), is therefore $\frac{3}{4} \times 2^{m-1}$. This is because there are 2^{m-1} groups of three elements (W_{2n},W_{2n+1} and $W_{2n,2n+1}$), one of which will be used to add to S_0 with probability $\frac{3}{4}$.

If $x = 3$ then the following combinations of waveforms will be created from W_{3n}, W_{3n+1} and W_{3n+2}:

$$W_{3n,3n+1} = W_{3n} + W_{3n+1}$$

$$W_{3n+1,3n+2} = W_{3n+1} + W_{3n+2}$$

$$W_{3n,3n+2} = W_{3n} + W_{3n+2}$$

$$W_{3n,3n+1,3n+2} = W_{3n} + W_{3n+1} + W_{3n+2}$$

Following the same reasoning as above, the number of additions per Δ_n is $\frac{7}{8} \times \frac{2^m}{3}$. However, if 2^m is not divisible by three, then there will be a set of waveforms (of size $2^m \bmod 3$) that cannot be combined in the same way. These waveforms will have to be combined separately. The therefore the number of additions per Δ_n is $\frac{7}{8} \times \left\lfloor \frac{2^m}{3} \right\rfloor + \frac{2^{2^m \bmod 3} - 1}{2^{2^m \bmod 3}}$

For all values of x this can be generalised to:

$$\text{Additions per Hypothesis} = \left(\frac{2^x - 1}{2^x} \right) \left\lfloor \frac{2^m}{x} \right\rfloor + \frac{2^{2^m \bmod x} - 1}{2^{2^m \bmod x}}$$

This comprises two expressions, as x will not always divide evenly into 2^m, which will leave $2^m \bmod x$ elements to be grouped together separately (as described above for the case where $x = 3$). Note that this is only an approximation for the amount of additions required for each set. In practice the number of additions involved in generating Δ_n will depend on the contents of the set, and whether the pre-computed combinations appear or not.

The combined waveforms need to be generated before DPA traces can be generated. This involves generating the $2^x - 1$ combinations necessary for each group of x waveforms (this is not 2^x, as one combination excludes all of the x waveforms). This will involve $2^x - x - 1$ additions, as x waveforms already exist and one addition will be required for each unknown combination. The waveform combinations that are formed out of more than two waveforms still only require one addition, as they can be formed from previously computed combinations, e.g. when $x = 3$, $W_{3n,3n+1,3n+2}$ can be calculated using $W_{3n,3n+1}$ and W_{3n+2}. This can be repeated for each of the $\left\lfloor \frac{2^m}{x} \right\rfloor$ groups of size x. The last group that will be present when x does not divide into 2^m will produce a smaller group of containing $2^m \bmod x$ waveforms, which will require $2^{2^m \bmod x} - (2^m \bmod x) - 1$ additions to form its own group, following the

reasoning above.

Generating W_n and G will involve adding together will require $2^m\left(\frac{N}{2^m}-1\right)+(2^m-1)\equiv N-1$ additions, following the reasoning given in Section A.2. The amount of pre-calculation involved (incorporating G, W_n and the pre-computed combinations) is therefore:

$$\text{Pre-Additions} = N-1+(2^x-x-1)\left\lfloor\frac{2^m}{x}\right\rfloor+2^{2^m \bmod x}-(2^m \bmod x)-1$$

The total number of additions required to generate 2^m DPA waveforms is:

$$\text{Total Calculations} = 2^m \times \text{Additions per Hypothesis} + \text{Pre-Additions}$$

The memory requirement is a vital factor, as the more pre-computed values that are required, the more memory they will take up, and the more time it will take to load these values into memory. In order to balance the time-memory trade off and achieve the optimal attack, the number of memory that is required for any value of x needs to be derived.

The amount of waveforms that need to stored in memory for each of the $\left\lfloor\frac{2^m}{x}\right\rfloor$ groups of x waveforms that are combined is 2^x-1. There will be a further smaller group of 2^m mod x waveforms that is created when x does not divide into 2^m, which will require that a further $2^{2^m \bmod x}-1$ waveforms are stored in memory.

$$\text{Memory Required} = (2^x-1)\left\lfloor\frac{2^m}{x}\right\rfloor+(2^{2^m \bmod x}-1)$$

This value gives the number of representative waveforms W_n that need to be stored in memory. Each point in this waveform will need to be stored in a 32-bit word so that no information is lost. The value generated will therefore need to be multiplied by four times the size of one acquisition (assuming that each point of an acquisition occupies one byte of memory).

A.4 Chosen Plaintext Differential Power Analysis

The pre-calculations previously made for S-box ℓ_j will be redundant for ℓ_{j+1}, as the power consumption waveforms, when classified according to the partial input $P_{i,j}$, will fall into different equivalence classes to those that apply for $P_{i,j+1}$. Therefore, for S-box ℓ_{j+1}, regeneration of the representative waveforms will be required.

In classical DPA the message given to the algorithm under attack is random. However, if plaintexts can be chosen, the precalculated W_i can be used to attack subsequent S-boxes. The simplest case is where the input to S-box ℓ_1 can be made the same for $\ell_1, \ell_2, \ldots, \ell_{N_S}$ (where N_S is the number of S-boxes used in one round of the algorithm under consideration). For example, suppose we construct the plaintext so that all plaintext bytes are equal, i.e. byte[1] = byte[2] = ... = byte[16] in AES. This means that there are 256 possible values for the plaintext. Calculating the DPA waveform for the first S-box will calculate the DPA waveform for all others at the same time, giving sixteen peaks at the points in time at which the sixteen key bytes are being manipulated. Using this method may not always be advantageous, as some confusion can arise as to which peak corresponds to which key byte.

A similar approach can be applied to an implementation of DES, where the plaintext can be generated such that the value $P_{i,j}$ entered into the even-numbered S-boxes ($\ell_{j \bmod 2=0}$) are all equal, and the value of $P_{i,j}$ entered into the odd numbered S-boxes ($\ell_{j \bmod 2=1}$) are all equal. All the even numbered S-boxes can use the same set of data generated during the pre-calculation for the first S-box. The odd numbered S-boxes also use this data but with a permutation on the value associated with each representative waveform. This does not affect the quality of the results produced, as each S-box uses a different permutation. The DPA peak will be at the same level as if a random plaintext was used.

Note that these optimisations are applicable when the attack is concentrating on the first round of a block cipher. If the attack focuses on the last round, where the DPA waveforms are related to the ciphertext, then these optimisations will be useless as the data cannot be controlled.

A.5 Optimisation Results

The optimisations are described in terms of performing DPA on DES, where 1000 power consumption acquisitions have been taken. In experiments, this often appears to be sufficient to determine a relationship between the power consumption and the data manipulated. It may be necessary to use more acquisitions, or possibly to use less, but 1000 has usually proven to be a good starting point.

Given $N = 1000$ power consumption waveforms w_i, Table A.1 details the number of operations that must be performed to generate the differential traces for the key hypotheses.

Table A.1: Optimisation results

	Precalc.	Additions per hypothesis	Additions per S-box	Waveforms	Memory in Megabytes
Theoretical DPA	-	998	63872	-	-
Optimisation 1: Global Sum	999	487	32167	1	4
Optimisation 2: Equivalence Classes	999	31	2983	65	260
Optimisation 3:					
2 bits	1031	23.3	2519	96	384
3 bits	1083	18	2235	148	592
4 bits	1175	14.1	2075	240	960
5 bits	1322	11.6	2064	387	1548
6 bits	1580	9.8	2207	645	2580
7 bits	2079	8.4	2619	1144	4576
8 bits	2975	7.0	3421	2040	8160
9 bits	4513	6.5	4928	3578	14312
10 bits	7088	5.9	7468	6153	24612

The more bits that are grouped together, the more memory is required to conduct the attack. It has been assumed that the power consumption acquisitions consist of a million points, where each point occupies one byte; therefore each representative waveform requires the equivalent of four million points, because of the required increase in variable size.

As shown in Table A.1, the best results for a realistic amount of memory are

obtained when five bits are grouped together at a time. However, the memory requirement for this is 1.5 Gigabytes. The amount of additions when three bits are combined is slightly larger, but requires a much more modest amount of memory.

The optimal location for the storage of the representative waveforms is obviously in the computer's RAM. This is because the access times are significantly faster than those of a hard drive, especially as the amount of data being processed is too large for the hard drive to store in its cache.

A.6 Remarks

In the example given, the time taken for the processing of all the hypotheses for one six-bit section of the key is reduced by a factor of thirty. In actuality, this time will be further decreased, as the acquisitions on the computer's hard drive are only accessed during the construction of the representative curves W_n. The rest of the processing takes place in RAM.

The ideas expressed in Section A.4 have not been discussed in the example, as the gain for the overall attack depends on how the message can be manipulated. The gain for an attack where the plaintext can be freely manipulated should reduce execution time by a factor of approximately sixty.

In this analysis, it is assumed that each power consumption waveform consists of one million points. In practice this can vary depending on a number of factors, such as the storage capacity of the oscilloscope, the amount of time spent localising the S-boxes using Simple Power Analysis, and the algorithm being attacked. In the case of AES, larger S-boxes are used, and so there will be a greater number of key hypotheses, which will result in an increase in the number of precomputed waveforms that need to be stored in RAM. The worst possible scenario is where the memory requirement becomes unmanageable and pre-computation actually inhibits the attack. There are two approaches that an attacker can employ to combat this situation. In the case, where the acquisitions captured are large, the acquisitions can be split into sections (e.g. each waveform can be divided into two segments), and the DPA calculation performed on each section and the results concatenated to

construct the full DPA waveform Alternatively, the formula given for the memory usage in Section A.3 can be used to determine how much pre-computation is possible with the memory available, allowing an attacker to achieve the maximum benefit from the optimisations described.

Appendix B

Finding Glitch Parameters

Section 5.3 described the effect of a glitch, and showed that a glitch is capable of modifying both data and code that is being manipulated by a chip. In order to find a glitch that could be used to inject a fault in a process running on a Microchip Silvercard, an implementation of AES was used as a fault detection mechanism. Numerous different glitch configurations were applied to a smart card implementation of AES, using a known key and no effort was made to produce any particular effect. The glitch configuration was detected by observing when a ciphertext was returned that did not correspond to the expected ciphertext.

Various different configurations were considered within the functional limits imposed by the ISO/IEC standards [52]. The external clock speed was varied between one and five megahertz in steps of one megahertz. The size of the glitch was varied between one to ten clock cycles in steps of one clock cycle. The applied voltage started at three volts, and was incremented in steps of 0.5 volts to five volts. The 250 different combinations of these three parameters were each tested at 200 different positions during the AES computation. Different positions were tested in order to try to prevent the code being executed by the smart card from affecting the results.

The voltage level of a glitch that could potentially be used to inject a fault was determined by finding the limit at which the smart card did not respond correctly. In smart cards without a glitch sensor (or poorly implemented sensors), the voltage required to inject a fault should be slightly above the level required to provoke a reset. This process was automated using the equipment shown in Figure B.1, a

proprietary smart card reader called a Clio reader. This reader was modified to inject arbitrary glitches at any given point during a command. This allowed a program to be developed to entirely automate this process.

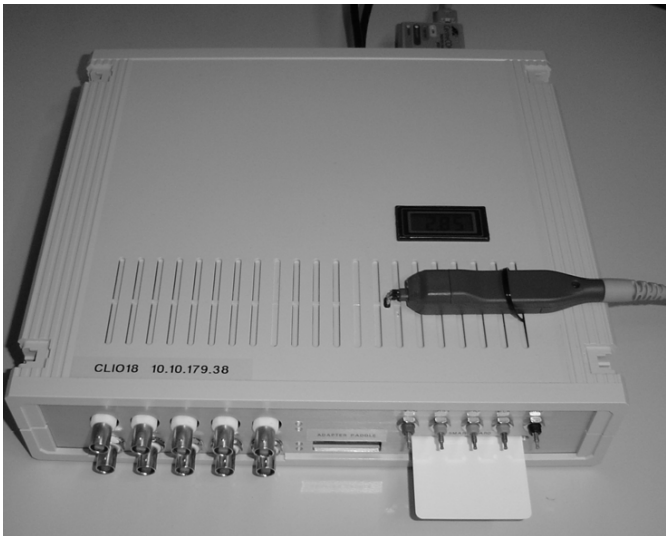


Figure B.1: A modified Clio reader.

Upper and lower boundaries were used to determine the magnitude of the voltage glitch on the Vcc. These were initially set to the voltage applied during the normal functioning of the smart card and 0.25 volts. The point midway between the two bounds was selected as the starting point for the voltage applied during the glitch. When the glitch was applied during the AES computation three different response could result:

1. The card could complete its calculation without any faults. In this case the voltage applied during the glitch will need to be lower than this value in order to achieve an effect, and hence this voltage became the new upper bound.

