UNIVERSITÀ DEGLI STUDI DI MILANO
Department of Computer Science "Giovanni Degli Antoni"
Doctoral School in Computer Science
Ph.D. in Computer Science
XXIX cycle

Analysis of cryptographic algorithms against theoretical and implementation attacks

INF/01

Ph.D Candidate:
Silvia Mella

Advisor:
Prof. Stelvio Cimato
Co-Advisor:
Dr. Gilles Van Assche
Coordinator:
Prof. Paolo Boldi

Academic year 2016/2017

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor in Computer Science

Ph.D. Candidate: Silvia Mella

Advisor: Prof. Stelvio Cimato

Co-Advisor: Dr. Gilles Van Assche

Coordinator: Prof. Paolo Boldi

Università degli Studi di Milano
Department of Computer Science "Giovanni degli Antoni"

Abstract

This thesis deals with theoretical and implementation analysis of cryptographic functions. Theoretical attacks exploit weaknesses in the mathematical structure of the cryptographic primitive, while implementation attacks leverage on information obtained by its physical implementation, such as leakage through physically observable parameters (side-channel analysis) or susceptibility to errors (fault analysis).

In the area of theoretical cryptanalysis, we analyze the resistance of the Keccak-f permutations to differential cryptanalysis (DC). Keccak-f is used in different cryptographic primitives: Keccak (which defines the NIST standard SHA-3), Ketje and Keyak (which are currently at the third round of the CAESAR competition) and the authenticated encryption function Kravatte. In its basic version, DC makes use of differential trails, i.e. sequences of differences through the rounds of the primitive. The power of trails in attacks can be characterized by their weight. The existence of low-weight trails over all but a few rounds would imply a low resistance with respect to DC. We thus present new techniques to efficiently generate all 6-round differential trails in Keccak-f up to a given weight, in order to improve known lower bounds. The limit weight we can reach with these new techniques is very high compared to previous attempts in literature for weakly aligned primitives. This allows us to improve the lower bound on 6 rounds from 74 to 92 for the four largest variants of Keccak-f. This result has been used by the authors of Kravatte to choose the number of rounds in their function. Thanks to their abstraction level, some of our techniques are actually more widely applicable than to Keccak-f. So, we formalize them in a generic way. The presented techniques have been integrated in the KeccakTools and are publicly available [BDPV15].

In the area of fault analysis, we present several results on differential fault analysis (DFA) on the block cipher AES. Most DFA attacks exploit faults that modify the intermediate state or round key. Very few examples have been presented, that leverage changes in the sequence of operations by reducing the number of rounds. In this direction, we present four DFA attacks that exploit faults that alter the sequence of operations during the final round. In particular, we show how DFA can be conducted when the main operations that compose the AES round function are corrupted, skipped or repeated during the final round. Another aspect of DFA we analyze is the role of the fault model in attacks. We study it from an information theoretical point of view, showing that the knowledge that the attacker has on the injected fault is fundamental to mount a successful attack. In order to soften the a-priori knowledge on the injection technique needed by the attacker, we present a new approach for DFA based on clustering, called J-DFA. The experimental results show that J-DFA allows to successfully recover the key both in classical DFA scenario and when the model does not perfectly match the faults effect. A peculiar result of this method is that, besides the preferred candidate for the key, it also provides the preferred models for the fault. This is a quite remarkable ability because it furnishes precious information which can be used to analyze, compare and characterize different specific injection techniques on different devices.

In the area of side-channel attacks, we improve and extend existing attacks against the RSA algorithm, known as partial key exposure attacks. These attacks on RSA show how it is possible to find the factorization of the modulus from the knowledge of some bits of the private key. We present new partial key exposure attacks when the countermeasure known as exponent blinding is used. We first improve known results for common RSA setting by reducing the number of bits or by simplifying the mathematical analysis. Then we present novel attacks for RSA implemented using the Chinese Remainder Theorem, a scenario that has never been analyzed before in this context.

List of Publications

Part of the contributions of this thesis has been published in the following papers.

In Journals:

- Silvia Mella, Joan Daemen, Gilles Van Assche. New techniques for trail bounds and application to differential trails in KECCAK. IACR Transactions on Symmetric Cryptology, 2017(1), pages 329-357, 2017.

In Proceedings of International Conferences/Workshops:

- Claudio Ferretti, Silvia Mella, and Filippo Melzani. The role of the fault model in DFA against AES. In Proceedings of the Third Workshop on Hardware and Architectural Support for Security and Privacy (HASP 2014), pages 4:1-4:8. Minneapolis, MN, USA, June 15, 2014. ACM.
- Silvia Mella, Filippo Melzani. Differential fault attacks against AES tampering with the instruction flow. In Proceedings of the 11th International Conference on Security and Cryptography (SECRYPT 2014), pages 1-6. Wien, Austria, August 28-30, 2014.
- Stelvio Cimato, Silvia Mella, Ruggero Susella. New results for partial key exposure on RSA with exponent blinding. In Proceedings of the 12th International Conference on Security and Cryptography (SECRYPT 2015), pages 136-147. Colmar, Alsace, France, July 20-22, 2015.
- Stelvio Cimato, Silvia Mella, Ruggero Susella. Partial key exposure attacks on RSA with exponent blinding. In 12th International Joint Conference on e-Business and Telecommunications (ICETE 2015) Revised and Extended Selected Papers, pages 364-385. Springer International Publishing.
- Luca Magri, Silvia Mella, Filippo Melzani, Beatrice Rossi, Pasqualina Fragneto. J-DFA: a novel approach for robust differential fault analysis. In proceedings of the 12th Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC 2015), pages 35-44. Saint Malo, France, September 13, 2015. IEEE Computer Society.
- Stelvio Cimato, Ernesto Damiani, Silvia Mella, Ching-Nung Yang. Key Recovery in public clouds: a survey on cross-VM side channel attacks. International Conference on Cloud Computing and Security (ICCCS 2016) Revised Selected Papers, pages 456-467. Nanjing, China, July 29-31, 2016. Springer International Publishing.

Contents

Abstract V

List of Publications VII

List of Figures XII

List of Tables XIV

1 Introduction 1
- 1.1 Cryptanalysis 1
  - 1.1.1 Theoretical attacks 2
  - 1.1.2 Implementation attacks 3
- 1.2 Thesis contributions 6
- 1.3 Thesis structure 7

Part I Trail search in Keccak-*f*

2 Introduction 11

3 Preliminaries on differential cryptanalysis 13
- 3.1 Differentials 13
- 3.2 Differential trails 15
- 3.3 Trail extension 16
- 3.4 Trail cores 16

4 A generic approach for trail generation 19
- 4.1 Generating trails 19
  - 4.1.1 The first-order approach: trails from extending round differentials 19
  - 4.1.2 The second-order method: trails from extending two-round trails 21
  - 4.1.3 Lower bounding the weight starting from two-round trail cores 22
- 4.2 Tree traversal approach 22
  - 4.2.1 Units and unit-lists 23
  - 4.2.2 Lower bound on the cost 25
  - 4.2.3 Traversing the unit-tree 25
- 4.3 Searching by canonicity 26

5 Keccak-*f* 27
- 5.1 The KECCAK-*f* state 27
- 5.2 The KECCAK-*f* round function 28
- 5.3 Difference propagation properties of the step mappings 28

5.3.1 Properties of χ . . . 30
5.3.2 Properties of θ . . . 30
5.3.3 Properties of π . . . 33
5.3.4 Properties of ρ . . . 33
5.3.5 Properties of ι . . . 35
5.3.6 Translation invariance along the z-axis . . . 35
5.4 Differential trails in KECCAK-f . . . 36
5.4.1 Extension of trails and trail cores in KECCAK-f . . . 37
5.4.2 Extension of trail cores in KECCAK-f . . . 38
5.4.3 The Matryoshka structure . . . 38

6 Lower bounding the weight of trails in Keccak-f . . . 39
6.1 Scanning the space of trails in Keccak . . . 39
6.1.1 Scanning the space of 6-round trail cores . . . 39
6.1.2 Scanning the space of 3-round trail cores . . . 40
6.1.3 Covering the space of 4 and 5 rounds . . . 42
6.2 Generating two-round trail cores in KECCAK-f as a tree traversal . . . 43
6.2.1 Generating trail cores in the kernel . . . 43
6.2.2 Generating trail cores outside the kernel . . . 47
6.2.3 An alternative approach to generate trail cores outside the kernel . . . 54
6.3 Extension of trails in KECCAK-f . . . 58
6.3.1 Extension as a tree traversal . . . 58
6.3.2 Forward extension . . . 59
6.3.3 Backward extension . . . 62
6.4 Experimental results . . . 65
6.4.1 2-round trail cores . . . 66
6.4.2 3-round trail cores . . . 67
6.4.3 Extension to 6-rounds . . . 67
6.4.4 Lightest trails . . . 68
6.4.5 Bounds on n_r-rounds . . . 69
6.5 Execution times . . . 81
6.6 Comparison with [DV12] . . . 82
6.6.1 Limitations of the previous techniques . . . 82
6.6.2 Covering the space of 3-round trails up to weight 36 . . . 84

7 Conclusions . . . 85

Part II Differential Fault Analysis against AES

8 Introduction . . . 91

9 Preliminaries on DFA and AES . . . 93
9.1 Fault injection Techniques . . . 94
9.2 Countermeasures . . . 95
9.3 The Advanced Encryption Standard . . . 96
9.3.1 AddRoundKey step . . . 96
9.3.2 SubBytes step . . . 97
9.3.3 ShiftRows step . . . 98
9.3.4 MixColumns . . . 98
9.3.5 Key schedule . . . 98
9.4 Previous DFA Results on AES . . . 99

10 DFA against AES tampering with the instruction flow 103
10.1 Tampering with the instruction flow 103
10.1.1 Hardware implementations 104
10.2 DFA against the round steps 105
10.2.1 Altering the execution of SubBytes 105
10.2.2 Jumping the ShiftRows 107
10.2.3 Executing the MixColumns 109
10.2.4 Jumping the AddRoundKey 109
10.3 Effectiveness against countermeasures 110

11 The role of the fault model in DFA 111
11.1 Information theory of DFA against AES 111
11.1.1 Giraud's attack 111
11.1.2 Piret and Quisquater's attack 113
11.1.3 Moradi, Shalmani and Salmasizadeh's attack 114
11.1.4 Recap on attacks 115
11.2 How the fault model affects the attack 116
11.2.1 Application to Giraud's attack 116
11.2.2 Application to Piret and Quisquater's attack 117
11.2.3 Application to Moradi, Shalmani and Salmasizadeh's attack 117
11.3 How the knowledge of the model affects the attack 117
11.3.1 Giraud 118
11.3.2 Piret and Quisquater 119
11.3.3 Moradi, Shalmani and Salmasizadeh 119
11.3.4 Recap on probabilities 120
11.4 A concrete example 120

12 J-DFA: a novel approach to differential fault analysis 123
12.1 DFA against last round of AES 124
12.2 J-Linkage: an overview 125
12.2.1 Conceptual representation 125
12.2.2 Clustering 126
12.3 J-DFA: DFA based on J-linkage 126
12.3.1 J-DFA steps 127
12.3.2 Robustness of J-DFA 129
12.4 Evaluation results 129
12.4.1 J-DFA with profiling 130
12.4.2 J-DFA without profiling 131

13 Conclusions 137

Part III Partial key exposure attacks against RSA

14 Introduction 143

15 Preliminaries on partial key exposure attacks and RSA 145
15.1 RSA Applications 145
15.1.1 CRT-RSA 145
15.1.2 Exponent Blinding 146
15.1.3 Common Parameters Setting 146
15.2 Partial key exposure attacks 147
15.3 General Strategy 147
15.3.1 Lattices 147

15.3.2 General Strategy ... 148

16 New attacks on RSA with exponent blinding ... 151
16.1 Attacks on RSA ... 151
16.1.1 Partial Information on LSBs of d^* ... 151
16.1.2 Partial Information on MSBs of d^* ... 153
16.2 Attacks on CRT-RSA ... 155
16.2.1 Partial Information on LSBs of $\mathbf{d_p^*}$ ... 155
16.2.2 Partial Information on MSBs of $\mathbf{d_p^*}$ ... 156
16.3 Experimental Results ... 157
16.3.1 Results with known LSBs of $\mathbf{d}^*$ ... 158
16.3.2 Results with known MSBs of $\mathbf{d}^*$ ... 158
16.3.3 Results with known LSBs of $\mathbf{d_p^*}$ ... 159
16.3.4 Results with known MSBs of $\mathbf{d_p^*}$ ... 160

17 Conclusions ... 161

18 Closing remarks and future directions ... 163

References ... 165

List of Figures

4.1 Number of round differentials in Keccak-f 20
4.2 Number of 2-round trail cores in Keccak-f 22

5.1 Naming conventions for parts of the Keccak-f state 29
5.2 χ applied to a single row.......... 30
5.3 θ applied to a single bit 31
5.4 The action of the θ map. 32
5.5 Action of θ on patterns in the kernel. 33
5.6 Example of a parity pattern and θ-effect superimposed. 34
5.7 π applied to a slice.......... 34
5.8 ρ applied to the lanes. 34
5.9 Translation of a state pattern by 2 positions along the z-axis. 35
5.10 Example of symmetric state pattern and symmetric parity pattern. 36
5.11 Example of non-symmetric state pattern and symmetric parity-pattern. 36
5.12 The Matryoshka consequence.......... 38

6.1 Possible orbital positions in a column. 44
6.2 Possible orbitals in a column. 44
6.3 Plane representation of the orbital-tree. 45
6.4 A difference pattern outside the kernel.......... 48
6.5 Plane representation of the free-orbital tree. 48
6.6 Plane representation of the run-tree. 52
6.7 Example of pattern that cannot be forward extended in the kernel. 60
6.8 Example of pattern that can be forward extended in the kernel. 60
6.9 Example on backward extension. 64
6.10 Number of 2-round trail cores $\langle a, b\rangle$ in $|K|$ with $\mathrm{w}^{\mathrm{rev}}(a) = T$.......... 70
6.11 Number of 2-round trail cores $\langle a, b\rangle$ in $|K|$ with $\mathrm{w}(b) = T$. 70
6.12 Number of 2-round trail cores $\langle a, b\rangle$ in $|N|$ with $2\mathrm{w}^{\mathrm{rev}}(a) + \mathrm{w}(b) = T$. 71
6.13 Number of 2-round trail cores $\langle a, b\rangle$ in $|N|$ with $\mathrm{w}^{\mathrm{rev}}(a) + 2\mathrm{w}(b) = T$. 71
6.14 Number of 2-round trail cores $\langle a, b\rangle$ in $|N|$ with $\mathrm{w}^{\mathrm{rev}}(a) + \mathrm{w}(b) = T$. 72
6.15 Number of 2-round trail cores $\langle a, b\rangle$ in $|K|$ with $\chi(b)$ in $|K|$ and $\mathrm{w}^{\mathrm{rev}}(a) + \mathrm{w}(b) = T$. 72
6.16 Number of 2-round trail cores $\langle a, b\rangle$ in $|K|$ with $\mathrm{w}^{\mathrm{rev}}(a) + 2\mathrm{w}(b) = T$. 73
6.17 Number of 2-round trail cores $\langle a, b\rangle$ in $|K|$ with $2\mathrm{w}^{\mathrm{rev}}(a) + \mathrm{w}(b) = T$. 73
6.18 Number of 2-round trail cores $\langle a, b\rangle$ in $|K|$ with $\mathrm{w}^{\mathrm{rev}}(a) \leq T$.......... 74
6.19 Number of 2-round trail cores $\langle a, b\rangle$ in $|K|$ with $\mathrm{w}(b) \leq T$. 74
6.20 Number of 2-round trail cores $\langle a, b\rangle$ in $|N|$ with $\mathrm{w}^{\mathrm{rev}}(a) + \mathrm{w}(b) \leq T$. 75
6.21 Number of 2-round trail cores $\langle a, b\rangle$ in $|N|$ with $2\mathrm{w}^{\mathrm{rev}}(a) + \mathrm{w}(b) \leq T$. 75
6.22 Number of 2-round trail cores $\langle a, b\rangle$ in $|N|$ with $\mathrm{w}^{\mathrm{rev}}(a) + 2\mathrm{w}(b) \leq T$. 76
6.23 Number of 2-round trail cores $\langle a, b\rangle$ in $|K|$ with $\chi(b)$ in $|K|$ and $\mathrm{w}^{\mathrm{rev}}(a) + \mathrm{w}(b) \leq T$. 76

6.24 Number of 2-round trail cores $\langle a, b\rangle$ in $|K|$ with $\mathrm{w}^{\mathrm{rev}}(a) + 2\mathrm{w}(b) \leq T$. 77
6.25 Number of 2-round trail cores $\langle a, b\rangle$ in $|K|$ with $2\mathrm{w}^{\mathrm{rev}}(a) + \mathrm{w}(b) \leq T$. 77
6.26 Number of 3-round trail cores with profile $|K|K|$ and weight T_3. 78
6.27 Number of 3-round trail cores with profile $|K|N|$ and weight T_3. 78
6.28 Number of 3-round trail cores with profile $|N|K|$ and weight T_3. 79
6.29 Number of 3-round trail cores with profile $|N|N|$ and weight T_3. 79
6.30 Number of all 3-round trail cores with weight T_3. 80

7.1 Number of 1,2 and 3-round trail cores in Keccak-f 86

9.1 Schematic representation of the AES transformation. 97
9.2 One round of the key schedule for AES-128. 99

10.1 The number of bytes x such that $x \oplus SubBytes(x) = \delta$. 106
10.2 The number of bytes x such that $InvSubBytes(x) \oplus SubBytes(x) = \delta$. 108

11.1 Fault's propagation in Giraud's attack. 112
11.2 Fault's diffusion caused by the MixColumns operation in Piret's attack. 113
11.3 Fault's diffusion in Moradi's attack. 115
11.4 Probability in Moradi's attack. 121

12.1 J-DFA in a nutshell. 128
12.2 Efficiency of the attack when the model H and the reality E differ. 133
12.3 Preferences of key candidates using P. 135
12.4 Preferences of key candidates using $\tilde{P}$. 135

List of Tables

5.1 Weight and minimum reverse weight 30
5.2 Affine space of possible output differences 31
5.3 The offsets of ρ 35

6.1 Results on extension in the kernel. 62
6.2 Results on backward extension in the kernel. 63
6.3 Number of 2-round trail cores generated for each parity profile. 66
6.4 Number of 3-round trail cores up to weight 45. 67
6.5 Lightest trails found for different variants of Keccak-f. 68
6.6 Bounds on the minimum weight up to 6 rounds. 68
6.7 Bounds on the weight of trails over the total number of rounds. 69
6.8 Execution time for the generation of 2-round trail cores. 81
6.9 Execution time for the extension of 2-round trail cores to 3 rounds. 82
6.10 Total execution time to cover the space of 3-round trail cores. 82
6.11 Report on parity patterns generated with KeccakTools. 83
6.12 Execution time for 3-round trail cores up to 36 using KeccakTools. 84
6.13 Execution time for 3-round trail cores up to 36. 84

7.1 Minimum weight or range for the minimum weight. 86

9.1 The S-box of AES. 98

11.1 Recap on known DFA from information theory point of view. 116
11.2 Probabilities to find correct, wrong or no key for DFA. 120

12.1 Number of experiments required by different fault models. 131
12.2 Number of experiments required in case $H \subset E$. 132
12.3 Number of experiments required in case $H \supset E$. 132

16.1 Experimental results on RSA using LSBs. 158
16.2 Experimental results on RSA using LSBs compared with [JL12]. 159
16.3 Experimental results on RSA using MSBs. 159
16.4 Experimental results on RSA using MSBs and LSBs. 159
16.5 Experimental results on CRT-RSA using LSBs. 160
16.6 Experimental results on RSA using MSBs. 160

17.1 Recap on experientals results on RSA and CRT-RSA. 162

1 Introduction

The importance of keeping information secret has been known since the times of ancient Egyptians and the adoption of cryptography throughout history is documented by several examples, like the substitution cipher that Julius Caesar used to communicate with his generals, the ancient Greek scytale transposition cipher and the XV century polyalphabetic ciphers of Alberti and Vigenère.

For centuries cryptography has been a prerogative of military and diplomatic services. With the proliferation of computers and digital communication systems, the amount of exchanged data (and the rate of its transmission) has seen a huge increase also in the private sector. Nowadays, many processes and applications rely on cryptography. They include, electronic commerce, internet banking, SIM cards, bank cards, electronic passport, pay-TV set-top box, smart meters and also vehicles.

All these applications come with security requirements and cryptography provides solutions to them. In fact, to achieve confidentiality, data integrity and authentication, cryptography provides encryption schemes, electronic signatures, message authentication codes, hash functions, authenticated encryption and many other systems.

The complexity of cryptographic algorithms increased with time. In the past it was mainly based on substitution schemes, while nowadays it uses tools from complexity theory, information theory and number theory. The need is of course that of contrasting cryptanalysis that is become more and more effective also thanks to the increasing computational resources available.

1.1 Cryptanalysis

The goal of cryptanalysis is to assess the security of cryptographic primitives. Finding attacks or properties not present in ideal instances typically contributes to the cryptanalysis of a given function. At the same time, cryptanalysis can also provide positive results excluding classes of attacks, thereby allowing research to focus on potentially weaker aspects of the system.

Cryptanalysis is also part of the design process of new primitives. Designers should perform it since early phases of the development to evaluate whether their system has potential vulnerabilities and consequently decide whether to amend it. It is common practice to publish a design and invite others to attack it. This is the approach that NIST decided to adopt for the selection of the AES and SHA-3 standards, announcing public competitions.

The starting point for the analysis of any cipher is that the system is known by the attacker. This is known also as the Kerckhoffs's second principle, which states that a system must be secure even if the attacker knows all the details of the system. For instance, for an encryption scheme security should depend only on the secrecy of the key. This principle is widely accepted for commercial systems, where the standard network protocols require that all parties involved in the communication must be able to participate and it is also an unavoidable consequence of the mass volume production of cryptographic devices. In addition, the idea of the algorithm being widely known favors the interoperability of devices from different manufacturers. In some circumstances, as in the military community, it might happen that the algorithm is kept secret, simply because it makes cryptanalysis harder to perform. However, it is secrecy might be compromised.

If the algorithm is known, anybody could, at least in principle, perform a brute force attack. Defeating the brute-force attack is the first, obvious but fundamental, requirement for any secure cryptosystem. Such a requirement influences the choice of some parameters during the design of a cryptographic algorithm, as the size of the secret key in an encryption scheme.

There exist different families of attacks, we can group them into two classes. The first are theoretical or algorithmic attacks and exploit weaknesses in the mathematical structure of the cryptographic algorithm. The second are implementation attacks. They leverage on information obtained by the physical implementation of the cryptographic primitive or the device on which it is performed. They can exploit physical measurements, as power consumption, or intentionally induced faults.

1.1.1 Theoretical attacks

There are different goals a cryptanalyst may want to achieve and these depend on the primitive under attack. The main tasks are usually the following:

- Key recovery: finding the secret key of an encryption scheme allows to decrypt all ciphertexts previously encrypted with that key (or the corresponding public key in the public-key scenario).
- Information deduction: as obtaining information on the plaintext given the ciphertext or on the message given its digest.
- Forgery: when it concerns authentication, the attacker may be interested into play the role of a trusted party and thus into forge a signature or a message authentication code.
- Distinguishing attacks: where the attacker can build a procedure that allows her to distinguish the output of the primitive from a random function. Such a procedure is called *distinguisher*. A distinguisher can be always built when the output of a cipher is biased. The analysis that leads to the detection of biased output could be further developed to obtain one of the previous tasks.

Most of the times, cryptanalysis is initially performed on simplified versions of the cryptographic function. For instance, in iterated primitives it is first conducted on reduced round variants. The aim of analyzing such simplified versions is to gain insights into the structural properties of the cipher. Building upon previous results, attacks can be improved over time, possibly up to a point where the security of the primitive is severely questioned. With iterated primitives, it means that the number of attacked rounds can be progressively increased until the full cipher is broken. Many attacks become exponentially more difficult as rounds are added, so it is possible that the full cryptosystem is strong even though reduced-round variants are weak.

Attacks can be classified by the kind of information available to the attacker. The canonical models are the following.

- Ciphertext-only attack: the attacker can only observe the ciphertexts output by the algorithm. This model assumes the weakest attacker and any cryptosystem susceptible to this kind of attack is considered completely insecure. Examples of ciphertext-only attacks are frequency attacks, which use statistical techniques to break monoalphabetic substitution ciphers. Also some modern ciphers are susceptible of this kind of attacks, such as the RC4 stream cipher and the Akelarre block cipher [KR00, MS01].
- Known-plaintext attack: the attacker knows some ciphertexts and the corresponding plaintexts. The Caesar cipher is an example of classical cipher prone to this kind of attacks. Another famous example is the Enigma machine, whose ciphertexts were decrypted exploiting the fact that some messages started or contained fixed words, like those used to communicate weather broadcasts. In modern ciphers, an example is the PKZIP cipher, used in older

versions of the ZIP format. Knowing only one file of an encrypted archive is sufficient to recover the key [BK94].
- (Adaptive) Chosen-plaintext attacks: the attacker can choose some plaintexts and obtain the corresponding ciphertexts. When the choice of the plaintexts is based on the knowledge of previous plaintext/ciphertext pairs, the attack is called adaptive. Ciphers that are secure against chosen-plaintext attacks, are also secure against known-plaintexts and ciphertext-only attacks. Differential and linear cryptanalysis are examples of chosen-plaintext attacks.
- (Adaptive) Chosen-ciphertext attacks: the attacker can choose some ciphertexts and obtain the corresponding plaintexts. When the choice of the ciphertext is based on the knowledge of previous plaintext/ciphertext pairs, the attack is called adaptive. A practical attack against systems using the RSA algorithm was presented in 1998 [Ble98], including the SSL protocol used at the time.
- Related-key attack: the attacker can observe ciphertexts obtained using different keys, which are related in some way that the attacker knows. These attacks may be possible when the key generation algorithm has some weakness that allows relationships between different keys. An example of protocol prone to this kind of attacks is WEP used in WiFi wireless networks [SIR04].

Any attack requires a number of resources, which are:

- the number of ciphertexts or of plaintext/ciphertext pairs needed;
- the number of computations (i.e. encryptions or decryptions) that must be performed:
- the amount of memory needed.

A system is broken when an attack is found that requires less resources than brute force. Notice that in some cases such attacks can be only theoretical. That means that the number of required resources, even if less than brute force, is still too high to be practical. For instance, the best known key-recovery attack on full AES128 takes 2^{126} operations and 2^{56} bits of data. So, it performs better than brute force attack, but still does not undermine AES security in practice.

Some cryptographic algorithms base their security on the infeasibility to solve a mathematical problem. This is typically the case for public key cryptography. For instance, integer factorization for RSA, the discrete logarithm problem for the Diffie-Hellman protocol or the shortest vector problem for lattice based cryptosystems. In all these cases, the cryptosystem is believed to be secure since all known methods for solving that problem are efficient enough to be practical. However, for any of these problems, efficient methods might be found in the future and such a discovery would make any cryptosystem based on it useless.

1.1.2 Implementation attacks

In implementation attacks, the adversary observes or manipulates some physical parameters of the device running the cryptographic algorithm. For instance, the attacker can observe execution duration, power consumption, electromagnetic radiation, or tamper with the clock signal and the supply voltage.

One of the system most affected by implementation attacks is smart card. Smart cards are very small, inexpensive and the attacker could easily get into possession of the complete system and operate it at will.

Implementation attacks can be classified based on the power the attacker has:

- Passive attacks: the attacker observes the execution of the algorithm. She can feed inputs but does not interfere with the execution. These attacks comprise side-channel attacks, where physical leakages are observed.
- Active attacks: the attacker manipulates an implementation such that it behaves abnormally. They comprise fault attacks where errors are introduced during the execution of the cryptographic algorithm.

Implementation attacks can also be classified based on the level of intrusion into the system:

- Non-invasive attacks: the device remains intact and the attacker can only observe environmental parameters;
- Invasive attacks: the case or package of the device is removed. For instance, a microchip is decapsulated, to get access to the internal components.

These two classifications are not independent. Both non-invasive and invasive attacks can be active or passive. Side-channel attacks are mainly non-invasive, but, for instance, attacks exploiting electromagnetic radiation benefit from partially removing the package of an integrated circuit, since this action enables to register stronger fields close to the silicon chip surface. Invasive attacks are mainly active, but there are special cases. Probing attacks, where the adversary physically connects to wires of an integrated circuit to read the transferred bits, do not manipulate the execution.

Side channel attacks

Side-channel attacks can be hard to detect since they are mainly non-invasive and do not interfere with the normal device operation.

Physical observables that are usually exploited in side-channel attacks are timing [Koc96], power consumption [KJJ99], electromagnetic radiation [QS02], acoustic [ST], optical [FH08, SA02] and thermal [BDK^{+}].

A typical side-channel attack consists of two main stages:

- Online stage: the attacker performs measurements of a certain physical observable while the target device is processing some sensitive data. The result is a set of side-channel traces, namely vectors of leakage samples observed over a period of time.
- Offline stage: the attacker analyzes the collected traces and the corresponding inputs or outputs to recover the key. The result is one or several key candidates. This stage usually includes pre-processing of traces, like selecting the most informative traces, de-noising or traces alignment in time dimension.

In some cases, these two stages may be preceded by a profiling stage. During this stage, the attacker can characterize the leakage, but it can be performed only if the attacker has a copy of the attacked device.

In general, all these attacks require a so-called leakage model. Such a model defines the dependency between the value of an internal variable processed by the algorithm and the corresponding physical observable. The two commonly used are Hamming weight model and Hamming distance model.

One of the best known attacks related to side-channel analysis is the so-called power analysis. Every electronic system requires a power supply to work. The basis of the attack is to notice that the power consumption is not constant in time, rather it is dependent on the operation currently performed by the system. Consider a system performing a modular exponentiation via the simple square-and-multiply algorithm. It is known that if the i-th bit of the secret key is equal to zero, then a square operation is performed, otherwise a square operation followed by a multiplication is executed. If it is practically feasible to measure the power consumption profile of the device by means of an ordinary oscilloscope or some other kind of measurement equipment, it becomes possible to understand, from the power profile, whether a single square operation or a square operation followed by a multiplication has been performed at a certain point of the computation, thus recovering the secret bit of the key. And this can be replicated for each bit of the key, recovering the whole secret. This kind of attack in its simplest form takes the name of Simple Power Analysis (SPA).

More complex but also more powerful is Differential Power Analysis (DPA). In a DPA attack, the adversary guesses the value for the chunk of the key processed in a selected time frame. For a set of known inputs, she calculates the values of a target variable, that depends on this key chunk. Then she calculates the predicted leakage of the device using an appropriate leakage model. Finally, using a certain statistical test, he compares the predicted leakage with the observed leakage. From all the possible candidates for the chunk of key, those exhibiting higher correlation values are more likely to be correct.

In order to contrast these attacks, several countermeasures have been proposed. The simplest one introduces some dummies operations. In fact, the attack against the square-and-multiply algorithm described above, is based on an asymmetry of the implementation. To balance the algorithm, a dummy multiplication when the bit of the key is equal to zero can be introduced, whose result in then discarded. This solution is known as square-and-multiply always algorithm. At the price of performance degradation and higher power consumption, the device becomes more resistant against power attacks.

Other solutions are based on reformulations of the cipher in order to make the computation independent of the secret key bits. One way is through randomization. The algorithm is executed using a different random input each time. In this way, the side channel information depends on an unknown variable which is changing continuously and thus is uncorrelated to the secret data. In the case of modular exponentiation, some examples are modulus randomization, message randomization and exponent randomization.

Fault attacks

Fault attacks are active attacks and, because of their active nature, they are harder to counter than side-channel attacks.

They consist of two phases:

- Online phase: the attacker alters the normal execution of the algorithm by injecting faults to introduce errors in the computation. The result is a wrong ciphertext, which is collected.
- Offline phase: the result of the faulted computation (usually along with the result of a correct computation with the same input) is exploited, for instance, to recover the key.

Faults can be injected using different techniques and producing different effects. Some examples are the following.

- Power spikes: this is a non-invasive way to induce faults. It can cause the skipping of an instruction or the gathering of a wrong data from a bus [KK99, BECN$^+$04];
- Clock Glitches: as power spikes, they are non-invasive and it can result in data corruption, or skipping of an instruction [KK99, BECN$^+$04, BGV11];
- Eddy currents: an external electromagnetic field can induce eddy currents on the surface of the chip, which can cause a single bit fault [QS02];
- Laser beam: this is an invasive technique. The laser beam can be focused on a particular region of the chip, for instance on the memory or register. Even a single bit can be set or reset [SA02, vWWM11];

The effect of fault injection can be either transient, when it is limited in time, or permanent, when the hardware is damaged in an irrecoverable way and the fault persists even if fault injection is not performed any more.

The first example of using hardware faults to attack a cryptographic system, was presented by Boneh et al. in 1997 [BDL97]. This attack could recover the secret key of an RSA implementation using the Chinese Remainder Theorem. Following the same idea of applying faults, the attacks have been extended to symmetric ciphers [BS97]. The technique introduced against block ciphers is referred to as Differential Fault Analysis (DFA) and it consists of analyzing the difference between correct and faulty ciphertexts in order to obtain information on the secret key. DFA attacks have been presented against several symmetric encryption schemes, such as DES and triple-DES, CLEFIA and AES.

Fault attacks require a certain fault model, that is the knowledge that a specific fault injection leads to a particular kind of corruption. For instance, a fault model is a flip of a single bit within an operand or the change of a byte to 0 or to 1 (this is known as the stuck-at fault model).

Countermeasures to fault attacks can be implemented at different levels. On the hardware level, they aim mainly at preventing fault injection. They consist, for instance, in sensors for detecting variations in power supply voltage and clock frequency to prevent power spikes and clock glitches. On the algorithm level, countermeasures generally aim at detecting faulty values. They are based on introducing redundancy and checks into computations. For instance, the algorithm is executed twice and the results are checked to be equal.

1.2 Thesis contributions

There are two directions in cryptanalysis. First, it attempts to find weaknesses in algorithms or their implementation, improving already existing attacks, finding new attacks and identifying sources of vulnerabilities. Second, it provides some kind of proof on the resistance against classes of attacks, allowing to exclude them as potential concerns.

In response to these challenges, in this thesis we work on differential cryptanalysis, fault attacks and partial key exposure attacks. Each technique is applied to a different cryptographic algorithm.

First, we study the resistance of the Keccak-f permutation against differential cryptanalysis (DC). In its basic version, DC makes use of differential trails, that are sequences of differences through the rounds of the primitive. The power of trails in attacks can be measured by their weight. The complexity of an attack using a trail of weight w is estimated by 2^w. Therefore, low-weight trails are a potential weakness. We analyze round-reduced versions of Keccak-f and find lower bounds for the minimum weight of trails in such versions. Then, building upon these results, we deduce lower bounds for trails over the total number of rounds of the primitive. The obtained results apply to all the cryptographic primitives based on the Keccak-f permutation: Keccak and thus the hash functions and the extendable-output functions that compose the SHA-3 and SHAKE families; Ketje, an authenticated encryption function with support for message associated data; Keyak, an authenticated encryption scheme with support for associated data and sessions; Kravatte, a permutation-based authenticated encryption scheme. It is worth noticing that the number of rounds adopted in Kravatte has been chosen based on the results of this thesis. Thanks to its abstraction level, the methodology we introduce to find lower bounds is more widely applicable than to Keccak-f. It can in principle be applied to those cryptographic primitives with a bit-oriented mixing layers and a relatively lightweight non-linear layer.

Then, we study differential fault attacks (DFA) against the AES algorithm, by presenting new attacks and improving existing ones. We focus on the offline phase and replace the online phase by software simulations. Since most of the attacks in literature exploit faults induced on intermediate data (i.e. on the state or the key), we investigate whether it is possible to perform DFA tampering with the sequence of operations. In particular, we show how to exploit faulted ciphertexts obtained by jumping one of the step composing the final round. Subsequently, we analyze the role that the fault model has in the offline phase of some well known DFA attacks. We will show how the effectiveness of these attacks can be compromised when the fault model does not correspond to the actual fault injection effect. To avoid this behavior, we present a new approach to analyze faulted ciphertexts using a clustering technique. We will show how the fault model constraints can be relaxed up to the point where all possible faults are admitted into the model.

Finally, we study a family of attacks against the RSA algorithm, known as partial key exposure attacks. These attacks make use of lattice reduction techniques to achieve full key recovery when some bits of the key are known. These attacks are interesting in the context of DPA, where chunks of the key are retrieved one at a time. We analyze the effectiveness of the exponent blinding countermeasure against this class of attacks. We first improve existing attacks on classic implementation of RSA and then present new attacks for the RSA variant implemented using the Chinese Reminder Theorem.

1.3 Thesis structure

The thesis is divided into three parts, each dealing with a different algorithm and type of cryptanalytic technique.

Part I is dedicated to Keccak-f and the study of its resistance against DC. In Chapter 2 we motivate our research and discuss related works. In Chapter 3 we first recall some basic concepts on differential cryptanalysis, especially on differential trails. In Chapter 4 we formalize our method to find lower bounds on the weight of trails in a generic way. In Chapter 5 we describe the Keccak-f permutation, focusing our attention on the difference propagation properties of the steps composing its round function. In Chapter 6 we apply our generic approach to Keccak-f, showing how to efficiently scan sets of trails. We provide experimental results and discuss more in detail the limitations of previous work that motivated our research. Finally, in Chapter 7 we give conclusions.

Part II is dedicated to differential fault analysis against AES. We start by motivating our research in Chapter 8. In Chapter 9 we first provide more details on fault attacks, then we describe the AES algorithm and discuss state of the art on DFA against it. In Chapter 10 we present four attacks based on faults targeting the operation sequence. In Chapter 11 we present our analysis on the role of the fault model from an information theoretical point of view. Then, in Chapter 12, we present J-DFA, i.e. a new approach to analyze collected faulted ciphertexts using a clustering technique. Finally, in Chapter 13 we give some closing remarks.