2. A "Card is Mute" status could be returned, i.e. the card stalls or is reset by the glitch. In this case the voltage applied during the glitch is known to be too low to achieve the desired effect, and this voltage became the new lower bound.

3. A faulty ciphertext could be returned, with a status word indicating that the AES enciphering was successful. In this case the configuration that induced the fault was noted and the next configuration was tested.

If a glitch configuration that can succesfully inject a fault was not found, the experiment was then repeated with modified bounds. The Clio reader used can set a voltage to an arbitrary level, with an accuracy of ± 0.01 volts. When the difference between the two bounds approached 0.01 volts, any subsequent results would be meaningless, and so in such a case it was assumed that the current configuration would not allow a glitch to inject a fault, and the next configuration was tested. The complete process took approximately 24 hours.

The corrupt responses were used to determine the various glitch configurations that succeeded in inducing a fault. The voltage applied to the smart card can be chosen arbitrarily from amongst the configurations that successfully induced a fault. Of the various different configurations that induced a fault, the smallest glitch size that created a fault was chosen for future work; this was to give as much precision as possible for a subsequent fault attack, since a large glitch may affect numerous processes at once, whereas a small glitch will minimise this possibility. In the implementation that was tested the smallest glitch size that was found to have the desired effect had a length of one clock cycle. The voltage applied during the glitch itself was not deemed to have any importance, as the effect of a particular glitch voltage was assumed to be variable, given that the chip's behaviour changes as it heats up through constant activity.

This process provides a list of possible glitches that could be applied to a Microchip Silvercard; however, the whole process would need to be repeated in order to find a glitch that could be applied to another chip.

Appendix C

Attacking a Key Transfer

In Section 5.3 faults are described that change the code being executed by a chip. These faults could force a **for** loop to be terminated before it has finished all of its iterations. An example of how this can be used to implement an attack against DES is described in Section 6.3, and an implementation of this method is described in this appendix.

If the memory where the result of the XOR between the message block and the key is to be stored has not previously been used, it has a high chance of being equal to either $\texttt{00}_{16}$ or $\texttt{FF}_{16}$, depending on the logical representation of the physical state. If a fault is applied to the **for** loop, an arbitrary number of bytes can be set to the empty physical state. In Section 6.3 an attack is described in which one byte is set to zero, then two, etc., as in Table C.1; this attack was originally proposed in [18].

Table C.1: The Biham–Shamir attack

Input	AES Key		Output
$M \rightarrow$	$K_0 =$	XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX	$\rightarrow C_0$
$M \rightarrow$	$K_1 =$	XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX 00	$\rightarrow C_1$
$M \rightarrow$	$K_2 =$	XX XX XX XX XX XX XX XX XX XX XX XX XX XX 00 00	$\rightarrow C_2$
$M \rightarrow$	$K_3 =$	XX XX XX XX XX XX XX XX XX XX XX XX XX 00 00 00	$\rightarrow C_3$
$\vdots$	$\vdots$		$\vdots$
$M \rightarrow$	$K_{14} =$	XX XX 00 00 00 00 00 00 00 00 00 00 00 00 00 00	$\rightarrow C_{14}$
$M \rightarrow$	$K_{15} =$	XX 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	$\rightarrow C_{15}$

Such an attack can be achieved using a total of 15 successful faults. The first byte of the key can then be found by searching through the 2^9 possible key values that could produce C_{15}, i.e. 2^8 values for XX and 2 possible values for the rest of the bits depending on the logical representation of the physical state. Once the first key

byte is known, the second key byte can be found with a further 2^8 AES executions using C_{14}. This process can be continued, with 2^8 AES executions required to determine each subsequent byte, giving a total of 2^{13} AES executions to derive the entire key.

The basic attack (where the memory content is set to 00_{16}) was implemented on an eight-bit smart card microprocessor using a glitch as a fault injector by scanning the entire loop that copies the key from NVM to a temporary working RAM buffer at the beginning of an AES execution (just before the XOR with the first subkey). This produced 127 different faulty ciphertext blocks (whereas 15 different ones were expected), giving as a result the 22 different keys listed below (in base 16). These were found by searching though the results in a bytewise fashion.

```
00000000000000000000000000000000
FE000000000000000000000000000000
FEDC0000000000000000000000000000
FEDCBA0000000000000000000000000000
FEDCBA9800000000000000000000000000
FEDCBA9876000000000000000000000000
FEDCBA9876540000000000000000000000
FEDCBA9876543200000000000000000000
FEDCBA9876543210000000000000000000
FEDCBA9876543210010000000000000000
FEDCBA9876543210012300000000000000
FEDCBA9876543210012345000000000000
FEDCBA9876543210012345600000000000
FEDCBA9876543210012345678900000000
FEDCBA9876543210012345678ABCDEF00
FEDCBA9876543210012345678ABCD0000
FEDCBA9876543210012345678ABCDCD00
FEDCBA9876543210012345678AB000000
FEDCBA9876543210012345678AB00EF00
```

FEDCBA98765432100123456789AB00AB

FEDCBA98765432100123456789ABEF00

FEDCBA98765432100123456789AB5300

Desynchronisation effects, and the exhaustive scanning of the copy loop, mean that unexpected faulty ciphertext blocks were produced. In the set of possible keys as listed above, there are several possible values for the last two bytes, which makes the attack slightly more complicated than originally supposed; however, the attack still remains practical. The correct key was:

FEDCBA98765432100123456789ABCDEF

Appendix D

Round Reduction of AES

If an attacker can modify the code being executed, as described in Section 5.3, there are numerous ways in which the number of rounds of a block cipher can be reduced. In this appendix the various possible faults that can be used to realise such an attack are discussed, and some experimental results obtained using a Microchip Silvercard are described. The faults described in the Appendix were induced using a glitch, derived using the method given in Appendix B.

In general, the implementation of a block cipher in the PIC assembly language (the Silvercard is based around a `PIC16F877`), will have the following format.

```
   movlw    0Ah
   movwf    RoundCounter
RoundLabel
   call     RoundFunction
   decfsz   RoundCounter
   goto     RoundLabel
```

The RAM variable (`RoundCounter`) is set to the number of rounds required; in AES this value is $\texttt{0A}_{16}$ in hexadecimal. The round function is executed, which has been represented here by a call to the function `RoundFunction`. The `RoundCounter` variable is then decremented, and the round is repeated until `RoundCounter` is equal to zero, at which point the loop exits. It is this loop that needs to be changed, so that it exits earlier than expected.

D.1 The Possible Effects

The target of the fault attack is the `decfsz` step, which consists of a decrement, a test, and a conditional jump. The conditional jump involves a jump of one instruction when the test is positive; otherwise the next instruction is executed. The aim of the attack is to reduce the algorithm to one round. It is not possible to remove the first round entirely, as the first conditional test is after the first round has been executed.

The `decfsz` instruction can be broken down into three different tasks: a decrement, a test and a modification to the program counter. The first step is therefore:

```
Decrement task:
        RoundCounter := RoundCounter - 1
```

Three tasks must be performed when a variable in RAM is decremented by one. All three of these tasks represent a potential target for a fault attack. They are:

1. The transfer of the RAM variable contents to the internal processor accumulator.
2. a decrement of the internal processor accumulator.
3. Transfer of the contents of the internal processor accumulator to RAM.

After the `RoundCounter` variable has been updated, its content is tested to see if it equals zero.

```
Testing task:
        If (RoundCounter = 0)
            Status := 1
        Else
            Status := 0
```

An injected fault could potentially have an effect during two different phases of the execution of the testing task. These are:

1. The register content test.

2. The change to the `Status` variable.

Following the status of the test the program counter (`PC`) will take one of two different values, i.e. either zero or one. The jump task will then assign one of two different values to the program counter* (PC).

```
Jump task:
        If (Status = 1)
            PC := PC1
        Else
            PC := PC2
```

If the `Status` variable is equal to one, the hardware sets the program counter to `PC1`. Otherwise it sets the program counter to `PC2`. In the case of the `decfsz` command, `PC1` will be equal to the program counter plus two, and `PC2` will be the program counter plus one. This gives two further potential targets for a fault attack:

1. The Status value test.

2. The modification to the value of `PC`.

In the case of the `PIC16F877` chip, the jump task executes in one clock cycle when the test is negative, and two when the following instruction is not executed. The next instruction is a `goto` that will change the program counter so that the round function is executed again, which takes two clock cycles to execute. This means that the target for the attack is three cycles long, where the first two tasks are present in the `decfsz` step and the last is in the `goto`.

D.2 Experimental Results

In order to perform this attack, one of the possible faults described in the previous section needs to be found within a command that computes AES. An AES computation can easily be detected by monitoring the power consumption of a smart

*The program counter is a register that contains the address of the next command that will be executed by the CPU. A jump in memory is achieved by modifying this register.

card, because the round function will create a pattern that will repeat itself (as described in Section 4.2). Figure D.1 gives an example of the power consumption during a command that computes AES. A pattern that repeats itself nine times is visible, with a tenth pattern that is slightly shorter because of the absence of the **MixColumn** function in the last round of AES.

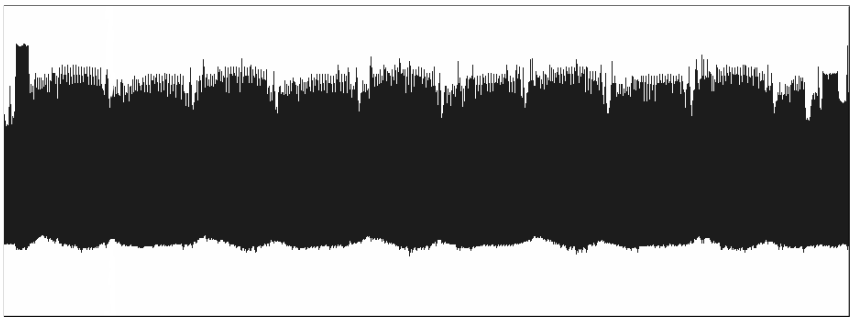


Figure D.1: A power consumption waveform that shows the rounds of AES visible as a repeating pattern.

This can be used to give the approximate position of the execution of the target opcode. The more effort that is put into this stage, the quicker the attack can proceed. I.e. if the functions within the round function can be identified, the number of positions that need to be tried to get the desired results can be reduced. There is, however, a lower limit to the amount of positions that will need to be tested, as the test that will exit the loop could be placed either before or after the **MixColumn** function.

In conducting this attack, the simplest case was taken by applying a voltage glitch, the size of which was determined using the method described in Appendix B, at what appears to be the middle of the first round, and incrementing the glitch position in steps of one clock cycle to the middle of the second round. Approximately 48 hours of testing were required to test 9000 different glitch positions. This process yielded five different positions which caused only one round of AES was computed, although it was not possible to determine how these relate to the theoretical targets given above.

If the key can be changed on a test card, the result of one round of AES can

searched for using the data acquired. This will show when the algorithm has been reduced to one round. Once the fault position has been found, an identical approach can be applied to other cards that use the same implementation of AES, i.e. the same product. This was the case chosen during the experiment described in this Appendix.

If, however, the key cannot be modified, the I/O channel needs to be acquired each time the glitch is applied. This can be used to signal when the attack has been successful, as the time between the command and the response will shorten, since nine rounds of AES have been removed. An example of this is shown in Figure 6.2, where the shortening of the command can be seen in the I/O and is confirmed by the power consumption. This shortening of the command time is only significant when the status returned by the smart card implies that everything has executed correctly, and sixteen bytes have been returned. Otherwise it could be confused with a reset provoked by an overly aggressive glitch.

The five different positions where a glitch had been found to reduce the number of rounds executed to one, were each used as the basis of an attack for three different message blocks. Again the voltage level of the glitch was varied, as the glitch configuration used for the initial search would not necessarily apply, since the chip will have been heated during the constant activity required by the search. This resulted in 150 different attempts to reduce AES to one round. All of the responses were kept for subsequent interpretation with the corresponding messages.

From the 150 glitch attempts made against the Silvercard, a pair of results that produced a correct key hypothesis was trivial to find, using the method described in Section 6.4.

Appendix E

Efficient Use of Random Delays

The use of random delays in embedded software is discussed in Section 7.2. In this appendix we describe the results from experiments in which the distribution behind the generation of each random delay was modified.

The lengths of individual software random delays are generally governed by a uniformly distributed random value. The distribution of the random value could be modified to improve the properties of the cumulative distribution. This section defines the design criteria for the modified distribution of random delays, and then explains how a solution was designed.

E.1 Design Criteria

The ideal criteria for the new distribution would be the following:

1. Random delays should provide greater variation when compared to the same number of random delays generated using a uniformly distributed random variable.