Part III is dedicated to partial key exposure attacks against RSA. In Chapter 14 we introduce our research. In Chapter 15 we describe the RSA algorithm and the methodology adopted in partial key exposure attacks. Then, in Chapter 16 we present our attacks against RSA and CRT-RSA. Finally, in Chapter 17 we provide conclusions.

In Chapter 18 we give closing remarks and provide interesting points for future work.

Part I

Trail search in Keccak-f

2 Introduction

Differential cryptanalysis (DC) is a discipline that attempts to find and exploit predictable difference propagation patterns to break iterative cryptographic primitives [BS90]. The basic version makes use of *differential trails* (also called *differential characteristics* or *differential paths*) that consist of a sequence of differences through the rounds of the primitive. Given such a trail, one can estimate its differential probability (DP), namely, the fraction of all possible input pairs with the initial trail difference that also exhibit all intermediate and final difference when going through the rounds.

A way to characterize the power of trails is by their weight w. For the round function of KECCAK-f (but also of, e.g., the AES), the weight equals the number of binary equations that a pair of inputs must satisfy to follow the specified differences [BDPV11b, DR02]. Assuming that these conditions are independent, the weight of the trail relates to its DP as DP $= 2^{-w}$ and exploiting such a trail becomes harder as the weight increases. The assumption of independence does not always apply. For instance, a trail with weight larger than the number of input bits implies redundant or contradictory conditions on pairs, for which satisfying pairs may or may not exist.

Finding lower bounds on the weight of differential trails is an interesting goal. In certain scenarios, low-weight (i.e., high-probability) differential trails give rise to attacks, so proving a lower bound ensures resistance against certain classes of attacks. In general, such bounds give an indication of the cryptographic strength of the primitive and, if they are not tight, narrowing the gap between the lower bounds and the weight of known trails contributes to its understanding.

The KECCAK-f permutation is designed according to the wide trail strategy [Dae95]. For some other primitives that follow this strategy, there exist compact and powerful proofs for the lower bounds of differential trails over multiple rounds. The best known example is the AES with its solid bound of weight 150 over 4 rounds [DR02]. It appears that compact proofs only exist for primitives with strong alignment, like the AES, which is defined in terms of byte operations rather than bit operations [BDPV11a]. With its 3-dimensional and bit-oriented structure, KECCAK-f has weak alignment. This property helps prevent attacks such as truncated differentials, but at the same time it appears to preclude compact proofs. For primitives with weak alignment the best results can be obtained by computer-assisted proofs, as for DES [Mat94], where a program scans the space of all possible n-round trails with weight below some target weight T. If such trails are found, the one with the lowest weight defines a tight bound, otherwise T provides a loose bound.

Scanning the space of trails can be done with dedicated programs or by using standard tools. In the latter case, standard tools include for instance (mixed) integer linear programming ((M)ILP) or SAT solvers. MILP has been used by Sun, Hu, Wang, Wang, Qiao, Ma, Shi and Song to prove a lower bound of weight 19 in the first 3 rounds of Serpent, and to find the best 4-round trail in PRESENT that turns out to have weight 12 [SHW$^+$14]. The highest weight per round using standard tools we know of is due to Mouha and Preneel, who used a SAT solver to scan all 3-round characteristics up to weight 26 in the ARX primitive Salsa20 [MP13]. Other examples using (M)ILP do not achieve numerically better results [BFL10, MWGP11].

Better results can be obtained by writing dedicated programs that leverage the structural properties of the primitive. The ability to reach a larger search space depends on how good they use the structure of the primitive. Of course, the better such a program exploits the particular structure of a cipher, the more specific it becomes to that cipher. For instance, Daemen, Peeters, Van Assche and Rijmen used a dedicated program to scan the full space of trails with less than 24 active S-boxes in Noekeon and showed that the best 4-round trails have weight 48 [DPVR00]. For Keccak-f, Daemen and Van Assche used dedicated programs for lower bounding differential trails in Keccak-f[1600], by generating all 3-round differential trails with weight up to 36, and showed that the best 3-round trails have weight 32 [DV12] and that a 6-round trail has weight at least 74.

Results obtained with standard tools cited above never reached a weight per round above 9. The dedicated efforts for Noekeon in [DPVR00] and Keccak-f[1600] in [DV12] both reached a weight per round of 12.

One of the goals of this work is to extend the space of trails in Keccak-f that can be scanned with given computation resources and thus reach a weight per round beyond 12. This allows us to improve the bound on 6-round trails. Indeed, in [DV12] Daemen and Van Assche showed that the space of 6-round trails in Keccak-f[1600] with weight up to 73 is empty, that implies that a 6-round trail has weight at least 74, but we do not know how large the gap between this bound and the actual minimum weight is.

Our first step was thus to apply the techniques of [DV12] to generate all 3-round differential trails with weight up to a target weight bigger than 36. But we encountered several bottlenecks, which made the search prohibitively expensive already for a target weight of 38. This has motivated us to investigate alternative approaches to generate trails more efficiently in order to increase the target weight. The new methods we present in this thesis allow us to scan the space of 3-round differential trails up to weight 45 and thus reach a weight per round of 15.

As said, the most direct consequence of scanning a bigger space of trails is an extension and an improvement over known bounds. But this is not the only goal of this work. Indeed, scanning a bigger space of trails for different variants of Keccak-f allows to compare results and trends among widths and it gives insight in how weight distributions of low-weight trails scale with the permutation width of Keccak-f. In addition, among the new trails generated there might be some that once extended give rise to trails that are lighter than the lightest known.

Thanks to their abstraction level, some of the techniques we developed are actually more widely applicable than to Keccak-f. They are indeed of independent interest and could be applied to those primitives with bit-oriented mixing layer and relatively lightweight non-linear layer. So, we have formalized them in a generic way.

This part is structured as follows. In Chapter 3, we recall some basic concepts about differential trails. In Chapter 4, we present the generic approach to scan the space of trails up to a given weight. Then in Chapter 5 we introduce Keccak-f and the propagation properties of its step mappings. In Chapter 6 we instantiate the generic approach for Keccak-f and provide experimental results. Finally, in Chapter 7 we give conclusions on this part.

3

Preliminaries on differential cryptanalysis

Differential cryptanalysis is one of the most powerful techniques used to analyze symmetric-key ciphers and hash functions. In the case of block ciphers, it is used to analyze how input differences in the plaintext lead to output differences in the ciphertext. These differences can then be used to assign probabilities to the possible key candidates and to identify the most probable key. Or they can be used to build a distinguisher. For stream ciphers, differential cryptanalysis can be used to show how a key difference or an IV difference can be used to predict the stream difference. For hash functions, it can be used to build a collision attack.

Therefore, security against differential cryptanalysis is a major security metrics for the design of such cryptographic primitives.

Differential cryptanalysis was first published by Biham and Shamir for the block cipher DES in 1990 [BS90, BS91]. Their results led to differential attacks on the full 16-Round of DES [BS92] and was then applied to many other block ciphers, stream ciphers and hash functions [dBB93, Dob98, CJ98, WLF$^+$05, WY05].

Differential cryptanalysis is a chosen plaintext attack, meaning that the attacker can select inputs and analyze outputs. The main idea is to consider the propagation of differences between input pairs through a multiple number of rounds of the primitive. The sequence of differences through the rounds of the primitive is called *differential trail.* The *differential probability* (DP) of a trail is the fraction of all possible input pairs with the initial trail difference that also exhibit all intermediate and final differences when going through the rounds of the primitive. An attacker thus tries to construct high-probability trails, so that it is expected to be easier to find one or more pairs which follow it. It follows that high-probability trails are a potential weakness for cryptographic primitives.

Differential attacks are usually very specialized by exploiting the internal structure of the primitive, starting from the kind of difference that best fits the design under attack. There are indeed many types of differences that can be considered. Some examples are XOR difference [BS90], modular difference [Dob98], signed bit difference [WY05, WYY05] and truncated difference [Knu94].

A significant effort has been dedicated to develop automatic tools to find differential trails and pairs following trails more efficiently. Some examples are automated differential path search [CJ98,CR06,FLN07,MNS11,MNS12,SO06,SLdW12] and advanced message modification [WY05, WYY05].

In this chapter provide some basic definitions and properties of differentials. Then we introduce some basic concepts related to differential trails, that are important for our analysis in the following chapters.

3.1 Differentials

In this thesis we focus on differences over the finite field GF(2).

For a transformation $f\colon \mathbb{Z}_2^n \to \mathbb{Z}_2^m$ an *input difference* is an element of $a \in \mathrm{GF}(2)^n$ representing the difference between two elements $a' \oplus a''$ and an *output difference* is an element $b \in \mathrm{GF}(2)^m$ representing the difference $f(a') \oplus f(a'')$.

For a given n, there are $2^n - 1$ possible non-zero input differences. For each non-zero input difference, there are 2^{n-1} pairs with that input difference.

We say that a bit in a difference pattern is active if it has value 1 and passive otherwise.

We consider the propagation of differences through a cryptographic primitive and we get the following basic definition:

Definition 3.1. *A differential over the transformation f is a pair of input difference a and output difference b and is denoted as (a, b).*

Among all pairs with input difference a, only a fraction of them will also exhibit the output difference b and we get the following definitions.

Definition 3.2. *The cardinality of a differential (a, b) is the number of pairs $\{a', a''\}$ with input difference a that have output difference b:*

$$N(a,b) = \#\left\{\{a', a''\} \mid a' \oplus a'' = a \textit{ and } f(a') \oplus f(a'') = b\right\}. \tag{3.1}$$

Definition 3.3. *The differential probability of a differential (a, b) is defined as the ratio between the cardinality of the differential and the number of pairs with input difference a:*

$$\mathrm{DP}(a,b) = \frac{\#\left\{\{a', a''\} \mid a' \oplus a'' = a \textit{ and } f(a') \oplus f(a'') = b\right\}}{\#\left\{\{a', a''\} \mid a' \oplus a'' = a\right\}} = \frac{N(a,b)}{2^{n-1}} \tag{3.2}$$

Note that the differential probability can be either zero or a multiple of 2^{1-n}. If it is zero, then the differential is an *impossible* differential and the two difference patterns composing the differential are *incompatible*. Otherwise, the differential is a *possible* differential and two differences composing it are *compatible*.

Example 3.1. Some trivial examples of differentials are the following:

- The differential $(0, 0)$ is called the *trivial differential* and it has cardinality equal to 2^{n-1} and differential probability equal to 1.
- A differential $(0, a)$ with $a \neq 0$ is an impossible differential, it has cardinality and probability equal to 0.
- For a linear function f, $f(a' \oplus a'') = f(a') \oplus f(a'')$. So, a differential (a, b) has cardinality 2^{n-1} and probability 1 if $b = f(a)$, otherwise it has cardinality and probability 0.

Definition 3.4. *The weight of a (possible) differential (a, b) is defined as the negative logarithm of its probability:*

$$\mathrm{w}(a,b) = -\log_2 \mathrm{DP}(a,b) \tag{3.3}$$

In round functions with degree 2 or in the round function of AES, the weight represents the number of binary conditions that the bits of input a' must satisfy to follow the specified differences, or equivalently the loss in entropy of a' due to the restriction imposed by (a, b). This becomes clear if we expand (3.3), obtaining

$$\mathrm{w}(a,b) = n - 1 - \log_2 \left|\left\{\{a', a''\} \mid a' \oplus a'' = a \text{ and } f(a') \oplus f(a'') = b\right\}\right| \tag{3.4}$$

The weight of a differential relates to its differential probability as $\mathrm{DP}(a,b) = 2^{-w(a,b)}$ and to its cardinality as $N(a,b) = 2^{n-1-w(a,b)}$.

3.2 Differential trails

In general, we can only determine the cardinality or the differential probability only for functions $f\colon \mathbb{Z}_2^n \to \mathbb{Z}_2^m$ with n and m small. For larger functions, we can split the function into smaller sub-functions and estimate the differential probability for the whole function on the differential probabilities of these sub-functions.

In this thesis we focus on iterated mappings, namely maps that consists in the repetition of a round function, as used in all block ciphers and permutations.

Let $f = R_r \circ R_{r-1} \circ \cdots \circ R_2 \circ R_1$ be an iterated map, where R_i denotes the round function at round i.

A differential trail through the function f is a sequence of $r+1$ differences specified by the input difference, the output difference and the intermediate differences, i.e. the differences after each round:

$$Q = a_1 \xrightarrow{R_1} a_2 \xrightarrow{R_2} \quad \dots \quad \xrightarrow{R_{r-1}} a_r \xrightarrow{R_r} a_{r+1} \tag{3.5}$$

and we denote it as $Q = (a_1, a_2, \dots, a_r, a_{r+1})$. We call a differential (a_i, a_{i+1}) a *round differential.*

Differential trails over a smaller number of rounds $n \leq r$ are often considered. We refer to such trails as n-round trails. The round functions composing a n-round trail can be any sequence $R_i, \dots, R_{i+n}$ within the map f. However, the round functions within an iterated map usually differ only by the addition of a round constant or a round key. This operation has no influence in the propagation of differences and thus it can be ignored in the analysis, making the round functions identical. Therefore, we will denote the round function by R and n-round trails as $Q = (a_1, a_2, \dots, a_n, a_{n+1})$.

The definition of cardinality and differential probability for a differential trail through an iterated map is similar the definition of cardinality and differential probability of a differential.

Definition 3.5. *For a differential trail $Q = (a_1, \dots, a_{n+1})$, the cardinality of Q is defined as:*

$$\begin{aligned} N(Q) = \#\{\{a', a''\} \mid\ & a' \oplus a'' = a \text{ and} \\ & R(a') \oplus R(a'') = a_2 \\ & R(R(a')) \oplus R(R(a'')) = a_3 \\ & \vdots \\ & R(\cdots(R(a'))) \oplus R(\cdots(R(a''))) = a_{n+1}\}. \end{aligned} \tag{3.6}$$

Namely, the cardinality is the number of pairs that exhibit all the differences specified by the differential trail, through the round of the primitive.

Definition 3.6. *The differential probability of a differential trail $Q = (a_1, \dots, a_{n+1})$ is defined as the ratio between its cardinality and the number of input pairs*

$$\mathrm{DP}(Q) = \frac{N(Q)}{2^{n-1}} \tag{3.7}$$

Definition 3.7. *The weight of a trail is the sum of the weight of the round differentials that compose the trail:*

$$\mathrm{w}(Q) = \sum_{i=1}^{n} \mathrm{w}(a_i, a_{i+1}). \tag{3.8}$$

In general, is not feasible to calculate the exact value of the differential probability, while it is easy to compute the weight, which represents the number of binary conditions that a pair must satisfy to follow all the differences specified by the differential trail, through the round of the primitive. Under the assumption that these restrictions imposed by the different round differentials are independent, the weight of the trail relates to its DP as

$$\mathrm{DP}(Q) \sim 2^{-w(Q)}. \tag{3.9}$$

The bigger the weight of a trail is, the smaller the differential probability is and thus the cardinality. The existence of low-weight trails (i.e. with high differential probability) over all but a few rounds are a potential weakness and would imply a low resistance with respect to DC.

It is thus interesting to understand the distributions of differential trail weights for estimating the safety margin of a primitive.

3.3 Trail extension

In general, trails can be generated starting from short trails (i.e. trails over a small number of rounds) and then extending them to more rounds.

Given a n-round trail $Q = (a_1, a_2, \ldots, a_{n+1})$, it is possible to extend it to $n + 1$ rounds. The extension can be done in both the forward and backward direction.

Forward extension consists in building all differences a_{n+2} that are compatible with a_{n+1} through R. This operation will thus produce a set of $n + 1$-round trails with Q as leading part which differ from each other for the pattern a_{n+2}. The weight of such trails is $\mathrm{w}(Q) + \mathrm{w}(a_{n+1}, a_{n+2})$.

Backward extension consists in building all differences a_0 compatible with a_1 through R^{-1}. This operation will give a set of trails with Q as trailing part and the weight of such trails is given by $\mathrm{w}(a_0, a_1) + \mathrm{w}(Q)$.

Of course, the operation of extension can be iterated and backward and forward extension can be combined. Starting from a n-round trail Q, we can generate all $n + m$-round trails, that have Q as subtrail, by performing all the following steps:

- extend forward by m rounds,
- extend backward by 1 round and forward by $m - 1$ rounds,
- ...
- extend backward by $m - 1$ rounds and forward by 1 rounds,
- extend backward by m rounds.

3.4 Trail cores

In this section we introduce so called *trail cores*. Namely, sets of trails sharing all round differentials except the first and the last. These sets are of particular interest, because for each set we can compute the minimum weight among the trails it contains without the need of generating all of them. Indeed, it is sufficient to generate a representative trail for each set and derive the minimum weight from such representative.

Thanks to such partition of trails, we can limit our search to the generation of representative trails, one for each set, instead of generating all trails. The rationale behind this approach is the following.

When extending a trail in the forward direction, all differences compatible with the last one in the trail are built. Among these compatible differences, there will be one defining a minimum weight.

Definition 3.8. *Given a difference pattern a_{n+1} at the input of R, we define the* minimum direct weight *of a_{n+1} as*

$$\mathrm{w}^{\mathrm{dir}}(a_{n+1}) = \min_{a_{n+2}} \mathrm{w}(a_{n+1}, a_{n+2}) \tag{3.10}$$

where a_{n+2} is compatible with a_{n+1} through R.

Therefore the weight of any $n+1$-round trail obtained by forward extension of an n-round trail Q can be lower bounded by $\mathrm{w}(Q) + \mathrm{w}^{\mathrm{dir}}(a_{n+1})$.

Similarly, when extending a trail in the backward direction, all differences compatible with the first one in the trail are built. Among these one will define a minimum weight.

Definition 3.9. *Given a difference pattern a_1 at the output of R, we define the* minimum reverse weight *of a_1 as*

$$\mathrm{w}^{\mathrm{rev}}(a_1) = \min_{a_0} \mathrm{w}(a_0, a_1) \tag{3.11}$$

where a_0 is compatible with a_1 through R^{-1}.

Therefore the weight of any $n+1$-round trail obtained by backward extension of an n-round trail Q can be lower bounded by $\mathrm{w}^{\mathrm{rev}}(a_1) + \mathrm{w}(Q)$.

It follows that a n-round trail Q defines a set of $n+2$-round trails as in the following definition.

Definition 3.10. *Given a n-round trail $Q = (a_1, a_2, \ldots, a_{n+1})$, the* trail core *defined by Q is the set of $n+2$-round trails with Q as central part, with a_0 compatible with a_1 through R^{-1}, with a_{n+2} compatible with a_{n+1} through R and with weight at least $\mathrm{w}^{\mathrm{rev}}(a_1) + \mathrm{w}(Q) + \mathrm{w}^{\mathrm{dir}}(a_{n+1})$. We denote it by $\langle Q \rangle$ or $\langle a_1, a_2, \ldots, a_{n+1} \rangle$.*

We define the weight of a trail core as follows.

Definition 3.11. *Given a n-round trail $Q = (a_1, a_2, \ldots, a_{n+1})$, the weight of the trail core $\langle Q \rangle$ is the quantity $\mathrm{w}^{\mathrm{rev}}(a_1) + \mathrm{w}(Q) + \mathrm{w}^{\mathrm{dir}}(a_{n+1})$.*

The minimum direct and reverse weight can usually be easily computed either analytically or by building the differential distribution table of the non-linear layer, which has usually low degree.

Since the aim of our search is to find the minimum weight of trails, we can restrict our search to trail cores and find the minimum weight for them. Indeed, the weight of a trail core corresponds to the minimum weight among the trails contained in the core. Namely, trails in the core will have weight equal or bigger than the weight of the trail core. Building all trails and computing the actual weight of such trails is not needed to reach our goal. This allows to significantly reduce the effort of our search, since it is sufficient to generate $(n-2)$-round trails to find the minimum weight over n rounds. This means covering a space that is much smaller.

4

A generic approach for trail generation

In this chapter we introduce our generic approach for the generation of all n-round trail cores up to a given target weight.

In Section 4.1 we show how starting from 2-round trail cores allows to achieve higher values of the target weight than the brute-force approach, which starts from round differentials (assuming the presence of a mixing layer between two non-linear layers).

In Section 4.2 we present our method to generate 2-round trail cores by constructing difference patterns. Indeed, following the definition of trail core, a 2-round trail core is fully determined by a difference pattern. We thus generate such difference patterns up to a given cost, where the cost is the weight of the 2-round trail core. Notice that this method can be extended to other patterns like linear masks and can use functions of the weight. We propose a representation of difference patterns by elementary components called *units* so that patterns are incrementally generated by adding units. By imposing an order relation on units, a difference pattern can be coded as an ordered list of units. This arranges all possible difference patterns in a tree, where the parent of a node is the pattern with the last unit of its list removed. We select cost bounding functions that allow to efficiently bound the cost of any pattern and that of its descendants. In this way, when the cost of a pattern is already above the budget, we can avoid generating other patterns by adding new units to it. The generation of trails up to a given cost can thus be performed as a tree traversal where we can efficiently prune any node whose bound on the cost is higher than the target weight.

Finally, in Section 4.3 we show how the tree traversal can be made more efficient when symmetry properties define a partition of patterns in equivalence classes. We can indeed reduce our effort, by generating only those patterns that are the representative of their equivalence class. This again allows to prune entire subtrees as soon as we find a node that is not a representative of an equivalence class.

4.1 Generating trails

To find lower bounds for n-round trails weight, we can scan the space of all possible n-round trails with weight up to some target weight T_n. If actual trails are found, then the one with the smallest weight establishes the minimum weight for all n-round trails. Otherwise, we can say that an n-round trail has weight at least $T_n + 1$ and this bound is not necessarily tight.

4.1.1 The first-order approach: trails from extending round differentials

The brute-force method to generate trails consists in starting from round differentials and extending them.

Lemma 4.1. *An n-round trail with weight W has at least one round differential with weight below or equal to $T_1 = \left\lfloor \frac{W}{n} \right\rfloor$.*

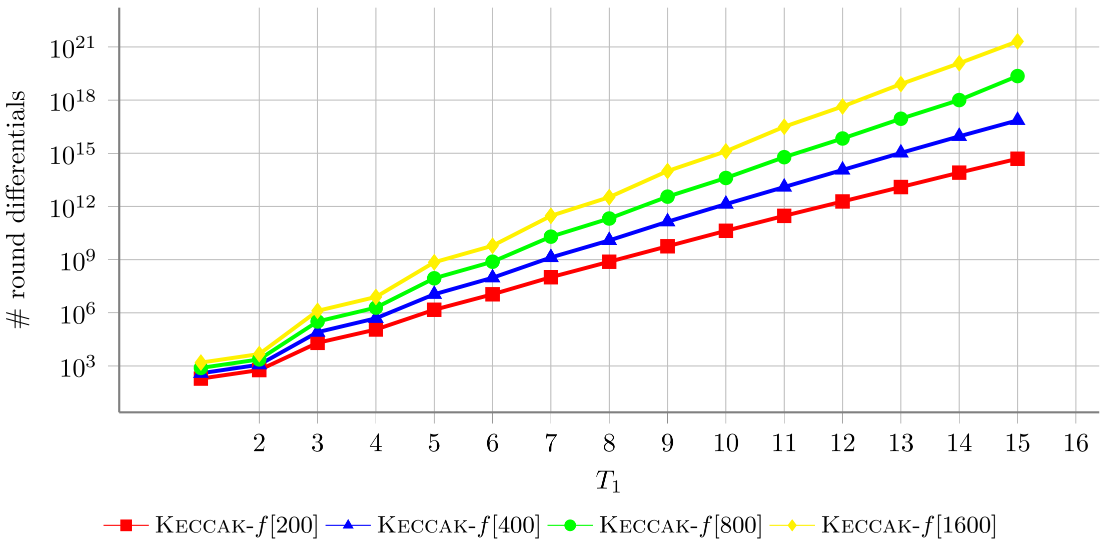


Fig. 4.1: *Number of round differentials in* KECCAK-f *with weight up to* T_1 *per round.*

Proof. The proof follows proof of Lemma 1 in [BDPV11b].
The weight of an n-round trail is the sum of the weight of its n-round differentials, thus $W = \sum_i w_i$, where w_i denotes the weight of the i-th differential in the sequence. Let $w_{\min} = \min_i(w_i)$. We want to prove that $w_{\min} \leq T_1$.
It holds that $W = \sum_i w_i \geq n \cdot w_{\min}$, which implies that $w_{\min} \leq \frac{W}{n}$. Since the weight is an integer quantity it holds that $w_{\min} \leq \left\lfloor \frac{W}{n} \right\rfloor$.

□

The following corollary follows immediately from Lemma 4.1 and deals with the generation of n-round trails from extending round differentials.

Corollary 4.1. *All n-round trails of weight smaller or equal to T_n, can be generated by extension of round differentials with weight smaller or equal to $T_1 = \left\lfloor \frac{T_n}{n} \right\rfloor$.*

So for a given target weight T_n, we can find all n-round trails with weight up to T_n by using a first-order approach that consists of the following two phases:

1. Generate all round differentials of weight below or equal to $\left\lfloor \frac{T_n}{n} \right\rfloor$.
2. Build for each of these round differentials all n-round trails that contain them by:
   - extending forward by $n-1$ rounds,
   - extending backward by 1 round and forward by $n-2$ rounds,
   - ...
   - extending backward by $n-2$ rounds and forward by 1 rounds,
   - extending backward by $n-1$ rounds.

The limiting factor of this method is the number of round differentials with weight up to T_1 and the effort per such differentials to extend them into n-round trails. In fact, the number of round differentials increases very steeply with the weight and the value of T_n we can achieve is limited by the mere quantity of round differentials with weight below T_1.

As an example, we list in Figure 4.1 the number of round differentials of KECCAK-f for several widths. As can be seen, the numbers increase exponentially with the weight and this increase is stronger as the width grows. When using standard tools, a weight per round of 7 for width $b = 1600$ already seems quite ambitious as it implies at least hundred million trail extensions starting from round differentials. This is in line with the results obtained with standard tools cited in previous chapters, that up to now never reached a weight per round above 9.

4.1.2 The second-order method: trails from extending two-round trails

In modern ciphers the round function has a relatively powerful mixing layer and the number of two-round trails with weight below $2T_1$ is much smaller than the number of round differentials with weight below T_1.

For example, AES has $16 \times 255 \times 127 + \binom{16}{2} \times 255^2 \times 127^2 > 10^{11}$ round differentials with weight below 15 while the mixing layer ensures that there are no 2-round trails with weight below 30. In fact, the mixing layer and the byte-aligned layout of AES allows demonstrating that there are no four-round trails with weight below 150 without the need for scanning the space. The price paid for this is a relatively costly mixing layer and the vulnerability of the cipher with respect to attacks that exploit the strong alignment. However, most weakly aligned ciphers and permutations have a mixing layer that has a similar effect.
Instead of building trails from round differentials, we can build them starting from two-round trails.

Lemma 4.2. *An n-round trail of weight W always contains a 2-round trail of weight below or equal to T_2, with $T_2 = \left\lfloor \frac{2W}{n} \right\rfloor$ if n is even, or $T_2 = \left\lfloor \frac{2(W-1)}{n-1} \right\rfloor$ if n is odd.*

Proof. The proof follows proof of Lemma 2 in [BDPV11b].
The weight of an n-round trail is the sum of the weight of its n-round differentials, thus $W = \sum_i w_i$, where w_i denotes the weight of the i-th differential in the sequence. Let $\delta_{\min} = \min_i(w_i + w_{i+1})$. We want to prove that $\delta_{\min} \leq T_2$.
If n is even, then $W = \sum_{i=1}^{n} w_i = \sum_{j=1}^{n/2}(w_{2j-1} + w_2 j) \geq \frac{n}{2} \cdot \delta_{\min}$, which implies that $\delta_{\min} \leq \frac{2W}{n}$. Since the weight is an integer quantity it holds that $\delta_{\min} \leq \left\lfloor \frac{2W}{n} \right\rfloor$.
If n is odd, we can always assume $w_n \geq 1$ and consider $i = 1, \ldots, n-1$ and $T_{n-1} = T_n - 1$. □

The following corollary follows.

Corollary 4.2. *All n-round trails of weight smaller or equal to T_n, can be generated by extension of 2-round trails with weight smaller or equal to T_2, with $T_2 = \left\lfloor \frac{2W}{n} \right\rfloor$ if n is even, or $T_2 = \left\lfloor \frac{2(W-1)}{n-1} \right\rfloor$ if n is odd.*

For a given target weight T_n, we can thus find all n-round trails with weight up to T_n by using a second-order approach that consists of the following two phases:

1. Generate all 2-round trails of weight below or equal to T_2.
2. Build for each of these 2-round trails all n-round trails that contain them by:
   - extending forward by $n - 2$ rounds,
   - extending backward by 1 round and forward by $n - 3$ rounds,
   - ...
   - extending backward by $n - 3$ rounds and forward by 1 rounds,
   - extending backward by $n - 2$ rounds.

In Figure 4.2 we depict the number of 2-round trails for different variants of Keccak-f. When we compare these numbers with those in Figure 4.1 considering the weight per round, we can immediately see that the number of round differentials with weight T_1 is much bigger than the number of 2-round trails with weight $2T_1$. For instance, for all widths there are several round differentials with weight 2 or 3, while there are no 2-round trails with weight below 8. Again, for Keccak-f[1600] there are around 2^{58} round differentials with weight 13 while there are only around 2^{32} 2-round trails with weight 26. Moreover, the extension of such 2-round trails allows for instance to generate all 4-round trail up to weight 52. The same number of round differentials is reached at weight 7, that allows to generate all 4-round trails up to weight only 28. Of course, the actual effort for extension is not directly comparable because the cost of extending a trail depends on the weight of the differential where the extension starts from. However, compared to the first-order approach, this second-order approach allows to target much higher values of T_n for the same number of rounds n.

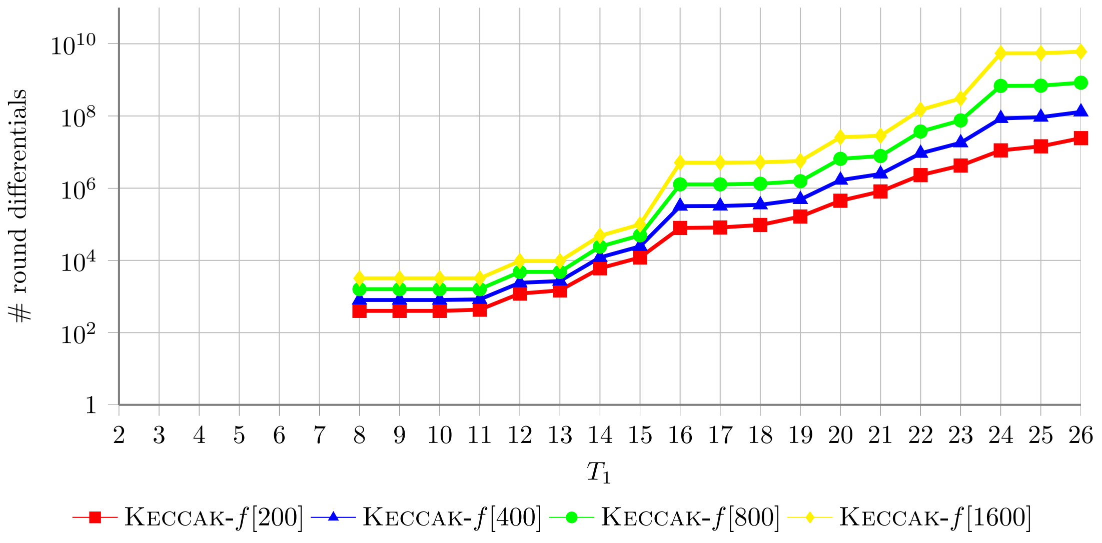


Fig. 4.2: *Number of 2-round trail cores in* Keccak-f *with weight up to* T_1 *per round.*

For this reason, this approach was used in [DV12] and we will adopt it, too. However, it introduces the problem of generating 2-round trails with weight below some limit T_2.

4.1.3 Lower bounding the weight starting from two-round trail cores

As explained in Section 3.4, to find the minimum weight of n-round trails, we don't need to build all trails but it is sufficient to restrict our search to trail cores. Indeed, the weight of a trail core lower bounds the weight of all the trails contained in the core and it can be easily computed once the (n-2)-round trail defining the core is known. It follows that we can reach our target, i.e. find the minimum weight of trails, by covering a space that is much smaller than the space of all trails.

We thus generate all n-round trail cores up to a given target weight T_n starting from 2-round trail cores $\langle a \rangle$ whose weight $\mathrm{w}^{\mathrm{rev}}(a) + \mathrm{w}^{\mathrm{dir}}(a)$ is below or equal to T_2, with T_2 defined as above. Therefore, the generation of 2-round trail cores comes down to the generation of difference patterns a from which we can compute the minimum reverse and direct weights.

In the following, we introduce our approach to efficiently generate difference patterns keeping the weight of the corresponding 2-round trail core below the budget T_2.

4.2 Tree traversal approach

As explained above, the weight of a 2-round trail core is fully determined by a single difference pattern.

Definition 4.1. *Given a difference pattern a, the* cost *of a is defined as the weight of the 2-round trail core $\langle a \rangle$ and is denoted by $\gamma(a)$, i.e. $\gamma(a) = \mathrm{w}^{\mathrm{rev}}(a) + \mathrm{w}^{\mathrm{dir}}(a)$.*

It now suffices to generate all difference patterns a with cost below T_2, i.e. with $\gamma(a) \leq T_2$.

To do this in an efficient way, we introduce a new approach where we arrange all possible patterns a in a tree, where the root of the tree is the all-zero (or empty) pattern. In its simplest and most intuitive form, the idea is to generate descendants by adding groups of active bits in a way that depends on the properties of the mixing layer and that makes the cost function monotonic in the addition of bits. Monotonicity is actually not required, what we need is a lower bound on the cost of a pattern and of all its descendants. Finding all patterns with cost below T_2 now simply corresponds to traverse the tree and prune any node (and all its descendants) whose lower bound on the cost is higher than T_2.

4.2.1 Units and unit-lists

We represent difference patterns as sets of elementary components that we call *units*. These units are groups of bits whose actual definition depends on the properties of the mixing layer of the round function. We denote the set of units by $\mathcal{U}$. Each pattern is thus represented by a subset of $\mathcal{U}$. Such subset of units defines the value of some bits of the pattern, while the other bits are set equal to zero.

Example 4.1 (Units as active bit positions). Each bit in a difference pattern can be either active or passive. We can list the positions of all active bits and the obtained list uniquely determines the pattern. Indeed, we can rebuild the pattern by setting the bits in the positions that appear in the list to one and all the other bits to zero. Suppose that the state of the primitive under analysis is a bi-dimensional state with n rows and m columns. We can represent an active bit by its position within the state, which is a pair (i, j) where i denotes the row and j denotes the column. Therefore, the set of units is $\mathcal{U} = \{0, \ldots, n-1\} \times \{0, \ldots, m-1\}$ and any difference pattern is represented by a subset of $\mathcal{U}$. △

We generate difference patterns by incrementally adding units. When building patterns, we do not want to generate them more than once. For example, we want to avoid generating a pattern with units u_1 and u_2 by first adding u_1 then u_2 and then again by first adding u_2 then u_1. A straightforward way to implement it is to define a total order relation [1] on the set of units. This arranges all patterns in a tree, where the root is the all-zero pattern and where the children of a node are obtained by adding a new unit to it.

Definition 4.2. *We call* unit-order *a total order relation on the set of units* $\mathcal{U}$ *and we denote it by* $\prec_u$.

Example 4.2. Let's consider again the case of Example 4.1 where units are single active bit positions. We can define the order relation $\prec_u$ on $\mathcal{U}$ as the lexicographic order $[i, j]$, i.e.:

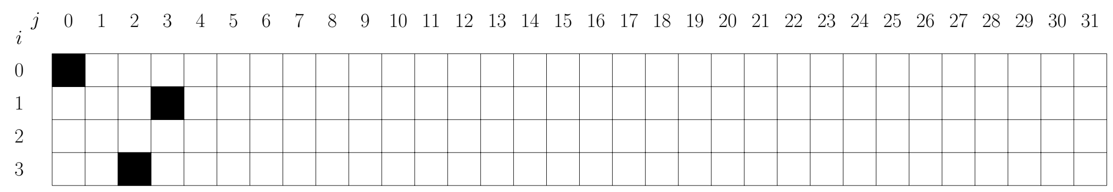
$$(i,j) \prec_u (i',j') \Leftrightarrow \begin{cases} i < i' \text{ or} \\ i = i' \text{ and } j < j'. \end{cases} \tag{4.1}$$

△

Equipped with such a total ordering, a difference pattern can thus be represented as an ordered list of units $[u_i]_{i=1,\ldots,n}$ that satisfies $u_1 \prec u_2 \prec \cdots \prec u_n$. Indeed, the set of units representing the pattern is a subset of the set $\mathcal{U}$ and therefore it is ordered.

Definition 4.3. *Given a difference pattern a, we call* unit-list *the list of units representing the pattern.*

Example 4.3. Consider the following difference pattern (as in Serpent):



[1] A total order on a set X is a binary relation, which is antisymmetric, transitive and total.

We define units as active bit positions and the order relation on units as the one defined in Equation 4.1. Then, the unit-list of such pattern is $[(0,0),(1,3),(3,2)]$. △

We incrementally generate difference patterns by incrementally adding units at the end of their unit-lists. This entirely defines the tree structure as follows.

Definition 4.4 (Unit-tree). *We call* unit-tree *the rooted tree whose nodes are difference patterns represented by unit-lists. The root of the tree is the all-zero pattern represented by the empty list. The children of a node with unit-list $[u_1,\dots,u_n]$ are those patterns with unit lists $[u_1,\dots,u_n,u_{n+1}]$ for all u_{n+1} such that $u_n \prec u_{n+1}$. Equivalently, the parent of a pattern is defined as the pattern with the last element of its unit-list removed. Two nodes are siblings if their unit-lists with the last unit removed are identical. The unit-list of a node defines the path from the node to the root and the depth of a node is thus the size of its unit-list.*

The order on the units naturally establishes an order on unit-lists and thus on difference patterns, as follows.