2. The overall decrease in performance arising from the inclusion of random delays should be similar or better than if the random delays are generated from a uniformly distributed random variable. This means that the mean of the cumulative distribution should be less than, or equal to, the mean of the cumulative uniform distribution.

3. Individual random delays should not make an attack trivial in the event that

there is only one random delay between the synchronisation point and the target process.

4. It should be a non-trivial task to derive the distribution of the random delay used.

These conditions represent the ideal; some compromise will be needed between the conditions, as the uniform distribution will probably be preferable for criterion 3.

Our main aim in this section is to optimise criteria 1 and 2, since the reverse engineering of a random delay distribution is rarely conducted (furthermore, it will be shown that this involves more effort than assuming the distribution is uniform) and individual efficiency is rarely applicable due to the numerous random delays used (as described above).

In order to be able to compare different distributions, the mean and the standard deviation are used to capture the characteristics of the cumulative distribution of random delays. This was a natural choice, as the cumulative distribution is based on a binomial distribution, which is the discrete version of a normal distribution (characterised by the mean and variance). However, the standard deviation was chosen rather than the variance, as this represents the mean deviation from the mean.

E.2 Deriving a Suitable Distribution

In order to increase the variation in the cumulative distribution, the probabilities of the extreme values occurring were increased, i.e. the minimum and maximum values of one random delay. The formation of a binomial distribution after several random delays cannot be avoided, as the number of combinations close to the mean value is far too large. Any modification is also required to be subtle, as one random delay needs to be able to provide desynchronisation between two sensitive areas (criterion 2). It was initially assumed that this could be achieved by a distribution of the form shown in Figure E.1.

To use this probability function in a constrained environment, for example a smart card, this function will need to be expressed as a table, where the number of

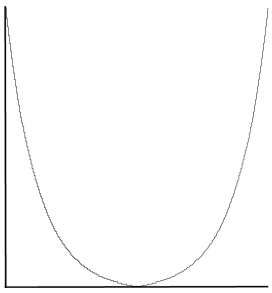


Figure E.1: An example of a modified probability function.

entries for a given value represents the probability of that value being chosen. A uniformly distributed random value can then be used to select an entry from this table. The function that was chosen to govern the amount of entries for each value of x, where x takes integer values in the interval $[0, N]$, was:

$$y = \lceil ak^x + bk^{N-x} \rceil$$

where a and b represent the values of the probability function when x takes the values 0 and N respectively, and y governs the number of entries in a table for each value of x that can be used to represent this function. The sum of y for all values of x will therefore give the total number of table entries. The variable k governs the slope of the curve, and can take values in the interval $(0, 1)$. Different values for a and b are used, so that a bias can be introduced into the sum distribution to lower the mean delay. The two elements ak^x and bk^{N-x} are both close to zero when $x \approx N/2$ for the majority of values of k that will be of interest. The ceiling function is therefore used to provide a minimum number of entries in the table for each value. This is important, because if values are removed from the distribution it will decrease the number of values the random delay can take, and therefore reduce its effect.

The parameters that generate this shape were changed, and the effect on the sum distribution was tested, in order to search for the best configuration that would satisfy the criteria given above. As given in the example, the length of each individual random delay can be an integer value in the interval $[0, 255]$.

For values of a and b that are approximately equal, the change in k will have an effect on the mean and the standard deviation, as shown in Figure E.2. This graph was generated by analysing a large number of random values generated by a random look-up in a table, as described above.

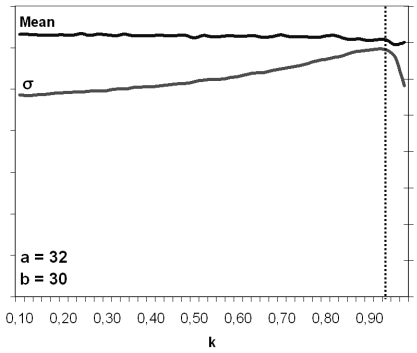


Figure E.2: The mean and the standard deviation against k for approximately equal values of a and b.

The mean in Figure E.2 is fairly constant for all the values of k tested. The variance shows an optimal value of $k = 0.92$. This experiment was repeated for various different values of a and b and the optimal value for k remained constant.

This value was used to look at the effect of varying b on the mean and the standard deviation, as shown in Figure E.3. As can be seen, the mean can be reduced by an asymmetric distribution that favours the lower extreme over the higher extreme values. The mean shows an almost linear relationship with b, and the standard deviation has a logarithmic relationship with b. The best configuration would be to minimise the mean value and maximise the standard deviation. This is not possible, and a compromise needs to be found between the two. The choice $b = 16$ seemed to be a good compromise, and this was used as the basis of further investigations.

To provide a table that can be efficiently implemented, the number of entries in the table should be a power of two. A random number generator will provide a random word, for which the relevant number of bits can be masked off and used to read the value at the corresponding index of the table in order to dictate the

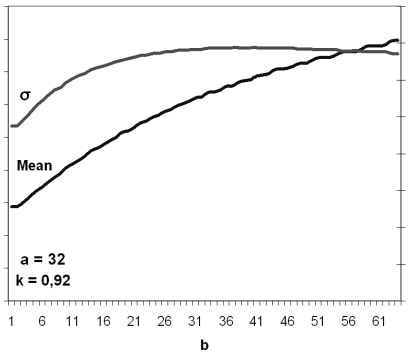


Figure E.3: The mean and the standard deviation against b for fixed values of a and k.

length of each random delay. If the number of entries is not equal to a power of two, any values generated between the number of entries and the next power of two will have to be discarded. This testing of values will slow down the process and have a potentially undesirable effect on the distribution of the random delays, as suitable random numbers will only be generated with a certain probability.

The parameters that were found to naturally generate a table of 2^9 entries are shown in Table E.1. It would be possible to choose some parameters and then modify the table so that it has 2^9 entries, but this was deemed overly complicated as tables with an appropriate number of entries occur naturally. The percentage changes shown are in comparison to the mean and standard deviation of a uniformly distributed random delay. The change in the mean and the standard deviation is not dependent on the number of random delays that are added together, which is not dependent on the number of random delays that occur.

As can be seen, a large variation is visible in the change in the mean and standard deviation. It can also be seen that we cannot achieve the best standard deviation increase and mean decrease at the same time. The designer will have to make a compromise between maximising the standard deviation and minimising the mean.

Table E.2 shows parameters that naturally generate a table of 2^{10} entries. The effect of the modified random delays is more pronounced when based on a table of 2^{10} entries, as it is possible to approach the optimal value of 0.92 for k. However, it may

Table E.1: Parameter Characteristics for Tables of 2^9 Entries

a	b	k	Mean	σ
			% decrease	% increase
22	13	0.88	13.4	34.5
23	12	0.88	16.3	33.5
23	15	0.87	11.6	35.4
24	11	0.88	19.7	32.1
24	14	0.87	13.4	34.9
25	10	0.88	22.6	30.7
26	6	0.89	32.8	23.5
26	9	0.88	25.6	29.0
26	12	0.87	19.7	32.5
32	8	0.86	31.4	25.8

not be realistic to have a table containing 1024 entries in a constrained environment such as a smart card, although in modern smart cards the problem of lack of code memory is beginning to disappear. An example of the effect of using this sort of method to govern the length of random delays on the cumulative distribution is shown in Figure 7.3, where $a = 26$, $b = 12$ and $k = 0.87$. As previously, the y-axis is normalised to show the change in the cumulative distribution. The last graph shows the cumulative distribution for 10 random delays generated using random values from the modified distribution, plotted alongside the cumulative distribution for 10 random delays generated using uniformly distributed random delays. The differences in the mean and standard deviation can be clearly seen, and the tail present on the left hand side is much smaller for the cumulative distribution of random delays governed by random values taken from the modified distribution.

This distribution satisfies criteria 1 and 2 as described in the introduction. It is unlikely to satisfy criterion 3, as the distribution does have some undesirable properties. The probability of a 0 being produced by the table is $(a + 1)/T$, where T is the number of entries in the table.

Table E.2: Parameter Characteristics for Tables of 2^{10} Entries

a	b	k	Mean % decrease	σ % increase
33	18	0.94	21.3	39.2
37	14	0.94	32.2	32.7
50	32	0.90	16.2	46.5
53	29	0.90	21.3	44.6
54	28	0.90	23.3	43.6
55	19	0.91	35.8	34.7
56	34	0.89	18.2	46.6
58	32	0.89	21.5	45.3
59	23	0.90	32.3	38.4
60	22	0.90	34.3	36.9

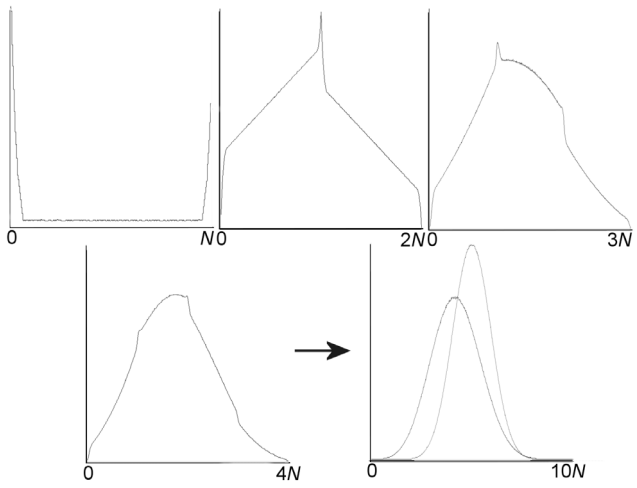
E.3 Reverse Engineering the Distribution

If an attacker knows the distribution of the random delays used, this could potentially be used to increase the speed of an attack, rather then hinder it. However, situations where this information is useful are likely to be rare. If one random delay can be isolated, attacks can be designed around the fact that the distribution of the length of the random delay has been modified. The aim of changing the distribution is to make an attacker work harder to conduct a side channel or fault attack. Situations where the modification allows an attacker to increase the efficiency of an attack are highly undesirable.

E.3.1 Potential Attack Scenarios

In the case of Statistical Power Analysis, an attacker is unlikely to be able to provoke a situation where the desynchronisation present at the point the attacker is interested in is caused by just one random delay. A suitable target for Statistical Power Analysis will only arise after several functions have already taken place within a cryptographic algorithm and, therefore, after several random delays (as described in Section 7.2). The same argument can be applied to faults injected within a cryptographic algorithm.

If an attacker synchronises a set of power consumptions at a given point in a



The cumulative random delay generated by a sequence of random delays generated from a biased distribution. The number of random delays considered are 1, 2, 3, 4, and 10, from top left to bottom right. The last graph also shows the distribution of 10 uniformly distributed random delays for comparative purposes.

Figure E.4: The cumulative random delay using a modified distribution.

cryptographic algorithm, then there will only be one random delay between that point and the following function. There will be a certain number of acquisitions that remain synchronised because of the use of a modified distribution. In theory an attacker could take $a + 1$ times as many acquisitions than would normally be necessary to conduct a Statistical Power Analysis, and then extract the curves that remain synchronised to have a large area synchronised at the desired point. However, this represents much more work than would be required to synchronise before and after the random delay in question.

The only point where an attacker is able to isolate one random delay is when attempting to inject a fault into the operating system running on an embedded device. In Section 7.2, mechanisms for synchronising with events in the operating system are described, followed by one random delay that hinders attacks on a large area of the operating system. If a modified distribution is used in this situation, an attacker could greatly increase the chances of injecting a fault at a desired point by attacking the point in time that would correspond to the minimum value of the

random delay. It is therefore of interest to know how to determine the distribution of a random delay at a given point in a command.

E.3.2 Hypothesis Testing

In order to determine whether the length of a random delay at a certain point in a command is uniformly distributed or not, a reasonable amount of data would need to be collected and the lengths of the random delays stored. These values could then be tested statistically to determine whether or not they correspond to a uniform distribution or not. This could be done by conducting a χ^2 test on the acquired data. An attacker will most likely be required to measure each delay by hand, which will be a lengthy and tedious process. It would be possible to develop a tool for a given chip that would generate this information, but, as soon as the chip is changed, the tool would need to be updated. This because different chips change the power consumption in different ways.

This process can be quite complex if hardware random delays are also present. These normally take the form of randomly inserted clock cycles (where the chip randomly insert clock cycles where no processing occurs) or an unstable internal clock. This is likely to add serious complexity to the resynchronisation process. These effects are ignored in this analysis for simplicity, but would make an attacker work much harder for the desired information.

It can be shown that the random delay being observed is not based on a uniformly distributed random value by using a χ^2 test with a null hypothesis that the random delays are uniformly distributed. In order to conduct this test, the minimum frequency threshold should be around $5(N+1)$ (a rule of thumb given in [58]). In the example, given the minimum frequency threshold, 1280 samples would be required. In practice, far fewer samples are required to consistently provide evidence against the null hypothesis when one of the proposed distributions is used. However, the attacker cannot trust these results without repeating the test with independent acquisitions, so the actual amount of data required will probably still be around $5(N+1)$.

Nevertheless, an attacker can base certain attacks on the assumption that the

length of a random delay may use the modified distribution, and at the same time not reduce the chances of the attack being successful. For example, an attacker injecting a fault after one random delay could attack the point in time corresponding to the minimum length of one random delay, thus maximising the chances of the attack being successful. If the length of a single random delay is uniformly distributed this strategy is as good as any other, as the chances of the minimum length occurring are the same as any other length.

E.4 Remarks

In this appendix we have demonstrated that the standard deviation of cumulative random delays can be increased by using a specific distribution for each individual random delay. It has also been shown that the expected amount of time lost due to random delays can be reduced to minimise the impact of this countermeasure on the performance of functions they are protecting. Given the heuristic design process, the solution is not claimed to be optimal, but it appears likely to be close.

The modified distribution presented satisfies all the design criteria given, except when one random delay is used after a synchronisation point to hinder attacks on the operating system, as described in Section E.3.1. The strategy for implementing this countermeasure is therefore to use the uniform distribution where one random delay is being used to protect the operating system, and to use the modified distribution where numerous random delays are going to be used.

This represents an unusual situation, where a method of increasing the security of a process can also reduce the amount of computational time required. Given the nature of random delays, this increase in efficiency is actually a reduction in the time lost due to the use of random delays. Typically, in smart card implementations of cryptographic algorithms the addition of more security features brings about a reduction in performance.

Appendix F

Differential Fault Analysis on DES

In this appendix we describe the results obtained from a practical implementation of the attack described in Section 8.2, which was conducted against a smart card that was enciphering the message block 0000000000000000_{16} to produce the ciphertext block

$$6FB23EAD0534752B_{16}.$$

Faults were injected into the fifteenth round producing the ciphertext blocks

$$EB327EBD0534712B_{16} \text{ and } DA8AE85B4C55F358_{16}.$$

The faulty ciphertext blocks were compared to the correct ciphertext block, as described in Section 8.2.1, and the key was found after 6.39×10^5 DES executions from a total of 1.13×10^6 possible keys.

The differential $R_{15} \oplus R'_{15}$ can be calculated by computing the inverse of the final permutation and taking the first 32 bits of the result. The change in the ciphertext block between a correct and a faulty ciphertext block can then be observed.

For $EB327EBD0534712B_{16}$, the inverse final permutation of this gives

$$456E3CD909EF8D87_{16}.$$

The last four bytes XORed with the value given by

$$\texttt{6FB23EAD0534752B}_{16}$$

gives

$$\texttt{03000000}_{16}.$$

This does not give much information by itself, but if the expansion permutation is conducted on this, the result is

$$\texttt{0006200000000000}_{16},$$

where the input into each S-box is represented by two digits. As can be seen, the fault changes three bits, creating a differential across two S-boxes. It is interesting to note that this was caused by two false S-box values, as can be seen if the inverse P-permutation is conducted on $\texttt{03000000}_{16}$ giving $\texttt{00008010}_{16}$, where each nibble represents the differential at the exit of an S-box of the fifteenth round.

The same calculation on $\texttt{DA8AE85B4C55F358}_{16}$ gives a differential in the S-box input of

$$\texttt{091B351622251918}_{16},$$

which demonstrates that a differential was created across each S-box. Looking at the differential on the exit of the S-boxes in the fifteenth round ($\texttt{AA7CA60A}_{16}$), it can be seen that the exit value of all the S-boxes were changed.

The breakdown of the key information gained from each faulty ciphertext block is shown in Table F.1. As can be seen, the first faulty ciphertext block does not reduce the total keyspace by very much. The second ciphertext block reduces the size of the keyspace for the last subkey to 2^{17}, reducing the complexity of a complete key search to 2^{26}. When combined with the information given by the first ciphertext block, the size of the keyspace can be reduced to 2^{20}, as the hypotheses for the key bits used before the second and third S-box are reduced to one possible value by the extra information.

Table F.1: The breakdown of hypotheses per ciphertext block.

Faulty Ciphertext Block	Differential on S-box entry	Hypotheses Derived	Total Subkey Hypotheses
$EB327EBD0534712B_{16}$	00_{16}	64	
	06_{16}	10	
	20_{16}	2	
	00_{16}	64	
	00_{16}	64	
	00_{16}	64	
	00_{16}	64	
	00_{16}	64	2^{40}
$DA8AE85B4C55F358_{16}$	09_{16}	2	
	$1B_{16}$	2	
	35_{16}	10	
	16_{16}	8	
	22_{16}	2	
	25_{16}	4	
	19_{16}	6	
	18_{16}	6	2^{17}

Appendix G

CFA of the First XOR of DPA-Resistant AES

An implementation of the attack described in Section 8.3.2 is desdescribed in this appendix. The implementation was conducted under the same conditions as described in Appendix C, with two exceptions. Firstly, the implementation used was DPA resistant; it implemented all the countermeasures described in Section 8.5 except for the random delays that were removed for simplicity. Secondly, the timing of the glitch injection was also fixed, as the random order provided the temporal variation. The key was known *a priori* so the dictionary was generated with a computer rather than with the smart card, which took around 4 minutes on a standard PC.

By conducting faults using glitches on the Vcc for approximately one hour, 118 faulty ciphertext blocks were obtained. Amongst these, 72 unique collisions were extracted. This provided information of the form shown in Table G.1, which has been re-ordered for clarity.

Table G.1: Information derived from 72 collisions.

Collision	Key Information
0	F11A----------
1	00------00----------
2	8E--------21----------
3	00--------88----------
4	5B----------C7----------
5	24------------A9----------
6	00------------88----------

```
7D--------------0F-------------
9C----------------FF-----------
AA------------------FA----------
14--------------------55--------
32----------------------15----
E7------------------------F3--
--DF--FD----------------------
--B4----------D2---------------
--06------------9F-------------
--B6--------------------6B------
----7F6E----------------------
----B2--D4--------------------
----82------C6----------------
----4B--------1E--------------
----3E----------94-------------
----79--------------F1----------
----F1------------------1F------
----03----------------------CF--
----D8------------------------05
------8D----D8----------------
------B3--------08-------------
------17------------8E----------
------A9--------------21--------
------C6------------------28----
------2D--------------------F0--
------FD----------------------31
--------89----BA---------------
--------0D------C1-------------
--------2D------E1-------------
--------C4----------2A----------
--------F2--------------7A------
--------1B------------------B1--
--------A7--------------------1C
----------AD9E------------------
----------8C----51-------------
----------89------45------------
----------FC--------03----------
----------6D------------F4------
----------DE--------------56----
------------E4F5----------------
------------37----C8------------
------------B0------7C----------
------------76------------CD----
------------C6----------------5F
--------------EF----32----------
--------------D6--------6D------
```

```
53    --------------CB----------61----
54    --------------87------------1E--
55    ----------------F0E1------------
56    ----------------62--40----------
57    ------------------F2C1----------
58    ------------------9E--BC--------
59    ------------------E1----B4------
60    ------------------19------5D----
61    ------------------58--------2F--
62    ------------------92----------F4
63    --------------------CD--AB------
64    --------------------83----F4----
65    --------------------59------1D--
66    --------------------F3--------A6
67    ----------------------EC9B------
68    ----------------------52----07--
69    ------------------------9A--B8--
70    ------------------------C2----F1
71    --------------------------1D0C
```

The information in these 72 collisions was processed in around 10 minutes on a standard PC to find 31 unique keys, where there are no linear relationships between any of the keys. These are shown below:

A6487B6A1D0C3F2ED1C0F3E29584B7A6

BA76455423320110EFFECDDCABBA8998

4A1F2C3D4A5B68798697A4B5C2D3E0F1

7D93A0B1C6D7E4F50A1B28394E5A6C7D

D61A29384F5E6D7C8392A1B0C7F1E5F4

D31F2C3D4A5B687986978392C2D3E0F1

E5291A0B7C6D5E4FB08692A4F4C2F1C7

F11F2C3D4A5B68798692A1B5C2D3E0F1

5EB08392E5F4C7D6293D0E1A6D7C4A5E

09E2D1C0B7A695847B6A59483F2E1D0C

7DB18293E4F5C6D7281E2D1B6C7D695F

D31F2C3D4A5B685B869783B5C2D3E0F1

BA76455423320132C8FECDFBABBA8998

84487B6A1D0C3F09D1C0F3E29584B7A6

2376455423AB0110EFFECDDCABBA3798

5EB08392E5D6C7D629380B1A6D794F5E

A7F2C1D0A72F85946BC449582F3E0D1C

1D487B6A1D953F2E6FC0F35C958409A6

1D487B6A1D953F95D1C0F35C958409A6

A7F2C1D0A72F852F6BC449582F80B31C

D61A2938D65E6D7C8392A1B0C7D6E5F4

BA764554BA320110EFFECDFBABBA8998

BA764554BA320132EFFECDDCABBA8998

D61A2938D65E6D5E8392A1B0C7F1E5F4

5EB083925ED6C7D629380B1A6D7C4F5E

9876455498100410EFFECDDCABBA8C98

A6487B6AA62E3A2ED1C0F3E79584B2A6

96784B5A961E0A1EE1F5C3D7A5B18296

9873455423100110EFFBCDDCABBA8C98

98734554983201l0EFFBCDDCABBA8C98

F11A2C3DF17968798697A4B5C2D3E0F1

As 31 hypotheses were produced, this shows that some fault injections have produced some collisions not related to the key values. However, the number of keys produced can easily be tested to determine the correct key, requiring $31 \times 2^8 \approx 2^{12}$ AES executions.

By exhaustive search the correct key information from the previous list is

09E2D1C0B7A695847B6A59483F2E1D0C

masked with the value

F3F3F3F3F3F3F3F3F3F3F3F3F3F3F3F3

giving as result the value of the key used

FA112233445566778899AABBCCDDEEFF.

Appendix H

CFA of the Key Masking of DPA-Resistant AES

An implementation of the attack described in Section 8.3.2 was tested using exactly the same conditions as the implementation described in Appendix G. The same software implementation was run on the same smart card.

The dictionary required for this attack was generated in a matter of minutes on a standard PC. Unlike the attack described in Appendix G, the dictionary does not depend on the value of the key used, so the dictionary generated would be valid for any key value.

After attacking the implementation for approximately one hour, around 60 collisions were generated from faulty ciphertext blocks. These ciphertext blocks, and the information derived from them, are shown in Table H.1.

Table H.1: Information derived from 60 collisions.

Ciphertext Block	Index	Key Information
F81E9C53601A9D27BF14A439CFB89329	13	CC
9589F701F254450A95B9ACE3F56CC525	8	77
D5B7691596141F967B8933B3EC19D80E	5	44
FA88725F36EED9A99DA1BC318861F1CA	5	44
0CA8BF1D394DA73B5DB36C03C6F19540	16	FF
7ED1484607BBCF135F90B460DADA1FCD	4	33
A1EDC486CAD6C32EA16DE3CFDD309201	4	33
0CA8BF1D394DA73B5DB36C03C6F19540	16	FF
B2C5E49D5B5AE03478A06D7212151870	16	FF
96FA183C668222C6094A5D5D2791F489	1	FA
15F059E21A7B3C549A3B8A008DBE1092	15	EE