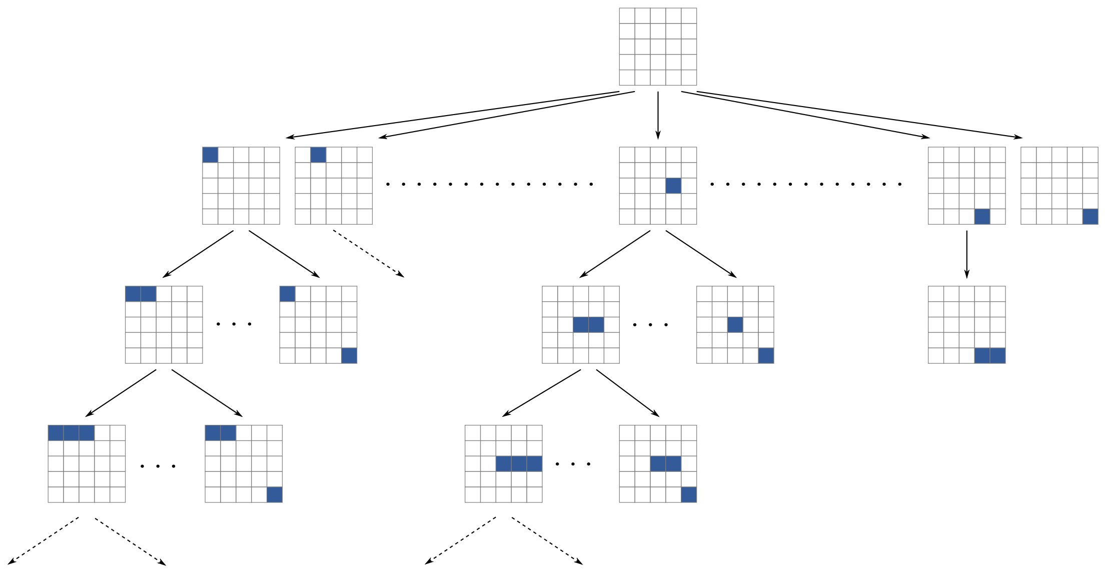
Definition 4.5 (Order relation on patterns). *Given two patterns $a = [u_i]_{i=1,\dots,n}$ and $a' = [u'_j]_{j=1,\dots,m}$, we define the order relation $\prec_p$ using the relation $\prec_u$ on their units as*

$$a \prec_p a' \textit{ iff } \begin{cases} \exists\, k \textit{ such that } u_i = u'_i \ \forall\, i < k \textit{ and } u_k \prec_u u'_k, \textit{ or} \\ n < m \textit{ and } u_i = u'_i \ \forall\, i \leq n. \end{cases} \tag{4.2}$$

In other words, either a' and a are in two different branches of the tree with the same ancestor at height $k-1$ and $u_k \prec u'_k$, or a' is a descendant of a.

Therefore, the unit-tree is a plane tree[2] and we represent it in the plane by ordering children from left to right.

Example 4.4. Consider again the case of Example 4.1 with $n = 5$, $m = 5$ (as in KECCAK-f[25]) and the order defined as in Example 4.2. With the convention that the bit position $(0,0)$ is the top-left bit, the representation of the unit-tree in the plane is the following:



△

[2] A *plane tree* (or *rooted tree*) is a rooted tree in which an ordering is specified for the children of each node.

4.2.2 Lower bound on the cost

To efficiently traverse the tree, we define a function that gives a lower bound on the cost of a node and all its descendants.

Definition 4.6. *Let the set of descendants of a, including a itself, be denoted by $\triangle(a)$. We call* cost bounding function *any map L that associates to a pattern a a value $L(a)$ such that $\gamma(a') \geq L(a)$ for all $a' \in \triangle(a)$. We call the value $L(a)$ the* bound *of a.*

Then, when traversing the tree, any sub-tree rooted in a can be safely pruned when $L(a)$ is bigger than the target weight. In other words, given a target weight T_2 as soon as the search encounters a pattern a such that $L(a) > T_2$, a and all its descendants can be ignored.

4.2.3 Traversing the unit-tree

Traversing the tree now simply consists of recursively generating the children of a given node by iterating over all units that come after the last unit of that node.

Definition 4.7. *Given a unit u, the set of successors of u is denoted by U_u and is defined as:*

$$U_u = \{u' \mid u \prec_u u'\}. \tag{4.3}$$

Of course, the set U_u is again a totally ordered set, being a subset of the set of units $\mathcal{U}$.

In the tree traversal, the move from a node $a = [u_i]_{i=1,\ldots,n}$ to the next one is defined following Algorithm 1. In particular we implement a depth-first search as follows. If possible, we try to add a new unit after u_n, by scanning the ordered set U_{u_n} from its smallest to largest element. This is done by the function `toChild()`, which returns true if a new unit has been added and false otherwise. There are different reasons why this function may return false. The most obvious is that the set U_{u_n} is empty. Another reason is that the addition of any other unit in U_{u_n} implies the cost to grow beyond the target weight. Still another reason is that most of the time we want to avoid that the addition of a new unit cancels bits already added (this might be the case when units are composed by more than a single active bit). If a unit cannot be added, we call it *non-valid*, otherwise we call it and *valid*. Therefore, if false is returned by the function `toChild()`, then we iterate the value of unit u_n. Namely, we look for the first valid element after u_n in $U_{u_{n-1}}$. This is done by the function `toSibling()`, which returns true if a new valid value for u_n is found, false otherwise. If neither this action is possible, we remove u_n from the unit-list (using function `toParent()`) and iterate u_{n-1}. And so on. The search ends when there are no more units left in the unit-list. This check is performed by the function `toParent()`, which returns true if the last unit has been popped from the unit-list, false if the unit-list is empty.

Algorithm 1 Next move in the tree traversal

```
1: procedure NEXT MOVE                          ▷ Next move in the tree traversal
2:     if toChild() then                         ▷ adds a new unit to the list
3:         return true
4:     do
5:         if toSibling() then                   ▷ iterates the highest unit of the list
6:             return true
7:         if !toParent() then                   ▷ pops the highest unit of the list
8:             return false
9:     while (true)
```

This method of generating state patterns as a tree traversal can be applied to all round functions that have a linear and a non-linear layer. However, the definition of units and cost bounding function depends on the details of these layers and thus it is specific of the target cryptographic primitive. We will define them for KECCAK-f in the next chapter.

4.3 Searching by canonicity

The tree traversal approach can be extended when there are symmetry properties that we wish to take into account. Indeed, these properties may define a partition of trails in equivalence classes, where the weight of trails in a class is the same. The size of these classes depends on the symmetry property. We can thus reduce our effort, by generating only those trails that are the representative of their equivalence class. To this end, we define a property that we call *canonicity*: a pattern is canonical if it is the *smallest* representative of its class.

Let $T = \{\tau\}$ be a set of transformations s.t. f is symmetric with respect to each $\tau \in T$. Namely, such that $f \circ \tau = \tau \circ f$.

Definition 4.8. *Given a total order relation $\prec_p$ among patterns, we say that a pattern a is* canonical *if*

$$a = \min_{\prec_p} \{\tau(a) \;:\; \tau \in T\} . \tag{4.4}$$

The following lemma gives an interesting property of canonical patterns when arranged in the tree-structure.

Lemma 4.3. *Using the order $\prec_p$, the parent of a canonical pattern is canonical.*

Proof. Let $a = [u_1, \dots, u_n]$ be a canonical pattern. The parent of a is a with its highest unit u_n removed. Let parent(a) denote the parent of a. We want to show that Equation 4.2 holds also for parent(a).
Assuming that any τ in T preserves the number of units in the list, each $a' \neq a$ in the equivalence class of a, is represented by a unit-list $[u'_j]_{j=1,\dots,n}$ with a bijective correspondence between $[u'{}_j]$ and $[u_i]$. Let Π denote this bijective function.
In general, the order of units in the list of a is not maintained in the list of a'. Therefore, $\Pi(u_n)$ is not necessarily the highest unit of a'.
Say that $\Pi(u_n) = u_h$, so that $\tau(\text{parent}(a)) = (u'_1, \dots, u'_{h-1}, u'_{h+1}, \dots, u'_n)$. Then, there are two possible cases:

- ◇ $h > k$: then the right hand of (4.2) still holds;
- ◇ $h \leq k$: then in (4.2) u'_h is replaced by u'_{h+1}, which is higher than u'_h, and we have $u_i = u'_i \; \forall \, i < h$ and $u_h \prec u'_{h+1}$.

In all cases, the parent of a is canonical. □

A direct consequence of Lemma 4.3 is the following corollary.

Corollary 4.3. *A pattern that is not canonical cannot have canonical descendants.*

So, when we encounter non-canonical patterns during our tree traversal, we can safely exclude them and all their descendants.

Therefore, there are two ways to prune entire subtrees. One when we encounter a node that is not canonical and the other when we encounter a node whose cost bound (and that of its descendants) is above the budget.

5

Keccak-f

Keccak-f is a family of seven permutations, indicated by Keccak-$f[b]$, where $b = 25 \times 2^{\ell}$ and ℓ ranges from 0 to 6. Keccak-$f[b]$ is an iterated permutation consisting of a sequence of rounds and the number of rounds depends on the width of the permutation.

The largest variant, i.e. Keccak-$f[1600]$, is adopted in the SHA-3 family consisting of the four cryptographic hash functions SHA3-224, SHA3-256, SHA3-384 and SHA3-512 and the two extendable-output functions (XOFs) SHAKE128 and SHAKE256.

Round reduced versions of Keccak-f were used to build several permutation-based cryptographic functions. In particular, round-reduced versions of Keccak-$f[200]$ to
Keccak-$f[1600]$ were used to construct Ketje, a set of four authenticated encryption functions with support for message associated data, that was submitted to the CAESAR competition and is currently part of the third round. For more details, see the Ketje documentation [BDP+14].

Round-reduced versions of Keccak-$f[800]$ and Keccak-$f[1600]$ were also used in the definition of Keyak, a permutation-based authenticated encryption scheme with support for associated data and sessions. As Ketje also Keyak was submitted to the CAESAR competition and is currently participating to the third round. For more details, see the Keyak documentation [BDP+15].

A round reduced version of Keccak-$f[1600]$ was also adopted in the definition of Kravatte, a very efficient instance of Farfalle, which is a permutation-based construction for building pseudorandom functions. For a detailed description of Farfalle and Kravatte see [BDP+16].

This chapter is dedicated to the description of the Keccak-f family and the properties we will exploit to generate trails. In Section 5.1 and Section 5.2 we describe the Keccak-f state and round function. Then in Section 5.3 we focus on the difference propagation properties of the step mappings composing the round function, since we will make extensive use of them in defining units and generating trails. Finally, in Section 5.4 we describe trails in Keccak-f in terms of patterns at the input of both the linear and non-linear layers. Following this formalism we describe trail extension and trail cores, too.

5.1 The Keccak-f state

The permutation Keccak-$f[b]$ is described as a sequence of operations on a state a that is a three-dimensional array of elements of GF(2), namely $a[5][5][w]$, with $w = 2^{\ell}$. The expression $a[x][y][z]$ with $x, y \in \mathbb{Z}_5$ and $z \in \mathbb{Z}_w$, denotes the bit in position (x, y, z). Expressions in the x and y coordinates should be taken modulo 5 and expressions in the z coordinate modulo w. We may sometimes omit the $[z]$ index, both the $[y][z]$ indices or all three indices, implying that the statement is valid for all values of the omitted indices.

Parts of the Keccak-f state are illustrated in Figure 5.1. The one-dimensional parts are the following:

◇ A *row* is a set of 5 bits with constant y and z coordinates.

- ◇ A *column* is a set of 5 bits with constant x and z coordinates.
- ◇ A *lane* is a set of w bits with constant x and y coordinates.

The two-dimensional parts are the following:

- ◇ A *sheet* is a set of $5w$ bits with constant x coordinate.
- ◇ A *plane* is a set of $5w$ bits with constant y coordinate.
- ◇ A *slice* is a set of 25 bits with constant z coordinate.

Note that the bit $x = y = 0$ is depicted at the center of the slice.

When working with difference states, a bit is called *active* if its value in the difference pattern is one. A row having at least one active bit is called *active row*. Similarly, for all the other parts of the state pattern.

5.2 The Keccak-f round function

KECCAK-$f[b]$ is an iterated permutation consisting of a sequence of n_r rounds R, indexed with i_r from 0 to $n_\mathrm{r} - 1$. The number of rounds n_r is determined by the width of the permutation, namely,

$$n_\mathrm{r} = 12 + 2\ell.$$

A round of KECCAK-f consists of five invertible steps:

$$\mathrm{R} = \iota \circ \chi \circ \pi \circ \rho \circ \theta$$

with

$$\begin{aligned}
\theta : a[x][y][z] &\leftarrow a[x][y][z] + \sum_{y'=0}^{4} a[x-1][y'][z] + \sum_{y'=0}^{4} a[x+1][y'][z-1],\\
\rho : a[x][y][z] &\leftarrow a[x][y][z - (t+1)(t+2)/2],\\
&\quad \text{with } t \text{ satisfying } 0 \le t < 24 \text{ and } \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix}^t \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} \text{ in } \mathrm{GF}(5)^{2\times 2},\\
&\quad \text{or } t = -1 \text{ if } x = y = 0,\\
\pi : \quad a[x][y] &\leftarrow a[x'][y'], \text{ with } \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} \begin{pmatrix} x' \\ y' \end{pmatrix},\\
\chi : \quad a[x] &\leftarrow a[x] + (a[x+1] + 1)a[x+2],\\
\iota : \quad a &\leftarrow a + \mathrm{RC}[i_\mathrm{r}].
\end{aligned}$$

The additions and multiplications between the terms are in GF(2). Except for the value of the round constants $\mathrm{RC}[i_\mathrm{r}]$, these rounds are identical. The round constants are given by (with the first index denoting the round number)

$$\mathrm{RC}[i_\mathrm{r}][0][0][2^j - 1] = \mathrm{rc}[j + 7i_\mathrm{r}] \text{ for all } 0 \le j \le \ell,$$

and all other values of $\mathrm{RC}[i_\mathrm{r}][x][y][z]$ are zero. The values $\mathrm{rc}[t] \in \mathrm{GF}(2)$ are defined as the output of a binary linear feedback shift register (LFSR):

$$\mathrm{rc}[t] = \left(x^t \bmod x^8 + x^6 + x^5 + x^4 + 1\right) \bmod x \text{ in } \mathrm{GF}(2)[x].$$

5.3 Difference propagation properties of the step mappings

In this section we discuss the step mappings more in details, highlighting the difference propagation properties that we will exploit in our trail search.

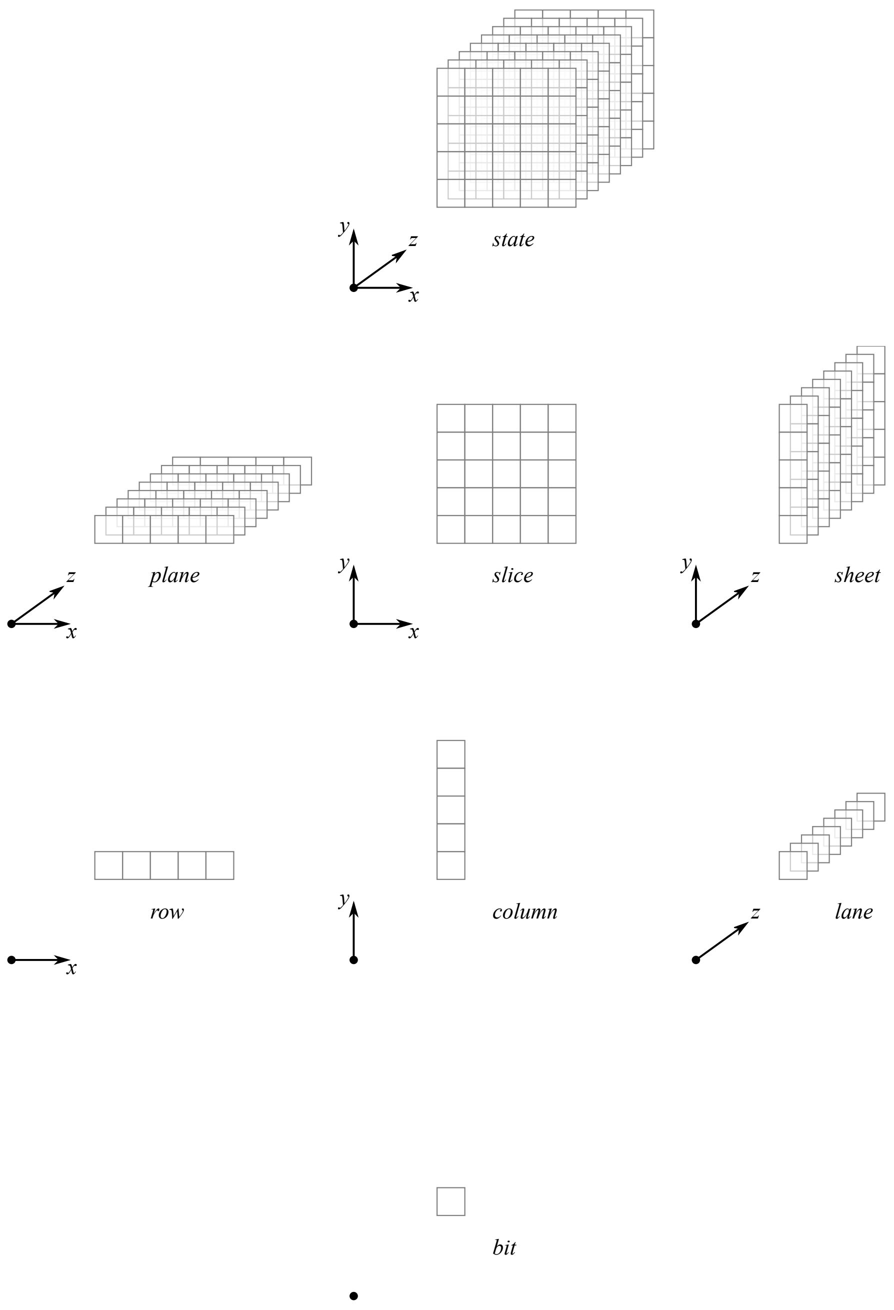


Fig. 5.1: *Naming conventions for parts of the* KECCAK-*f state*

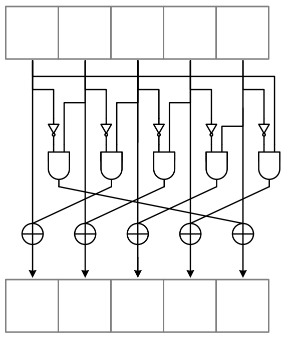


Fig. 5.2: χ *applied to a single row.*

Difference	$\mathrm{w}(\cdot)$	$\mathrm{w}^{\mathrm{rev}}(\cdot)$
00000	0	0
00001	2	2
00011	3	2
00101	3	2
10101	3	3
00111	4	2
01111	4	3
11111	4	3

Table 5.1: *Weight and minimum reverse weight of all row difference patterns, up to cyclic shifts.*

5.3.1 Properties of χ

Figure 5.2 contains a schematic representation of χ.

The χ map is the only nonlinear mapping in Keccak-f with algebraic degree of two. Without it, the Keccak-f round function would be linear. It can be seen as the parallel application of $5w$ S-boxes operating on 5-bit rows.

Since χ has algebraic degree 2, given an input difference pattern a, the space of differences b compatible with a through χ is an affine space [CLO07] that we denote by $\mathcal{A}(a)$. For any compatible $b \in \mathcal{A}(a)$, the weight of (a, b) depends only on a and is equal to $\log_2(|\mathcal{A}(a)|)$. We denote it by $\mathrm{w}(a)$. Obviously, $\mathrm{w}^{\mathrm{dir}}(a) = \mathrm{w}(a)$.

As χ operates on each row independently, the weight can be computed on each row and then summed. This is valid also for the minimum reverse weight. We list in Table 5.1 the weight and the minimum reverse weight for all row difference patterns at the input of χ.

Offsets and bases for constructing the affine space of all single-row differences are given in Table 5.2.

χ is invertible but its inverse is of a different nature than χ itself. For example, it does not have algebraic degree 2. Given a pattern a, the set of patterns that are χ^{-1}-compatible with a do not form an affine space.

5.3.2 Properties of θ

The θ mapping is linear and aimed at diffusion. Figure 5.3 contains a schematic representation of θ. Its effect can be described as adding to each bit $a[x][y][z]$ the bitwise sum of the parities of two columns: that of $a[x-1][\cdot][z]$ and that of $a[x+1][\cdot][z-1]$. It can thus be seen as the addition of a state pattern built using the sum of the parities of such columns. This pattern is called *θ-effect.* An example of the action of θ is given in Figure 5.4.

Difference	offset	base elements			
00000	00000				
00001	00001	00010	00100		
00011	00001	00010	00100	01000	
00101	00001	00010	01100	10000	
10101	00001	00010	01100	10001	
00111	00001	00010	00100	01000	10000
01111	00001	00011	00100	01000	10000
11111	00001	00011	00110	01100	11000

Table 5.2: *Affine space of possible output difference patterns for all input difference patterns, up to cyclic shifts.*

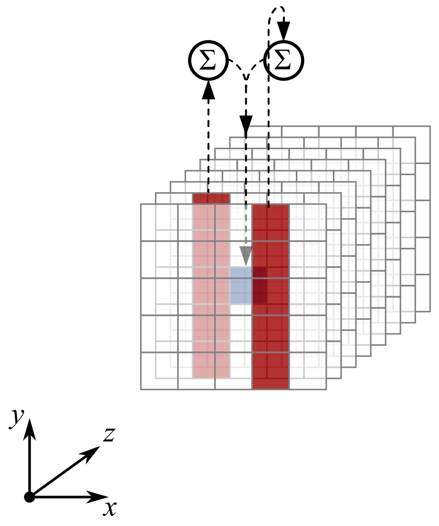


Fig. 5.3: *θ applied to a single bit*

Without θ, the KECCAK-f round function would not provide diffusion of any significance. Thanks to the interaction with χ each bit at the input of a round potentially affects 31 bits at its output and each bit at the output of a round depends on 31 bits at its input. Note that without the translation of one of the two sheet parities this would only be 25 bits.

Given a state pattern a of width $25w$, its *parity* $p(a)$ is a $5w$-bit plane defined by $p[x,z] = \sum_y a[x,y,z]$. The θ-effect is thus defined by $E[x,z] = p[x-1,z] + p[x+1,z-1]$. Hence $\theta(a)[x,y,z] = a[x,y,z] + E[x,z]$.

A column (x,z) is *even* if $p[x,z] = 0$, otherwise it is *odd*. If all columns of a are even we say that a is in the *(parity) kernel*, otherwise we say it is outside the (parity) kernel [BDPV11b]. We denote the set of states in the kernel with $|K|$ and the set of states outside the kernel with $|N|$, where the vertical lines symbolize the χ steps.

If a is in the kernel, then θ acts as the identity on it, being the parity and thus the θ-effect the all-0 pattern. An example is given in Figure 5.5.

A column (x,z) is called *affected* if $E(x,z) = 1$, otherwise it is *unaffected*. We thus distinguish four different types of columns: unaffected even (UE), unaffected odd (UO), affected even (AE) and affected odd (AO). A state in the kernel has only unaffected even columns.

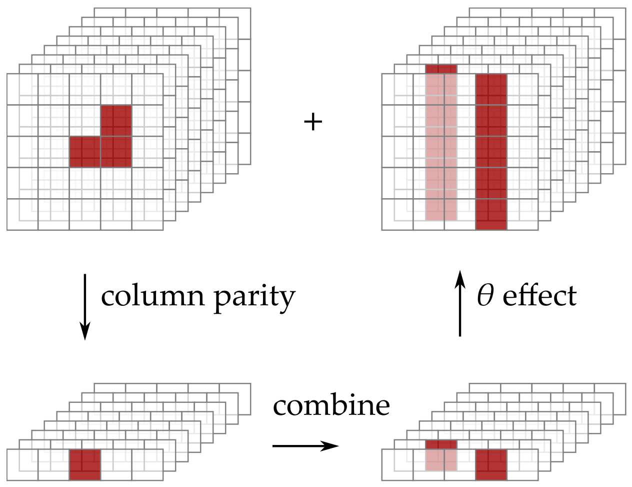


Fig. 5.4: *The action of the θ map. The parity pattern of the given state is used to build the θ-effect, which is then summed to the original state.*

5.3.2.1 Parity-bare states

Definition 5.1. *The Hamming branch number of a state a before θ is defined as the total Hamming weight before and after θ:*

$$B_h(a) = \|a\| + \|\theta(a)\| \,, \tag{5.1}$$

where $\|\cdot\|$ denotes the Hamming weight of a state.

Definition 5.2. *The Hamming branch number of a parity p before θ is defined as the minimum Hamming branch number over all states with the given parity:*

$$B_h(p) = \min_{a:p(a)=p} B_h(a). \tag{5.2}$$

We can compute the Hamming branch number of a parity p based on the following facts. The minimum number of active bits in an unaffected even column is zero, whereas in an unaffected odd column it is one. Since θ acts as the identity for unaffected columns, active bits before θ remain active after θ and passive bits remain passive. So, each unaffected even column contributes zero to the hamming branch number, while each unaffected odd column contributes by 2. On the contrary, active bits in affected columns become passive after θ and vice versa. The total number of active bits for each affected column before and after θ is thus always five. We can thus compute the Hamming branch number of a parity p as $5 \cdot (\#AE + \#AO) + 2 \cdot \#UO$.

The minimum Hamming branch number for a non-zero parity is 12 and corresponds to those parity patterns with a single unaffected odd column and two affected columns.

Definition 5.3. *A parity-bare state is a state pattern whose Hamming branch number is equal to the Hamming branch number of the corresponding parity:*

$$B_h(a) = B_h(p(a)).$$

An example of a parity-bare state is the one given in Figure 5.4.

5.3.2.2 The parity as a set of runs

Given a parity pattern, we can group its odd columns into *runs* [BDPV11b].

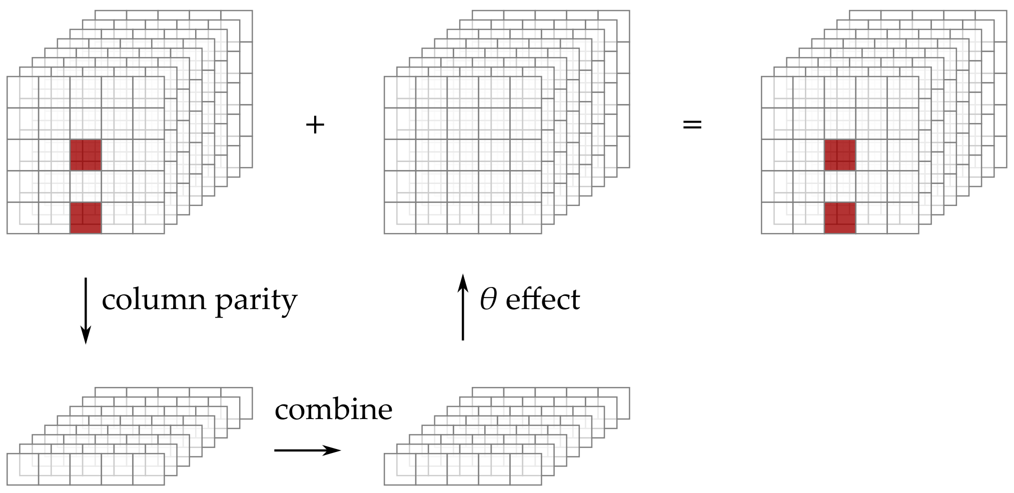


Fig. 5.5: *When the state pattern at the input of θ is in the kernel, then θ acts as the identity. The parity of a is indeed all-0 and thus also the θ-effect is all-0 as well.*

Definition 5.4. *Given a state pattern a, a* run *of* length ℓ *in* $p(a)$ *is a sequence of odd columns of a specified by a* starting point (x_0, z_0) *and such that the i-th element of the sequence is* $(x_i, z_i) = (x_{i-1} - 2, z_{i-1} + 1)$ *for all* $0 < i < \ell$.

A run is fully specified by its starting point (x_0, y_0) and its length ℓ. Each run contains ℓ odd columns and affects two columns:

- starting affected column: the right neighbor of its initial point at $(x_0 + 1, z_0)$ and
- ending affected column: the top-left neighbor of its final point at $(x_0 + 1 - 2\ell, z_0 + \ell)$.

Indeed, two consecutive columns in a run, say (x_i, z_i) and (x_{i+1}, z_{i+1}), contribute to the θ-effect of column $(x_i - 1, z_i + 1)$. So, $E(x_i - 1, z_i + 1) = 0$ for $0 \leq i \leq \ell - 2$. While $E(x_0 + 1, z_0) = E(x_0 + 1 - 2\ell, z_0 + \ell) = 1$.

A parity pattern can contain several runs and an odd column of a run might be affected by another run. Figure 5.6 gives an example of a parity pattern as a set of runs indicating their odd columns and the affected columns.

5.3.3 Properties of π

Figure 5.7 contains a schematic representation of π.

The mapping π is a transposition of the lanes that provides dispersion aimed at long-term diffusion. Without it, Keccak-f would exhibit periodic trails of low weight. The π map operates in a linear way on the coordinates (x, y) and is specified by the matrix $\begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix}$: the lane in position (x, y) goes to position $(y, 2x + 3y)$. It follows that the lane in the origin $(0, 0)$ does not change position.

The inverse of π is specified by $\begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix}^{-1}$: the lane in position (x, y) is mapped to position $(\frac{1}{2}(-3x + y), x)$.

5.3.4 Properties of ρ

The mapping ρ consists of translations within the lanes aimed at providing inter-slice dispersion. Figure 5.8 contains a schematic representation of ρ, while Table 5.3 lists its translation offsets. Without π, diffusion between the slices would be very slow.

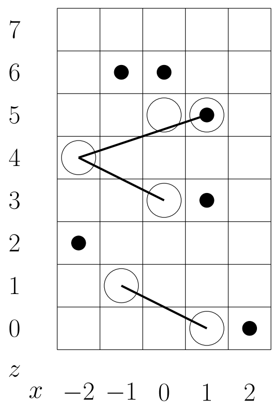


Fig. 5.6: *Example of a parity pattern and θ-effect superimposed. A circle represents an odd column, a dot represents an affected column. The odd columns of a run are connected through a line. There are three runs in the pattern: one of length* 1 *starting at* $(0, 5)$, *one of length* 2 *starting at* $(1, 0)$ *and one of length* 3 *starting at* $(0, 3)$. *Note that* $(1, 5)$ *is an odd column belonging to the run of length* 3 *and affected by the run of length* 1.

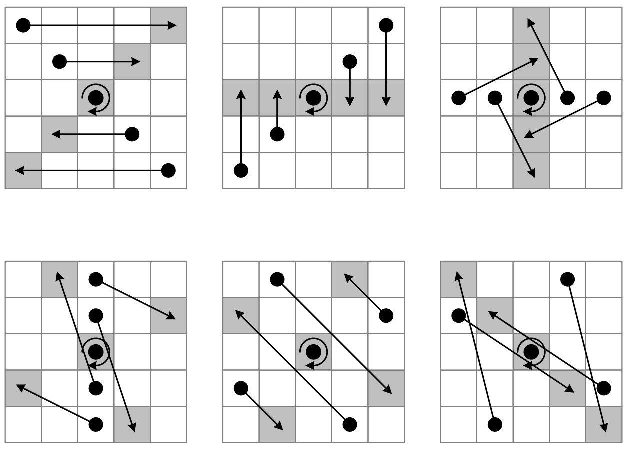


Fig. 5.7: π *applied to a slice. Note that* $x = y = 0$ *is depicted at the center of the slice.*

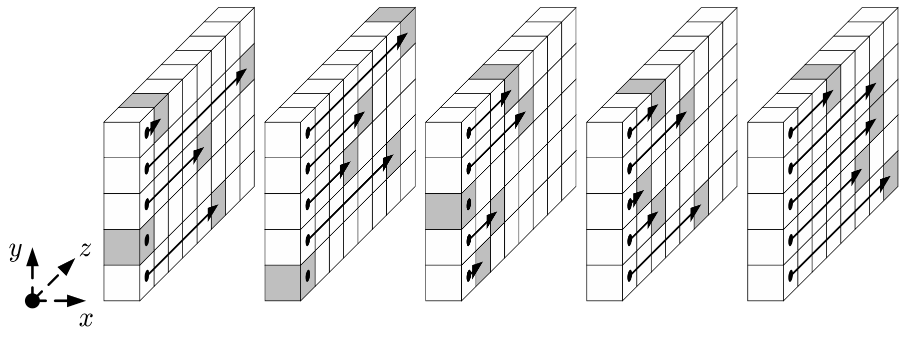


Fig. 5.8: ρ *applied to the lanes.*

	$x=3$	$x=4$	$x=0$	$x=1$	$x=2$
$y=2$	153	231	3	10	171
$y=1$	55	276	36	300	6
$y=0$	28	91	0	1	190
$y=4$	120	78	210	66	253
$y=3$	21	136	105	45	15

Table 5.3: *The offsets of* ρ

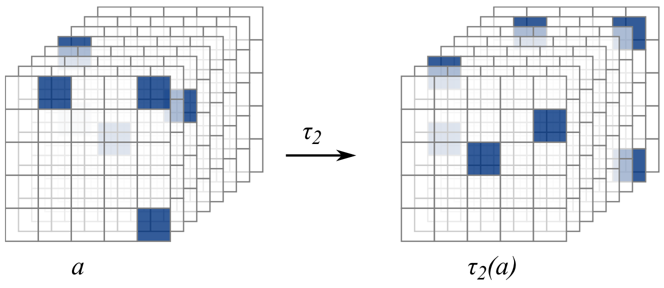


Fig. 5.9: *Translation of a state pattern by* 2 *positions along the* z*-axis.*

The 25 translation constants are the values defined by $i(i+1)/2$ modulo the lane length. It can be proven that for any ℓ, the sequence $i(i+1)/2 \bmod 2^\ell$ has period $2^{\ell+1}$ and that any sub-sequence with $n2^\ell \leq i < (n+1)2^\ell$ runs through all values of $\mathbb{Z}_{2^\ell}$. From this it follows that for lane lengths 64 and 32, all translation constants are different. For lane length 16, 9 translation constants occur twice and 7 once. For lane lengths 8, 4 and 2, all translation constants occur equally often except the translation constant 0, that occurs one time more often.

The inverse of ρ is the set of lane translations where the constants are the same but the direction is reversed.

5.3.5 Properties of ι

The mapping ι consists of the addition of round constants and is aimed at disrupting symmetry. Without it, the round function would be translation-invariant in the z direction and all rounds would be equal making Keccak-f subject to attacks exploiting symmetry such as slide attacks. The number of *active bit positions* of the round constants, i.e., the bit positions in which the round constant can differ from 0, is $\ell+1$. As ℓ increases, the round constants add more and more asymmetry.

The bits of the round constants are different from round to round and are taken as the output of a maximum-length LFSR. The constants are only added in a single lane of the state. Because of this, the disruption diffuses through θ and χ to all lanes of the state after a single round.

Since the ι map consists in the addition of constants, it has no effect on difference propagation.

5.3.6 Translation invariance along the z-axis

Let τ_t a mapping that translates the state by t bits in the direction of the z axis. Then, $\tau_t(a)[x][y][z] = a[x][y][(z-t) \bmod w]$.

Except for ι, all step mappings of the Keccak-f round function are translation-invariant in the direction of the z axis. Namely, for each step α it holds:

$$\tau_t \circ \alpha = \alpha \circ \tau_t \ \forall t \in \mathbb{Z}_w.$$

Let us now recall the z-period of a state.

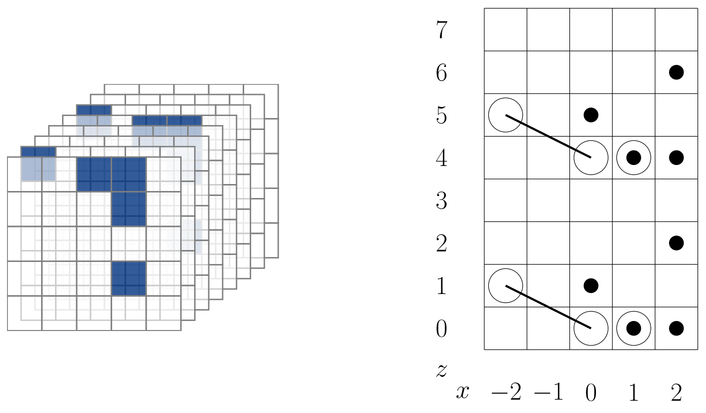


Fig. 5.10: *Example of a symmetric state pattern with period 4 and its corresponding parity pattern, which has period 4 as well.*

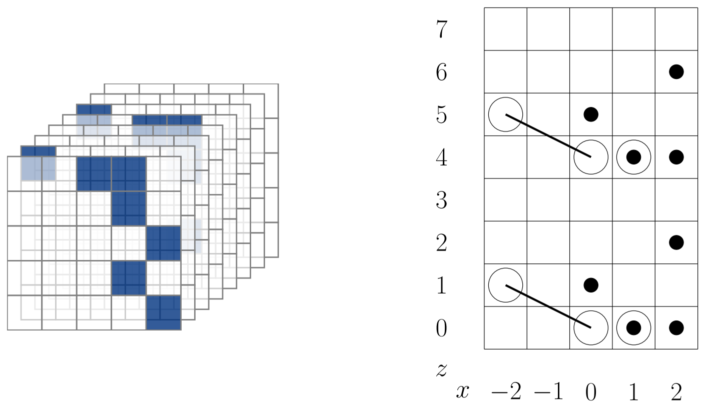


Fig. 5.11: *Example of a state pattern which is not periodic but whose parity pattern is periodic.*

Definition 5.5. *The z-period of a state a is the smallest integer $d > 0$ such that:*

$$\forall x, y \in \mathbb{Z}_5 \text{ and } \forall\ z \in \mathbb{Z}_w : a[x][y][(z+d) \bmod w] = a[x][y][z]\ .$$

It is easy to prove that the z-period of a state divides w. Therefore, the state space consists of the states with z-period 1, 2, 2^2 up to $2^\ell = w$. The number of states with z-period 1 is 2^{25}. The number of states with z-period 2^d for $d \geq 1$ is $2^{2^d 25} - 2^{2^{d-1} 25}$. If α is injective and translation-invariant in the direction of the z axis, α preserves the z-period.