7F0BC5470530ACEB8FEED67EFB05DE67	12	BB
DCFBF05B975F16C7D9F7AF1D2C7F261D	14	DD
301E921E4DDC1406857EA9085A518668	7	66
06D7E8AD7EF7E3996150365A2BF9F0CC	11	AA
80B7F9B4DDE348C06569A5DCBBD95583	16	FF
30062C6D66449A243C5D8E977DD52877	15	EE
7AA9849D02F539FB45E8AF200EBCD3CA	14	DD
E25DF37A1A72327DF0B850C2ABFEC1E4	13	CC
4EB5A2152C7369137B2F8713051A5685	8	77
CAEB530099B8C71195D8EFA435C8D44B	11	AA
66222AF67A6B9901CBCDA3307AAC000D	5	44
E6C3A6C4A9E166C790A63CE245CEA16F	1	FA
E6CFEBD771ADBC6CF3DAFD3242D629C9	4	33
431DFF0951C5013A00A48C458A3174AB	16	FF
9B676E73FB6FE09199C3926FB260C7E3	12	BB
EC39B70B7EF824DF75F04A7A71497A60	5	44
83215DE1E76D80284614FAC872567D94	6	55
0227F87F6164E5D5658FC2B66C15ED6C	13	CC
CEA80CF8E7B8D19647C0D0A0C6444E1C	10	99
A98BF5AA64719312AB186914246FD798	11	AA
33996299B4A6997066AF8EF3EDDBE249	8	77
A9E394105305283D108CE4A6104E5042	8	77
DC5F1D3F237590837A4E4AE3A6ED839A	9	88
620B8D1257593AB2C5D163AAC07C65E0	10	99
B9E733C89AF5F84D34984412A70C1650	4	33
8C31C9275D27A07992A4806252299A10	8	77
1770FCDC7DA9D70A26071F8D3C2F315E	4	33
79DDB2428F1045946CCBB602C18C72EF	13	CC
AA5CCACBBA4C821D46163E7D29A99CBA	10	99
3EE40ED0F8FA4885FA4E83C679AACDE0	12	BB
1E69C676370BB86D254707D27E6B75EF	16	FF
FF7259BB2952C2E8362BC88257F4FB7A	3	22
DD390F4428D9C7C64DC06A93D04D0E59	14	DD
2BE37107241BFE78C82EECEA2713FAAD	2	11
704EBDB101AC884CB0CF01BC4DE2308E	16	FF
E9F20F24D4A570F57A087CED882D861E	13	CC
EC012644E475C01AF37DD00745EEAEF8	7	66
403FB21648A1A5D85B91069643D38133	16	FF
67D4DF6B38A9300B2F161A63B86442FB	15	EE
5EE7DCD69164B870FDC6EEAC861328C3	4	33
1F1D2DC23B8BED58C26E671ABC973BB4	1	FA
D88983BD1FBEE5808E4A624F37DEF88D	15	EE
87B4D54527B230B351FEEB7EF60DB086	10	99
BEE66E610C45F758EB807993D698A5B7	16	FF
BCFF001FD8876BEF2CBB26895FFA4968	12	BB
7E3C2C98E9B4CE0F0F6C0A6CA82748FD	11	AA

37BE59099E6F6B12CF3107F853B944A7	3	22
CF9B0E7CB5C86085814F53CACDE04943	1	FA
8F68CEDF861567D593965008B26DED9B	10	99

After acquiring these ciphertext blocks, the *a posteriori* processing was trivial, as no incorrect hypotheses were produced by the collisions found. The key can be seen to be:

FA11223344556677889AABBCCDDEEFF.

Appendix I

S-Box Modification in DPA-Resistant DES

In attempting to implement the attacks described in [18], it was observed that when the duty cycle* of the clock given to the smart card was too small, an incorrect ciphertext block was produced. Further study revealed that if the duty cycle was below 15%, data written to certain areas of the chip's memory would be written incorrectly. This essentially meant that random bytes were written to the correct memory address, and this was therefore a different fault injection effect to the other experiments mentioned in this thesis. Once this effect was found it was relatively trivial to implement fault attacks based on this effect. However, this fault effect exploits a bug in a particular chip, and is only likely to apply to one chip type.

The attack described in Section 8.4.1 could be implemented against a smart card using this effect. The attack was conducted with three different message blocks, followed by a small exhaustive search to find the key. The entire attack took 45 minutes using tools created specifically for this purpose. This was possible because of the manner in which the fault was injected, but is unlikely to be possible with other fault injection methods. An attacker cannot always be as certain that a fault has occurred, as a fault is generally expected to be successful with a certain probability when it is applied to a chip [22]. In this case the probability of success was equal to one.

The attack described in Section 8.4.2 was also implemented against the same

*The duty cycle is the amount of time that a voltage is applied to the clock pin compared to the time no voltage is applied, e.g. a standard clock will have a duty cycle of 50%.

chip, as the fault effect discovered was ideal for modifying the S-box values as they were created. The first attempt at this attack was against a DES implementation that used data masking and constructed S-boxes in RAM, as described in Section 7.4. Tools were developed to automate this attack; these tools waited until at least one differential had been found across each S-box before conducting an exhaustive search of the hypotheses derived from the fault injection. These tools found the key after eight minutes.

A second attempt was conducted with the addition of random execution ordering (see Section 7.3) random delays (in both hardware and software), as described in Sections 7.2, so that a fault would be produced with a lower probability. The same tools took 20 minutes to derive the key.

This second attack was easier to implement, as only one fault injection position was needed to attack a random S-box entry. In the first attack it was necessary to shift the position of the fault injection for each new fault injection attempt.

Appendix J

Security Analysis of Montgomery Multiplication with Redundancy Check

We present here a security analysis of the Montgomery multiplication algorithm proposed in Section 9.8.2. The work described in this appendix was conducted by Khanh Nguyen, but remains unpublished [81].

For clarity, let $\mathbf{A}_i$ and A_i respectively be the values of $\mathbf{A}$ and A after iteration i of the loop in step 3 of Algorithm 9.14. First, the completeness of the algorithm is analysed.

Completeness: Consider the integrity relation

$$f(A) - \mathbf{A} \equiv 0 \pmod{b-1}$$

between A and $\mathbf{A}$. This integrity relation needs to holds at the end of the algorithm, but not necessarily at every step of the algorithm. After step 2, this function holds if both A and $\mathbf{A}$ are zero. In step 3, at iteration i, we have

$$f(A_i) - \mathbf{A}_i \equiv \sum_{j=0}^{i} x_j Y \pmod{b-1}.$$

Thus at the end of step 3,

$$
\begin{aligned}
f(A_{z-1}) - \mathbf{A}_{z-1} &\equiv \sum_{j=0}^{z-1} x_j Y \pmod{b-1} \\
&\equiv \mathbf{XY} \pmod{b-1}.
\end{aligned}
$$

At the end of step 4, we have

$$
\begin{aligned}
f(A) - \mathbf{A} &\equiv f(A_{z-1}) - \mathbf{A}_{z-1} - \mathbf{XY} \\
&\equiv \mathbf{XY} - \mathbf{XY} \pmod{b-1} \\
&\equiv 0 \pmod{b-1}.
\end{aligned}
$$

This establishes the completeness of the algorithm.

To analyse the security of the algorithm, we consider the effects of a single fault, whose effect will correspond to the data randomisation, data reset or opcode modification attacks described above.

In the analysis, steps five and six of the algorithm are ignored, as they could be eliminated using techniques from [46]. It is assumed that N, X and Y are random and unknown to the adversary. This is the case in RSA implementations using CRT, or where exponents are multiplied by a small random value before performing the exponentiation (see Section 9.6.1).

Resistance to Opcode Modification Attacks: An opcode modification attack allows the adversary to cause the algorithm implementation to skip a number of operations. Attacks against steps one and two are trivial, and will clearly fail. If the attack is against step four, it would cause the checksum evaluation to fail, as $\mathbf{A}$ would not be equal to $f(A)$ but to $f(A) - \sum_{i=0}^{n-1} x_i \mathbf{Y}$ at the end of (the skipped) step four. The probability of the checksum remaining valid is $1/(b-1)$.

If the attack is against step three, the attack could skip a number of iterations in step three, or skip calls to the instructions $3(a)$, $3(b)$ or $3(c)$ at some iteration j. Skipping the iteration j would cause the final value of $f(A) - \mathbf{A}$ to differ by $x_j Y \bmod (b-1)$. In this case, the probability that the integrity relation $\mathbf{A} \equiv A \pmod{b-1}$ holds is approximately equal to $2/(b-1)$, as the checksum will be valid if either x_j or Y are equal to 0 mod $(b-1)$.

Similarly, skipping either $3(b)$ or $3(c)$ would cause A to change by $(x_j Y + u_j N)$

and $f(A)$ by $-u_j\mathbf{N} \bmod (b-1)$ respectively. This would cause the checksum evaluation to be valid with a probability of $1/(b-1)$. Note that when X and Y are known and $(x_jY + u_jN) \equiv 0 \pmod{b-1}$, the algorithm cannot detect the opcode modification attack which skips the instruction $3(b)$.

Resistance to Data Randomisation Attacks: A data randomisation attack allows an adversary to randomise a variable at a time of the adversary's choosing. As there is no common variable used in computing A and $\mathbf{A}$ except the u_i's and i, a single fault injection against any variable, except the u_i's and i, will change only one of A or $\mathbf{A}$. This would cause the integrity relation $\mathbf{A} = f(A)$ to be valid with a probability of $1/(b-1)$, as before. Randomising i would cause the algorithm to skip some iterations at step three. This is an opcode modification attack and has been considered above. Randomising variable u_i to a random and different value u_i' at iteration i would cause both A_i and $\mathbf{A}_i$ to change after step $3(b)$ and $3(c)$. Here $\mathbf{A}_i = \mathbf{A}_{i-1} + u_i'\mathbf{N}$ and $A_i = (A_{i-1} + x_iY + u_i'N)/b$. As $u_i' \neq u_i$, b does not divide $A_{i-1} + x_iY + u_i'N$ and thus $f(A_i) \neq f(A_{i-1} + x_iY + u_i'N)$. This gives $f(A_{i-1}) \equiv \mathbf{A}_{i-1} + \sum_{j=0}^{i-1} x_jY \pmod{b-1}$. Hence $f(A_i) \not\equiv f(A_{i-1} + x_iY + u_i'N) \equiv \mathbf{A}_{i-1} + \sum_{j=0}^{i-1} x_jY + x_iY + u_i'N \equiv \mathbf{A}_i + \sum_{j=0}^{i} x_jY \pmod{b-1}$. This means the intermediate state would no longer correspond to that given above at the end of iteration i, and therefore the checksum evaluation would fail at the end of the algorithm.

Resistance to Data Reset Attacks: Resetting a random value to its virgin state causes the variable to change its value by a random quantity. When X, Y and N are unknown to the adversary, values u_i, A and $\mathbf{A}$ are also unknown to the adversary at all stages of the algorithm. A data reset attack against any of them would cause a similar impact to a data randomisation attack. A data reset attack against b or $b-1$ would cause a division by zero error from the processor.

Appendix K

Implementation of Montgomery Multiplication with Redundancy Check

In order to evaluate the performance of the algorithm proposed in Section 9.8.2, the standard Montgomery multiplication was implemented on a chip with a MIPS instruction set, with and without the checksum mechanism. The checksum itself is the size of one machine word, which, in the case of the MIPS architecture, is a 32-bit word. In order to be able to give a comparison with existing countermeasures, an implementation using the sort of multiplication required for the countermeasures proposed for RSA in [98] was also implemented. This involves calculating $XY \mod rN$ instead of $XY \mod N$, where r is a small random number. The random value used was assumed to be the same size as the checksum used, i.e. the size of one machine word.

The functions were written in C, and we used the MIPSsde tool chain to generate the code loaded into the chip. The code was written to avoid using any of the standard functions that do not exist in the assembly language to avoid any unnecessary function calls. The ALU was known to be able to hold a 64-bit variable, so variables of this size were freely used in the code.

The calculation of the checksum was optimised to avoid using the modulus function available in C to calculate the modulo $b-1$ reduction, i.e. the % operator. This was because it was assumed that this function call would take a considerable amount of time using the C libraries, as such a function will be written for parameters of

any size. A specific solution will typically be considerably faster than a generic one provided in a C library. This algorithm used to calculate $\mathbf{A} \leftarrow \mathbf{A} + u_i\mathbf{N} \mod b-1$ is given in Algorithm K.1. Step 3 involves an incrementation to take into account the case where $X = b - 1$.

Algorithm K.1: Calculate $\mathbf{A} \leftarrow \mathbf{A} + u_i\mathbf{N} \mod (b-1)$

Input: $\mathbf{A}$, u_i, $\mathbf{N}$
Output: $\mathbf{A} + u_i\mathbf{N} \mod b-1$

1. $X \leftarrow (u_i \times \mathbf{N}) + \mathbf{A}$
2. $X \leftarrow X/b + (X \mod b)$
3. $\mathbf{A} \leftarrow (X+1)/b + (X \mod b)$

return A

This algorithm is efficient as the functions that divide by the word size can be implemented in one assembly instruction, and bit shifts of an arbitrary length are available as one instruction in the MIPS instruction set. Instruction sets will usually have functions that allow an efficient implementation of this algorithm. The same is true of the modulo b function, as this is just an AND function to mask off the lower $\log_2 b$ bits.

The maximum variable size produced at each stage is given below:

1. $(b-1)(b-1) + (b-1) = (b^2 - 2b + 1) + (b-1) = b^2 - b$
2. $(b-1) + (b-2) = 2b - 3$
3. $b - 2$

This shows that X needs to be able to contain $\log_2 b^2$ bits in order to be able to receive the output of the first step of the equation; so this will need to be taken into account in any implementation. As the ALU used was known to be of length 64 bits, this was trivial to implement, but would be more problematic in cases where the ALU is the same size as the machine word size.

The computation times for a variety of different modulus lengths are given in the table below. The algorithm proposed in [98], and described in Section 9.6.2,

was also implemented for comparative purposes. The implementation was limited to a modular multiplication, so this algorithm was reduced to a Montgomery multiplication with a word size one larger than the implementation using the checksum algorithm.

The timing figures are given in clock cycles, as this gives an indication of the speed of the algorithm without being affected by the speed of the clock being used by the chip. This was measured by incorporating an I/O peak* either side of the relevant code. The communication was then observed during the execution of this code using a proprietary card reader. The tool used for this observation can give the number of clock cycles between any two events on the I/O. This is usually used to debug I/O routines, but for our purposes provided a quick and accurate tool for measuring algorithm performance.

Table K.1: Speed comparisons.

Modulus Bit Length	Normal Implementation	Random Modulus	% increase	Checksum	% increase
256	11233	13945	24.1	12167	8.3
512	41563	46721	12.4	43011	3.5
768	91483	99081	8.3	93491	2.2
1024	156759	166536	6.2	159336	1.6

As can be seen, both types of fault protection have an additional cost in time; the use of a checksum to protect the Montgomery multiplication is approximately four times more efficient than using a modulus multiplied by a small random value.

The figures given should not be taken as a reference for the speed of the Montgomery multiplication on a MIPS platform. The code doubtless has room for optimisation, especially in an assembly language implementation. It should also be noted that some of the parameters used in the Montgomery multiplication were declared as constants, which will have been taken into account by the compiler. The values are therefore given only as an indication of the performance levels of a MIPS implementation.

*An I/O peak is a quick transition from 1 to 0 and back again. This is ignored by the card reader as it will be much shorter than one elementary time unit in the $T = 0$ and $T = 1$ protocols, but can be seen with an oscilloscope or proprietary tools.

The algorithm implementation was tested by subjecting the chip to a laser shot during its execution. The chip used was known to be vulnerable to fault injections by laser[†], so a location on the chip's CPU was readily found that produced a fault with a high probability. The first attempts targeted the middle of the algorithm execution to make sure that a fault could be provoked. Once a laser configuration was found that produced a fault, a more methodical analysis was conducted.

A 512-bit implementation of the Montgomery multiplication with redundancy was then scanned, starting near the beginning of the multiplication. Two laser shots were taken at each clock cycle, so that there would be a high probability of producing a fault, as faults can rarely be produced with a 100% success rate. The chip was clocked with an external clock so that the best possible coverage of the code could be achieved. This resulted in 9000 fault attack attempts at 4500 successive clock cycles, 2521 of which produced erroneous data and 438 produced a "Card is Mute" status, where the laser provoked a fault that the chip could not recover from. All of the faulty results produced had an incorrect checksum. This validated the countermeasures for large changes in the output data. This was considered adequate, as the 4500 successive clock cycles cover 1.5 loops of the algorithm, so a fault injection had been attempted at every instruction in the main algorithm loop.

A second series of fault injections was tried at the end to make sure that a small change in the algorithm could also be detected. The end of the algorithm was scanned in the same way. The difference here was that the position was decremented from just after the end of the algorithm. This produced 1300 fault attempts on the last 650 clock cycles of the algorithm, 309 of which gave corrupted results and a further 874 resulted in the "Card is Mute" status. The first faults seen were the algorithm returning zero with the correct checksum; it was assumed that these faults were affecting the transfer of data from the results buffer to the I/O buffer. This problem is beyond the scope of this thesis. The next set of faults seen were modifications in the checksum with the correct answer given, corresponding to the last two manipulations of the checksum. The rest of the results were modifications

[†]The chip used was a prototype that was vulnerable to fault attacks; subsequent versions of this chip have proven to be more robust. This is in no way a negative comment on the MIPS architecture or smart cards in general.

in both the result and the checksum, where the fault attacked the last loop in the algorithm. The faults in this set modified one 32-bit word of the result, then two, etc. As before, no faulty answer and checksum were found that were valid.

This gives an indication of the effectiveness of fault injection attacks on the checksum algorithm proposed in Section 9.8. These results are by no means exhaustive and it is not known which model described in Section 5.3 corresponds to the faults produced.

Appendix L

Glitching k During DSA Computations

An implementation of the attack described in Section 10.3 is described below. DSA was implemented on a chip known to be vulnerable to Vcc glitches. For testing purposes a separate implementation for the generation of k was also developed.

This attack on DSA proceeds as follows: several DSA signatures are generated where the random value generated for k has been modified so that a few of the least* significant bytes of k are set to zero†. This faulty k will then be used by the card to generate a (valid) DSA signature. Using lattice reduction, the secret key α can be recovered from a collection of such signatures (see Section 10.3). In this section each of the stages in the attack is detailed in turn, showing first how k can be modified and how this technique can then be applied to a complete implementation.

In DSA a 160-bit nonce is generated and compared to q. If $k \geq q - 1$ the nonce is discarded and a new k is generated. This is done in order to ascertain that k is drawn uniformly from $\mathbb{Z}_q$ (assuming that the source used for generating the nonce is perfect). The code fragment (modified for simplicity) that was used to generate k is given below:

```
PutModulusInCopro(PrimeQ);
RandomGeneratorStart();
```

*It is also possible to run a similar attack by changing the most significant bytes of k. This is determined by the implementation.

†It would have also been possible to run a similar attack where these bytes are set to $\texttt{FF}_{16}$.

```
status = 0;
do {
   IOpeak();
   for (i=0; i<PrimeQ[0]; i++)
   {
      acCoproMessage[i+1] = ReadRandomByte();
   }
   IOpeak();
   acCoproMessage[0] = PrimeQ[0];
   LoadDataToCopro(acCoproMessage);
   status = 1;
   (j=0; j<(PrimeQ[0]+1); j++)
   {
      if (acCoproResult[j] != acCoproMessage[j])
         status = 0;
   }
} while (status == 0);
RandomGeneratorStop();
```

Note that the `IOpeaks`[‡] featured in the above code were also included in the implementation of DSA. The purpose of this is to be able to easily identify the code sections in which a fault can be injected to produce the desired effect. This could also have been done by Simple Power Analysis (see Section 4.2), but would have greatly increased the complexity of the task.

The tools used to create the glitches can be seen in Figure L.1. The card reader is a modified Clio reader, which is a specialised high precision reader that allows one glitch to be introduced, following an arbitrarily chosen number of clock cycles after the command is sent to the card. A BNC connector is present on the Clio reader which allows the I/O to be easily read; another connector produces a signal

[‡]An I/O peak is a quick transition from 1 to 0 and back again. This is ignored by the card reader as it will be much shorter than one elementary time unit in the $T = 0$ and $T = 1$ protocols, but can be seen with an oscilloscope or proprietary tools.

when a glitch is applied (in this case used as a trigger). The power consumption was measured using a differential probe situated on top of the Clio reader.

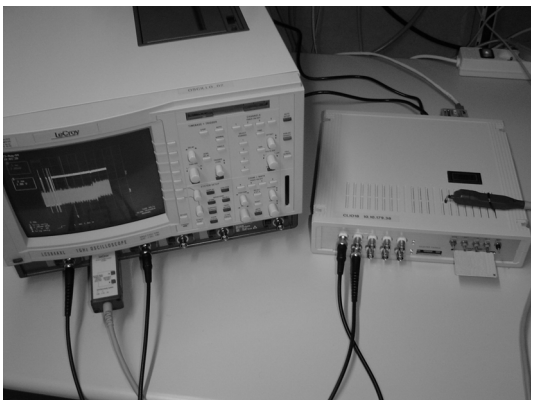


Figure L.1: Experimental set up

The command that generated k was attacked in every position between the two `IOpeak`s in the code. It was found that the fault did not affect the assignment of k to the RAM, i.e. the instruction `acCoproMessage[i+1] = ReadRandomByte();` which was always executed correctly. However, it was possible to change the evaluation of `i` during the loop. This enabled the number of least significant bytes that were reset to be chosen arbitrarily. In theory, this would produce the desired fault in k with probability $q/2^{160}$, as, if the modified k happens to be larger than q, it is discarded anyway. In practice the probability of inducing the desired fault is likely to be lower, as it is unusual for a fault induction method to work correctly every time.

An evaluation of a position that reset the last two bytes was performed. Out of 2000 attempts, 857 were corrupted. This is significantly less than would be expected, as the theoretical probability is $\simeq 0.77$. The practical results were expected to have a success rate lower than the theory would suggest, due to a slight variation in the amount of time that the smart card takes to arrive at the position where the data corruption is performed. There were other positions in the same area that return k with the same fault but with a lower frequency.

The position found was equated to the generation of k in the command that

computed the DSA signature. This was done by using the last I/O event at the end of the command sent as a reference point, and gave a rough position of where the fault needed to be injected.

As changes in the value of k were not visible in the signature, results would only be usable with a certain probability. This made the attack more complex, as the subset signatures having faulty k values had to be guessed from amongst those acquired.

To be able to identify the correct signatures, the I/O and the power consumption signals were monitored during the attacks. An example of a trace from such a monitoring is given in Figure L.2. The object of these acquisitions was to measure the time T elapsed between the end of the command sent to the card and the beginning of the calculation of r. This can be seen in the power consumption, as the chip will require more energy when the cryptographic coprocessor is activated.

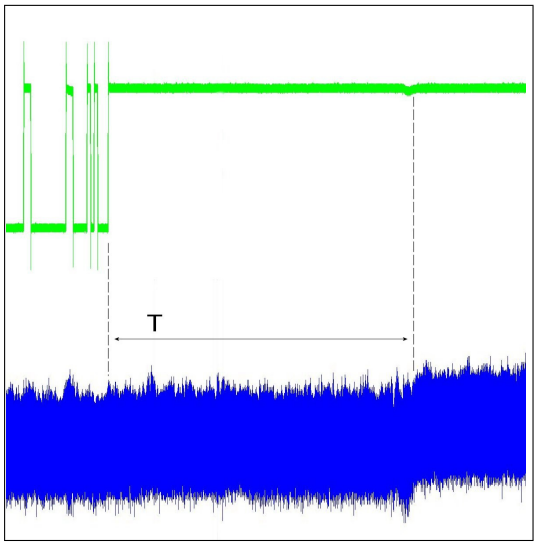


Figure L.2: I/O and power consumption (beginning of the trace of the command used to generate signatures).

If we denote by t the time taken to reach the start of the calculation of r, knowing that the generated k was smaller that q (i.e. that it was not necessary to restart this process), then, if $T = t$ we know that the command has executed properly and that k was picked correctly the first time. If $T > t$ then any fault targeting k would be a

miss (as k was regenerated given that the value of k originally produced was greater than q). Signatures resulting from commands that feature such running times can be discarded as the value of k will not present any exploitable weaknesses. When $T < t$ we know that the execution of the code generating k has been cut short, so some of the least significant bytes will be equal to zero. This allows signatures generated from corrupted k values to be identified *a posteriori*.

As the position where the fault should be injected was only approximately identified, glitches were injected in twenty different positions until a position that produced signatures with the correct characteristics (as described above) was found. The I/O peaks left in the code were used to confirm these results. Once the correct position was identified, more attacks were conducted at this position to acquire a handful of signatures. From 200 acquisitions, 38 signatures where $T < t$ were extracted.

This interpretation had to be done by a combination of the I/O and the power consumption, as after the initial calculation involving k the command no longer takes the same amount of time. This is because $0 < k \leq q$ and therefore k does not have a fixed size; consequently any calculations involving k will not always take the same amount of time. In particular, the algorithm used to calculate the modular exponentiation and the modular inverse will not take the same amount of time from one execution to another. This problem is amplified if random delays are included, as described in Section 7.2.

The lattice attack described in Section 10.3 was applied to these 38 DSA signatures, using the NTL [99] implementation of the Schnorr–Euchner BKZ Algorithm [96] with block size 20 as the lattice basis reduction algorithm. Out of the 38 signatures, 30 were taken at random to launch the lattice attack, and these turned out to be enough to disclose the DSA private key α after a few seconds on an Apple PowerBook G4. 30 signatures were taken as this number should be sufficient to reveal the private key, and to allow for any signatures in the set of 38 that did not have the relevant bits set to zero. If the attack had been unsuccessful another subset of 30 signatures would have been selected and the analysis restarted.

Bibliography

[1] L. Adams, E. J. Daly, R. Harboe-Sorensen, R. Nickson, J. Haines, W. Schafer, M. Conrad, H. Griech, J. Merkel, T. Schwall, and R. Henneck, *A verified proton induced latchup in space*, IEEE Transactions on Nuclear Science **39** (1992), 1804–1808.