A state a with z-period d can be represented by the lane size w, its z-period d, and its d first slices $a[.][.][z]$ with $z < d$. This is called the *z-reduced representation* of a.

Similarly to state pattern, a parity pattern $p(a)$ is z-periodic if there exists an integer $\bar{z}$ such that $\tau_{\bar{z}}(p(a)) = p(a)$. The smallest integer for which this equality holds is called the *period* of $p(a)$. The parity pattern of a z-periodic state pattern is z-periodic but the opposite is not always valid. Two examples are given in Figure 5.10 and Figure 5.11.

5.4 Differential trails in Keccak-f

Since ι has no effect on difference propagation, the linear layer when working with differentials reduces to $\lambda = \pi \circ \rho \circ \theta$.

We describe differential trails in Keccak-f by a redundant representation with the differences before and after each layer of each round. So, an n-round trail is of the form

$$Q = a_1 \xrightarrow{\lambda} b_1 \xrightarrow{\chi} a_2 \xrightarrow{\lambda} b_2 \xrightarrow{\chi} a_3 \xrightarrow{\lambda} \quad \dots \quad \xrightarrow{\chi} a_{n+1} \tag{5.3}$$

and we denote it as $Q = (a_1, b_1, \dots, a_n, b_n, a_{n+1})$. This representation is redundant since the pattern a_i fully specifies the pattern $b_i = \lambda(a_i)$ and vice versa, being λ linear.

The weight of the trail is given by $\mathrm{w}(Q) = \sum_i \mathrm{w}(a_i \overset{\lambda}{\to} b_i \overset{\chi}{\to} a_{i+1})$. Since λ is linear and thus $b_i = \lambda(a_i)$, this expression simplifies to $\mathrm{w}(Q) = \sum_i \mathrm{w}(b_i \overset{\chi}{\to} a_{i+1})$.

Since χ has algebraic degree 2, the weight of (b_i, a_{i+1}) depends only on b_i and we denote it by $\mathrm{w}(b_i)$. Hence the weight of Q can be written as $\mathrm{w}(Q) = \sum_i \mathrm{w}(b_i)$. It follows that all n-round trails Q that share the sequence of patterns $\tilde{Q} = (a_1, b_1, a_2, b_2, \dots, b_n)$ and end with $a_{n+1} \in \mathcal{A}(b_n)$ have the same weight $\sum_i \mathrm{w}(b_i)$. Obviously, the minimum direct weight of b_n is exactly $\mathrm{w}(b_n)$.

Definition 5.6. *Given a n-round trail* $Q = (a_1, b_1, \dots, a_n, b_n, a_{n+1})$, *its weight profile is the sequence of weights* $(\mathrm{w}(b_1), \dots, \mathrm{w}(b_n))$.

Similarly, we define the parity profile of a trail.

Definition 5.7. *Given a n-round trail* $Q = (a_1, b_1, \dots, a_n, b_n, a_{n+1})$, *its parity profile is the sequence*

$$\underset{\chi}{\Big|}\; X_1 \;\underset{\chi}{\Big|}\; \dots \;\underset{\chi}{\Big|}\; X_n \;\underset{\chi}{\Big|}\; X_{n+1}$$

where

$$X_i = \begin{cases} K \text{ if } a_i \text{ is in the kernel} \\ N \text{ if } a_i \text{ is outside the kernel} \end{cases}$$

and the vertical lines symbolize the χ steps. For short we denote the parity profile of a trail by $|X_1| \dots |X_n| X_{n+1}$.

5.4.1 Extension of trails and trail cores in Keccak-f

Using the redundant representation for trails in Keccak-f, the concepts of trail extension and trail cores can be reformulated as follows.

Given an n-round trail $Q = (a_1, b_1, \dots, a_n, b_n, a_{n+1})$, forward extension consists in first computing the pattern $b_{n+1} = \lambda(a_{n+1})$ and then building all differences a_{n+2} that are compatible with b_{n+1} through χ. Therefore, the weight of any $n+1$-round trail obtained by forward extension of an n-round trail Q is $\mathrm{w}(Q) + \mathrm{w}(\lambda(a_{n+1}))$. Offsets and bases for constructing the affine spaces, and thus state patterns a_{n+2}, are given in Table 5.2.

Similarly, backward extension consists in first building all patterns b_0 compatible with a_1 through χ^{-1} and then computing $a_0 = \lambda^{-1}(b_0)$ for each b_0. Therefore, the weight of any $n+1$-round trail obtained by backward extension of an n-round trail Q can be lower bounded by $\mathrm{w}^{\mathrm{rev}}(a_1) + \mathrm{w}(Q)$.

It follows that a n-round trail $Q = (a_1, b_1, \dots, a_n, b_n, a_{n+1})$ defines a $n+2$-round trail core, which is a set of $n+2$-round trails with Q as central part, b_0 compatible with a_1 through χ^{-1}, $a_0 = \lambda^{-1}(b_0)$, $b_{n+1} = \lambda(a_{n+1})$, a_{n+2} compatible with b_{n+1} through χ and with weight at least $\mathrm{w}^{\mathrm{rev}}(a_1) + \mathrm{w}(Q) + \mathrm{w}(b_{n+1})$. We denote a $n+2$-round trail core as the sequence $\langle a_1, b_1, a_2, b_2, \dots, a_{n+1}, b_{n+1} \rangle$.

Similarly to trails, the weight profile of a $n+2$-round trail core is the sequence of weights $(\mathrm{w}(b_1), \dots, \mathrm{w}(b_{n+1}))$ and its parity profile is the sequence $|X_1| \dots |X_n| X_{n+1}|$.

As explained previously, to find lower bounds on the weight of trails, we limit our search to trail cores. Indeed, a trail core is a set of trails for which we can easily compute the minimum weight. Targeting trail cores results into covering a space that is much smaller than the space of all trails. It is indeed sufficient to generate the representative of the trail core, namely a n-round trail for a $n+2$-round trail core.

5.4.2 Extension of trail cores in Keccak-f

In our search we will need to perform extension of trail cores, that slightly differ from extension of trails as follows.

Given an n-round trail core $\langle a_1, b_1, a_2, b_2, \dots, a_{n-1}, b_{n-1}\rangle$ of weight $\mathrm{w}^{\mathrm{rev}}(a_1) + \mathrm{w}(Q)$, doing forward extension consists in constructing all differences a_n that are compatible with b_{n-1} through χ and then compute b_n. The weight of the obtained $n+1$-round trail cores $\langle a_1, b_1, a_2, b_2, \dots, a_{n-1}, b_{n-1}, a_n, b_n\rangle$ will be $\mathrm{w}^{\mathrm{rev}}(a_1) + \mathrm{w}(Q) + \mathrm{w}(b_n)$.

Doing backward extension consists in constructing all states b_0 that are compatible with a_1 through χ^{-1} and computing the corresponding a_0. To compute the weight of the resulting $n+1$-round trail cores $\langle a_0, b_0, a_1, b_1, a_2, b_2, \dots, a_{n-1}, b_{n-1}\rangle$, we must replace $\mathrm{w}^{\mathrm{rev}}(a_1)$ by $\mathrm{w}(b_0)$ and then add $\mathrm{w}^{\mathrm{rev}}(a_0)$.

5.4.3 The Matryoshka structure

The translation-invariance in the direction of the z axis allows to use structures for small width versions as symmetric structures embedded in larger width versions. This property is called Matryoshka property [BDPV11b].

A practical example involves trails and their weight. A trail for a given width b implies a trail for all larger widths b'. The patterns of this new trail are defined by the z-reduced representations of the original patterns. In other words, each pattern is built by repeating the slices of the original pattern b'/b times. To compute the weight of the trail for the larger width, the weight of the trail for the smaller width must be multiplied by b'/b. Note that the same is not true for the cardinality of differential trails as it depends on the round constants.

The same property holds for trail cores as well. An example is given in Figure 5.12.

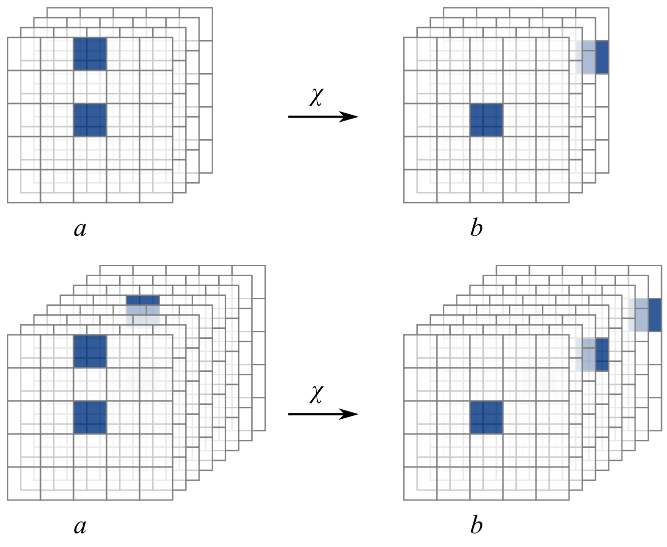


Fig. 5.12: *The Matryoshka consequence. A 2-round trail core in* Keccak-f[100] *(above) gives rise to a 2-round trail core in* Keccak-f[200] *(below). The patterns composing the trail in* Keccak-f[200] *have period 2. Their z-reduced representation are equal to the patterns of* Keccak-f[100].

6

Lower bounding the weight of trails in Keccak-f

In this chapter we apply the techniques described in Chapter 4 to Keccak-f. We make use of the properties of the step functions to define the tree structures and to present efficient methods to extend trails. We use them to scan the space of all trail cores over 3, 4, 5 and 6 rounds up to a given weight.

This chapter is organized as follows. In Section 6.1 we explain how we cover the space of 3, 4, 5 and 6-round trail cores starting by the generation of 2-round trail cores. In Section 6.2 we show how to instantiate the tree traversal approach and in Section 6.3 we present the techniques to efficiently extend trails based on their parity profile. In Section 6.4 and Section 6.5 we provide experimental results for the generation of all 3-round trail cores up to weight 45, from which we can generate all 4-round trail cores up to weight 47, all 5-round trail cores up to weight 49 and all 6-round trail cores up to weight 91. Finally, in Section 6.6 we compare our approach with the one presented in [DV12].

6.1 Scanning the space of trails in Keccak

In our search, we target the space of all 6-round trail cores in Keccak-f up to a given weight. Covering 6 rounds in fact turns out to be feasible, since it comes down to the generation and extension by 3 rounds of 3-round trail cores up to half the given weight and generating trail cores of 3 rounds has the advantage that only a single layer of χ must be passed by forward or backward extension. Three-round trail cores are in fact generated by extending 2-round trail cores by one round.

The same set of 3-round trail cores can actually be used to cover also the spaces of all 4 and 5-round trail cores up to certain limit weights, by extending such 3-round trail cores by one and two rounds respectively. The limit weights we can reach for such spaces depends on the initial target weight for 6-round trail cores and this dependency will be clarified in the following sections.

To generate 2-round trail cores, we define units and cost-bounding functions to apply the tree-traversal method described in Section 4.2. Then, using the same approach, we perform optimized extensions based on the properties of the mixing layer of Keccak-f.

6.1.1 Scanning the space of 6-round trail cores

We start our search based on the following lemma, that is actually generic and not only specific to Keccak-f.

Lemma 6.1. *A 6-round trail of weight W always contains a 3-round trail of weight below or equal to T_3, with $T_3 = \left\lfloor \frac{W}{2} \right\rfloor$.*

Proof. Any 6-round trail $Q = (a_0, b_0, a_1, b_1, a_2, b_2, a_3, b_3, a_4, b_4, a_5, b_5)$ can be seen as the concatenation of two sub-trails $(a_0, b_0, a_1, b_1, a_2, b_2)$ and $(a_3, b_3, a_4, b_4, a_5, b_5)$. Therefore, its weight can be seen as the sum of the weights of the two sub-trails, i.e. $W = \mathrm{w}(a_0, b_0, a_1, b_1, a_2, b_2) + \mathrm{w}(a_3, b_3, a_4, b_4, a_5, b_5)$.
Let $\delta = \min(\mathrm{w}(a_0, b_0, a_1, b_1, a_2, b_2), \mathrm{w}(a_3, b_3, a_4, b_4, a_5, b_5))$. Then, $W \geq 2 \cdot \delta$, which implies that $\delta \leq \frac{W}{2}$. Since the weight is an integer quantity it holds that $\delta \leq \lfloor \frac{W}{2} \rfloor$. □

It follows that given all 3-round trail cores with weight up to T_3, we can generate all 6-round trail cores of weight up to $2T_3 + 1$ by extension of these trail cores by three rounds either in the forward or backward direction. If a trail of weight $2T_3 + 1$ or less is found, then it yields a lower bound on 6-round trail weights. Otherwise, the bound is $2T_3 + 2$, not necessarily tight.

6.1.2 Scanning the space of 3-round trail cores

To scan the space of 3-round trail cores $\langle a_1, b_1, a_2, b_2 \rangle$ up to some limit weight T_3, we distinguish between different sets depending on a_1 and a_2 being in the kernel or not. In fact, the generation of 3-round trail cores starts with the generation of 2-round trail cores $\langle a, b \rangle$. The difference between the two collections of trail cores with a in the kernel or a outside the kernel is the Hamming branch number. For any given limit, the number of trail cores in the kernel is much higher than those outside the kernel. The reason is that in the former case θ acts as the identity and patterns a with small Hamming weight imply small Hamming weight at b, too, keeping the weight low. On the contrary, patterns outside the kernel with small Hamming weight before λ result in patterns with larger Hamming weight after λ, making the weight increase beyond the limit more easily. This calls for different approaches and it is reflected in our strategy.

We denote the set of 2-round trail cores in the kernel with $|K|$ and the set of 2-round trail cores outside the kernel with $|N|$, where the vertical lines symbolize the χ steps.

To cover the whole space of 3-round trail cores $\langle a_1, b_1, a_2, b_2 \rangle$ up to some limit weight T_3, we split trails based on their parity profile, thus obtaining the following classes:

(a) $|K|N|$: a_1 is in the kernel and a_2 is outside the kernel;
(b) $|N|K|$: a_1 is outside the kernel and a_2 is in the kernel.
(c) $|N|N|$: both a_1 and a_2 are outside the kernel;
(d) $|K|K|$: both a_1 and a_2 are in the kernel;

Thus, there are four classes and any 3-round trail core belongs to one and only one of these classes.

For brevity we denote the weights of the three round differentials by shortcut notations: $\mathrm{w}^{\mathrm{rev}}(a_1) = w_0$, $\mathrm{w}(b_1) = w_1$ and $\mathrm{w}(b_2) = w_2$.

To cover the different sets, we extend 2-round trail cores $\langle a, b \rangle$ by one round in the forward or backward direction. Extension is limited to compatible patterns in the kernel or outside the kernel, depending on the set we are targeting.

For $|K|N|$, we partition the space based on the value of w_0.

Lemma 6.2. *All trail cores in $|K|N|$ up to weight T_3 can be generated by forward extending by one round outside the kernel all trail cores in $|K|$ with $\mathrm{w}^{\mathrm{rev}}(a) \leq T_1$ and backward extending by one round in the kernel all trail cores in $|N|$ with $\mathrm{w}^{\mathrm{rev}}(a) + \mathrm{w}(b) < T_3 - T_1$.*

Proof. For a given T_1, any trail core Q in $|K|N|$ satisfies either $w_0 \leq T_1$ or $w_0 > T_1$.

If $w_0 \leq T_1$, then the trail Q starts with a two-round trail core in $|K|$ with $\mathrm{w}^{\mathrm{rev}}(a) \leq T_1$ and hence can be built by forward extending this by one round outside the kernel.

If $w_0 > T_1$, then $w_1 + w_2 < T_3 - T_1$. So, the trail core Q ends with a two-round trail core in $|N|$ with $\mathrm{w}^{\mathrm{rev}}(a) + \mathrm{w}(b) < T_3 - T_1$ and hence can be built by backward extending this by one round in the kernel. □

The optimal value of T_1 is determined by the ratio of 2-round trails in $|K|$ and $|N|$ for the relevant weight functions and the cost of extension.

For $|N|K|$, we apply the same partitioning technique, but based on the weight w_2.

Lemma 6.3. *All trail cores in $|N|K|$ up to weight T_3 can be generated by backward extending by one round outside the kernel all trail cores in $|K|$ with $\mathrm{w}(b) \leq T_1$ and forward extending by one round in the kernel all trail cores in $|N|$ with $\mathrm{w}^{\mathrm{rev}}(a) + \mathrm{w}(b) < T_3 - T_1$.*

Proof. For a given T_1, any trail core Q in $|N|K|$ satisfies either $w_2 \leq T_1$ or $w_2 > T_1$.

If $w_2 \leq T_1$, then the trail Q ends with a two-round trail core in $|K|$ with $\mathrm{w}(b) \leq T_1$ and hence can be built by backward extending this by one round outside the kernel.

If $w_2 > T_1$, then $w_0 + w_1 < T_3 - T_1$. So, the trail core Q starts with a two-round trail core in $|N|$ with $\mathrm{w}^{\mathrm{rev}}(a) + \mathrm{w}(b) < T_3 - T_1$ and hence can be built by forward extending this by one round in the kernel. □

Here T_1 is again a parameter that can be varied for optimizing performance.

For $|N|N|$, we build 3-round trail cores in $|N|N|$ by forward extension of trail cores in $|N|$ for obtaining trail cores with $w_2 \geq w_0$ and by backward extension of trail cores in $|N|$ for obtaining trail cores with $w_0 > w_2$.

Lemma 6.4. *All trail cores in $|N|N|$ up to weight T_3 can be generated by forward extending by one round outside the kernel all trail cores in $|N|$ with $2\mathrm{w}^{\mathrm{rev}}(a) + \mathrm{w}(b) < T_3$ and backward extending by one round outside the kernel all trail cores in $|N|$ with $\mathrm{w}^{\mathrm{rev}}(a) + 2\mathrm{w}(b) \leq T_3$.*

Proof. Any trail core Q in $|N|N|$ satisfies either $w_2 > w_0$ or $w_2 \leq w_0$.

If $w_2 > w_0$, combination with $w_0 + w_1 + w_2 \leq T_3$ gives $2w_0 + w_1 < T_3$. It follows that all trail cores Q in $|N|N|$ with $w_2 > w_0$ start with a two-round trail core in $|N|$ with $2w_0 + w_1 < T_3$ and hence can be built by forward extending these by one round outside the kernel.

If $w_2 \leq w_0$, combination with $w_0 + w_1 + w_2 \leq T_3$ gives $w_1 + 2w_2 \leq T_3$. It follows that all trail cores Q in $|N|N|$ with $w_2 \leq w_0$ end with a two-round trail core in $|N|$ with $w_1 + 2w_2 \leq T_3$ and hence can be built by backward extending these by one round outside the kernel. □

Similarly, we cover the space $|K|K|$.

Lemma 6.5. *All trail cores in $|K|K|$ up to weight T_3 can be generated by forward extending by one round in the kernel all trail cores in $|K|$ with $2\mathrm{w}^{\mathrm{rev}}(a) + \mathrm{w}(b) < T_3$ and backward extending by one round in the kernel all trail cores in $|K|$ with $\mathrm{w}^{\mathrm{rev}}(a) + 2\mathrm{w}(b) \leq T_3$.*

Proof. Any trail core Q in $|K|K|$ satisfies either $w_2 > w_0$ or $w_2 \leq w_0$.

If $w_2 > w_0$, combination with $w_0 + w_1 + w_2 \leq T_3$ gives $2w_0 + w_1 < T_3$. It follows that all trail cores Q in $|K|K|$ with $w_2 > w_0$ start with a two-round trail core in $|K|$ with $2w_0 + w_1 < T_3$ and hence can be built by forward extending these by one round in the kernel.

If $w_2 \leq w_0$, combination with $w_0 + w_1 + w_2 \leq T_3$ gives $w_1 + 2w_2 \leq T_3$. It follows that all trail cores Q in $|K|K|$ with $w_2 \leq w_0$ end with a two-round trail core in $|K|$ with $w_1 + 2w_2 \leq T_3$ and hence can be built by backward extending these by one round in the kernel. □

To cover $|K|K|$, a method is presented also in [DV12, Section 7]. This consists in generating all 2-round trail cores with a in the kernel for which there exists a compatible state $\chi(b)$ in the kernel. Then, forward extension (in the kernel) of these trails is performed. This method is summarized below. It in principle very suitable for the tree structure but very specific for differential trails in Keccak. On the contrary, the approach used in Lemma 6.5 results more generic and can be applied also to linear trails.

We have not implemented the method of [DV12] within the tree framework, also because we believe it would not bring much gain, since covering the case $|K|K|$ is not a bottleneck. An implementation was already available in the KeccakTools and we leverage on it for our experiments. It is worth noticing that for Keccak-f[800] and Keccak-f[1600] the method of [DV12] is still more efficient than ours. While for 200 our approach performs significantly better. For Keccak-f[400], the two methods are comparable from a computational point of view. Experimental results will be provided in Section 6.4.

Covering the case $|K|K|$ with vortices and chains

Here we briefly summarize the approach presented in [DV12, Section 7] to cover the case $|K|K|$. The idea is to consider groups of active bits that form orbitals at a and that results in orbitals also at $\chi(b)$. These groups of active bits are actually defined as sequences. They are called *vortices* and *chains*.

We say that two active bits p_i and p_j are *peer* if they are in the same column at a. We say that they are *chained* if they lie in the same column at b. A *chain* is a sequence of bit positions $\{p_0, p_1, \ldots, p-2n-1\}$ of even length such that p_{2k} and p_{2k+1} are peer and that p_{2k+1} and p_{2k+2} are chained. Additionally, bits p_0 and p_{2n-1} are in so called *knots* (either the same one or different ones) at b. A knot is a slice with three or more active bits. A *vortex* is a chain where p_0 and p_{2n-1} are chained.

By construction, chains and vortices have an even number of active bits per column at a and hence $p(a) = 0$. To ensure that $p(\chi(b)) = 0$ the slices of b should be characterized under this constraint. If the slice has no active bits, then it will be passive also at $\chi(b)$. If a slice contains a single active bit, it will not be in the kernel at $\chi(b)$. A slice with two active bits must result after χ in a slice with orbitals. It happens only if the two bits are in the same column at b. If a slice is a knot, it may result in a slice in the kernel after χ. The active points of a knot are the end points of chains that lead to other knots or that connect back to the same knot.

Any state b for which there exists at least one compatible state $\chi(b)$ with $p(\chi(b)) = 0$ can be represented as a set of vortices and chains connecting knots. To generate all trail cores up to a given weight, states a and b are built by progressively adding chains and vortices. At each step a lower bound on the weight of the trail core is computed to efficiently limit the search. Then forward extension is used to extend (a, b) to 3 rounds.

It is natural to see this approach with the tree structure. Vortices and chains play the role of units and the lower bound defined in [DV12] can be used to lower bound the weight of descendants of a node.

However, we believe that implementing this method within the tree framework would not lead to significant improvements. So, we did not do that to focus on other aspects of our search.

As said, our orbital-based method performs significantly better than the chain-based method for Keccak-f[200], while it does not for the largest variants. This means that for smaller variants of Keccak-f, checking the restrictions imposed by requiring $p(\chi(b)) = 0$ dominates the construction and in these cases it is more convenient to just ignore it and generate all trails with $p(a) = 0$. Subsequently, with extension in the kernel, only the desired patterns will survive. On the contrary, for larger variants, using the orbital-based approach generates too many trails, most of which cannot be extended in the kernel. In these cases, adopting the chain-based approach is more efficient, since it helps to cut the search in advance.

6.1.3 Covering the space of 4 and 5 rounds

Scanning the space of 3-round trail cores up to weight T_3 allows to also cover the whole space of 4 and 5-round trail cores up to weight $T_3 + 2$ and $T_3 + 4$ respectively.

Indeed, since the weight over a single round is at least 2, any 4-round trail core with weight smaller or equal to $T_3 + 2$ satisfies either $\mathrm{w}(b_0) + \mathrm{w}(b_1) + \mathrm{w}(b_2) \leq T_3$ or $\mathrm{w}(b_1) + \mathrm{w}(b_2) + \mathrm{w}(b_3) \leq T_3$. So, all 4-round trail cores with weight below or equal to $T_3 + 2$ can be generated by extending all 3-round trail cores up to T_3 by one round in the forward or backward direction. If no trail is found below the limit, then we can say that the lower bound on the weight of 4-round trail cores is $T_3 + 3$.

Similarly, any 5-round trail core with weight up to $T_3 + 4$ satisfies the condition $\mathrm{w}(b_1) + \mathrm{w}(b_2) + \mathrm{w}(b_3) \leq T_3$ and $\mathrm{w}(b_0) = \mathrm{w}(b_4) \geq 2$. Therefore, by extending all 3-round trail cores up to T_3 by 2 rounds we can generate all 5-round trail cores with weight smaller or equal to $T_3 + 4$. If no trail core is found, we can say that a lower bound on the weight of 5-round trail cores is $T_3 + 5$.

6.2 Generating two-round trail cores in Keccak-f as a tree traversal

To generate 3-round trail cores, we start from 2-round trail cores either in $|K|$ or in $|N|$. In this section we show how to generate such 2-round trail cores in KECCAK-f by applying the tree-traversal method described in Section 4.2. In order to do this, we will need to define units, order relations among units and cost-bounding functions.

Our definitions will make use of order relations among components of the KECCAK-f state. In particular, we define the following orderings.

Definition 6.1 (bit-order). *The* bit-order *is defined as the lexicographic order on* $[z, x, y]$ *and is denoted by* $\prec_b$. *Namely, given two bit positions* $b = (x, y, z)$ *and* $b' = (x', y', z')$ *then*

$$b \prec_b b' \Leftrightarrow \begin{cases} z < z' \text{ or} \\ z = z' \text{ and } x < x' \text{ or} \\ z = z' \text{ and } x = x' \text{ and } y < y'. \end{cases}$$

Definition 6.2 (column-order). *The* column-order *is defined as the lexicographic order on* $[z, x]$ *and is denoted by* $\prec_c$. *Namely, given two columns* $c = (x, z)$ *and* $c' = (x', z')$ *then*

$$c \prec_c c' \Leftrightarrow \begin{cases} z < z' \text{ or} \\ z = z' \text{ and } x < x'. \end{cases}$$

Additionally, in the tree traversal, we will take into account the symmetry properties of KECCAK-f and hence consider only canonical difference patterns. Indeed, except for ι, all step mappings of the KECCAK-f round function are translation-invariant in the direction of the z axis. For a given order relation among units, we can define an order relation $\prec_p$ over patterns as in Equation 4.2. Hence, we can compare each pattern with its translated variants along the z-axis and the next definition follows.

Definition 6.3. *A difference pattern* a *is* z-canonical *if and only if*

$$a = \min_{\prec_p} \{\tau_t(a) \ : \ t \in \mathbb{N}\}, \tag{6.1}$$

where τ_t *is the translation by* t *bits in the direction of the* z *axis.*

The z-canonicity imposes some restrictions on the active bit positions, that we can use to optimize our search, as shown in next sections.

6.2.1 Generating trail cores in the kernel

In this section, we deal with the generation of 2-round trail cores in the kernel.

As explained in Section 6.1.2, we use such trail cores as starting point for the generation of trails in $|K|N|$, $|N|K|$ and $|K|K|$. Depending on the target space, we consider trail cores in $|K|$ satisfying different conditions on their weight. Namely, $\mathrm{w}^{\mathrm{rev}}(a) \leq T_1$ (as in Lemma 6.2), $\mathrm{w}(b) \leq T_1$ (as in Lemma 6.3), and $2\mathrm{w}^{\mathrm{rev}}(a) + \mathrm{w}(b) < T_3$ and $\mathrm{w}^{\mathrm{rev}}(a) + 2\mathrm{w}(b) \leq T_3$ (as in Lemma 6.5).

Therefore, to cover the different cases, we consider the cost function of the form $\alpha\mathrm{w}^{\mathrm{rev}}(a) + \beta\mathrm{w}(b)$ with $\alpha, \beta \in \{0, 1, 2\}$.

6.2.1.1 Defining units and unit-order

Difference patterns a in the kernel have an even number of active bits in each column, namely zero, two or four.

Definition 6.4. *A pair of active bits in the same column is called an* orbital. *We call* bottom bit *the bit with smaller y-coordinate and* top bit *the other one.*

An active bit is fully specified by its coordinates $(x, y, z) \in \mathbb{Z}_5 \times \mathbb{Z}_5 \times \mathbb{Z}_w$. Therefore, an orbital is determined by the coordinates of its active bits and we represent it as a 4-tuple (x, y_1, y_2, z) with the convention that $y_1 < y_2$ to avoid duplicates.

Definition 6.5 (Orbital-order). *The* orbital-order *is defined as the lexicographic order on $[z, x, y_1, y_2]$ and is denoted by $\prec_\omega$. Namely, given two orbitals $\omega = (x, y_1, y_2, z)$ and $\omega' = (x', y_1{}', y_2{}', z')$ then*

$$\omega \prec_\omega \omega' \Leftrightarrow \begin{cases} z < z' \text{ or} \\ z = z' \text{ and } x < x' \text{ or} \\ z = z' \text{ and } x = x' \text{ and } y_1 < y_1{}' \text{ or} \\ z = z' \text{ and } x = x' \text{ and } y_1 = y_1{}' \text{ and } y_2 < y_2{}' \end{cases}$$

The number of possible orbitals in a column is $\binom{5}{2} = 10$ and they are depicted in Figure 6.1 ordered by using $\prec_\omega$ of Definition 6.5.

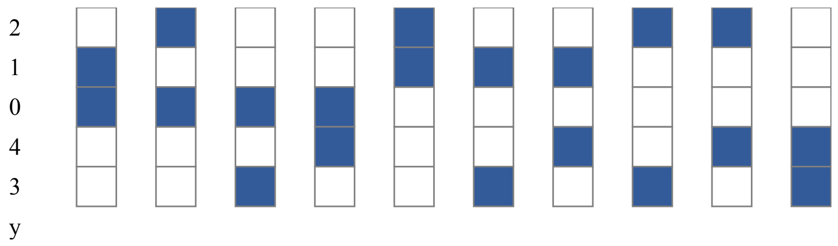


Fig. 6.1: *Possible orbital positions in a column.*

To keep the difference pattern in the kernel when adding units, but also to avoid duplicates, we must impose some restrictions.

The first restriction has already been specified and requires that $y_1 < y_2$. This allows to uniquely represent orbitals.

The second restriction requires that in a given column with two orbitals (x, y_1, y_2, z) and (x, y'_1, y'_2, z) it holds that $y_2 < y'_1$, namely, the first bit of the second orbital is higher than the second bit of the first orbital. This allows to represent a column containing two orbitals in a unique way.

The third restriction requires avoiding overlapping of orbitals. Namely, an orbital (x, y_1, y_2, z) can be added to a pattern only if its bits (x, y_1, z) and (x, y_2, z) are both passive in the pattern. This allows to uniquely represent columns and to guarantee the parity zero.

A column can thus have either zero, one or two orbitals. The possible combinations are depicted in Figure 6.2.

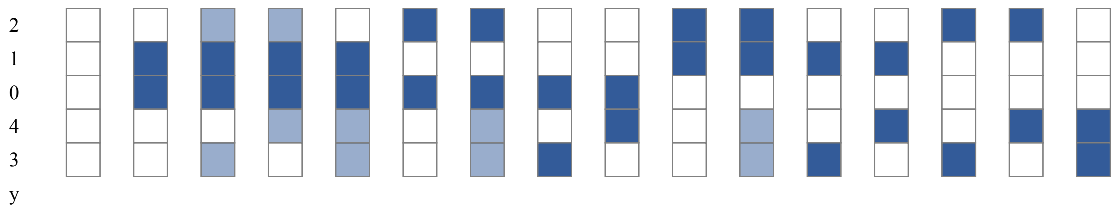


Fig. 6.2: *Possible orbitals in a column.*

Lemma 6.6. *Any state pattern in the kernel can be represented as an ordered list of orbitals.*

Proof. We describe a recursive procedure for decomposing a difference pattern by removing orbitals from the state one by one in a unique way. We start from the difference pattern a and an empty list of orbitals ℓ.
Over all active columns of a, choose the one with the highest (x, z) coordinates, where the order among column positions is the column-order of Definition 6.2. Remove the two active bits in this column with the highest y coordinates from the state. Build an orbital from these two bits and add it at the beginning of the list of orbitals ℓ. Repeat this step operation until there are no active columns left in the state. This procedure ends as the number of active bits in a state is finite (since the state is finite) and even (since the state is in the kernel). □

Lemma 6.6 allows us to arrange difference patterns in the kernel in a tree where the parent of a node is the pattern with the last orbital removed. At the root of the tree there is the empty pattern.

Definition 6.6. *The* orbital-tree *is a unit-tree whose units are orbitals.*

A plane representation of the orbital-tree is given in Figure 6.3

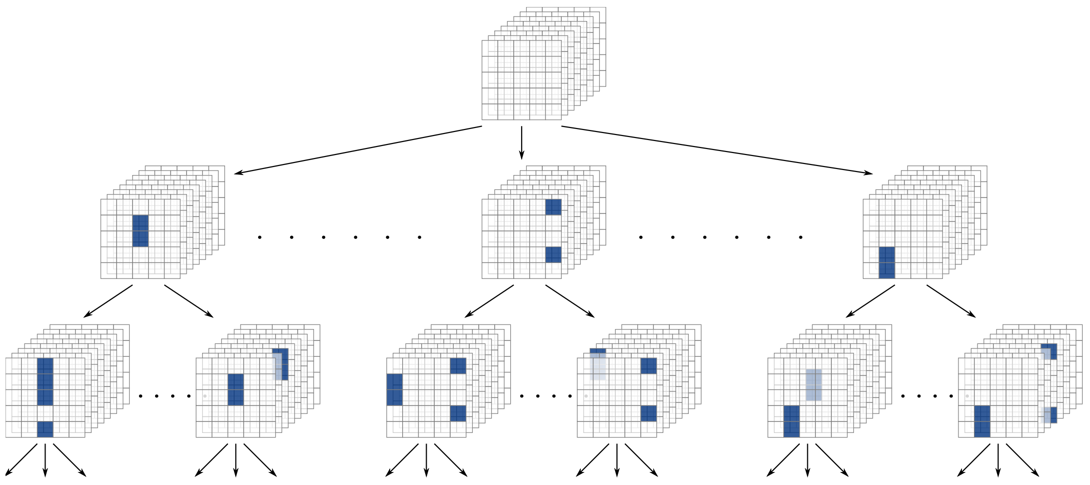


Fig. 6.3: *Plane representation of the orbital-tree. Units are orbitals and at the root there is the empty pattern.*

To concretely build difference patterns in the kernel we start from the empty pattern and iteratively add orbitals after the last orbital already in the state, with restrictions specified above.

6.2.1.2 Lower bounding the cost

To bound the cost of all descendants of a given pattern, we can use the cost of the pattern itself as proved in the following lemma.

Lemma 6.7. *In an orbital-tree, the cost of a pattern lower bounds the cost of all its descendants.*

Proof. Given a 2-round trail core (a, b), adding an orbital to a gives rise to a new trail core (a', b') with the following properties. The difference pattern a' is equal to a with two more active bits, due to the addition of the orbital. Since the θ map acts as the identity, these two bits remain active after θ and are then moved across the state difference by ρ and π. It follows that b' is equal to b with two additional bits. Hence, adding an orbital to a adds two active bits *at* a and two active bits *at* b. Hence, whether we consider $\mathrm{w}^{\mathrm{rev}}(a)$, $\mathrm{w}(b)$, $2\mathrm{w}^{\mathrm{rev}}(a)+\mathrm{w}(b)$ or $\mathrm{w}^{\mathrm{rev}}(a)+2\mathrm{w}(b)$, the cost is monotonic with respect to addition of orbitals, because both $\mathrm{w}^{\mathrm{rev}}$ and w are monotonic when adding active bits [DV12]. □

From Lemma 6.7 it follows that we can avoid adding new orbitals to a pattern when its cost is already above the budget.

Estimating the cost bound

Computing the exact weight of a pattern, and thus its cost, is a time-consuming operation. So, in order to optimize our search, we define a function Γ to estimate the cost due to the addition of a new orbital and understand if the addition is feasible. The function Γ should be a trade-off between efficiency and accuracy. If it underestimates the cost, it will cause the search losing time trying to add orbitals in positions leading to patterns exceeding the target cost. On the other hand, a too precise function will make the search losing time by the very computation of Γ for each individual orbital.

The definition of Γ is based on the following fact. In general, if an active bit is added to a passive row of a, then $\mathrm{w}^{\mathrm{rev}}(a)$ increases by 2. If it is added to an active row, then it may leave $\mathrm{w}^{\mathrm{rev}}(a)$ as it is. Similarly, if the active bit added to a is moved into a passive row of b by λ, then $\mathrm{w}(b)$ is increased by 2 and if it is moved into an active row of b then it may leave $\mathrm{w}(b)$ unchanged. For these reasons, our choice for the function Γ is the following.