[2] M.-L. Akkar and C. Giraud, *An implementation of DES and AES secure against some attacks*, Cryptogaphic Hardware and Embedded Systems — CHES 2001 (C. K. Koç, D. Naccache, and C. Paar, eds.), Lecture Notes in Computer Science, vol. 2162, Springer-Verlag, 2001, pp. 309–318.

[3] American National Standards Institute, *Financial institution key management (wholesale)*, April 1985.

[4] F. Amiel, C. Clavier, and M. Tunstall, *Collision fault analysis of DPA-resistant algorithms*, Fault Diagnosis and Tolerance in Cryptography 2006 — FDTC 06 (L. Breveglieri, I. Koren, D. Naccache, and J.-P. Seifert, eds.), Lecture Notes in Computer Science, vol. 4236, Springer-Verlag, 2006, pp. 223–236.

[5] R. Anderson and M. Kuhn, *Tamper resistance — a cautionary note*, Proceedings of the Second USENIX Workshop of Electronic Commerce, 1996, pp. 1–11.

[6] ———, *Low cost attacks on tamper resistant devices*, Security Protocols (B. Christianson, B. Crispo, T. M. A. Lomas, and M. Roe, eds.), Lecture Notes in Computer Science, vol. 1361, Springer-Verlag, 1997, pp. 125–136.

[7] Anonymous, *Season 2 interface*, `http://www.maxking.co.uk/`.

[8] C. Aumüller, P. Bier, P. Hofreiter, W. Fischer, and J.-P. Seifert, *Fault attacks on RSA with CRT: Concrete results and practical countermeasures*, Cryptographic Hardware and Embedded Systems — CHES 2002 (B. S. Kaliski, C. K. Koç, and C. Paar, eds.), Lecture Notes in Computer Science, vol. 2523, Springer-Verlag, 2002, pp. 260–275.

[9] F. Bao, R. H. Deng, Y. Han, A. Jeng, A. D. Narasimhalu, and T. Ngair, *Breaking public key cryptosystems on tamper resistant devices in the presence of transient faults*, Security Protocols (B. Christianson, B. Crispo, T. M. A. Lomas, and M. Roe, eds.), Lecture Notes in Computer Science, vol. 1361, Springer-Verlag, 1997, pp. 115–124.

[10] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan, *The sorcerer's apprentice guide to fault attacks*, Workshop on Fault Diagnosis an Tolerance in Cryptography, in association with DSN 2004 – The International Conference on Dependable Systems and Networks, June 2004.

[11] ———, *The sorcerers apprentice guide to fault attacks*, Proceedings of the IEEE **94** (2006), no. 2, 370–382.

[12] O. Benoit and M. Tunstall, *Efficient use of random delays*, Cryptology ePrint Archive, Report 2006/272, 2006, `http://eprint.iacr.org/`.

[13] G. Berger, G. Ryckewaert, R. Harboe-Sorensen, and L. Adams, *The heavy ion irradiation facility at CYCLONE — a dedicated SEE beam line*, IEEE Radiation Effects Data Workshop (1996), 78–83.

[14] D. J. Bernstein, *Cache timing attacks on AES*, 2005, `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf`.

[15] G. Bertoni, V. Zaccaria, L. Breveglieri, M. Monchiero, and G. Palermo, *AES power attack based on induced cache miss and countermeasures*, International Symposium on Information Technology: Coding and Computing — ITCC 2005, IEEE Computer Society, 2005, pp. 586–591.

[16] E. Biham and A. Shamir, *Differential cryptanalysis of DES-like cryptosystems*, Advances in Cryptology — CRYPTO '90 (A. Menezes and S. Vanstone, eds.), Lecture Notes in Computer Science, vol. 537, Springer-Verlag, 1991, pp. 2–21.

[17] ———, *Differential cryptanalysis of DES-like cryptosystems.*, Journal of Cryptology **4** (1991), no. 1, 3–72.

[18] ———, *Differential fault analysis of secret key cryptosystems*, Advances in Cryptology — CRYPTO '97 (B. S. Kaliski, ed.), Lecture Notes in Computer Science, vol. 1294, Springer-Verlag, 1997, pp. 513–525.

[19] J. Blömer and V. Krummel, *Fault based collision attacks on AES*, Fault Diagnosis and Tolerance in Cryptography — FDTC 2006 (L. Breveglieri, I. Koren, D. Naccache, and J.-P. Seifert, eds.), Lecture Notes in Computer Science, vol. 4236, Springer-Verlag, 2006, pp. 106–120.

[20] J. Blömer, M. Otto, and J.-P. Seifert, *A new RSA-CRT algorithm secure against bellcore attacks*, ACM Conference on Computer and Communications Security — CCS '03 (S. Jajodia, V. Atluri, and T. Jaeger, eds.), 2003, pp. 311–320.

[21] ———, *Wagners attack on a secure CRT-RSA algorithm reconsidered*, Fault Diagnosis and Tolerance in Cryptography 2006 — FDTC 06 (L. Breveglieri, I. Koren, D. Naccache, and J.-P. Seifert, eds.), Lecture Notes in Computer Science, vol. 4236, Springer-Verlag, 2006, pp. 13–23.

[22] J. Blömer and J.-P. Seifert, *Fault based cryptanalysis of the advanced encryption standard (AES)*, Financial Cryptography — FC 2003 (R. N. Wright, ed.), Lecture Notes in Computer Science, vol. 2742, Springer-Verlag, 2003, pp. 162–181.

[23] D. Boneh, R. A. DeMillo, and R. J. Lipton, *On the importance of checking computations*, Advances in Cryptology — EUROCRYPT '97 (W. Fumy, ed.), Lecture Notes in Computer Science, vol. 1233, Springer-Verlag, 1997, pp. 37–51.

[24] D. Boneh and R. Venkatesan, *Hardness of computing the most significant bits of secret keys in diffie-hellman and related schemes*, Advances in Cryptology — CRYPTO '96 (N. Koblitz, ed.), Lecture Notes in Computer Science, vol. 1109, Springer-Verlag, 1996, pp. 126–142.

[25] E. Brier, B. Chevallier-Mames, M. Ciet, and C. Clavier, *Why one should also secure RSA public key elements*, Cryptographic Hardware and Embedded Systems — CHES 2006 (L. Goubin and M. Matsui, eds.), Lecture Notes in Computer Science, vol. 4249, Springer-Verlag, 2006, pp. 324–338.

[26] E. Brier, C. Clavier, and F. Olivier, *Correlation power analysis with a leakage model*, Cryptographic Hardware and Embedded Systems — CHES 2004 (M. Joye and J.-J. Quisquater, eds.), Lecture Notes in Computer Science, vol. 3156, Springer-Verlag, 2004, pp. 16–29.

[27] P. Cazenave, P. Fouillat, X. Montagner, H. Barnaby, R. D. Schrimpf, L. Bonora, J. P. David, A. Touboul, M.-C. Calvet, and P. Calvel, *Total dose effects on gate controlled lateral pnp bipolar junction transistors*, IEEE Transactions on Nuclear Science **45** (1998), 2577–2583.

[28] S. Chari, C. S. Jutla, J. R. Rao, and P. Rohatgi, *Towards approaches to counteract power-analysis attacks*, Advances in Cryptology — CRYPTO '99 (M. Wiener, ed.), Lecture Notes in Computer Science, vol. 1666, Springer-Verlag, 1999, pp. 398–412.

[29] C.-N. Chen and S.-M. Yen, *Differential fault analysis on AES key schedule and some countermeasures*, Australasian Conference on Information Security and Privacy — ACISP 2003 (G. Goos, J. Hartmanis, and J. van Leeuwen, eds.), Lecture Notes in Computer Science, vol. 2727, Springer-Verlag, 2003, pp. 118–129.

[30] B. Chevallier-Mames, M. Ciet, and M. Joye, *Low-cost solutions for preventing simple side-channel analysis: Side-channel atomicity*, IEEE Transactions on Computers **53** (2004), no. 6, 760–768.

[31] H. Choukri and M. Tunstall, *Round reduction using faults*, Workshop on Fault Diagnosis and Tolerance in Cryptography 2005 — FDTC 05 (L. Breveglieri and I. Koren, eds.), 2005, pp. 13–24.

[32] M. Ciet and M. Joye, *Practical fault countermeasures for chinese remaindering based RSA*, Workshop on Fault Diagnosis and Tolerance in Cryptography 2005 — FDTC 2005 (L. Breveglieri and I. Koren, eds.), 2005, pp. 124–131.

[33] C. Clavier, *Private communication*, 2005.

[34] C. Clavier, J.-S. Coron, and N. Dabbous, *Differential power analysis in the presence of hardware countermeasures*, Cryptographic Hardware and Embedded Systems — CHES 2000 (C. K. Koç and C. Paar, eds.), Lecture Notes in Computer Science, vol. 1965, Springer-Verlag, 2000, pp. 252–263.

[35] J.-S. Coron, *Resistance against differential power analysis for elliptic curve cryptosystems*, Cryptographic Hardware and Embedded Systems — CHES 99 (C. K. Koç and C. Paar, eds.), Lecture Notes in Computer Science, vol. 1717, Springer-Verlag, 1999, pp. 292–302.

[36] P. Dusart, G. Letourneux, and O. Vivolo, *Differential fault analysis on A.E.S.*, Applied Cryptography and Network Security — ACNS 2003 (J. Zhou, M. Yung, and Y. Han, eds.), Lecture Notes in Computer Science, vol. 2846, Springer-Verlag, 2003, pp. 293–306.

[37] P. Fouillat, *Contribution à l'étude de l'interaction entre un faisceau laser et un milieu semiconducteur, applications à l'étude du latchup et à l'analyse d'états logiques dans les circuits intégrés en technologie CMOS*, Ph.D. thesis, University of Bordeaux, 1990.

[38] J. Fournier and M. Tunstall, *Cache based power analysis attacks on AES*, Australasian Conference on Information Security and Privacy — ACISP 2006 (L. M. Batten and R. Safavi-Naini, eds.), Lecture Notes in Computer Science, vol. 4058, Springer-Verlag, 2006, pp. 17–28.

[39] K. Gandolfi, C. Mourtel, and F. Olivier, *Electromagnetic analysis: Concrete results*, Cryptographic Hardware and Embedded Systems — CHES 2001 (C. K. Koç, D. Naccache, and C. Paar, eds.), Lecture Notes in Computer Science, vol. 2162, Springer-Verlag, 2001, pp. 251–261.

[40] C. Giraud, *DFA on AES*, International Conference Advanced Encryption Standard — AES 2004 (H. Dobbertin, V. Rijmen, and A. Sowa, eds.), Lecture Notes in Computer Science, vol. 3373, Springer-Verlag, 2004, pp. 27–41.

[41] C. Giraud and E. W. Knudsen, *Fault attacks on signature schemes*, Australasian Conference on Information Security and Privacy — ACISP 2004 (H. Wang, J. Pieprzyk, and V. Varadharajan, eds.), Lecture Notes in Computer Science, vol. 3108, Springer-Verlag, 2004, pp. 478–491.

[42] C. Giraud and H. Thiebeauld, *A survey on fault attacks*, Smart Card Research and Advanced Applications VI — 18th IFIP World Computer Congress (Y. Deswarte and A. A. El Kalam, eds.), Kluwer Academic, 2004, pp. 159–176.

[43] Global Platfom, *Global platform card specification, version 2.1*, 2001.

[44] S. Govindavajhala and A. W. Appel, *Using memory errors to attack a virtual machine*, IEEE Symposium on Security and Privacy 2003, 2003, pp. 154–165.

[45] D. H. Habing, *The use of lasers to simulate radiation-induced transients in semiconductor devices and circuits*, IEEE Transactions On Nuclear Science **39** (1992), 1647–1653.

[46] G. Hachez and J.-J. Quisquater, *Montgomery exponentiation with no final subtractions: Improved results*, Cryptographic Hardware and Embedded Systems — CHES 2000 (C. K. Koç and C. Paar, eds.), Lecture Notes in Computer Science, vol. 1965, Springer-Verlag, 2000, pp. 293–301.

[47] Y. Han, A. Jeng, A. D. Narasimhalu, F. Bao, R. H. Deng, and T. Nagir, *Breaking public key cryptosystems on tamper resistant devices in the presence of faults*, International Workshop Security Protocols (M. Lomas, B. Christianson, and B. Crispo, eds.), Lecture Notes in Computer Science, vol. 1361, Springer-Verlag, 1998, pp. 115–124.

[48] L. Hemme, *A differential fault attack against early rounds of (triple-)DES.*, Cryptographic Hardware and Embedded Systems — CHES 2004 (M. Joye and J.-J. Quisquater, eds.), Lecture Notes in Computer Science, vol. 3156, Springer-Verlag, 2004, pp. 254–267.