Definition 6.7. *Given a pattern a with cost $\gamma = \alpha\mathrm{w}^{\mathrm{rev}}(a) + \beta\mathrm{w}(b)$ and an orbital $\omega \notin a$, the* cost estimation function *is defined as $\Gamma(a,\omega) = \alpha \cdot \Gamma_a(\omega) + \beta \cdot \Gamma_b(\omega)$ with*

$$\Gamma_s(\omega) = \begin{cases} \gamma + 4 & \text{if both the bits are added to passive rows at } s \\ \gamma + 2 & \text{if one of the bits is added to a passive row at } s \\ & \text{and the other to an active row at } s \\ \gamma & \text{if both the bits are added to active rows at } s \end{cases} \tag{6.2}$$

for $s = a$ or b.

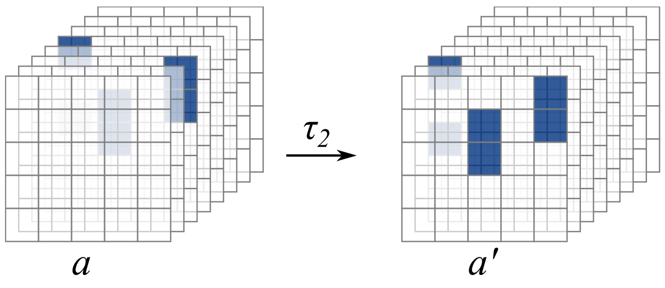
The function Γ allows to limit the addition of orbitals. Indeed, if the cost of a pattern is already T or $T-1$ then we cannot add orbitals if either bit goes to passive rows. Similarly, if the cost is equal to $T-2$ or $T-3$, then at most one passive row can be turn to active. This is valid both at a and at b.

6.2.1.3 Remark on z-canonicity

When generating trail cores in the kernel, z-canonicity imposes restrictions on the position of orbitals in the difference pattern.

A z-canonical pattern must have an orbital in the first slice. In fact, for any pattern consider the first orbital in its orbital-list, say $(x, \{y_1, y_2\}, \bar{z})$. If we translate the pattern by $\bar{z}$ positions along the z-axis, we obtain an orbital-list whose first orbital is $(x, \{y_1, y_2\}, 0)$ that is smaller than $(x, \{y_1, y_2\}, \bar{z})$ if $\bar{z} \neq 0$, using the lexicographic order defined above.

Example 6.1. Consider the following state a and its translated variant along the z-axis by two positions.



The orbital-list of a is $[(0,1,2,2),(2,1,2,2),(-2,0,2,4)]$. When translated by 2 positions along the z-axis, the orbital list becomes $[(0,1,2,0),(2,1,2,0),(-2,0,2,2)]$, which is smaller according to the order $\prec_\omega$ since $(0,1,2,0) \prec_\omega (0,1,2,2)$. △

Let $(\bar{x},\bar{y}_0,\bar{y}_1,0)$ be the smallest orbital of a pattern. Then no orbital can be added to sheet x if $x < \bar{x}$. Indeed, any orbital in position (x,y_0,y_1,z), with $x < \bar{x}$ once translated by z positions along the z-axis becomes the orbital $(x,y_0,y_1,0)$. When comparing $(\bar{x},\bar{y}_0,\bar{y}_1,0)$ with $(x,y_0,y_1,0)$, the latter is smaller than the former, following Definition 6.5.

Similarly, no orbital can be added to sheet $\bar{x}$ if its bottom bit is smaller than $\bar{y}_0$. No orbital with bottom bit in lane $(\bar{x}_0,\bar{y}_0)$ can be added if its top bit is smaller than $\bar{y}_1$.

6.2.2 Generating trail cores outside the kernel

In this section, we deal with the generation of 2-round trail cores outside the kernel.

As explained in Lemma 6.2, Lemma 6.3, and Lemma 6.4 we start from such trail cores to generate trails in $|K|N|$, $|N|K|$ and $|N|N|$. Depending on the target space, we consider different conditions on the weight of trail cores in $|N|$. Namely, $\mathrm{w}^{\mathrm{rev}}(a) + \mathrm{w}(b) < T_3 - T_1$, $2\mathrm{w}^{\mathrm{rev}}(a) + \mathrm{w}(b) < T_3$, and $\mathrm{w}^{\mathrm{rev}}(a) + 2\mathrm{w}(b) \leq T_3$.

Therefore, to cover the different cases, we consider the cost function of the form $\alpha\mathrm{w}^{\mathrm{rev}}(a) + \beta\mathrm{w}(b)$ with $\alpha,\beta \in \{1,2\}$.

6.2.2.1 Parity-bare states and free-orbital-trees

For a given parity p, parity-bare states are those with the minimum Hamming branch number. These states have the minimum number of active bits in unaffected columns, that is zero if it is even and one if it is active. If we add an orbital to an unaffected column of a parity-bare state a, the cost of the corresponding 2-round trail core cannot decrease, since θ acts as the identity for such columns and bits active before θ are active also after it and thus also at b.

Hence, we can generate all difference patterns up to some given cost by starting from the generation of parity-bare patterns up to the given cost and then add orbitals to their unaffected columns.

Definition 6.8. *An orbital in an unaffected column is called* free-orbital.

We can use parity-bare states and free-orbitals as building blocks of any pattern outside the kernel.

Lemma 6.8. *Each difference pattern can be decomposed in a unique way in a parity-bare state and a list of free-orbitals, where the active bits of the parity-bare state do not overlap with those of the orbitals, the active bits of the orbitals do not overlap mutually and the parity-bare state and the original state have the same parity.*

Proof. We describe a recursive procedure for performing the decomposition by removing free-orbitals from the state one by one in a unique way.

We start from a pattern a and an empty list of free-orbitals ℓ. Over all unaffected columns with more than a single active bit, choose the one with the highest (x,z) coordinates, where the order among column positions is the column-order of Definition 6.2. Remove the two active bits in this column with the highest y coordinates from the pattern. Build a free-orbital from these two bits and add it at the beginning of the list of free-orbitals ℓ. Repeat this step operation until there are no unaffected columns left in the state with more than a single active bit.

This procedure ends as the number of active bits in a state is finite and each step removes two active bits. The resulting pattern is parity-bare as any unaffected column has either 0 (even) or 1 (odd) affected bits. That parity-bare state has the same parity as the original state as removing an orbital does not affect the parity. Finally, the active bits of the free-orbitals and parity-bare state do not overlap as every free-orbital is taken from an unaffected column and hence removes bits at a and at b. □

An example of a difference pattern outside the kernel as the union of a parity-bare state and a list of free-orbitals is given in Figure 6.4.

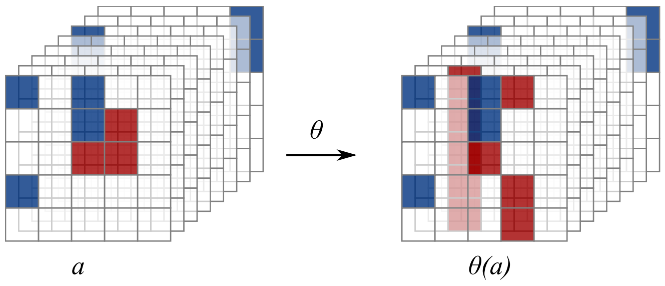


Fig. 6.4: *A difference pattern outside the kernel. In red the active bits of the parity-bare pattern and in blue the free-orbitals.*

Lemma 6.8 allows us to arrange difference patterns with a given parity in a tree where the parent of a node is the pattern with the last free-orbital removed. At the root of the tree there is a parity-bare pattern. We thus have a forest with a tree for each parity-bare state.

Definition 6.9. *We call* free-orbital-tree *a unit-tree whose units are free orbitals and whose root is a parity-bare pattern.*

In Figure 6.5 a plane representation of a free-orbital-tree is given.

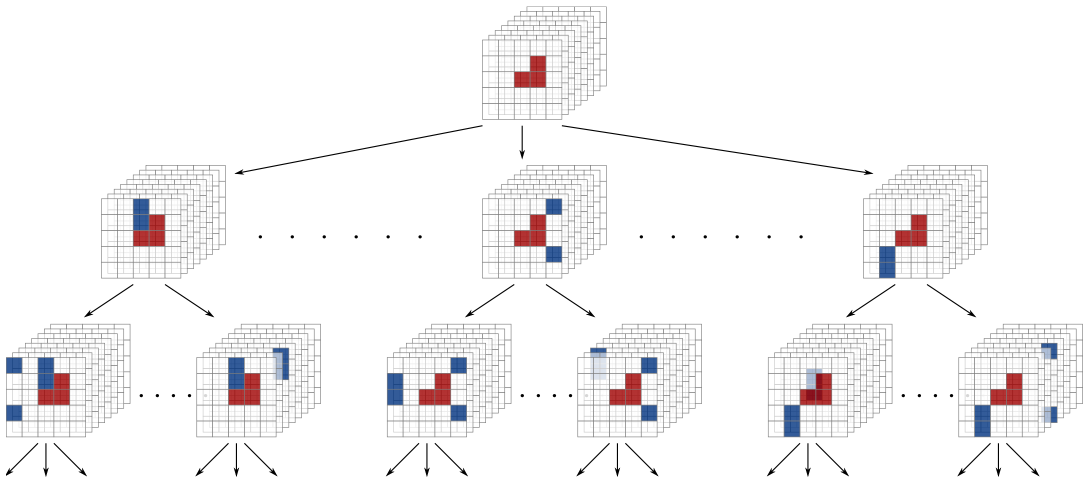


Fig. 6.5: *A free-orbital-tree. At the root of the tree there is a parity-bare state. Units are free-orbitals.*

The orbital-tree (i.e. the tree of difference patterns in the kernel) is a special case of free-orbital-tree: the free-orbital tree with the empty state at its root. All nodes in a free-orbital-tree have the same parity as adding orbitals does not modify the parity.

To bound the cost of all descendants of a node, we can apply the same result of Lemma 6.7 and use the cost of the node itself. In the same way, we make use of the Γ function of Definition 6.7 to estimate the cost bound and make our search more efficient.

It now remains to explain how to generate parity-bare states and we do it in next section.

6.2.2.2 Column assignments and the run-tree

In a parity-bare state an affected column can assume 16 possible values and an unaffected odd column can assume 5 possible values.

Definition 6.10. *An unaffected odd column or affected column with specified value is called* column assignment.

For our purposes, it is natural to represent a parity-bare state as a list of column assignments. We thus build parity-bare states by incrementally adding column assignments. To do it in an efficient way, we group them by using runs. As explained in Section 5.3.2.2, each run contains a number of odd columns and affects two columns.

When there is more than one run, the odd column of a run can be affected by another run, resulting in an affected odd column. In such a case, to preserve the grouping into runs, we express the column assignment of an affected odd column as the bitwise sum of two column assignments:

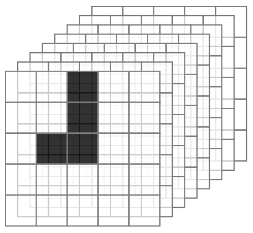
- the unaffected odd column of one run,
- the affected even column of the other run.

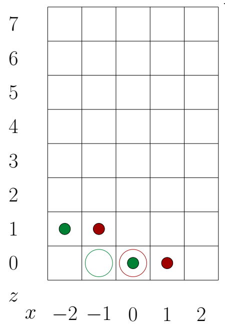
To make this representation unique, we fix the value of the unaffected odd component.

Definition 6.11. *An unaffected odd column with a single active bit in $y = 0$ is called an (unaffected)* odd-0 *column.*

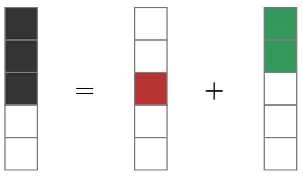
Thus, an affected odd column is expressed as the sum of an odd-0 column and an affected even column.

Example 6.2. Consider the following difference pattern and its corresponding parity pattern.



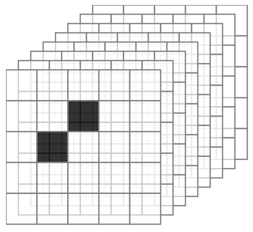


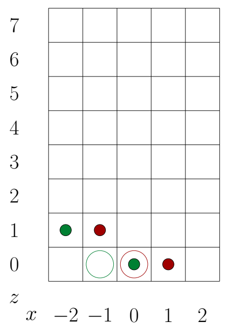
Column $(0, 0)$ is affected odd. It is an odd column of the red run, but it is also affected by the green run. Therefore, we can see it as the sum of two column assignments:



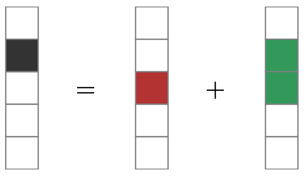
The affected even column assignment is the one with green active bits, while the odd-0 column assignment is the one with red active bit. △

Example 6.3. Consider the following difference pattern and its corresponding parity pattern.





Column $(0, 0)$ is affected odd. In this case the bit $y = 0$ is passive. Therefore, we can see it as the sum of two column assignments that cancel the bit at $y = 0$:



The affected even column assignment is the one with green active bits, while the odd-0 column assignment is the one with red active bit. △

It follows that an unaffected odd column can be added to an affected even column only if it is an odd-0 column and vice versa. This also optimizes the search.

We define an order relation over column assignments based on their type and on the associated runs. The idea is that for a run starting at (x_0, z_0), the starting affected column $(x_0 + 1, z_0)$ comes before the first odd column (x_0, z_0), which comes before the second odd column in $(x_0 - 2, z_0 + 1)$ and so on until the last odd column $(x_0 - 2\ell, z_0 + \ell)$. The ending affected column $(x_0 + 1 - 2\ell, z_0 + \ell)$ comes at the end. When comparing two columns belonging to two different runs, the one corresponding to the smallest run comes before, where the smallest run is given by an order relation defined over runs.

It follows that in the incremental generation of state patterns, a new run is not started until the previous one has been completed.

We first introduce an order relation over runs. Since a run is fully specified by its starting point (x, z) and its length ℓ, we represent it as the triple (x_0, z_0, ℓ).

Definition 6.12 (Run-order). *The* run-order *is defined as the lexicographic order on* $[z_0, x_0, \ell]$ *and is denoted by* $\prec_r$. *Namely, given two runs* $r = (x_0, z_0, \ell)$ *and* $r' = (x'_0, z'_0, \ell')$ *then*

$$r \prec_r r' \Leftrightarrow \begin{cases} z_0 < z'_0 \text{ or} \\ z_0 = z'_0 \text{ and } x_0 < x'_0 \text{ or} \\ z_0 = z'_0 \text{ and } x_0 = x'_0 \text{ and } \ell < \ell' \end{cases}$$

A column assignment is determined by the coordinates of its column position and the value of its bits but also by the run it belongs to. We represent it as a 4-tuple (x_0, z_0, i, v) where (x_0, z_0) is the starting point of the run it belongs to, i indicates which column of the run it is and v is the column value. i is an integer between -1 and ℓ, where -1 identifies the starting affected column, ℓ the ending affected column and the other values identify the odd columns.

We can now define the order on column assignments.

Definition 6.13 (Column assignment-order). *The* column assignment-order *is denoted by* $\prec_{ca}$ *and is defined as follows. Given two column assignments* $c = (x_0, z_0, i, v)$ *and* $c' = (x'_0, z'_0, i', v')$ *then*

$$c \prec_{ca} c' \Leftrightarrow \begin{cases} z_0 < z_0' \text{ or} \\ z_0 = z_0' \text{ and } x_0 < x_0' \text{ or} \\ z_0 = z_0' \text{ and } x_0 = x_0' \text{ and } i < i' \text{ or} \\ z_0 = z_0' \text{ and } x_0 = x_0' \text{ and } i = i' \text{ and } p(v) < p(v') \text{ or} \\ z_0 = z_0' \text{ and } x_0 = x_0' \text{ and } i = i' \text{ and } p(v) = p(v') \text{ and } v < v' \end{cases}$$

where $p(v)$ denotes the parity of the column.

The fourth condition is the one that ensures that a shorter run comes before a longer run.

Lemma 6.9. *Any parity-bare pattern can be represented in a unique way as an ordered list of column assignments.*

Proof. We describe a recursive procedure for decomposing a difference pattern that is parity-bare by removing column assignments from the state one by one in a unique way. We start from the difference pattern a and an empty list of column assignments ℓ. First, compute the parity pattern of a and identify the runs composing it. Among the runs choose the highest one, where the order among runs is the run-order of Definition 6.12.

Identify the affected ending column of this run. If it is even, remove all active bits of the column from the state, build a column assignment from them and add it at the beginning of the list ℓ. If it is odd, we distinguish between the following two cases. If the bit at $y = 0$ is active, then remove all active bits of the column except the one at $y = 0$ from the state, build a column assignment from them and add it at the beginning of the list ℓ. If the bit at $y = 0$ is passive, then remove all active bits of the column from the state, build a column assignment, complement the $y = 0$ bit, add the column assignment at the beginning of the list ℓ and finally set the bit at $y = 0$ to one in the state.

Then identify the last odd column of the run and do the following. If the column is not affected, then remove the active bit of the column from the state, build a column assignment from it and add it at the end of ℓ. Otherwise, if the column is affected and the bit at $y = 0$ is active, remove it from the state, build an odd-0 column and add it to ℓ. If the bit $y = 0$ is passive, set it to one in the state and add an odd-0 column to ℓ.

Then do the same with all the odd columns of the run and for the starting affected column proceed as done for the ending affected column. Repeat this operation until there are no more runs left in a.

This procedure ends as the number of runs in a parity pattern is finite and each step removes columns from them.

The resulting list ℓ will be such that column assignments of the same runs are in consecutive positions. □

Lemma 6.9 allows us to arrange all parity-bare states in a tree. In this tree the parent of a node is the state with the last column assignment removed and its root is the empty state.

Definition 6.14. *A unit-tree whose units are column assignments and whose root is the empty pattern is called* run-tree*.*

A plane representation of the run-tree is given in Figure 6.6.

Clearly, not all nodes of the run-tree define a valid difference pattern. Indeed, a pattern is valid only if its last run is complete, i.e. if the last column in the column assignment-list is an ending affected column. Only such patterns are concretely output by our search algorithm.

Therefore, building all difference patterns outside the kernel can be done by traversing the run-tree and then for each parity-bare pattern traverse its free-orbital-tree.

6.2.2.3 Lower bounding the cost

Adding an affected even column assignment to a in a column position where there is already an unaffected odd column assignment (or vice versa) is not monotonic in the weight. Indeed, this

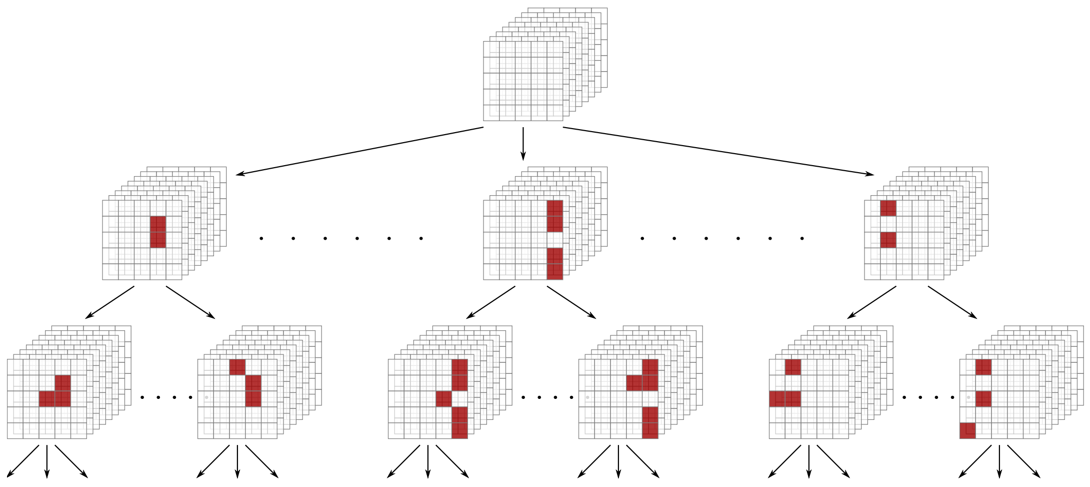


Fig. 6.6: *Plane representation of the run-tree. The root is the empty pattern. At depth 1, the starting affected column of the first run is added. At depth 2, the first odd column is added. Then at depth 3, some nodes will have the second odd column, while others will have the ending affected column of the run.*

action turns an active bit at a or at b into passive and it may happen that the weight loss due to this removal is larger than the weight gain due to the new added bits. For this reason, we cannot use the cost of a pattern to prune the tree, but must define a lower bound for the cost of all its descendants. It turns out that we can define such a lower bound that is reasonably tight.

By considering the worst-case loss due to the addition of overlapping column assignments, we can lower bound the cost of all descendants of a parity-bare state:

1. Adding an odd-0 column assignment to an affected even column moves the active bit at $y = 0$ from a to b or vice versa.
2. Adding an affected even column to an odd-0 column makes either its active bit at a or its active bit at b passive, but never both. It depends on the particular affected even assignment and hence is not known in advance. As a special case, if the bit of the odd-0 column is the only active bit in its slice at a, adding an even column assignment cannot reduce the number of active bits in that slice to 0 as it adds an even number of bits and the contribution of the bit at a remains the same.

From this we derive a cost function that lower bounds the weight of a parity-bare state and its descendants.

Definition 6.15. *The bound on the descendants of a pattern is defined as follows. Given the parity-bare state a and its corresponding state b compute reduced states $\bar{a}$ and $\bar{b}$ as follows. For each affected even column remove the active bit in $y = 0$ at a or in the corresponding position at b. For every odd-0 column, if it is in a slice whose only active row is $y = 0$, then remove the active bit in the corresponding position at b. Let $N_0(a)$ be the set of unaffected odd-0 columns whose slices at a contain at least one active row different from $y = 0$. The bound on the descendants of a is $L(a) = \alpha \cdot \mathrm{w}^{\mathrm{rev}}(\bar{a}) + \beta \cdot \mathrm{w}(\bar{b}) - 2 \cdot \max(\alpha, \beta) \cdot \#N_0(a)$.*

Lemma 6.10. *The bound $L(a)$ of Definition 6.15 lower bounds the cost of the parity-bare state a and all its descendants.*

Proof. First of all, it cannot overestimate the cost of a parity-bare state. This follows from the fact that $\bar{a}$ and $\bar{b}$ are obtained by removing bits at a and/or at b, so the weight of reduced states is smaller than the weight of original states, since the weight functions $\mathrm{w}^{\mathrm{rev}}(a)$ and $\mathrm{w}(b)$ are monotonic in the addition of bits [BDPV11b]. It follows that $L(a) \leq \alpha \cdot \mathrm{w}^{\mathrm{rev}}(\bar{a}) + \beta \cdot \mathrm{w}(\bar{b}) \leq \alpha \cdot \mathrm{w}^{\mathrm{rev}}(a) + \beta \cdot \mathrm{w}(b)$.

It remains to prove that the cost function lower bounds the weight of descendants.

For those descendants obtained by adding column assignments that do not remove bits from a and b, the weight is lower bounded by the weight of the reduced states, since they have less active bits. This is the case for states obtained by adding unaffected odd columns that are not odd-0 columns and affected even columns that are not added to odd-0 columns.

For descendants obtained by adding an odd-0 column to an affected even column, the active bit in $y = 0$ at a can be moved to b or vice versa. The reduced states don't have these bits, so again the weight of reduced states lower bounds the weight of such descendants.

The only cases where bits can be removed from reduced states are when affected even columns are added to odd-0 columns.

For those columns not in $N_0(a)$: if the bit in $y = 0$ of the affected even column is passive, then the bit remains active at a and becomes passive at b. The weight of such states is lower bounded by the weight of reduced states because they don't have such bits at b. If the bit in $y = 0$ of the affected even column is active, then the bit at b remains active and the bit at a is moved to another row, not changing its contribution to the weight.

For those columns in $N_0(a)$: if the bit in $y = 0$ of the affected even column is passive, then the bit remains active at a and becomes passive at b. This may reduce the weight by a factor 2β. If the bit in $y = 0$ of the affected even column is active, then the bit at b remains active and the bit at a is moved to another row, possibly reducing the weight by at most a factor of 2α. The maximum weight reduction is thus by a factor of $2\max(\alpha, \beta)$. This can happen at most $\#N_0(a)$ times and hence the weight can never become smaller than the weight of the reduced states minus $2 \cdot \max(\alpha, \beta) \cdot \#N_0(a)$. □

Estimating the lower bound

As for the case of patterns in the kernel, we define a function Γ to estimate the cost due to the addition of a new column assignment and understand if the addition is feasible without explicitly compute the cost function. Again, the function Γ should be a trade-off between efficiency and accuracy.

To be able to efficiently determine whether adding a new column assignment is worth it, we consider its type.

For unaffected odd columns, the definition of Γ is based on the following fact. If an unaffected odd column is added to an empty slice, then its active bit contributes 2α to w_0. If an affected even column is later added in the same column position, then the bit at a might be canceled but for sure another bit will be added in the same column and this will never decrease the original contribution 2α. Instead, the contribution to w_1 depends on the exact position of the bit at b and we will not consider it in the definition of Γ, since too expensive. Similarly, if the slice is not empty.

For these reasons, our choice for the function Γ is the following.

Definition 6.16. *Given a pattern a with cost γ and an unaffected odd column assignment $c \notin a$, the* cost estimation function *is defined as*

$$\Gamma_{UOC}(a, c) = \begin{cases} \gamma + 2\alpha & \text{if the slice } a[z(c)] \text{ is empty} \\ \gamma & \text{otherwise} \end{cases} \tag{6.3}$$

where $z(c)$ is the z-coordinate of column c.

So, if the cost of the pattern is already $T - 1$, no unaffected odd columns can be added to empty slices.

For affected even columns, the definition of Γ is based on the following fact. Each affected column has a total of 5 active bits, where each bit appears either at a or at b. Each bit can contribute at most 2 to the weight and this happens when all bits are in separate rows. Let $\tilde{a}$ be the pattern resulting from a when all 5 active bits are added to it and $\tilde{b}$ the pattern resulting from b when all 5 active bits are added to it. Then, $\mathrm{w}^{\mathrm{rev}}(\tilde{a}) + \mathrm{w}(\tilde{b})$ is the weight of the trail core

$\langle \tilde{a}, \tilde{b} \rangle$ with 5 active bits too many. Since removing each of them decreases the weight by at most 2, we can lower bound the real weight with $\mathrm{w}^{\mathrm{rev}}(\tilde{a}) + \mathrm{w}(\tilde{b}) - 2 \cdot 5$.

But we need to estimate the cost of the pattern and its descendants. So, our choice for the function Γ is the following.

Definition 6.17. *Given a pattern a with cost γ and an affected even column assignment $c \notin a$, the* cost estimation function *is defined as*

$$\Gamma_{AEC}(a, c) = \alpha \cdot \mathrm{w}^{\mathrm{rev}}(\bar{\tilde{a}}) + \beta \cdot \mathrm{w}(\bar{\tilde{b}}) - 2 \cdot 5 - 2 \cdot \max(\alpha, \beta) \cdot \#N_0(\tilde{a}) \tag{6.4}$$

where $\bar{\tilde{(\cdot)}}$ denotes the reduced state computed from $\tilde{(\cdot)}$.

6.2.2.4 Remark on z-canonicity

Verifying z-canonicity in the run-tree requires comparing lists of column assignments. Similar to the case in the kernel with orbitals, also in this case the first column assignment must be restricted to the first slice.

For state patterns in a free-orbital-tree, verifying z-canonicity is only required if the parity-bare state at its root exhibits symmetry. In that case it just requires comparing lists of orbitals. Note that this is a rare case, except for the orbital-tree with zero parity.

6.2.3 An alternative approach to generate trail cores outside the kernel

When traversing the orbital-tree and the free-orbital-tree we use the cost of a pattern as lower bound for the cost of its descendants. This is not the case for the run-tree, but we are able to define a lower bounding function that makes our search still efficient. The result is a quite simple definition of units at the cost of a non-trivial cost bounding function.

With a different factorization of parity-bare states and thus a different definition of units, it is possible to make the cost function monotonic and thus make the cost bounding function trivial. The side effect is a quite involved definition of units and order relation over these units.

In this section we describe this alternative approach, since it might be of interest for the reader.

6.2.3.1 From runs to run clusters

We plan on grouping the generation of difference patterns a by their parity $p(a)$.

Parities can be represented as unit-lists with runs as units and we can incrementally build them by adding and removing runs. While building parity patterns we contemporary build difference patterns with given parity, by adding the corresponding column assignments. The difference with the approach of Section 6.2.2 is that in this case column assignments of a run are added all at once. Instead, in Section 6.2.2 column assignments were incrementally added and the bound was updated after each single column assignment addition.

Definition 6.18. *For a given run r of length ℓ, the* run assignment *of r is the set of column assignments that consists of the column assignments of the (unaffected) odd columns of r and of the two (even) columns affected by r.*

When naively iterating over parities with runs as units, when adding a run, it cannot be excluded that the resulting patterns have smaller cost. In fact, the addition of the run might imply that either an affected even column becomes affected odd or an unaffected odd column becomes affected odd.

Therefore, we are going to group runs in clusters. The important benefit of using clusters is to be able to fix the value of active bits in odd and/or affected columns and thus make the cost function monotonic.

We now introduce the concepts of entanglement and run-cluster.

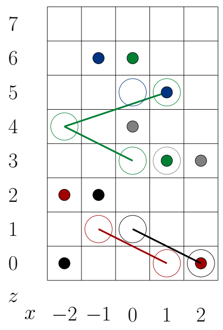
Definition 6.19. *Given two runs r and r', we say that they are* entangled *if at least one of the columns affected by r is an odd column of r' or vice versa.*

Adding a run to a parity p that is not entangled with any other run already in p is monotonic in the Hamming branch number $B_h(p)$. This follows from the fact that such a run only activates new columns, resulting in the addition of new active bits at a and at b. So, also the cost function is monotonic. The same does not hold when adding a run that is entangled with another run in p.

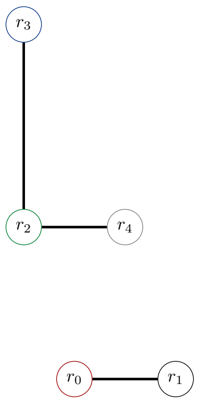
Definition 6.20. *The graph associated to a set of runs $R = \{r_1, \ldots, r_n\}$ is the graph formed by taking R as vertices and whose edges are determined by the entanglement relation. A set of runs is said to form a* run-cluster *if and only if its graph is connected.*

In other words, any run in a run-cluster is entangled with at least another run, but is not entangled with any other run outside the run-cluster.

Example 6.4. Consider the following parity pattern p.



It has five runs: $r_0 = (1, 0, 2)$, $r_1 = (2, 0, 2)$, $r_2 = (0, 3, 3)$, $r_3 = (1, 3, 1)$ and $r_4 = (0, 5, 1)$. Runs r_0 and r_1 are entangled, since run r_0 affects an odd column of r_1. Run r_2 is entangled with both r_3 and r_4. In fact, r_2 affects the odd column of r_3 and r_4 affects an odd column of r_2. Therefore, the graph associated to the set of runs of p is:



The graph has two connected components, each representing a run-cluster. Therefore, the parity p is composed by 2 run-clusters: one is $\{r_0, r_1\}$ and the other is $\{r_2, r_3, r_4\}$. △

By adding runs to the parity as disjoint run-clusters, before adding a new run-cluster we specify the column values consistent with run-clusters already present. This allows computing the cost due to these specified bits and use the cost of a pattern as lower bound on the cost of all its descendants. We can thus stop adding run-clusters as soon as the cost exceeds the budget. This calls for the following definition.

Definition 6.21. *A* run-cluster assignment *is a set of column assignments, one for each column specified by the runs composing the run-cluster.*

A run-cluster assignment can be seen as the sum of run-assignments, where affected odd column assignments are computed as the sum of an unaffected odd and an affected even column assignment. When working with run-clusters, there is no more need to split affected columns as the sum of one unaffected odd and one affected even column, because its nature is fixed and will not change with the addition of new run-cluster assignments.

Any parity-bare pattern a can be decomposed into a set of run-cluster assignments. To show that it is always possible to do so, we describe a procedure for performing the decomposition. First, compute $p(a)$ and compute its representation as a set of runs R. Then compute the graph associated to R. The run-clusters are the connected components in the graph associated to R. For each run-cluster, form the associated run-cluster assignment by taking the column value of each of its affected and unaffected odd columns.

The interesting property of this decomposition is that adding a run-cluster assignment to a can never decrease the cost. This monotonic behavior allows better lower bounds on the cost function. In particular, the cost of the descendants of a pattern can be lower bounded by the cost of the pattern itself.

All run-cluster assignments for a given run-cluster can be generated by assigning all 16 possible values to affected columns and a single active bit to unaffected odd columns, letting it run over the five possible positions.

Iterating on patterns can be cast with the formalism of Section 4.2, where run-cluster assignments are used as units.

It remains to specify an ordering relation among run-clusters and run-cluster assignments.

6.2.3.2 The runs inside a run cluster

We see the need to iterate at two levels:

- over run-cluster assignments in an outer loop;
- over the runs that make a run-cluster in an inner loop.

In particular, we wish to use the same principles as in Section 4.2, recursively inside a run cluster. The order relation among the runs inside a run cluster must satisfy following requirements:

- When removing the highest run in the run-cluster, the graph must remain connected.
- When removing the highest run in the highest run-cluster of a z-canonical pattern, it must remains z-canonical. This ensures that we can safely exclude non-z-canonical patterns also in the inner loop.

The first requirement excludes using a simple lexicographic order as the one defined in Definition 6.12.

We relate the order of runs in a run cluster to the choice of an arbitrary run $\bar{r} \in R$ of the run cluster, that we call the *anchor run*. How the anchor run is chosen will be explained later.

Definition 6.22. *The* rank *of a run $r \in R$ is its distance to $\bar{r}$, i.e.,* $\text{rank}_{\bar{r}}(\bar{r}) = 0$, $\text{rank}_{\bar{r}}(r) = 1$ *for all runs $r \neq \bar{r}$ that are entangled to $\bar{r}$, and* $\text{rank}_{\bar{r}}(r) = i$ *for all runs r that are entangled to a run of rank $i - 1$ but not lower. And so on.*

Definition 6.23. *We call the* relative coordinates *of a run r the sequence of numbers* $(\text{rank}_{\bar{r}}(r), x, z - \bar{z}, \ell)$*, with $\bar{z}$ the z coordinate of the starting point of $\bar{r}$ and x, z, ℓ the coordinates of the starting point and the length of the run.*

We are now ready to define the order among runs in a run cluster.

Definition 6.24. *Within a run cluster, the order $\prec_{\bar{r}}$ of the runs is the lexicographic order on* $[\text{rank}_{\bar{r}}(r), x, z - \bar{z}, \ell]$.

Given a run-cluster R, we represent it as an offset $\bar{z}$ and an ordered list of relative coordinates of its runs. This representation is called the *run-list* of R.

There are thus n possible representations, where n is the number of runs in R. The run $\bar{r}$ to which the other runs are relative can be identified by the only run whose rank is 0 and then translated by $\bar{z}$. The ordering of runs in a run-list is invariant with respect to translation along the z-axis thanks to the use of relative z-coordinates.

The choice of $\bar{r}$ is so far arbitrary and we would like to have a unique representation of a run cluster R. To this purpose, we define the canonical representation of a run-cluster R as follows.

Definition 6.25. *Let R be a run-cluster. The* canonical representation *of R is the minimal among its n run-lists. Where the minimal is given by the lexicographic order of the runs with their relative coordinates defined earlier.*

Definition 6.26. *Given a run-cluster R, the 0-ranked run of the canonical representation of R is called the* anchor run *of R and is denoted by $r^*(R)$. The* offset *of R is the z-coordinate of the starting point of $r^*(R)$ and is denoted by z^*.*

Notice that the ordering is only on the run-list with relative coordinates, so clearly, the choice of the anchor run of a run cluster is not affected by translation along the z axis.

Hence, the *canonical representation* of a run cluster R is (z^*, R_0) with its offset z^* and its minimal run-list R_0.

The ordering among run-lists of Definition 6.24 satisfies our requirements:

1. The use of the rank as primary ordering guarantees that if R is a run cluster, then so is parent(R).
2. When removing the highest unit from a canonical representation of a run cluster, it remains canonical. This is a consequence of Lemma 4.3, as at run-list level a canonical run-list is also z-canonical.

6.2.3.3 Difference patterns as run-cluster assignments

We now come back to the outer loop, where difference patterns a are represented as a set of run-cluster assignments. A run-cluster assignment is represented by its run cluster R and the set of column assignments C, i.e., the set of active bits in the odd and/or affected columns of R. We represent a run-cluster assignment as (z^*, R_0, C_0), where

- (z^*, R_0) is the canonical representation of R;
- C_0 is a set of column assignments with $C = \tau_{z^*}(C_0)$.

This representation was chosen to make it easy a translation along the z axis, i.e., $\tau_t(z^*, R_0, C_0) = (z - t, R_0, C_0)$, so that R_0 and C_0 remain unchanged. The order relation among run-cluster assignments is the lexicographic order on $[z^*, R_0, C_0]$, where the order on R_0 is in Section 6.2.3.2 and the order on column assignments is the lexicographic order of active bits given in Definition 6.1. Since the order is primarily determined by z^*, there is often no need to compare run clusters.

Being able to say that a pattern is z-canonical requires defining an order among the patterns. We propose to simply use the lexicographic order over its run-cluster assignments, to be able to use Lemma 4.3. This automatically guarantees that the parent of a z-canonical pattern, obtained by removing the highest run-cluster assignment, is still z-canonical.

This principle also extends at the level of runs in a run cluster. As discussed above, removing the highest unit from a canonical representation of a run-cluster, keeps it canonical. Also, the order among runs is preserved by a translation along z. This means that removing the highest run of the highest run-cluster assignments (z^*, R_0, C_0) preserves the order of R_0 among translated instances, in a similar way as in Lemma 4.3. Consequently, when iterating on runs, one can discard patterns that are not z-canonical and all its descendants.

6.2.3.4 Lower-bounding the cost

As said above, the cost functions we are considering are all monotonic in the addition of a new run-cluster assignment, since it will only activate new columns and thus add new bits both at a and at b. Therefore, we can use the cost of a pattern as lower bound on the cost of all its descendants.

To prune the tree as soon as possible, we wish to define a function Γ to estimate the cost of trail cores resulting from the addition of a new run-cluster.

To this end we analyze the contribution of affected and unaffected odd columns similarly to what has been done in Definition 6.17. Each affected column has 5 active bits, where each bit appears either at a or at b. Each bit can contribute at most 2 to the weight and this happens when all bits are in separate rows. Suppose that a newly added run cluster defines n new affected columns. Let w_1 be the minimum reverse weight of a when all $5n$ active bits are added to a and w_2 be the weight of b when all $5n$ active bits are added to b. Then, $\mathrm{w}_1 + \mathrm{w}_2$ is the weight of the new trail core with $5n$ active bits too many. Since removing each of them decreases the weight by at most 2, we can lower bound the weight with $\mathrm{w}_1 + \mathrm{w}_2 - 10n$. An unaffected column that appears as the only active column in a slice at a contributes 2 to the weight. Let m be the number of such columns. Then our choice for the function Γ is the following.

Definition 6.27. *Given a pattern a with cost γ and a run-cluster R, the* cost estimation function *is defined as*

$$\Gamma_R(a, R) = \alpha \mathrm{w}_1 + \beta \mathrm{w}_2 - 10 \max(\alpha, \beta) n + 2m. \tag{6.5}$$

6.3 Extension of trails in Keccak-f

In this section we present techniques to make extension of trails in Keccak-f more efficient and so, to increase the size of the trail space we can scan.

Indeed, as the target weight increases, the amount of 2-round trail cores to extend increases as well. Moreover, for a given pattern, the number of compatible patterns to investigate through χ or χ^{-1} grows exponentially with the weight. Therefore, we need to improve extension and make it more efficient.

The general principles are similar to those for two-round trail core generation. In particular, we generate compatible patterns by applying the tree traversal method where the tree structure is defined by the generators of a vector space. We make use of the linearity of λ, grouping of state patterns by their parity and we use monotonic weight bounding functions.

6.3.1 Extension as a tree traversal

In forward and backward extension, we need to compute difference patterns that are compatible with a given pattern through χ and χ^{-1} respectively. As explained in Section 6.1.2 we restrict this search to patterns in the kernel or outside the kernel depending on the specific case. However, in all cases these compatible patterns are elements of an affine space that we denote by U. This affine space U can be described as $U = V + e$ with V a vector space and e a difference pattern that forms the offset. The vector space V is generated by a basis $B = \{v_i\}_{i=1,\ldots,n}$ for some integer n, where v_i are difference patterns.

Every element $u \in U$ can be expressed as the bitwise sum of the offset e and a subset of the basis vectors $A \subseteq B$. In particular, $u = e + \sum_{v_i \in A} v_i$. This allows us to arrange the elements of U in a tree in the following way.

First of all, we establish an order relation over the vectors of B. It follows that the set A becomes an ordered list of vectors, being a subset of an ordered set. Now, the parent of an element a is obtained by removing the largest element of A. In other words, the parent of a difference pattern is obtained by *removing its last basis vector*. Or alternatively, the children of a difference pattern a are the patterns obtained by adding a basis vector after the highest vector of A. So the units of the tree are basis vectors and the root of the tree is the pattern consisting only of the offset e.

The cost of nodes in this case is either $\mathrm{w}(\lambda(a))$ or $\mathrm{w}^{\mathrm{rev}}(\lambda^{-1}(b))$ depending on whether we are performing forward or backward extension, respectively. To make the tree traversal efficient, we define cost bounding functions to prune the tree as soon as possible. The definition of the lower bounding function depends on whether we are extending in the kernel or outside the kernel, in the forward or backward direction.

6.3.2 Forward extension

Given a trail core Q ending in pattern b, forward extension up to some weight T by one round can be seen as scanning the space of all state patterns a that are χ-compatible with b and checking whether the weight of the extended trail core $\langle Q, a, \lambda(a)\rangle$ is below T.

Since χ has algebraic degree 2, the set of values a that are compatible with b through χ forms an affine space that here we denote as $U(b)$ (or just U if b is clear from the context). This affine space U can be described as $U = V + e$ with V a vector space of dimension $\mathrm{w}(b)$ and e a state pattern that forms the offset. If b has large weight, brute-force scanning all $a \in U$ becomes prohibitively expensive and therefore we seek for more efficient methods.

First of all, recall from Section 6.1.2 that we treat extension with a inside the kernel or a outside the kernel separately. The former can be made efficient in a particularly simple way, the latter requires some more sophistication.

6.3.2.1 Forward extension inside the kernel

The kernel is a vector space and we will denote it by K. The set of state patterns that are χ-compatible with b and are in the kernel is given by $(V + e) \cap K$. The intersection is either empty, or it is an affine space $U' = V' + e'$, with V' the vector space $V' = V \cap K$ and e' an offset in the kernel.

We use the structure of χ to construct an efficient algorithm that, given b, returns an offset $e' \in U(b) \cap K$ or a message stating that $U(b) \cap K$ is empty.

This algorithm operates slice by slice, where it makes use of the fact that χ operates on individual rows and K is computed over individual columns.

In particular, an active row $b[y, z]$ defines an affine space covering all row-patterns that are χ-compatible with it. We denote this space by $U[y, z] = V[y, z] + e[y, z]$, where the offset $e[y, z]$ and bases for $V[y, z]$ are given in Table 5.2.

The active rows in a slice $b[z]$ thus define an affine space $U[z]$ covering all slice-patterns that are χ-compatible with $b[z]$. To generate its offset, we take an empty slice and add the offsets of all active rows to it. We denote this action by

$$e[z] = \sum_y e[y, z]. \tag{6.6}$$

Then we take a basis vector of an active row and add it to an empty slice. The obtained slice is added to the set of basis vectors for $U[z]$. We repeat the process for all basis vectors of all active rows. We denote this action by

$$V[z] = \bigcup_y V[y, z]. \tag{6.7}$$

The affine space U is defined by the set of basis vectors of all active slices and an offset that is the sum of offsets of all active slices. Using notation introduced above, we say

$$e = \sum_z e[z] \tag{6.8}$$

and

$$V = \bigcup_z V[z]. \tag{6.9}$$

For e to be in the kernel, all $e[z]$ must be in the kernel so this can be handled slice by slice. For all z we can compute parities of offset and of basis vectors. These in turn generate the space of parity patterns that are χ-compatible with b. We denote this space as $U_p = e_p + V_p$.

If $e_p[z] \notin \text{Span}(V_p[z])$, then all elements in $U[z]$ will have at least one odd column and so will not be in the kernel. This is the case for instance when $e[z]$ has a single active row or has two active bits in different columns. Therefore, slices with single active row and slices with two non-orbital active bits allow eliminating in a very efficient way the vast majority of candidates b for extension. A non-trivial example is given in Figure 6.7.

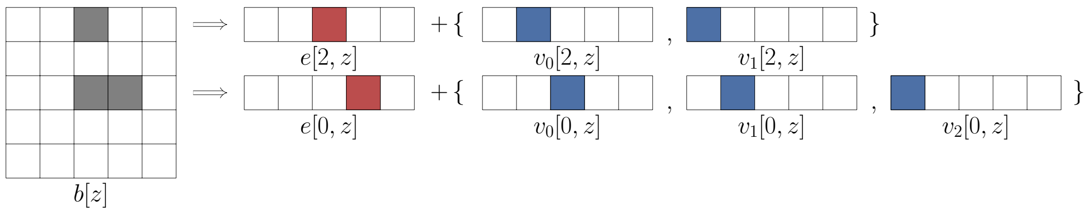


Fig. 6.7: *Example of pattern that cannot be forward extended in the kernel. For each active row of slice $b[z]$, offset and basis vectors for the space of χ-compatible rows are depicted in the format "offset + {basis vectors}". Basis for row $y = 2$ has no basis vector with active bit in $x = 1$ to complement the offset at row $y = 0$. Thus, since no offset in the kernel can be constructed, the pattern cannot be extended in the kernel.*

If $e_p[z] \in \text{Span}(V_p[z])$, then we generate $e'[z]$ by adding to $e[z]$ a number of basis vectors such that the sum of their parities is equal to $e_p[z]$. A non-trivial example is given in Figure 6.8.

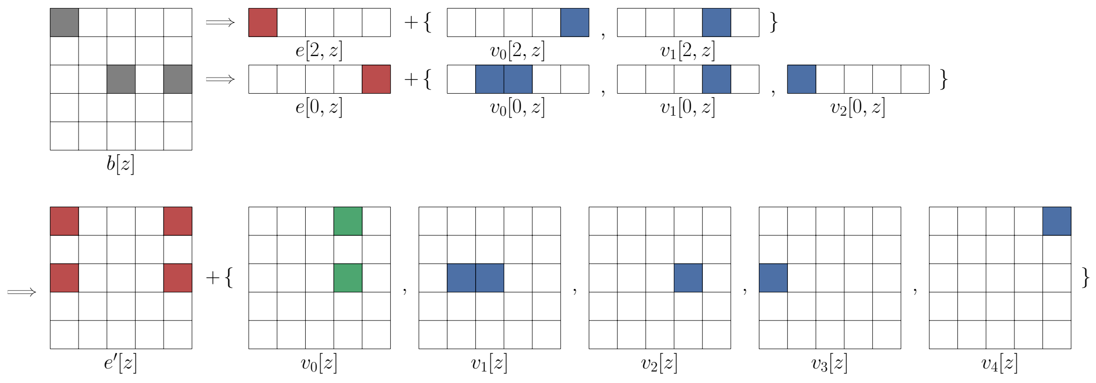


Fig. 6.8: *Example of pattern that can be forward extended in the kernel. Above, offset and basis vectors for the space of χ-compatible rows are depicted. Below, offset and basis vectors for the space of χ-compatible slices are depicted, with separation between basis vectors in the kernel (in green) and outside the kernel (in blue). The offset in the kernel can be built as $e'[z] = e[0,z] + e[2,z] + v_0[2,z] + v_2[0,z]$, thus the pattern can be forward extended in the kernel. Vector $v_0[z] = v_1[0,z] + v_2[2,z]$ forms the basis for $V_K[z]$, while $v_1[z], v_2[z], v_3[z], v_4[z]$ form a basis for $V_N[z]$.*

For state patterns b having an offset e' in the kernel, we construct the basis of $V(b)$. Then we do a basis transformation resulting in basis vectors in the kernel and outside the kernel.

This basis transformation simply consists of taking linear combinations of basis vectors yielding vectors containing one or more orbitals. We can thus partition the basis vectors of V in a set of in-kernel basis vectors generating $V_K = V \cap K$ and a set of outside-kernel basis vectors orthogonal generating V_N with $V = V_K + V_N$. This can be done again slice by slice. Specifically, for each z, let $w = \dim(\text{Span}(V[z]))$ and $d = \dim(\text{Span}(V_p[z]))$. We add to $V_N[z]$ d vectors of $V[z]$ whose corresponding parities are linearly independent. Then, for each of the $w - d$ vectors $v \in V[z] \setminus V_N[z]$ we add to it a number of basis vectors of $V_N[z]$ to make it in the kernel. We add the obtained vectors to $V_K[z]$. An example is given in Figure 6.8.

The procedure described above to generate e in the kernel and a basis for V, with separation between V_K and V_N, is reported in Algorithm 2.

Algorithm 2 Offset and basis creation with separation between in-kernel and outside-kernel vectors

```
1: Initialize $\text{flag}_{\text{in\_kernel\_offset}}$ = **true**
2: **for** Slices $[z = 0]$ to $[z = w - 1]$ **do**
3:     Initialize slice offset $e[z] = 0$ and slice basis $V[z] = \emptyset$
4:     **for** Rows $[y = 0]$ to $[y = 4]$ **do**
5:         $e[z] \leftarrow e[z] + e[y, z]$
6:         $V[z] \leftarrow V[z] \cup V[y, z]$
7:     **end for**
8:     Compute $e_p[z]$ and $V_p[z]$
9:     **if** $e_p[z] \in \text{Span}(V_p[z])$ **then**
10:        $e[z] \leftarrow e[z] + \sum_i v_i$ such that $e[z] \in K$        ▷ where $v_i \in V[z]$
11:    **else**
12:        Set $\text{flag}_{\text{in\_kernel\_offset}}$ = **false**
13:    **end if**
14:    Initialize $V_K[k] = \emptyset$ and $V_N[z] = \emptyset$
15:    Add to $V_N[z]$ $d$ vectors of $V[z]$ with linearly independent parities
16:    **for** All $w - d$ vectors $v \in V[z] \setminus V_N[z]$ **do**
17:        $v \leftarrow v + \sum_i v_i$ such that $v \in K$        ▷ where $v_i \in V_N[z]$
18:    **end for**
19: **end for**
20: $e \leftarrow \sum_z e[z]$, $V_N \leftarrow \bigcup_z V_N[z]$, $V_K \leftarrow \bigcup_z V_K[z]$ =0
```

Building all elements of $U(b) \cap K$ then simply corresponds to scanning the space $e' + V_K$. This is done as a tree traversal, where the root of the tree is the pattern e' and units are basis vectors of V_K.

In Table 6.1 we report some experimental results on the extension in the kernel of 2-round trail cores in $|N|$ with weight below 33. We give the number of trail cores with non-empty intersection with the kernel among all the trail cores to extend and the dimension of the basis V of the affine space and of its intersection V_K with the kernel.

Summarizing, thanks to the fact that most trail cores cannot be extended in the kernel and that for the remaining ones V_K has low dimension, the cost of forward extension in the kernel is dramatically reduced.

6.3.2.2 Forward extension outside the kernel

To perform forward extension outside the kernel we proceed as the case in the kernel to build the basis and partition the basis vectors of V in a set of in-kernel basis vectors generating $V_K = V \cap K$ and a set of outside-kernel basis vectors orthogonal generating V_N with $V = V_K + V_N$.

b	# trail cores to extend	# trail cores s.t $V_K \neq \emptyset$	$\dim V$ avg	$\dim V$ stdev	$\dim V_K$ avg	$\dim V_K$ stdev
200	$37 \cdot 10^6$	$9 \cdot 10^3$	14.66	2.95	4.72	1.73
400	$30 \cdot 10^6$	$2 \cdot 10^2$	12.19	2.23	4.46	1.43
800	$28 \cdot 10^6$	6	9.75	2.24	3.70	0.90
1600	$38 \cdot 10^6$	0	0	0	0	0

Table 6.1: *Results on extension in the kernel. Column two gives the total number of trail cores in $|N|$ to extend in the forward direction in the kernel. Among them, the number of trail cores for which V_K is not empty is reported in the third column. For such trail cores the average dimension of V and V_K is reported in the fourth and fifth column respectively.*

Building all patterns outside the kernel corresponds to scanning the space $e' + V_K + V_N$. This is done again as a tree traversal, where the root of the tree is the pattern e' and units are basis vectors of V_K and V_N.

The cost of a pattern a is the weight of the pattern $\lambda(a)$. In this case, adding base vectors is not monotonous, since adding a basis vector may decrease the weight of $\lambda(a)$. However, the potential weight loss can be bound and this potential weight loss depends on the type of basis vector and its relation with a. Moreover, if the weight of a is high, the worst-case total weight loss (due to the contribution of all base vectors that can still be added to a) may still give a weight above the limit and the full subtree of a and its descendants can be discarded.

Note that basis vectors outside the kernel either only have a single active bit, or consist of two neighboring active bits in the same row (see Table 5.2). The former implies 2 affected columns, the latter 4 affected columns.

As for the worst-case weight loss we can say the following:

- A basis vector in the kernel consisting of n orbitals has a weight loss of at most $4n$. An orbital in a column that is unaffected and that does not overlap with a has no weight loss, an orbital that overlaps with a has a weight loss of at most 2.
- For a basis vector outside the kernel we consider the number of affected columns overlapping with affected columns of a. If this is n, the weight loss is at most $10n+2q$ with q the Hamming weight of the basis vector.

Clearly, in-kernel basis vectors and basis vectors with affected columns not overlapping with those of a have the smallest potential weight loss. We take this into account when deciding the ordering of the basis vectors, putting them last. This leads to a great improvement in forward extension outside the kernel.

6.3.3 Backward extension

Given a trail core Q starting with state a, backward extension up to some weight T by one round is a scan over all state patterns b such that a is χ-compatible with b and checking whether the weight of the extended trail is below T.

Unlike for forward extension, the set of patterns b that are χ^{-1}-compatible with a do not form an affine space. However, we can define and compute an affine space that contains all such patterns.

As in forward extension, the offset and basis vectors of this space can be generated at row level. For the offset we make use of the property of χ that for an active row in a with a single active bit, the corresponding bit will be active in all states b that are χ^{-1}-compatible with a.

Definition 6.28. *Given a pattern a, we call the* χ^{-1}-envelope space *of a the space built as follows. For each active row of a with a single active bit in position (x, y, z), take as offset for this row in b the row with a single active bit in position (x, y, z) and as basis vectors the four basis vectors each with a single active bit of coordinates (x', y, z), with $x' \neq x$. For each active* *row of a with more than a single active bit, we consider the 5-dimensional vector space where each basis vector has a single active bit.*

The resulting χ^{-1}-envelope space of a has an offset with Hamming weight equal to the number of single-bit active rows in a and dimension equal to $5n - \mathrm{HW}(\text{offset})$ with n the number of active rows in a. An example is given in Figure 6.9[a].

6.3.3.1 Backward extension in the kernel

As kernel membership is defined at the input and output of θ, we map the χ^{-1}-envelope space of a through π^{-1} and ρ^{-1} to the output of θ.

Definition 6.29. *Given a pattern a, the* envelope *space of a is the image of the χ^{-1}-envelope space of a through π^{-1} and ρ^{-1} and it is denoted as $U = V + e$.*

Since ρ and π are transpositions, the Hamming weight of the offset e remains the same and the number of active bits in each basis vector remains the same, too. We use techniques similar to those of forward extension to determine whether $U \cap K$ is empty or not. In particular, if e has a slice with odd columns and there are no basis vectors with active bits in that column, then all elements of U will have at least an odd column, disqualifying a for backward extension inside the kernel. An example is given in Figure 6.9[b].

In Table 6.2 we report some experimental results on the backward extension in the kernel of 2-round trail cores in $|N|$ with weight below 33. We report the number of trail cores whose envelope space has non-empty intersection with the kernel among all the trail cores to extend. For these trails we report the dimension of the basis of the envelope space and of its intersection with the kernel. Even if the envelope space is much larger than the space of compatible patterns, the space we concretely investigate is the intersection with the kernel that is significantly smaller, making the search effort feasible.

b	# trail cores to extend	# trail cores s.t $V_K \neq \emptyset$	dim V		dim V_K	
			avg	stdev	avg	stdev
200	$37 \cdot 10^6$	$4 \cdot 10^6$	33.29	5.98	8.37	3.37
400	$30 \cdot 10^6$	$5 \cdot 10^5$	33.33	6.07	4.16	2.10
800	$28 \cdot 10^6$	$2 \cdot 10^5$	32.67	6.22	2.61	1.58
1600	$38 \cdot 10^6$	$2 \cdot 10^5$	32.08	6.25	1.35	1.12

Table 6.2: *Results on backward extension in the kernel. Column two gives the total number of trail cores in $|N|$ to extend in the backward direction in the kernel. Among them, the number of trail cores for which the intersection between the envelope space and the kernel is not empty is reported in the third column. For such trail cores the average dimension of V and V_K is reported in the fourth and fifth column respectively.*

6.3.3.2 Backward extension outside the kernel

We limit our search of values by restricting the set of parity patterns for $\lambda^{-1}(b)$, that we will denote by a', to investigate. To this end we iteratively construct them, limiting the generation by lower bounding the minimum reverse weight of a'. We start from the envelope space U at the output of θ as in Section 6.3.3.1. The offset and basis vectors of U define offset and basis vectors for the space of parity patterns at the output of θ. We denote it by $U_p = e_p + V_p$.

Let p be the parity after θ and p' the parity before. Then $p[x,z] = p'[x,z] + p'[x-1,z] + p'[x+1,z-1]$ or, equivalently,

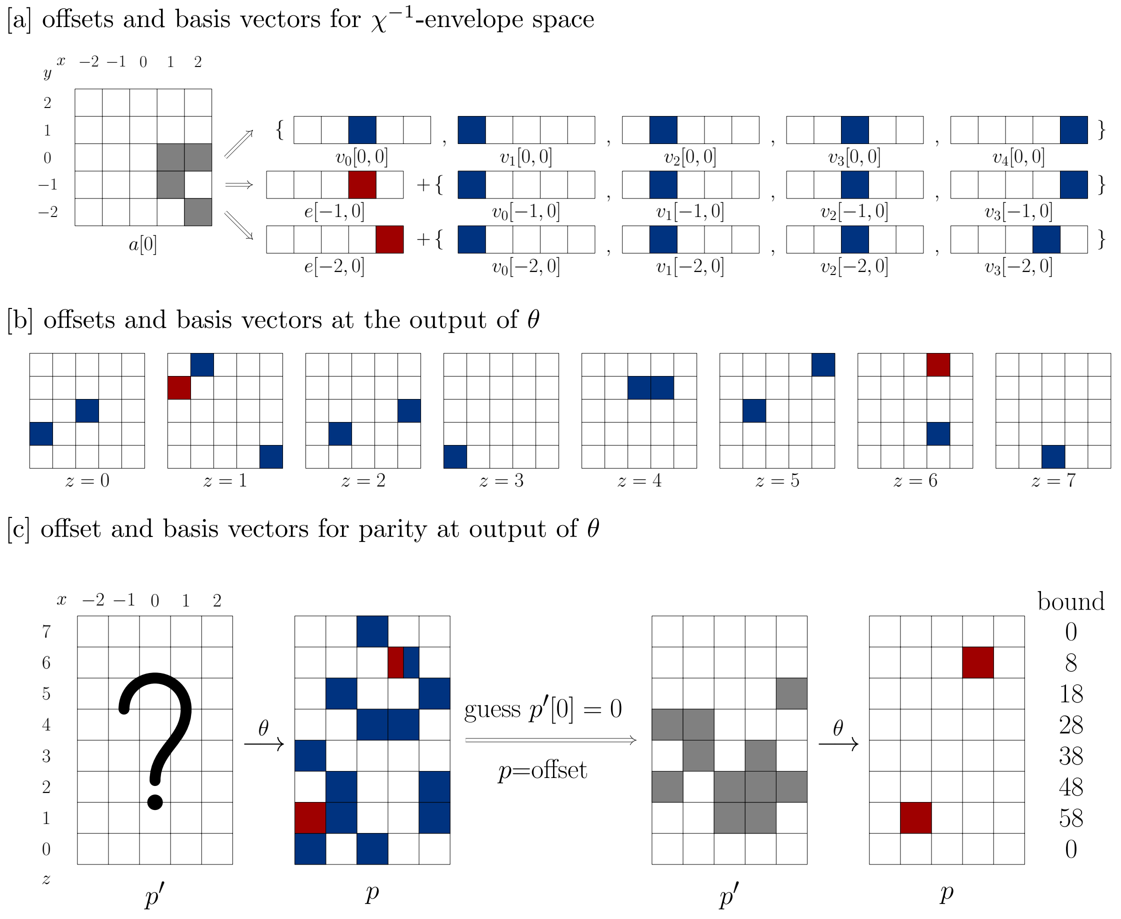

[a] offsets and basis vectors for χ^{-1}-envelope space
x −2 −1 0 1 2
y 2 1 0 −1 −2
$a[0]$
$v_0[0,0]$ $v_1[0,0]$ $v_2[0,0]$ $v_3[0,0]$ $v_4[0,0]$
$e[-1,0]$ $v_0[-1,0]$ $v_1[-1,0]$ $v_2[-1,0]$ $v_3[-1,0]$
$e[-2,0]$ $v_0[-2,0]$ $v_1[-2,0]$ $v_2[-2,0]$ $v_3[-2,0]$
[b] offsets and basis vectors at the output of θ
$z=0$ $z=1$ $z=2$ $z=3$ $z=4$ $z=5$ $z=6$ $z=7$
[c] offset and basis vectors for parity at output of θ
x −2 −1 0 1 2
z 7 6 5 4 3 2 1 0
?
θ
guess $p'[0]=0$
p=offset
θ
p' p p' p
bound 0 8 18 28 38 48 58 0

Fig. 6.9: *Example on backward extension. In (a) the active rows of a define offset and basis vectors for the χ^{-1}-envelope space. In (b) offset and basis vectors are mapped through $\pi^{-1} \cdot \rho^{-1}$. For the sake of compactness, offset and basis vectors are grouped by slices. This representation allows immediately noticing that there are no basis vectors in slice $z = 1$ with an active bit in column $x = -2$ to complement the offset bit in that column. It follows that a cannot be backward extended in the kernel. In (c) on the left, we derive offset and basis vectors for the space of parity patterns p after θ. In (c) on the right, parity patterns p' and p are incrementally generated and the bound is incrementally computed in the case guess on $p'[0]$ is 0 and p is the offset. The computed value of $p'[1]$ will imply value 11010 for $p'[0]$, which is inconsistent with the guess.*

$$p'[x+1, z-1] = p[x, z] + p'[x, z] + p'[x-1, z] \,. \tag{6.10}$$

Equation 6.10 allows computing the parity in row $z-1$ of p' from the parity in row z of p and p'. This can be done recursively. So, making an assumption for the value of the parity in some slice of p' allows computing p' completely, for a given p. There are 32 possible assumptions and due to the injectivity of θ, only one of them can be correct.

The knowledge of $p[z]$ and $p'[z]$ allows lower bounding the contribution of slice $a'[z]$ to the minimum reverse weight of a'. Indeed, given the positions of the affected columns determined by $p[z] + p'[z]$ and the offset and basis of U, we can determine a lower bound for the number of active rows in $a'[z]$ and each active row contributes at least 2 to the minimum reverse weight of a'. We build the parity pattern p' incrementally slice by slice using Equation 6.10 and keep track of the resulting bound on the minimum reverse weight. For each bit of a basis vector encountered in $p[z]$ we have to consider two possibilities, one with the basis vector present and one with the basis vector absent. So, this results again in a tree search where children have a higher cost (lower bound on the minimum reverse weight) than its parent. Hence, we truncate as soon as our partially reconstructed parity exceeds the limit weight. When the scan reaches the value of z where it started and the initial assumption turns out to be correct, it has constructed a valid parity pattern. In that case, we generate all possible states in U that have p as parity and that can be generated by the basis of U. An example on the incremental construction of p and p' (with bound on the minimum reverse weight of a' at each step) is given in Figure 6.9[c] and the procedure is translated in Algorithm 3 and Algorithm 4.

Algorithm 3 Generation of valid parity patterns

```
1: Initialize $p = e_p$, $p' = 0$
2: Initialize $L = \emptyset$
3: for 32 possible row values $v$ do
4:     $p'[0] \leftarrow v$
5:     Call addRow($w-1, p, p', L$)                    ▷ see Algorithm 4
6: end for
```

Algorithm 4 addRow

```
1: if bound on minimum reverse weight is above limit then
2:     return
3: end if
4: if $z = w$ then
5:     if $(p, p')$ is consistent then
6:         $L \leftarrow L \cup \{p\}$
7:     end if
8: else
9:     for all $p[z] \in e_p[z] + V_p[z]$ do
10:        $p' \leftarrow p' + p'[z-1]$ where $p'[z-1]$ is computed as in Equation 6.10
11:        Call addRow($z-1, p, p', L$)
12:    end for
13: end if
```

6.4 Experimental results

The techniques described in previous sections have been implemented in C++ and are available as part of the KeccakTools project [BDPV15]. The generic tree-traversal approach is available in class `Tree`, while class `Keccak-fTree` implements the approach for Keccak-f, defining units, lower bounding functions and all the other needed objects. The optimization for extension have been included in class `Keccak-fTrailExtensionBasedOnParity`.

To check correctness of our algorithm, we have tested it against [BDPV15] by generating all 3-round trail cores up to weight 36, i.e. by covering the same space covered in [DV12]. We have found the same number of trails using both codes and thus the same bounds. This result gave us confidence on the correctness of our implementation.

A comparison between the two approaches will be discussed in Section 6.6.2. We will show how the new techniques presented in this thesis allow to scan the space of trails more efficiently, thus allowing to push the target weight to higher values. In fact, we have used our code to cover the space of all 3-round trail cores up to weight 45, for the widths of Keccak-f between 200 and 1600. This allowed us to cover the space of all 6-round trail cores up to weight $T_6 = 91$.

2-round trail cores	$b = 200$	$b = 400$	$b = 800$	$b = 1600$
$\|K\|\, \mathrm{w}^{\mathrm{rev}}(a) \le 11$	$1.0 \cdot 10^7$	$2.1 \cdot 10^7$	$4.3 \cdot 10^7$	$8.6 \cdot 10^7$
$\|K\|\, \mathrm{w}(b) \le 11$	$3.0 \cdot 10^5$	$6.3 \cdot 10^5$	$1.3 \cdot 10^6$	$2.6 \cdot 10^6$
$\|N\|\, \mathrm{w}^{\mathrm{rev}}(a) + \mathrm{w}(b) \le 33$	$3.7 \cdot 10^7$	$3.0 \cdot 10^7$	$2.8 \cdot 10^7$	$3.8 \cdot 10^7$
$\|N\|\, \mathrm{w}^{\mathrm{rev}}(a) + 2\mathrm{w}(b) \le 45$	$3.9 \cdot 10^6$	$3.6 \cdot 10^6$	$4.8 \cdot 10^6$	$8.2 \cdot 10^6$
$\|N\|\, 2\mathrm{w}^{\mathrm{rev}}(a) + \mathrm{w}(b) \le 44$	$7.6 \cdot 10^6$	$5.4 \cdot 10^6$	$6.8 \cdot 10^6$	$1.1 \cdot 10^7$
$\|K\|\, \mathrm{w}^{\mathrm{rev}}(a) + 2\mathrm{w}(b) \le 45$	$3.0 \cdot 10^7$	$9.2 \cdot 10^7$	$2.6 \cdot 10^8$	$8.4 \cdot 10^8$
$\|K\|\, 2\mathrm{w}^{\mathrm{rev}}(a) + \mathrm{w}(b) \le 44$	$8.2 \cdot 10^7$	$3.1 \cdot 10^8$	$1.2 \cdot 10^9$	$1.2 \cdot 10^9$
$\|K\|$ with $\chi(b)$ in $\|K\|$	$3.6 \cdot 10^4$	$3.1 \cdot 10^3$	$5.2 \cdot 10^2$	$1.3 \cdot 10^2$

Table 6.3: *Number of 2-round trail cores generated for each parity profile. Last line refers to the chain-based method of [DV12].*

6.4.1 2-round trail cores

To generate 2-round trail cores, we apply Lemma 6.2 and Lemma 6.3 setting $T_1 = 11$. This implies the generation of 2-round trail cores in the kernel with either $\mathrm{w}^{\mathrm{rev}}(a)$ or $\mathrm{w}(b)$ smaller than 11 and the generation of all outside-kernel trail cores with $\mathrm{w}^{\mathrm{rev}}(a) + \mathrm{w}(b) \le 33$. Then, we generated all 2-round trail cores outside the kernel with either $2\mathrm{w}^{\mathrm{rev}}(a) + \mathrm{w}(b) \le 44$ or $\mathrm{w}^{\mathrm{rev}}(a) + 2\mathrm{w}(b) \le 45$ as stated in Lemma 6.4. Finally, to cover the case $|K|K|$ we applied both the chain-based method and the orbital-based method to compare them. So, we first generated all 2-round trail cores in the kernel with either $2\mathrm{w}^{\mathrm{rev}}(a) + \mathrm{w}(b) \le 44$ or $\mathrm{w}^{\mathrm{rev}}(a) + 2\mathrm{w}(b) \le 45$ as stated in Lemma 6.5. Then we generated all 2-round trail cores in the kernel for which there exists $\chi(b)$ in the kernel, by using the method of [DV12] already available in the KECCAKTOOLS.

In Table 6.3 we report the total number of 2-round trail cores generated for each scenario, while execution times are reported in Section 6.5.

More details on the distribution of trail weights are given in Figure 6.10 to Figure 6.17, where we depict the number of 2-round trail cores per weight and per parity profile, to show the trend among the different variants of KECCAK-f.

It is worth noticing that, while the number of trails globally increases with the weight, the local increase in not monotonic. In particular, the number of trails drops in correspondence of odd values. This can be easily explained by the fact that odd weights imply having a number of rows with more than one active bit. This imposes restrictions on the position that such bits can assume. It follows that the number of difference patterns satisfying such restrictions is relatively small.

A significant example is given by Figure 6.10, where there are no trail cores with $\mathrm{w}^{\mathrm{rev}}(a) = 5$. In fact, weight 5 implies one active row with a single bit (which contributes 2) and one active row with two active bits (which contributes 3), for a total of 3 active bits. Since we are considering trail cores in the kernel, such condition is never satisfied. This explains the absence of such trail cores.

Another trend we can notice is that in Figure 6.12 and Figure 6.14 there is a monotonic increase for KECCAK-$f[200]$, but there is not for larger variants of KECCAK-f. Also in this case, the number of trails drops for odd weights. This can be explained by the fact that odd weights imply rows with more than a single active bit either before or after λ. In both cases, such bits are spread across the difference pattern when passing through λ or λ^{-1}, respectively. For smaller variants of KECCAK-f, it is more likely that such bits will go in already active rows, limiting the weight increase. On the contrary, for larger variants this is less likely to happen and the weight of the trail core grows beyond the limit more easily. Therefore, the number of trails below the limit is much smaller.

In Figure 6.18 to Figure 6.25 we report the number of 2-round trail cores below a given weight T per each parity profile for increasing values of T. These are just the cumulative versions of the graphs in Figure 6.10 to Figure 6.17, but we depict them to show a trend that did not appear in previous graphs.

It is evident that as the width of Keccak-f increases, the number of 2-round trail cores in the kernel doubles. This because a higher number of possible positions for orbitals are allowed for larger widths.

For the case outside the kernel, the behavior is different. Indeed, up to a certain value of T (which is 29 for the case $w_0 + w_1 \leq T$, 37 for the case $2w_0 + w_1 \leq T$ and 39 for the case $w_0 + 2w_1 \leq T$), the number of cores decreases as the width increases. This can be explained again by the action of ρ, which moves active bits along the lanes. Indeed, for smaller variants it is more likely that active bits after θ are grouped into the same rows, making the weight increasing more slowly. On the contrary, for larger widths these active bits are spread within the state resulting in more active rows and thus a larger weight. When the target weight is pushed beyond this threshold value, the trend changes. The effect of ρ is hidden by the increasing number of active columns that a pattern may have, or equivalently by the number of possible runs in a parity.

6.4.2 3-round trail cores

For all 2-round trail cores generated, we performed extension in the kernel or outside the kernel, in the backward or forward direction. The number of 3-round trail cores obtained is reported in in Table 6.4 for each parity profile.

For each width, the majority of cores have profile $|K|K|$ since θ acts as the identity in each round and thus the number of active bits is kept small. On the contrary, the number of cores with profile $|N|N|$ is very small, until it reaches 0 for the two largest widths. Indeed, for a pattern outside the kernel, the number of active bits increases quickly through the rounds due to the action of θ and thus also the weight. The same principle explains the low number of trail cores with profile $|N|K|$ and $|K|N|$.

3-round trail cores	$b = 200$	$b = 400$	$b = 800$	$b = 1600$
$\|K\|K\|$	$2.0 \cdot 10^4$	$2.0 \cdot 10^3$	$5.6 \cdot 10^2$	233
$\|K\|N\|$	$4.9 \cdot 10^3$	$3.4 \cdot 10^2$	28	11
$\|N\|K\|$	$4.3 \cdot 10^3$	$3.0 \cdot 10^2$	23	0
$\|N\|N\|$	$2.8 \cdot 10^2$	2	0	0
Total	$2.9 \cdot 10^4$	$2.7 \cdot 10^3$	$6.1 \cdot 10^2$	244

Table 6.4: *Number of 3-round trail cores up to weight* 45.

In Figure 6.26 to Figure 6.29 we depict the number of 3-round trail cores per weight and per parity profile, to show the trend among the different variants of Keccak-f.

Even though these figures only give a limited view on the number of trail cores up to some weight, they allow us to detect some trends. First, for the number of trails with the smallest weight the pattern appears irregular, but as the weight increases, the number of trails seems to increase exponentially with the weight. Second, we see a dependence on the width. As the width increases, the minimum weight of 3-round trail cores increases, but not in a smooth way. However, the exponent by which the number of trails grows, decreases with the width.

In Figure 6.30 we accumulate trail cores with different parity profiles and give the total number of 3-round trail cores per weight. Note that each such trail core represents a class of w trail cores that can be obtained by translation along z. The results provide insight in how the number of exploitable 3-round trails scales with the width of the permutation. In Table 6.9 we report the execution time for each extension we performed.

6.4.3 Extension to 6-rounds

By extending all 3-round trail cores up to weight $T_3 = 45$, we can cover the space of all 6-round trails up to weight $2T_3 + 1 = 91$. We found no such trail and so we can say that this space is empty. This means that 92 is a lower bound for the weight of 6-round differential trails in Keccak-f for widths from 200 to 1600. For Keccak-f[1600] this result is an improvement of the bound found in [DV12], which was 74. The authors of [DV12] indeed proved that the space of 6-round trails with weight smaller than 73 is empty. For the other variants, from Keccak-f[200] to Keccak-f[800], it is a new result.