[49] J. L. Hennessy and D. A. Patterson, *Computer architecture: A quantitative approach*, Morgan Kaufmann, 2003.

[50] N. A. Howgrave-Graham and N. P. Smart, *Lattice attacks on digital signature schemes*, Design, Codes and Cryptography **23** (2001), 283–290.

[51] Infineon Technologies AG Secure and Mobile Solutions Security Group, *Security & chip cards ICs SLE88CX4000P, preliminary short product information 04.03*, 2003.

[52] International Organization for Standardization, *ISO/IEC 7816–3 information technology – identification cards – integrated circuit(s) cards with contacts – part 3: Electronic signals and transmission protocols*, 1997.

[53] International Organization for Standardization, *ISO/IEC 7816–1 identification cards – integrated circuit(s) cards with contacts – part 1: Physical characteristics*, 1998.

[54] International Organization for Standardization, *ISO/IEC 7816–2 identification cards – integrated circuit cards – part 2: Cards with contacts – dimensions and location of the contacts*, 1999.

[55] M. Joye and F. Olivier, *Side-channel attacks*, Encyclopedia of Cryptography and Security (H. van Tilborg, ed.), Kluwer Academic Publishers, 2005, pp. 571–576.

[56] M. Joye, J.-J. Quisquater, F. Bao, and R. H. Deng, *RSA-type signatures in the presense of transient faults*, Cryptography and Coding (M. Darnell, ed.), Lecture Notes in Computer Science, vol. 1355, Springer-Verlag, 1997, pp. 155–160.

[57] M. Joye and S.-M. Yen, *Checking before output may not be enough against fault based cryptanalysis*, IEEE Transactions on Computers **49** (2000), no. 9, 967–970.

[58] D. Knuth, *The art of computer programming*, third ed., vol. 2, Seminumerical Algorithms, Addison–Wesley, 2001.

[59] C. K. Koç, *Analysis of sliding window techniques for exponentiation*, Computers and Mathematics with Applications **30** (1995), no. 10, 17–24.

[60] P. Kocher, *Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems*, Advances in Cryptology — CRYPTO '96 (N. Koblitz, ed.), Lecture Notes in Computer Science, vol. 1109, Springer-Verlag, 1996, pp. 104–113.

[61] P. Kocher, J. Jaffe, and B. Jun, *Differential power analysis*, Advances in Cryptology — CRYPTO '99 (M. J. Wiener, ed.), Lecture Notes in Computer Science, vol. 1666, Springer-Verlag, 1999, pp. 388–397.

[62] R. Koga and W. A. Kolasinski, *Heavy ion induced snapback in CMOS devices*, IEEE Transactions on Nuclear Science **36** (1989), 2367–2374.

[63] R. Koga, M. D. Looper, S. D. Pinkerton, W. J. Stapor, and P. T. McDonald, *Low dose rate proton irradiation of quartz crystal resonators*, IEEE Transactions on Nuclear Science **43** (1996), 3174–3181.

[64] O. Kommerling and M. Kuhn, *Design principles for tamper resistant smartcard processors*, USENIX Workshop on Smartcard Technology, 1999, pp. 9–20.

[65] S. Kuboyama, S. Matsuda, T. Kanno, and T. Ishii, *Mechanism for single-event burnout of power MOSFETs and its characterization technique*, IEEE Transactions On Nuclear Science **39** (1992), 1698–1703.

[66] T. May and M. Woods, *A new physical mechanism for soft erros in dynamic memories*, In 16*th* International Reliability Physics Symposium, 1978.

[67] A. Menezes, P. van Oorschot, and S. Vanstone, *Handbook of applied cryptography*, CRC Press, 1997.

[68] T. S. Messerges, *Using second-order power analysis to attack DPA resistant software*, Cryptographic Hardware and Embedded Systems — CHES 2000 (Ç. K. Koç and C. Paar, eds.), Lecture Notes in Computer Science, vol. 1965, Springer-Verlag, 2000, pp. 71–77.

[69] T. S. Messerges, E. A. Dabbish, and R. H. Sloan, *Investigations of power analysis attacks on smartcards*, USENIX Workshop on Smartcard Technology, 1998, pp. 151–161.

[70] ———, *Power analysis attacks of modular exponentiation in smartcards*, Cryptogaphic Hardware and Embedded Systems — CHES '99 (Ç. K. Koç and C. Paar, eds.), Lecture Notes in Computer Science, vol. 1717, Springer-Verlag, 1999, pp. 144–157.

[71] MIPS-Technologies, *SmartMIPS ASE*, `http://www.mips.com/content/Products/`.

[72] *MIPS^{TM} architecture for programmers volume I: Introduction to the MIPS32^{TM} architecture*, Technical Report MD00082, Revision 0.95, March 2001.

[73] P. Montgomery, *Modular multiplication without trial division*, Mathematics of Computation **44** (1985), 519–521.

[74] J. Muir, *Seiferts RSA fault attack: Simplified analysis and generalizations*, Cryptology ePrint Archive, Report 2005/458, 2005, `http://eprint.iacr.org/`.

[75] M. Murdocca and V. P. Heuring, *Principles of computer architecture*, Addison-Wesley, 2000.

[76] D. Naccache, P. Q. Nguyen, M. Tunstall, and C. Whelan, *Experimenting with faults, lattices and the DSA*, Public Key Cryptography — PKC 2005 (S. Vaudenay, ed.), Lecture Notes in Computer Science, vol. 3386, Springer-Verlag, 2005, pp. 16–28.

[77] D. Naccache, M. Tunstall, and C. Whelan, *Computational improvements to differential side channel attacks*, NATO Security through Science Series D: Information and Communication Security, vol. 2, IOS Press, 2006, pp. 26–35.

[78] National Institute of Standards and Technology, *Data encryption standard (DES) (FIPS–46-3)*, 1999.

[79] National Institute of Standards and Technology, *Advanced encryption standard (AES) (FIPS–197)*, 2001.

[80] National Institute of Standards and Technology, *Security requirements for cryptographic modules (FIPS–140-2)*, 2002.

[81] K. Nguyen and M. Tunstall, *Montgomery multiplication with redundancy check*, 2006.

[82] P. Q. Nguyen and I. E. Shparlinski, *The insecurity of the digital signature algorithm with partially known nonces*, Journal of Cryptology **15** (2002), no. 3, 151–176.

[83] National Institute of Standards and Technology, *Digital signature standard (DSS) (FIPS–186-2)*, 2000.

[84] T. J. O'Gorman, *The effect of cosmic rays on soft error rate of a DRAM at ground level*, IEEE Transactions On Electronics Devices **41** (1994), 553–557.

[85] D. A. Osvik, A. Shamir, and E. Tromer, *Cache attacks and countermeasures: the case of AES*, Topics in Cryptology — CT-RSA 2006 (D. Pointcheval, ed.), Lecture Notes in Computer Science, vol. 3860, Springer-Verlag, 2006, pp. 1–20.

[86] E. Oswald, S. Mangard, C. Herbst, and S. Tillich, *Practical second-order DPA attacks for masked smart card implementations of block ciphers*, Topics in Cryptology — CT-RSA 2006 (D. Pointcheval, ed.), Lecture Notes in Computer Science, vol. 3860, Springer-Verlag, 2006, pp. 192–207.

[87] D. Page, *Theoretical use of cache memory as a cryptanalytic side–channel*, Cryptology ePrint Archive, Report 2002/169, 2002, `http://eprint.iacr.org/`.

[88] J. C. Pickel and J. T. Blandford Jr., *Cosmic ray induced errors in MOS memory circuits*, IEEE Transactions On Nuclear Science **25** (1978), 1166–1171.

[89] G. Piret and J.-J. Quisquater, *A differential fault attack technique against SPN structure, with application to the AES and KHAZAD*, Cryptographic Hardware and Embedded Systems — CHES 2003 (C. D. Walter, Ç. K. Koç, and C. Paar, eds.), Lecture Notes in Computer Science, vol. 2779, Springer-Verlag, 2003, pp. 77–88.

[90] V. Pouget, *Simulation expérimentale par impulsions laser ultra-courtes des effets des radiations ionisantes sur les circuits intégrés*, Ph.D. thesis, University of Bordeaux, 2000.

[91] W. Rankl and W. Effing, *Smart card handbook*, Wiley, 2003.

[92] J. R. Rao, P. Rohatgi, H. Scherzer, and S. Tinguely, *Partitioning attacks: or how to rapidly clone some GSM cards*, IEEE Symposium on Security and Privacy, 2002, pp. 31–41.

[93] B. G. Rax, C. I. Lee, A. H. Johnston, and C. E. Barnes, *Total dose and proton damage in optocouplers*, IEEE Transactions on Nuclear Science **43** (1996), 3167–3173.

[94] R. Rivest, A. Shamir, and L. M. Adleman, *Method for obtaining digital signatures and public-key cryptosystems*, Communications of the ACM **21** (1978), no. 2, 120–126.

[95] D. Samyde, S. P. Skorobogatov, R. J. Anderson, and J.-J. Quisquater, *On a new way to read data from memory*, Proceedings of the First International IEEE Security in Storage Workshop, 2002, pp. 65–69.

[96] C. P. Schnorr and M. Euchner, *Lattice basis reduction: improved practical algorithms and solving subset sum problems*, Math. Programming **66** (1994), 181–199.

[97] J.-P. Seifert, *On authenticated computing and RSA-based authentication*, ACM Conference on Computer and Communications Security – CCS 2005, 2005, pp. 122–127.

[98] A. Shamir, *Method and apparatus for protecting public key schemes from timing and fault attacks*, U.S. Patent Number 5,991,415, 1997, Also presented at the rump session of EUROCRYPT '97.

[99] V. Shoup, *Number theory C++ library (NTL)*, `http://www.shoup.net/ntl/`.

[100] S. Skorobogatov and R. Anderson, *Optical fault induction attacks*, Cryptographic Hardware and Embedded Systems — CHES 2002 (B. S. Kaliski, Ç. K. Koç, and C. Paar, eds.), Lecture Notes in Computer Science, vol. 2523, Springer-Verlag, 2002, pp. 2–12.

[101] E. G. Stassinopoulos, G. J. Brucker, P. Calvel, A. Baiget, C. Peyrotte, and R. Gaillard, *Charge generation by heavy ions in power MOSFETs, burnout space predictions and dynamic SEB sensitivity*, IEEE Transactions On Nuclear Science **39** (1992), 1794–1711.

[102] Sun Microsystems, *Java card 2.2.1 virtual machine specification*, 2003.

[103] TechnoData Interware, *Matrix software protection system*, `http://www.matrixlock.de/english/e_allgem.htm`.

[104] Y. Tsunoo, T. Saito, T. Suzaki, M. Shigeri, and H. Miyauchi, *Cryptanalysis of DES implemented on computers with cache*, Cryptographic Hardware and Embedded Systems — CHES 2003 (C. D. Walter, Ç. K. Koç, and C. Paar, eds.), Lecture Notes in Computer Science, vol. 2779, Springer-Verlag, 2003, pp. 62–76.

[105] K. Villegas, *Private communication*, 2006.

[106] D. Wagner, *Cryptanalysis of a provable secure CRT-RSA algorithm*, ACM Conference on Computer and Communications Security — CCS '04 (B. Pfitzmann and P. Liu, eds.), 2004, pp. 82–91.

[107] C. D. Walter, *Montgomery exponentiation needs no final subtractions*, Electronic Letters **35** (1999), no. 21, 1831–1832.

[108] ———, *Montgomery's multiplication technique: How to make it smaller and faster*, Cryptogaphic Hardware and Embedded Systems — CHES '99 (Ç. K. Koç and C. Paar, eds.), Lecture Notes in Computer Science, vol. 1717, Springer-Verlag, 1999, pp. 80–93.

[109] ———, *Data integrity in hardware for modular arithmetic*, Cryptographic Hardware and Embedded Systems — CHES 2000 (Ç. K. Koç and C. Paar, eds.), Lecture Notes in Computer Science, vol. 1965, Springer-Verlag, 2000, pp. 204–215.

[110] S.-M. Yen and D. Kim, *Cryptanalysis of two protocols for RSA with CRT based on fault infection*, Workshop on Fault Diagnosis an Tolerance in Cryptography, in association with DSN 2004 — The International Conference on Dependable Systems and Networks, June 2004.

[111] S.-M. Yen, S. Kim, S. Lim, and S.-J. Moon, *RSA speedup with residue number system immune against hardware fault cryptanalysis*, Information Security and Cryptology — ICISC 2001 (K. Kim, ed.), Lecture Notes in Computer Science, vol. 2288, Springer-Verlag, 2001, pp. 397–413.

[112] J. Ziegler, *Effect of cosmic rays on computer memories*, Science **206** (1979), 776–788.