Scanning the space of 3-round trail cores up to weight 45 allows to cover also the space of 4 and 5-round trail cores up to weight 47 and 49 respectively. Indeed, any 4-round trail satisfies either $\mathrm{w}(b_0) + \mathrm{w}(b_1) + \mathrm{w}(b_2) \leq 45$ or $\mathrm{w}(b_1) + \mathrm{w}(b_2) + \mathrm{w}(b_3) \leq 45$ and the weight over a single round is at least 2. For Keccak-f[200] there exists a 4-round trail of weight 46, which is thus the lightest trail. For Keccak-f above 200 no trail is found below the limit. Therefore, we can say that the lower bound on the weight of 4-round trail cores for these variants is 48.

Any 5-round trail may satisfy the condition $\mathrm{w}(b_1)+\mathrm{w}(b_2)+\mathrm{w}(b_3) \leq 45$ and $\mathrm{w}(b_0) = \mathrm{w}(b_4) = 2$. For all Keccak-$f$ from 200 to 1600 no trail core with weight smaller or equal to 49 is found, therefore we can say that a lower bound on the weight of 5-round trail cores is 50.

6.4.4 Lightest trails

During our search we have found some interesting trail cores exceeding the limit weight, but that are, to the best of our knowledge, the lightest trails found up to now with no particular symmetry properties. We report them in Table 6.5 giving their weight and parity profiles.

These trails allow to define upper bounds for the minimum weight of trails. In the definition of these upper bounds, we must take into account the Matryoshka structure Section 5.4.3. For the 6-round trail of weight 278 in Keccak-f[400] there will be a corresponding trail in Keccak-f[800] of weight $2 \cdot 278$ and in Keccak-f[1600] of weight $2 \cdot 556$. In this case, the upper bound is given by this trail and not by the trail reported in Table 6.5.

We recap the obtained results in Table 6.6 reporting the minimum weight of trails or the range where the minimum weight lives, for the variants of Keccak-f from 200 to 800.

rounds	profiles	$b = 200$	$b = 400$	$b = 800$	$b = 1600$
3	weight	4 8 8	8 8 8	4 4 24	4 4 24
	parity	$\|K\|K\|$	$\|K\|K\|$	$\|K\|N\|$	$\|K\|N\|$
4	weight	21 9 8 8	29 6 4 24	20 9 8 67	16 13 12 93
	parity	$\|N\|K\|K\|$	$\|N\|K\|N\|$	$\|N\|K\|N\|$	$\|K\|K\|N\|$
5	weight	36 13 9 15 16	67 14 12 16 38	124 21 14 11 77	166 22 16 16 152
	parity	$\|N\|N\|K\|K\|$	$\|N\|K\|K\|N\|$	$\|N\|K\|K\|N\|$	$\|N\|K\|K\|N\|$
6	weight	45 44 13 9 15 16	81 35 6 4 24 128	192 17 12 12 73 438	166 22 16 16 152 841
	parity	$\|N\|N\|N\|K\|K\|$	$\|N\|N\|K\|N\|N\|$	$\|N\|K\|K\|N\|N\|$	$\|N\|K\|K\|N\|N\|$

Table 6.5: *Lightest trails found for different variants of* Keccak-f.

rounds	$b = 200$	$b = 400$	$b = 800$	$b = 1600$
2	8	8	8	8
3	20	24	32	32
4	46	[48,63]	[48,104]	[48,134]
5	[50,89]	[50,147]	[50,247]	[50,372]
6	[92,142]	[92,278]	[92,556]	[92,1112]

Table 6.6: *Minimum weight or range for the minimum weight of trails over a given number of rounds for widths b of* Keccak-f *from* 200 *to* 1600.

6.4.5 Bounds on n_r-rounds

From the bounds we have found, we can derive lower bounds on the total number of rounds for each variant of Keccak-f. We report them in Table 6.7. Each bound is derived considering the worst case scenario when extending 6-round trails to n_r rounds.

For Keccak-f[200], the total number of rounds is 18. A 18-round trail can be obtained by combining three 6-round trails that in the worst case contribute $3 \cdot 92 = 276$.

For Keccak-f[400], the total number of rounds is 20. A 20-round trail can be obtained by first combining three 6-round trails that in the worst case contribute only 276. Then, for the two rounds left, the worst case is when they are one a the beginning and one at the end of the trail, thus contributing only 2 to the weight. This gives a lower bound of 280.

For Keccak-f[800], the total number of rounds is 22. Again, we can consider the worst case contribution of three 6-round trails as 276. Then, for the four rounds left, the minimal contribution is when they are two at the beginning and two at the end of the trail, thus contributing only 8 to the weight. This gives a lower bound of 292.

Finally, for Keccak-f[1600], the total number of rounds is 24. A 24-round trail can be obtained by combining four 6-round trails that in the worst case contribute $4 \cdot 92 = 368$.

Similarly, we can also infer lower bounds for the number of rounds used in Kravatte, Keyak and Ketje. Actually, the number of rounds in Kravatte was chosen based on the lower bounds found in this work.

	b=200	b=400	b=800	b=1600
n_r	18	20	22	24
bound	276	280	292	368

Table 6.7: *Bounds on the weight of trails over the total number of rounds.*

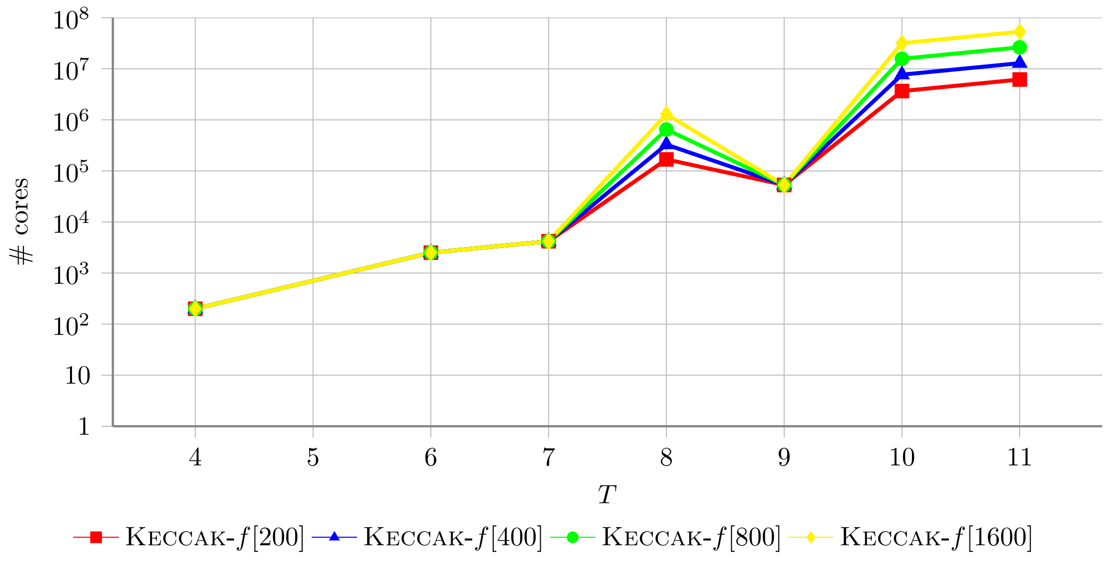


Fig. 6.10: *Number of 2-round trail cores* $\langle a, b\rangle$ *in* $|K|$ *with* $\mathrm{w}^{\mathrm{rev}}(a) = T$.

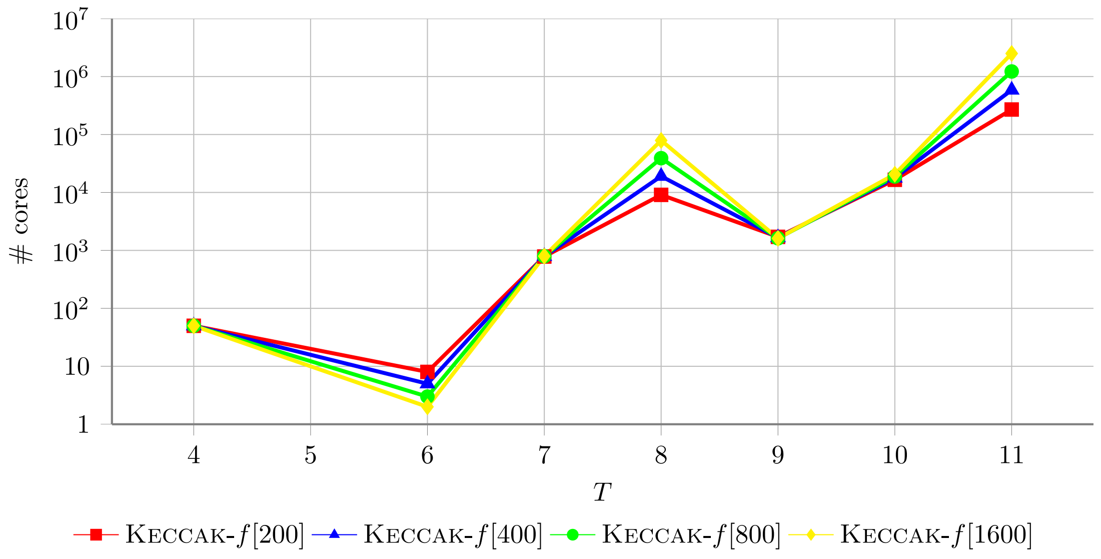


Fig. 6.11: *Number of 2-round trail cores* $\langle a, b\rangle$ *in* $|K|$ *with* $\mathrm{w}(b) = T$.

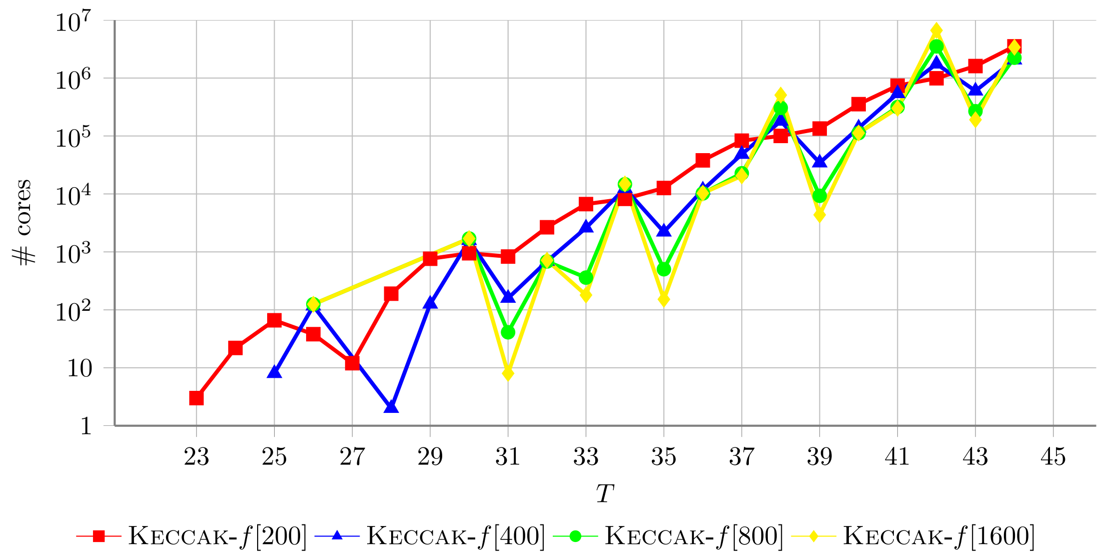


Fig. 6.12: *Number of 2-round trail cores* $\langle a, b\rangle$ *in* $|N|$ *with* $2\mathrm{w}^{\mathrm{rev}}(a) + \mathrm{w}(b) = T$.

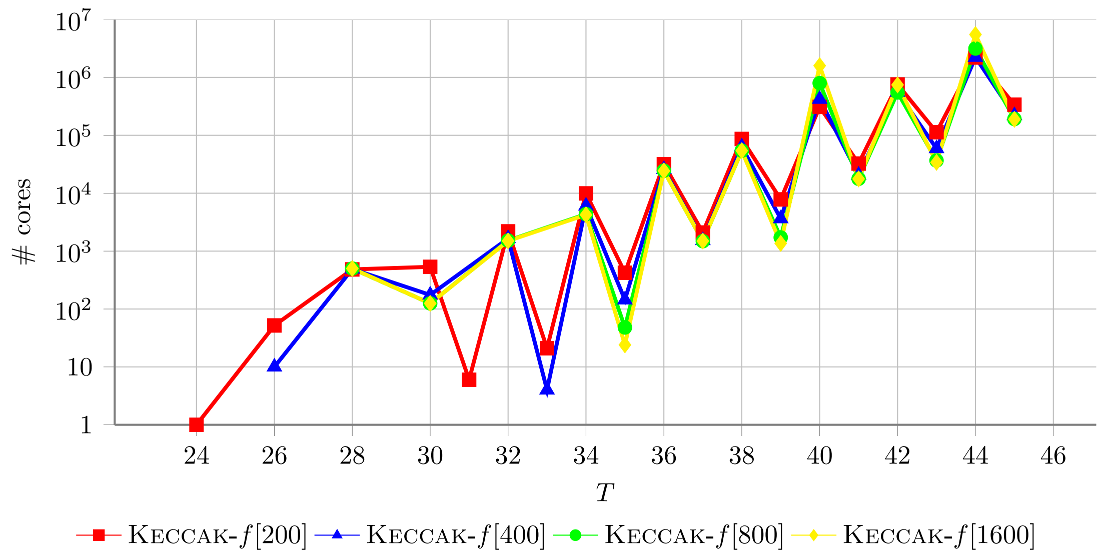


Fig. 6.13: *Number of 2-round trail cores* $\langle a, b\rangle$ *in* $|N|$ *with* $\mathrm{w}^{\mathrm{rev}}(a) + 2\mathrm{w}(b) = T$.

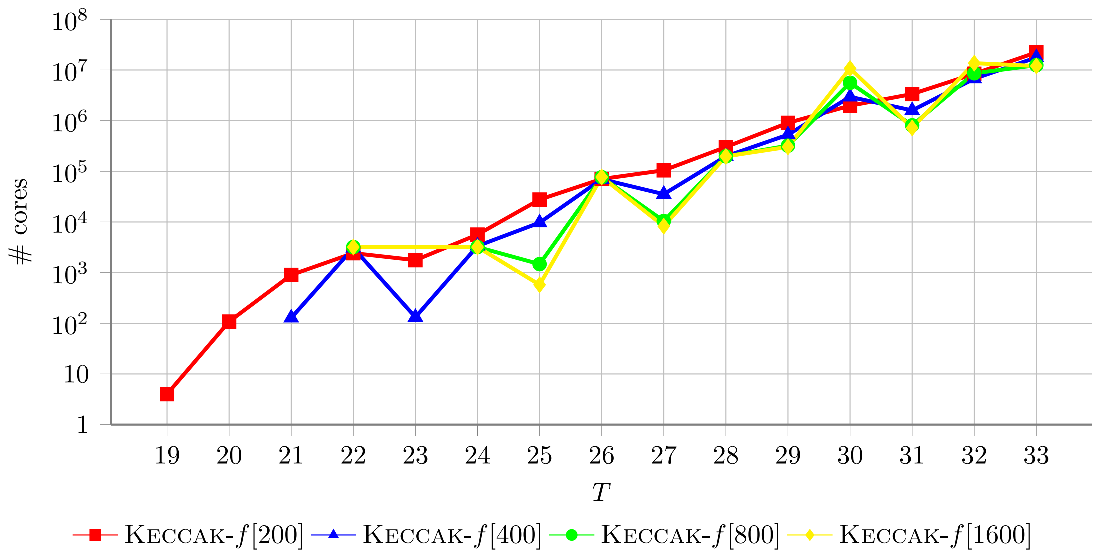


Fig. 6.14: *Number of 2-round trail cores* $\langle a, b\rangle$ *in* $|N|$ *with* $\mathrm{w}^{\mathrm{rev}}(a) + \mathrm{w}(b) = T$.

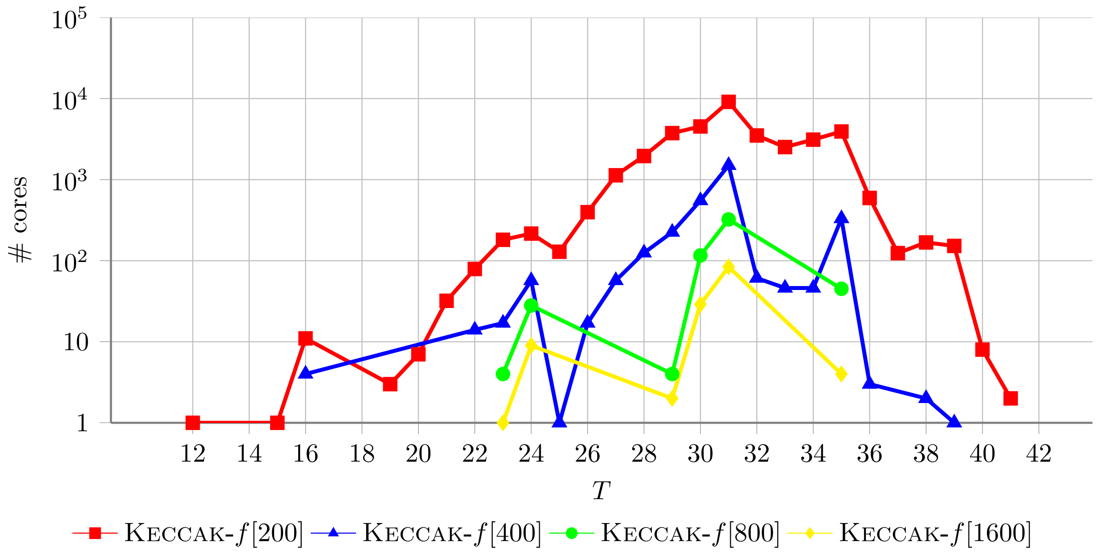


Fig. 6.15: *Number of 2-round trail cores* $\langle a, b\rangle$ *in* $|K|$ *with* $\chi(b)$ *in* $|K|$ *and* $\mathrm{w}^{\mathrm{rev}}(a) + \mathrm{w}(b) = T$. *Generated with the chain-based method.*

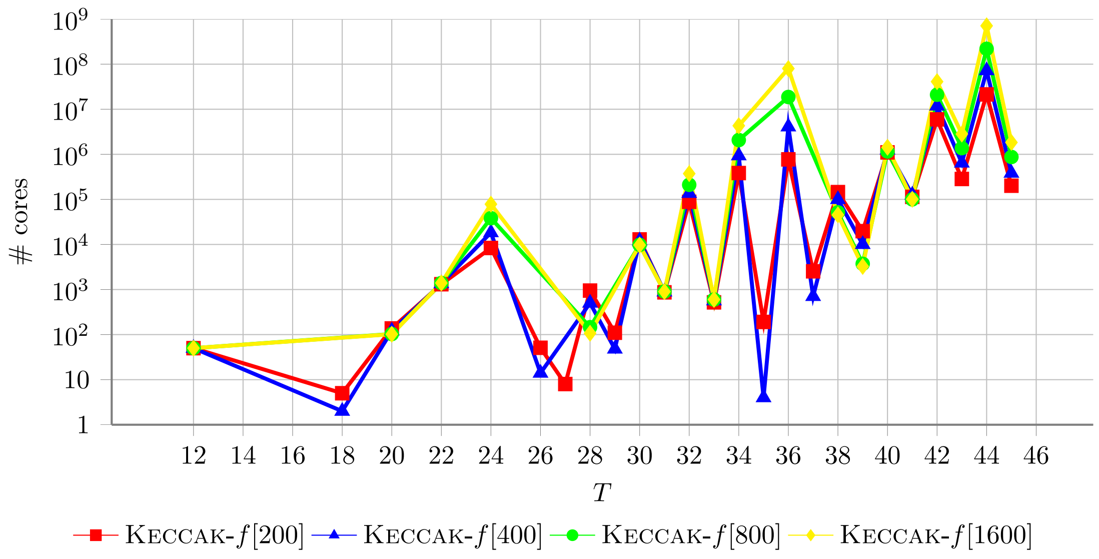


Fig. 6.16: *Number of 2-round trail cores* $\langle a, b\rangle$ *in* $|K|$ *with* $\mathrm{w}^{\mathrm{rev}}(a) + 2\mathrm{w}(b) = T$.

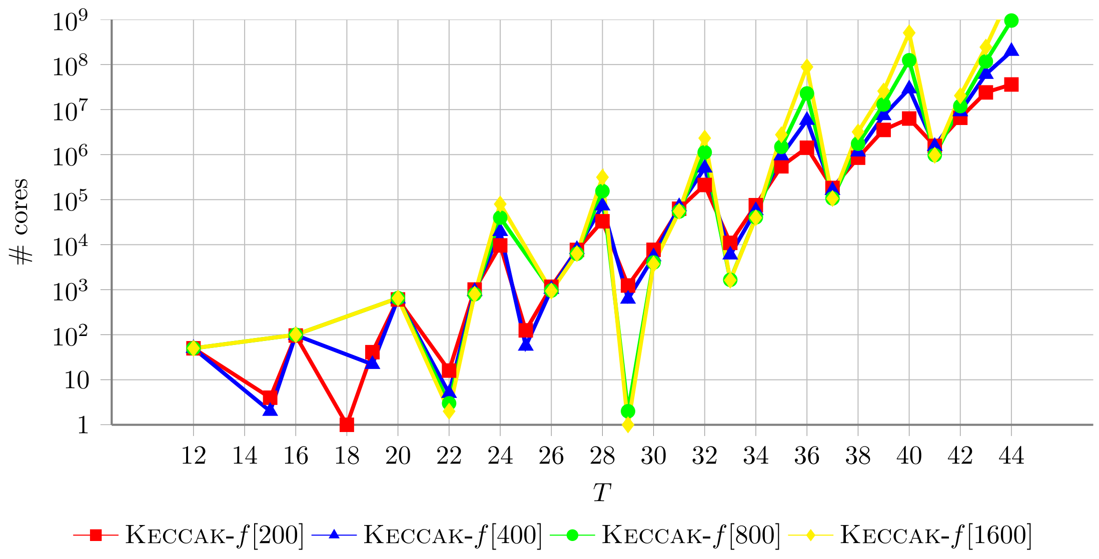


Fig. 6.17: *Number of 2-round trail cores* $\langle a, b\rangle$ *in* $|K|$ *with* $2\mathrm{w}^{\mathrm{rev}}(a) + \mathrm{w}(b) = T$.

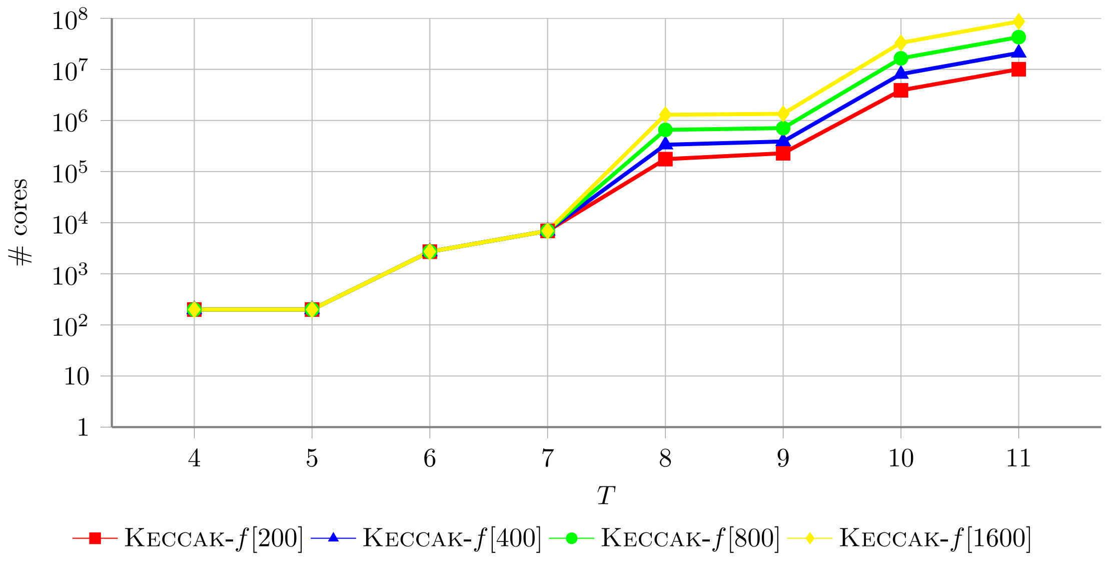


Fig. 6.18: *Number of 2-round trail cores* $\langle a, b\rangle$ *in* $|K|$ *with* $\mathrm{w}^{\mathrm{rev}}(a) \leq T$.

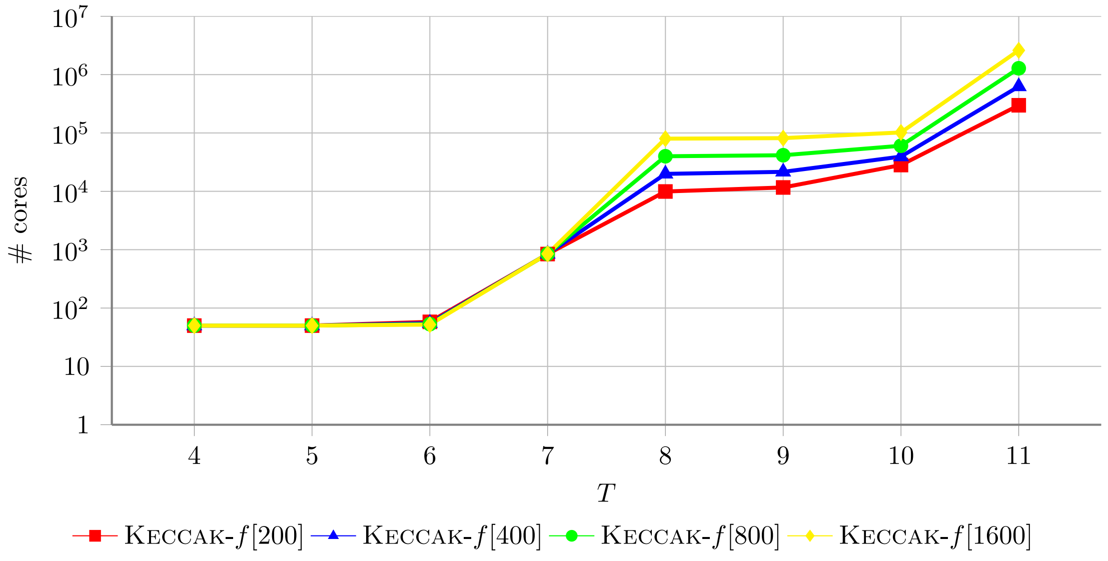


Fig. 6.19: *Number of 2-round trail cores* $\langle a, b\rangle$ *in* $|K|$ *with* $\mathrm{w}(b) \leq T$.

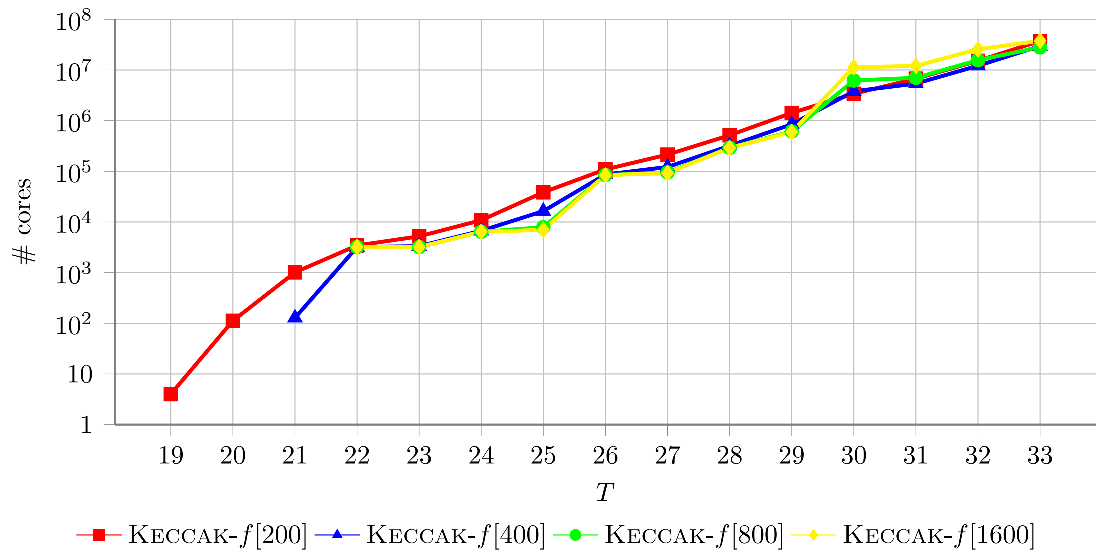


Fig. 6.20: *Number of 2-round trail cores* $\langle a, b\rangle$ *in* $|N|$ *with* $\mathrm{w}^{\mathrm{rev}}(a) + \mathrm{w}(b) \leq T$.

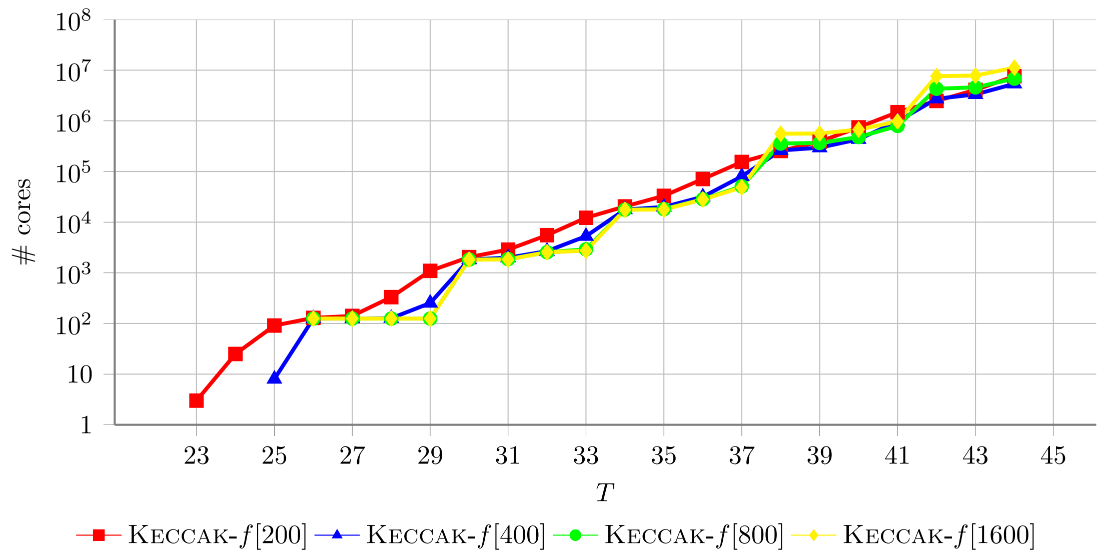


Fig. 6.21: *Number of 2-round trail cores* $\langle a, b\rangle$ *in* $|N|$ *with* $2\mathrm{w}^{\mathrm{rev}}(a) + \mathrm{w}(b) \leq T$.

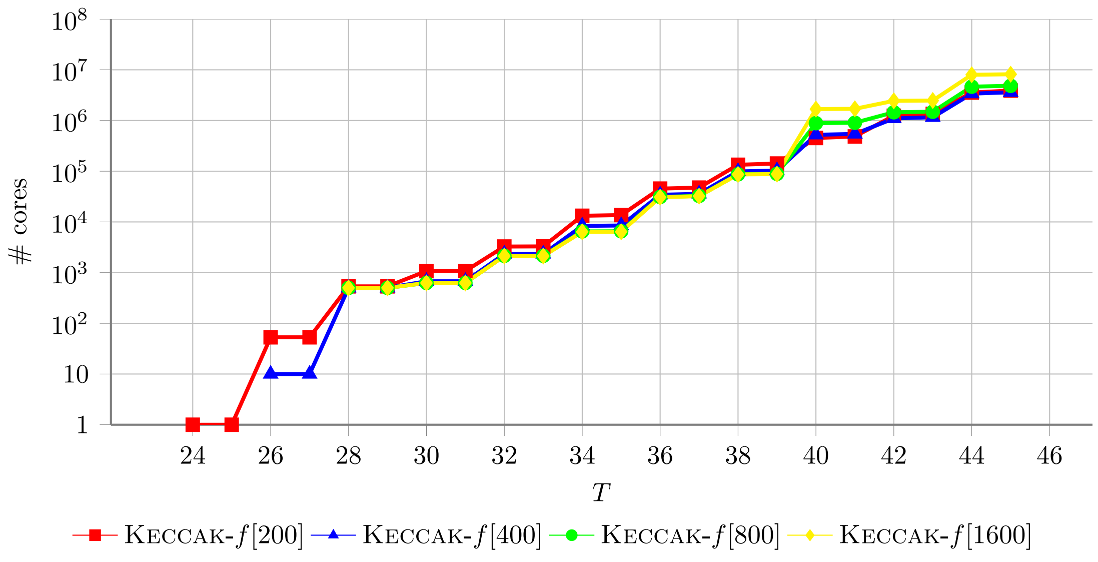


Fig. 6.22: *Number of 2-round trail cores* $\langle a, b\rangle$ *in* $|N|$ *with* $\mathrm{w}^{\mathrm{rev}}(a) + 2\mathrm{w}(b) \leq T$.

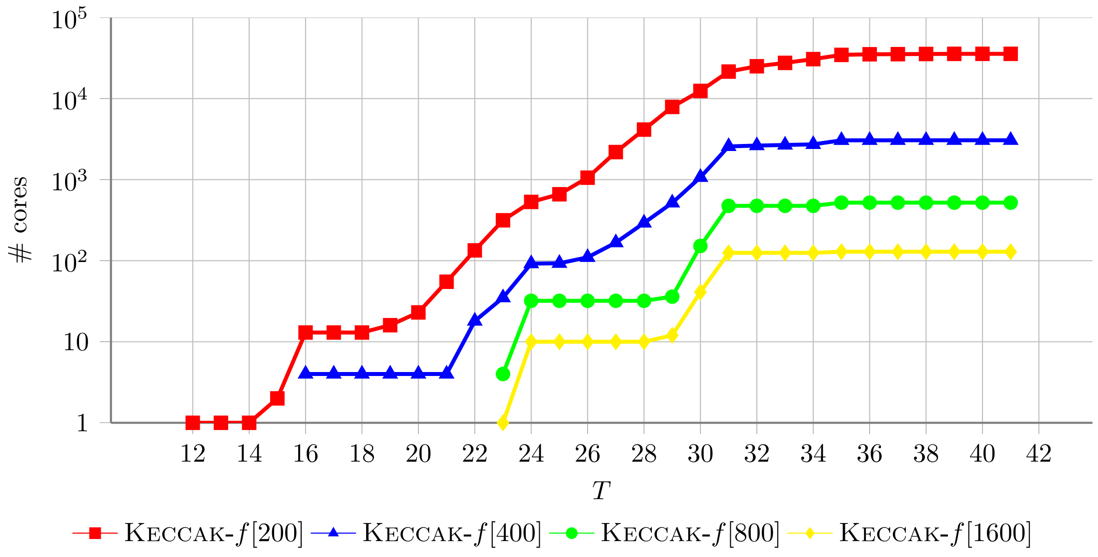


Fig. 6.23: *Number of 2-round trail cores* $\langle a, b\rangle$ *in* $|K|$ *with* $\chi(b)$ *in* $|K|$ *and* $\mathrm{w}^{\mathrm{rev}}(a) + \mathrm{w}(b) \leq T$.

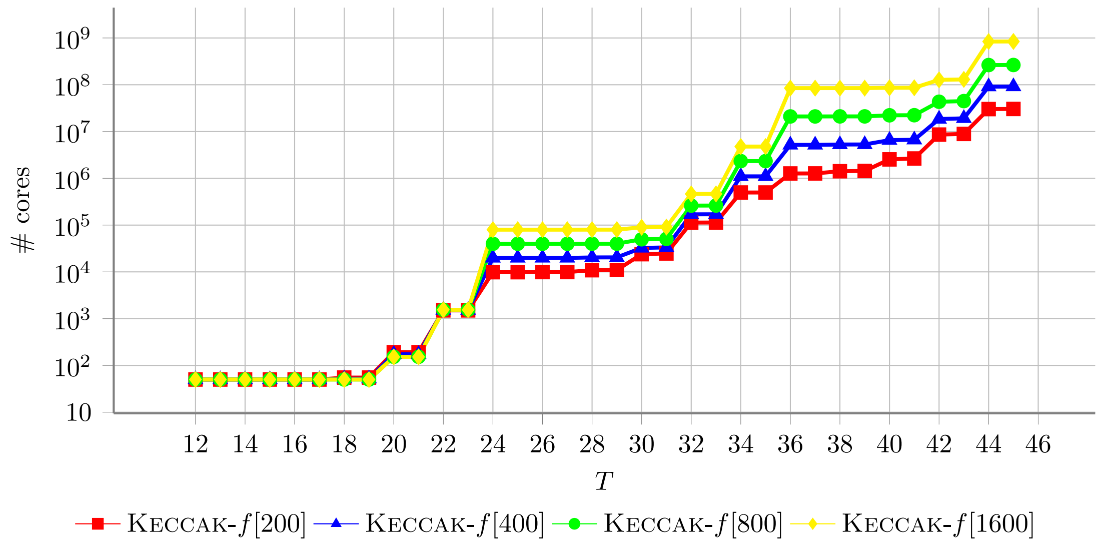


Fig. 6.24: *Number of 2-round trail cores* $\langle a, b\rangle$ *in* $|K|$ *with* $\mathrm{w}^{\mathrm{rev}}(a) + 2\mathrm{w}(b) \leq T$.

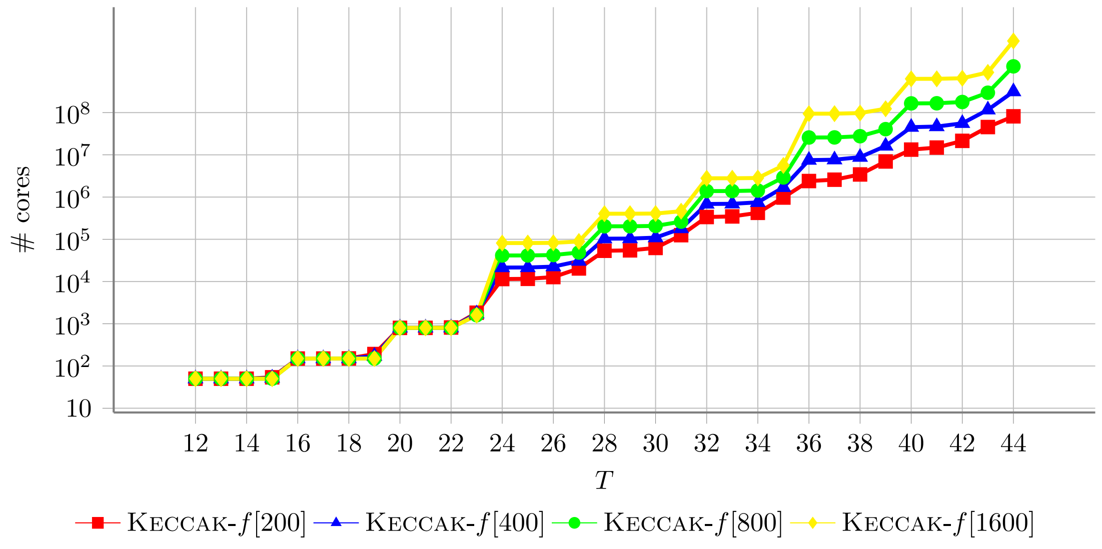


Fig. 6.25: *Number of 2-round trail cores* $\langle a, b\rangle$ *in* $|K|$ *with* $2\mathrm{w}^{\mathrm{rev}}(a) + \mathrm{w}(b) \leq T$.

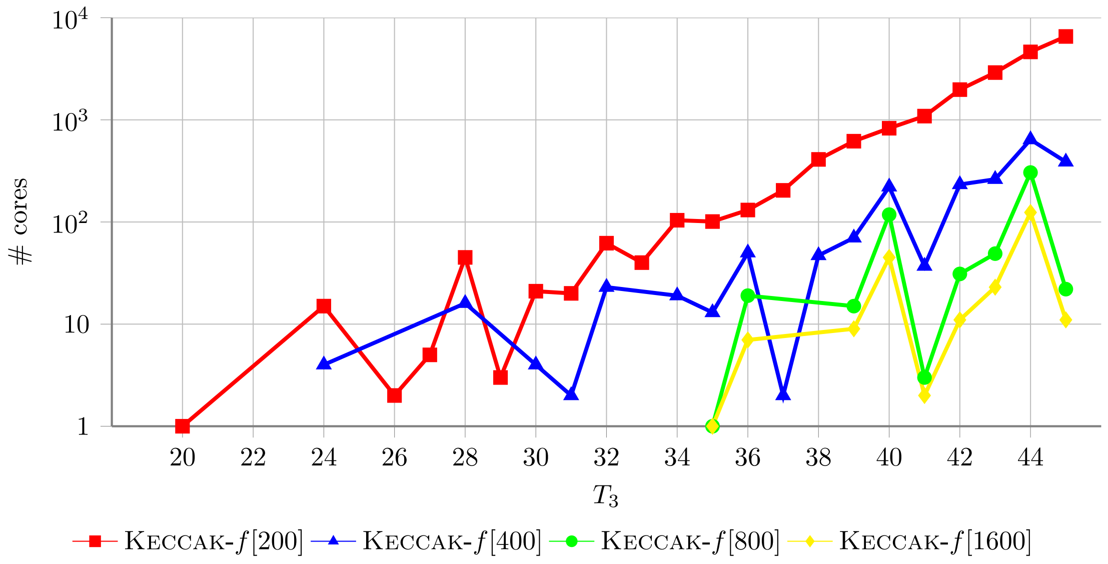


Fig. 6.26: *Number of 3-round trail cores with profile* $|K|K|$ *and weight* T_3*.*

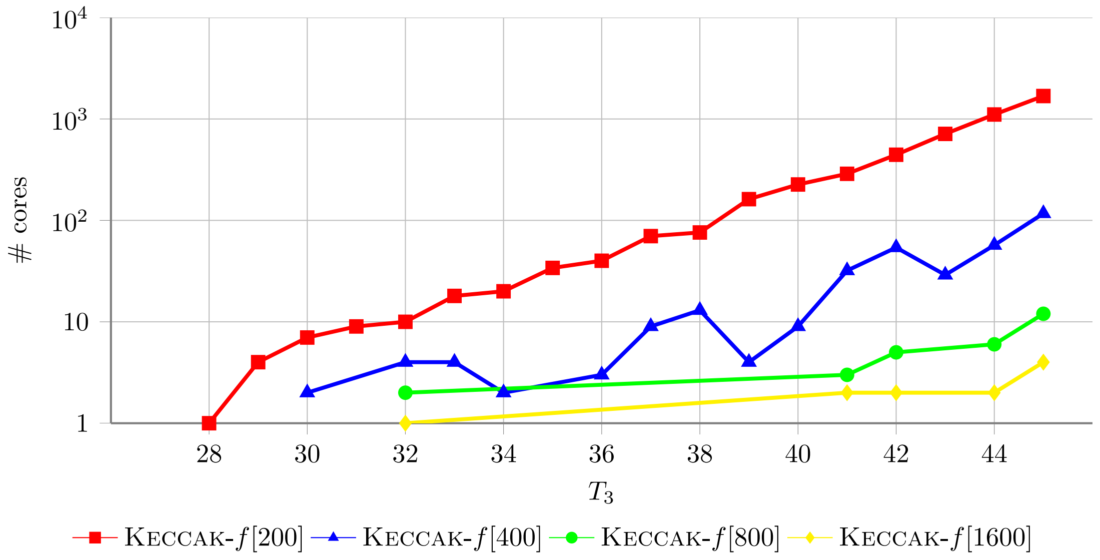


Fig. 6.27: *Number of 3-round trail cores with profile* $|K|N|$ *and weight* T_3*.*

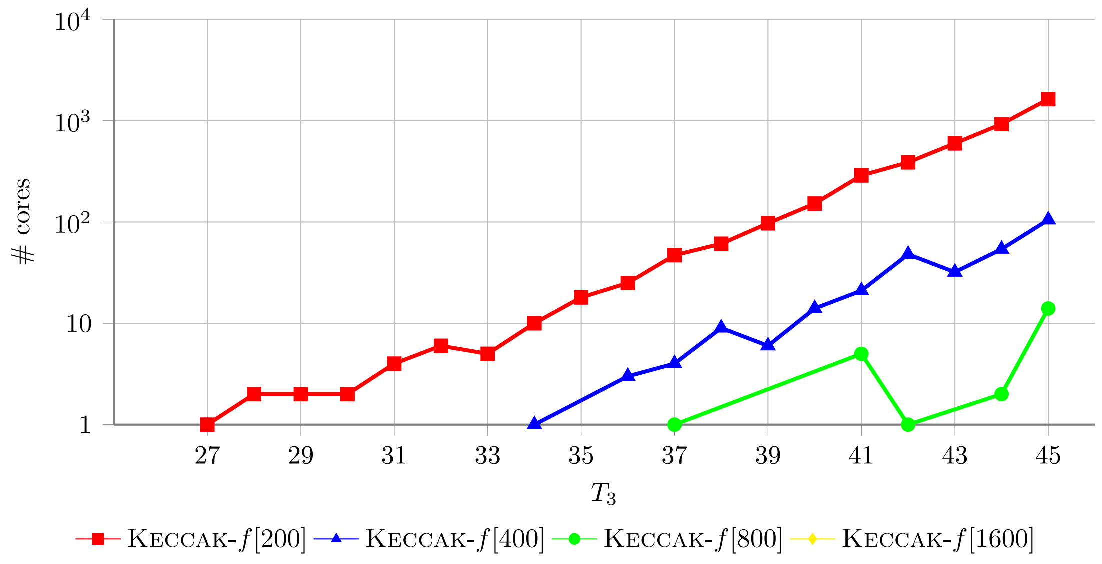


Fig. 6.28: *Number of 3-round trail cores with profile $|N|K|$ and weight T_3.*

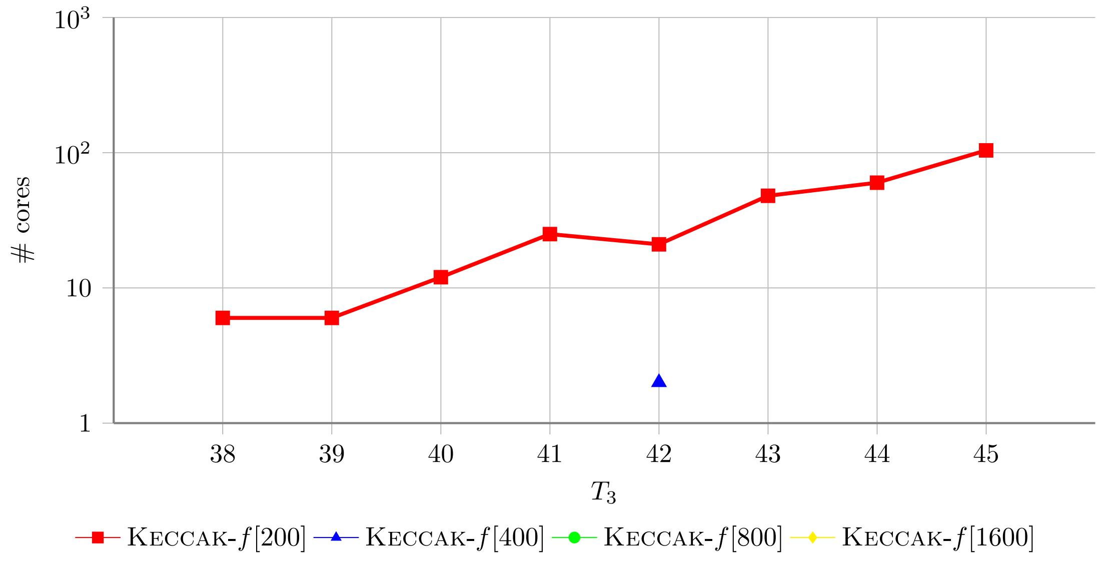


Fig. 6.29: *Number of 3-round trail cores with profile $|N|N|$ and weight T_3.*

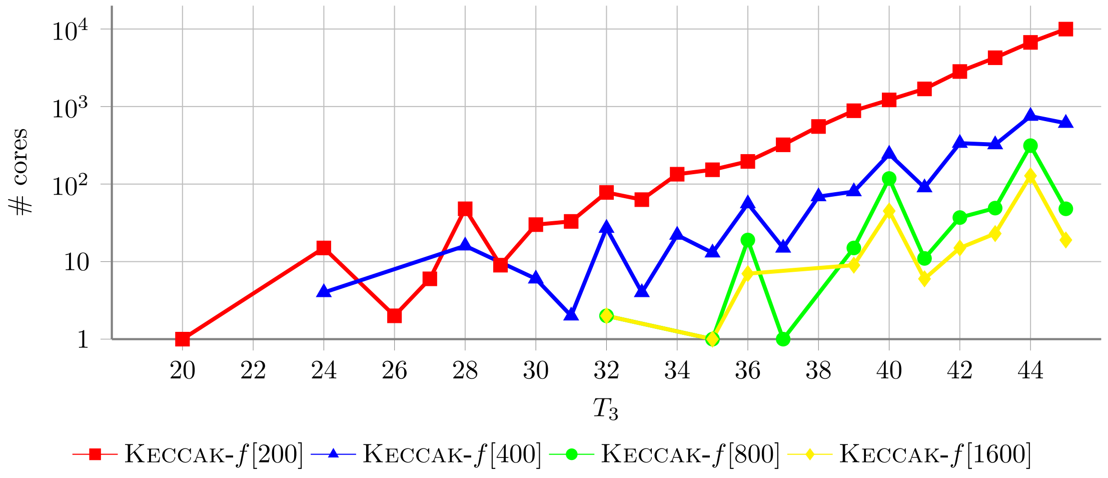


Fig. 6.30: *Number of all 3-round trail cores with weight T_3.*

6.5 Execution times

In Table 6.8 we report the execution time needed to generate the different sets of 2-round trail cores, while in Table 6.9 we report the time required to extend those trails in the backward or forward direction inside or outside the kernel. Experimental results have been obtained running the code on an Intel Xeon X5660 processor running at 2.8GHz, using a single core for each experiment.

For the case $|K|K|$, we report the execution time for either approaches. It is worth noticing that the orbital-based technique we presented in this thesis is more efficient than the chain-based technique of [DV12] for Keccak-f[200]. Indeed, covering the space (i.e. generating 2-round trail cores plus extending them) with the orbital-based approach takes around 2 hours, against the 37 hours of the chain-based approach. For Keccak-f[400] the two approaches are comparable in terms of execution time, while for the two largest variants of Keccak-f the chain-based approach is significantly faster. For instance, for Keccak-f[1600] the orbital-based approach takes more than 700 hours. Indeed, as shown in Figure 6.16 and Figure 6.17 the orbital-based approach generates a huge amount of trails whose extension becomes very expensive. On the contrary, the chain-based approach of allows to generate only those trail cores for which there might be a compatible pattern $\chi(b)$ in the kernel. This results in a very small number of trails given as output. It follows that extension of such trails is very cheap. On the contrary, for Keccak-f[200], the search of 2-round trail cores with such properties is more expensive than simply generating all 2-round trail cores in the kernel and extend them. However, it is worth noticing that the chain-based approach is very specific to differential trails in Keccak-f, while the orbital-based approach may be applied also to linear trails.

Finally, in Table 6.10 we report the total amount of time required to cover the space of all 3-round trail cores up to weight 45. In the computation of the total timings we consider the more efficient method to cover $|K|K|$.

2-round trail cores	$b = 200$	$b = 400$	$b = 800$	$b = 1600$
$\|K\|$ $\mathrm{w}^{\mathrm{rev}}(a) \leq 11$	2m20s	6m37s	19m50s	6h06m10s
$\|K\|$ $\mathrm{w}(b) \leq 11$	5s	14s	44s	2m27s
$\|N\|$ $\mathrm{w}^{\mathrm{rev}}(a) + \mathrm{w}(b) \leq 33$	1h36m51s	1h22m45s	1h42m27s	2h58m43s
$\|N\|$ $\mathrm{w}^{\mathrm{rev}}(a) + 2\mathrm{w}(b) \leq 45$	4h09m18s	2h50m14s	3h46m41s	6h44m40s
$\|N\|$ $2\mathrm{w}^{\mathrm{rev}}(a) + \mathrm{w}(b) \leq 44$	12h19m54s	25m36s	37m14s	1h12m48s
$\|K\|$ $\mathrm{w}^{\mathrm{rev}}(a) + 2\mathrm{w}(b) \leq 45$	4m54s	1h20m11s	1h30m35s	6h58m45s
$\|K\|$ $2\mathrm{w}^{\mathrm{rev}}(a) + \mathrm{w}(b) \leq 44$	12m49s	20m54s	5h12m47a	25h47m55s
$\|K\|$ with $\chi(b)$ in $\|K\|$	37h26m17s	12h38m33s	11h29m54s	16h11m16s

Table 6.8: *Execution time for the generation of 2-round trail cores. Last line refers to the chain-based technique of [DV12].*

2-round trail cores	extension	$b = 200$	$b = 400$	$b = 800$	$b = 1600$
$\lvert K\rvert\, \mathrm{w}^{\mathrm{rev}}(a) \leq 11$	$\rightarrow \lvert N\rvert$	18h03m01s	1h40m36s	19h11m45s	136h01m47s
$\lvert K\rvert\, \mathrm{w}(b) \leq 11$	$\leftarrow \lvert N\rvert$	17m57s	5m55s	31m13s	2h36m01s
$\lvert N\rvert\, \mathrm{w}^{\mathrm{rev}}(a) + \mathrm{w}(b) \leq 33$	$\rightarrow \lvert K\rvert$	2m51s	2m25s	3m17s	6m32s
	$\leftarrow \lvert K\rvert$	18h17m07s	2h16m52s	17h40m33s	33h28m14s
$\lvert N\rvert\, \mathrm{w}^{\mathrm{rev}}(a) + 2\mathrm{w}(b) \leq 45$	$\leftarrow \lvert N\rvert$	8h54m39s	39m34s	4h19m50s	45h52m52s
$\lvert N\rvert\, 2\mathrm{w}^{\mathrm{rev}}(a) + \mathrm{w}(b) \leq 44$	$\rightarrow \lvert N\rvert$	1h00m39s	24m37s	2h14m51s	10h06m26s
$\lvert K\rvert\, \mathrm{w}^{\mathrm{rev}}(a) + 2\mathrm{w}(b) \leq 45$	$\leftarrow \lvert K\rvert$	45m01s	6h57m40s	49h32m18s	> 700h
$\lvert K\rvert\, 2\mathrm{w}^{\mathrm{rev}}(a) + \mathrm{w}(b) \leq 44$	$\rightarrow \lvert K\rvert$	56m12s	9h44m11s	> 300h	> 1000h
$\lvert K\rvert$ with $\chi(b)$ in $\lvert K\rvert$	$\rightarrow \lvert K\rvert$	3s	1s	2s	5s

Table 6.9: *Execution time for the extension of 2-round trail cores to 3 rounds. Symbols $\rightarrow \lvert K\rvert$ and $\rightarrow \lvert N\rvert$ denote forward extension in the kernel and outside the kernel respectively; $\leftarrow \lvert K\rvert$ and $\leftarrow \lvert N\rvert$ denote backward extension in the kernel and outside the kernel respectively.*

operation	$b = 200$	$b = 400$	$b = 800$	$b = 1600$
generation of 2-round trail cores	18h26m11s	17h23m59s	17h56m06s	33h16m04s
extension to 3 rounds	48h17m27s	5h10m00s	43h30m18s	225h35m56s
Total	66h43m38s	22h33m59s	61h26m24s	258h52m00s

Table 6.10: *Total execution time to cover the space of all 3-round trail cores with weight up to 45, consider the fastest method for the case $\lvert K\rvert K\rvert$.*

6.6 Comparison with [DV12]

As explained in Chapter 2, this work derived from the necessity of overcoming some bottlenecks encountered in the application of the techniques of [DV12]. Indeed, what we tried to do as first step was to use the KeccakTools to cover the space of 3-round trails up to a target weight larger than 36. But we encountered a number of obstacles, which motivated us to develop new and more efficient techniques. We describe these obstacles here and report the execution time needed to cover the space of 3-round trails up to weight 36 in Keccak-$f[1600]$ using both approaches. These results will show how the new techniques presented in this thesis allow to scan the space of trails more efficiently, thus allowing to push the target weight to higher values.

6.6.1 Limitations of the previous techniques

In [DV12], the space of 3-round trail cores (a_1, b_1, a_2, b_2) up to weight T_3 is covered by splitting it into subspaces that are scanned using different techniques:

1. all 3-round trail cores with both a_1 and a_2 in the kernel. This is covered by first generating all 2-round trail cores with a in the kernel for which there exists a compatible state $\chi(b)$ in the kernel. Then, forward extension (in the kernel) of these trails is performed.
2. all 3-round trail cores such that $\mathrm{w}^{\mathrm{rev}}(a_1) < T_1$ or $\mathrm{w}(b_1) < T_1$ or $\mathrm{w}(b_2) < T_1$. This set is covered by first generating all 2-round trail cores such that a or b has $\lfloor \frac{T_1}{2} \rfloor$ or less active rows. In fact, a weight up to $T_1 - 1$ implies at most $\lfloor \frac{T_1}{2} \rfloor$ active rows, since each row contributes at least 2 to the weight. Then, forward and backward extension of these trails is performed.
3. all 2-round trail cores such that $\mathrm{w}^{\mathrm{rev}}(a_1) \geq T_1$, $\mathrm{w}(b_1) \geq T_1$ and $\mathrm{w}(b_2) \geq T_1$ and not both a_1 and a_2 in the kernel. This is covered by first generating all 2-round trail cores outside the kernel with weight up to $T_3 - T_1$ and then extending them in forward and backward direction.

The above techniques have been applied to cover the space of 3-round trail cores up to weight 36, by choosing $T_1 = 8$ and thus $T_2 = 28$. Thanks to this the authors of [DV12] have proved that the minimum weight for 3-round trails is 32 and that there are no 6-round trails with weight smaller than 73.

While trying to use these methods with a limit weight above 36 (looking for 6-round trails with small weight), we have faced the following bottlenecks:

1. in point 2 above: going for weight 8 or more explodes the number of cases to investigate and to extend. In fact, in Keccak-f[1600] the number of states with up to 3 active rows is about 2^{30}. Considering 8 or more implies to generate all states with up to 4 or more active rows, and thus to build more than 2^{40} states.
2. in point 3 above: the generation of 2-round trail cores outside the kernel is performed in two phases. In the first phase parity patterns are incrementally generated by adding runs and computing a lower bound on the weight of trail cores with a given parity. This bound exploits the fact that each affected column has a total of five active bits before and after and that an unaffected odd column has two active bits, one at a and one at b. So, the bound is computed starting from the total Hamming weight and the minimum number of active rows at a and b. Only those parity patterns with small lower bound (namely, below 28) are kept. Namely, only those parity patterns that might give rise to a trail core with weight smaller than 28. In the second phase, difference patterns are concretely built for those parities generated during the first phase, by iterating over the column values. Lemma 3 in [DV12] is tailored to the initial target weight of 36; aiming for a bigger target gives rise to thousands of parity patterns to investigate, most of which do not actually give rise to valid 2-round trail cores. In fact, the lower bound computed in the first step is loose and the result is that many of the parity patterns generated in the first phase do not actually give rise to trail cores whose weight is below the limit. In Table 6.11 we report the ratio between the number of parity patterns that really give rise to trails with weight below T_2 and the number of parity patterns with bound below T_2, for different values of T_2.

T_2	$b = 200$	$b = 400$	$b = 800$	$b = 1600$
26	22/181	18/61	15/37	15/35
28	30/534	27/127	24/48	22/41
30	42/1505	34/364	31/99	31/87
32	66/4028	45/938	36/342	35/277

Table 6.11: *Ratio between the number of parity patterns that give rise to trails with weight below T_2 and the number of parity patterns with bound below T_2, according to [DV12].*

As T_2 increases, the gap between the bound and the actual minimum weight of cores per given parity increases. Indeed, the number of parities for which the bound underestimate the minimum weight grows. This gap suggests the need for better techniques to lower bound the weight. In fact, we decided to generate parity patterns and difference patterns simultaneously, by making use of the run structure of parity patterns. This allows us to get better bounds on the weight of resulting trail cores, since the bound is computed from the active bits of the current trail core instead of being computed only from lower bounds on runs as in [DV12].

3. in general, increasing the target weight results in the generation of a larger amount of 2-round trail cores which must be extended to 3 rounds (see Figures from 6.18 to 6.23 for some examples). Moreover, the bigger the weight of a pattern is, the bigger is the number of its active rows and thus the bigger the number of (χ or χ^{-1}) compatible states to build is. Therefore, increasing the target weight makes extension much more expensive because the number of trail cores to investigate and the number of compatible pattern for each trail core grows. This motivated us to optimize extension, in order to avoid the investigation of patterns that will not give rise to valid trail cores (i.e. below the target weight).

6.6.2 Covering the space of 3-round trails up to weight 36

In order to compare our new techniques to previous work, we repeated the experiments described in [DV12]. In particular, we generated all 3-round trail cores for KECCAK-f[1600] up to weight 36 using the routines available in [BDPV15]. Then, we covered the same space using our code, by setting $T_1 = 8$, $T_2 = 27$ and $T_3 = 36$.

In Table 6.12 and Table 6.13 we report the execution time for both the approaches. Note that with the new techniques presented in this thesis we can cover the same space in a smaller amount of time and this allows to target higher weights in a still reasonable amount of time, as shown in previous section.

Notice that using the chain-based approach of [DV12] to cover the case $|K|K|$ (last line of Table 6.12 and Table 6.13) is more efficient than the orbital-based approach introduced in Lemma 6.5. Indeed, the former generates 129 trail cores in less than 5 minutes that are subsequently extended in 3 seconds. By using the alternative approach that we proposed, we generate trail cores with $2w_0 + w1 \leq 35$ in less than 4 minutes. The number of such trail cores is above 5 millions and forward extension (in the kernel) takes more than 2 hours. Then we generate trail cores with $w_0 + 2w1 \leq 36$ in around 40 minutes. The number of such trail cores is above 84 millions and backward extension (in the kernel) takes more than 60 hours. Therefore, we consider only the most efficient method when computing the total execution time.

generation of 2-round trail cores		extension to 3 rounds	
type	time	type	time
$\|\|a\|\|_{row} \leq 3$	34h38m14s	$\rightarrow$	2h07m13s
		$\leftarrow$	32m28s
$\|\|b\|\|_{row} \leq 3$	14h55m43s	$\rightarrow$	7s
		$\leftarrow$	more than 10 days
$\|N\|\ \mathrm{w}^{\mathrm{rev}}(a) + \mathrm{w}(b) \leq 28$	3m28s	$\rightarrow$	2h00m50s
		$\leftarrow$	more than 10 days
$\|K\|$ with $\chi(b)$ in $\|K\|$	4m50s	$\rightarrow$	4s
Total	49h42m15s	Total	more than 20 days

Table 6.12: *Execution time for the generation of 3-round trail cores up to weight* 36 *using the techniques of [DV12] and the* KECCAKTOOLS.

generation of 2-round trail cores		extension to 3 rounds	
type	time	type	time
$\|K\|\ \mathrm{w}^{\mathrm{rev}}(a) \leq 8$	53s	$\rightarrow \|N\|$	39m53s
$\|K\|\ \mathrm{w}(b) \leq 8$	9s	$\leftarrow \|N\|$	3m56s
$\|N\|\ \mathrm{w}^{\mathrm{rev}}(a) + \mathrm{w}(b) \leq 27$	1m23s	$\rightarrow \|K\|$	2m36s
		$\leftarrow \|K\|$	5m06s
$\|N\|\ \mathrm{w}^{\mathrm{rev}}(a) + 2\mathrm{w}(b) \leq 36$	4m27s	$\leftarrow \|N\|$	2m56s
$\|N\|\ 2\mathrm{w}^{\mathrm{rev}}(a) + \mathrm{w}(b) \leq 35$	32s	$\rightarrow \|N\|$	48s
$\|K\|$ with $\chi(b)$ in $\|K\|$	4m50s	$\rightarrow \|K\|$	4s
Total	12m14s	Total	55m19s

Table 6.13: *Execution time for the generation of 3-round trail cores up to weight* 36 *using the new techniques presented in this work. Last line refers to the chain-based technique of [DV12].*

7 Conclusions

In this part of the thesis, we presented new techniques to scan the space of all 6-round differential trail cores in Keccak-f up to a given weight. To do this, we started by generating all 3-round trail cores up to half the given weight. Three-round trail cores were in turn generated starting by 2-round trail cores.

At first, we have introduced a general formalism to express differential patterns as ordered lists of units. This allows arranging all patterns in a tree where the children of a node are obtained by adding a new unit at the end of the list. Exploiting the fact that the weight of a 2-round trail core is fully specified by a single differential pattern, we translated the problem of generating all 2-round trail cores below a given weight into the problem of traversing such tree. In fact, we defined a metric that lower bounds the weight of the node and that of its descendants. As soon as the search encounters a node whose bound is already above the limit, we can safely discard this node all its descendants. Thanks to its abstraction level, the tree traversal approach can be applied to primitives with round functions consisting of a bit-oriented mixing layer, a bit transposition layer and relatively lightweight non-linear layer. Of course, the specific definition of units, order relations and cost bounding functions depends on the targeted primitive and its properties.

For the specific case of Keccak-f, we instantiated the tree traversal method exploiting the propagation properties of the step mappings and their translation-invariance along the z-axis. In particular, we split difference patterns based on their parity and defined two different tree structures, one for the patterns in the kernel and one for those outside the kernel.

Then, we presented new techniques that exploit the parity profile of trails in Keccak-f to extend them efficiently to longer trails. We showed how extension can be done as a tree traversal, too.

We implemented these methods in the KeccakTools [BDPV15] and used them to scan the space of all 3-round trail cores in Keccak-f[200] to Keccak-f[1600] up to weight 45. The most direct consequence is an extension and an improvement over known bounds for differential trails over 3, 4, 5 and 6 rounds, as summarized in Table 7.1. In particular, the authors of [DV12] proved that the space of 6-round trail cores with weight below or equal to 73 is empty, that means that such a trail has weight at least 74. In this thesis we showed that there are no 6-round trails with weight below 91, setting the lower bound for the minimum weight of such trails to 92. The results for Keccak-$f[b]$ for $b \in \{200, 400, 800\}$ are instead new. We have also found new trails, that are, for the best of our knowledge, the lightest trails known in literature.

Furthermore, the obtained results provide insight in how the number of exploitable trails scales with the width of the permutation. It appears that low-weight trails over more than 4 rounds become sparser with increasing width. This makes sense as the sheer number of 3-round trails to be extended decreases with the width (see Figure 6.30) and the probability that a given 3-round trail can be extended with a low-weight pattern decreases with the width. As rounds are combined, the number of trails up to a given weight per round decreases significantly. We depict these numbers in Figure 7.1 for 1, 2 and 3-round trail cores. For instance, for Keccak-f[1600]

rounds	$b = 200$	$b = 400$	$b = 800$	$b = 1600$
2	8 [BDPV11b]	8 [BDPV11b]	8 [BDPV11b]	8 [BDPV11b]
3	20 [BDPV11b]	24 [this work]	32 [this work]	32 [DV12]
4	46 [BDPV11b]	[48,63] [this work]	[48,104] [this work]	[48,134] [this work]
5	[50,89] [this work]	[50,147] [this work]	[50,247] [this work]	[50,372] [this work]
6	[92,142] [this work]	[92,278] [this work]	[92,556] [this work]	[92,1112] [this work]

Table 7.1: *Minimum weight or range for the minimum weight of trails for different variants of* KECCAK-f *and number of rounds. A range $[x, y]$ means that x is a lower bound and y is the weight of the lightest known trail*

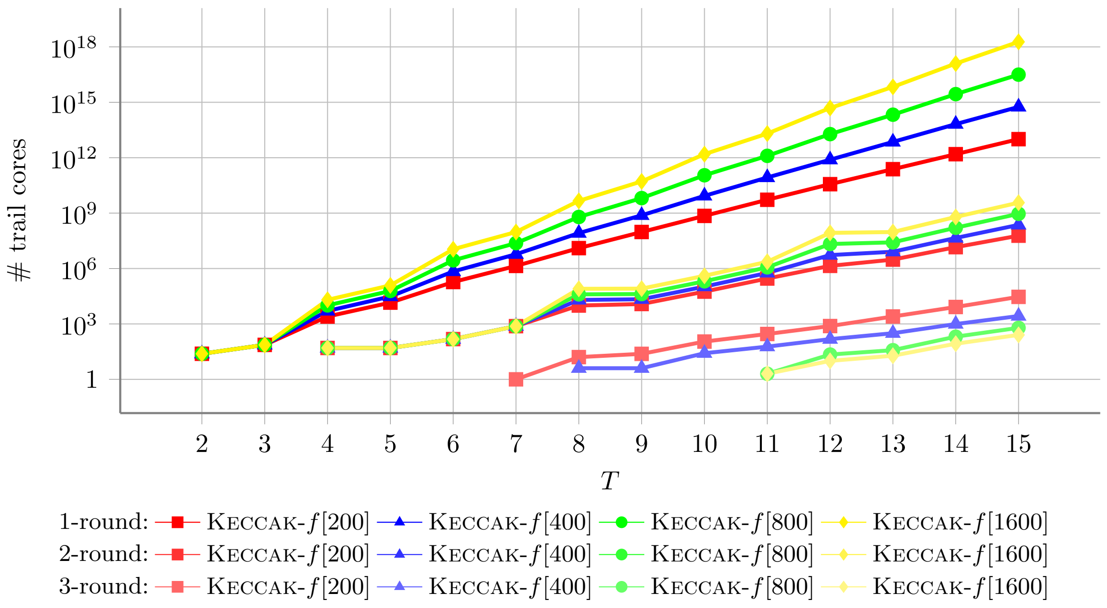


Fig. 7.1: *Number of 1,2 and 3-round trail cores in* KECCAK-f *modulo z-symmetry with weight per round up to T.*

the number of single-round differentials with weight up to 15 is $2^{60.7}$. The number of 2-round trails with weight up to $15 \cdot 2$ is $2^{31.7}$, whereas the number of 3-round trails up to weight $15 \cdot 3$ is just 2^8. After extending to 6 rounds, we see that there are no trails with weight below $15 \cdot 6$.

Developing dedicated programs is a time-consuming effort, but it is still worthwhile when compared to using standard tools. Assume we want to generate the space of all n-round trails with weight up to T. In a naive scan of the space, one has no choice but to consider all round differentials with weight up to $\lfloor T/n \rfloor$ and try to build trails from them. The limiting factor is the number of single-round differentials with weight up to $\lfloor T/n \rfloor$ and the effort per such differential to extend it into n-round trails. As can be seen in Figure 4.1, the numbers increase exponentially with the weight and this increase is stronger as the width grows. When using standard tools, a weight per round of 9 for width $b = 200$ and a weight per round of 7 for width $b = 1600$ already seems quite ambitious as both imply at least hundred million trail extensions starting from a round differential. This is in line with the results obtained with standard tools cited in Chapter 2 that up to now never reached a weight per round above 9. The dedicated efforts for Noekeon in [DPVR00] and KECCAK-f[1600] in [DV12] both reached a weight per round of 12. For Noekeon this can be compared with weight 12 for width 200, where in a naive approach between 2^{30} and 2^{35} round differentials would have to be investigated. For KECCAK-f[1600] this would be even $2^{48.8}$. In this thesis we scan the space of 3-round differential trails of weight up to 45. As this gives a weight of 15 per round for KECCAK-f[1600] doing this the naive way would imply extending $2^{60.7}$ round differentials to 3-round trails. From this perspective, our new techniques allow us to cover a space that is a factor $2^{60.7}/2^{48.8} \approx 4000$ times larger than in [DV12].

Part II

Differential Fault Analysis against AES

8 Introduction

The use of hardware faults to attack a cryptographic system, was originally presented by Boneh et al. in 1997 [BDL97]. This attack was applied to recover the secret key of an RSA implementation using the Chinese Remainder Theorem.

Subsequently, the idea of applying faults to attack implementations of cryptographic algorithms was extended to symmetric ciphers [BS97]. The technique introduced against block ciphers is referred to as Differential Fault Analysis (DFA) and it exploits techniques from differential cryptanalysis to study the difference between correct and faulty ciphertexts in order to obtain information on the secret key.

DFA attacks have been presented against several symmetric encryption schemes, such as DES and triple-DES [BS97, Hem04], CLEFIA [CWF07, TF08a], AES [BS03, Gir03, PQ03, TM09, MSS06, PMC+11].

In this thesis we focus our attention on the AES algorithm. Being a NIST standard and the most used symmetric cipher, the AES has attracted and still attracts much attention from the cryptanalysis community. Hence, a considerable number of results has been presented on it, including results about differential fault analysis. For these reasons, even if some of our results are more generically applicable to DFA, we conduct our analysis on the AES in order to explain and compare our findings with well-known examples available in literature.

The best known DFA attacks, including those against AES, assume a fault induced to alter an intermediate state or a round key. Very few examples have been presented, based on the modification of the total amount of round iterations. This is an example of exploiting alteration of the sequence of operations instead of data, which is the most common approach.

We thus wonder whether it is possible to conduct DFA on pairs of ciphertexts obtained by faults causing a change in the sequence of instructions, different from reducing the number of rounds. To answer this question, we present four DFA attacks against AES that exploit faults causing a misbehavior of the algorithm during the final round. In particular, we assume that one of the steps composing the AES final round (i.e. the AddRoundKey, the SubBytes and the ShiftRows operations) is jumped or that the MixColumns operation is executed also in the final round.

It is important to underline that these attacks are presented here only theoretically. Our aim is to bring attention on the importance of protecting parts of the implementation that are usually not covered by countermeasures. In fact, it has been shown in [PMC+11, CT05] that clock glitches and power spikes can cause the jumping of instructions during the computation, making our DFA attacks feasible in principle. Moreover, these fault injection techniques are actually cheaper and require a simpler equipment with respect to those techniques used to target specific bits of the state or key, like laser beams.

Another aspect of DFA that we want to examine is the impact that the fault model, and the knowledge the attacker has on it, has on the effectiveness of an attack. In fact, all DFA attacks against AES rely on a fault model implied by the attacker. In our attacks it corresponds to jumping a specific instruction, while in the most common attacks it corresponds to altering a bit, a byte or a word of the intermediate state or key.

We thus investigate what happens if the fault injection technique used to mount the attack does not always have the effect expected by the attacker. Namely, what happens if the attacker performs DFA on ciphertexts that do not follow the assumed fault model (we call such ciphertexts *bad* and those that follow the model *good*). We conduct our analysis on some of the most known DFA attacks against AES, which fall in the class of attacks that exploit bit or byte faults on the intermediate state. So, we study the consequence of treating a faulty ciphertext as the result, for instance, of a single-bit fault injection while it actually is the result of a two-bit fault injection.

One might expect that performing the analysis on bad ciphertexts (or some good and some bad) brings to no solution and thus that the attacker can simply discard those experiments and perform DFA on new ciphertexts. But, we will show that the probability that the analysis brings to a wrong solution is not negligible. Actually, also in the case of no solution, the attacker is not able to distinguish which ciphertexts are good and which are bad. Hence, the a-priori knowledge of the fault model by the attacker can fundamentally impact the effectiveness of the attack.

In order to relax such constraint, we introduce a novel approach for DFA based on the application of a clustering technique. The aim is to increase the robustness of the attacks and soften the requirements on the a-priori knowledge by the attacker. We name this new approach J-DFA, since the clustering method it is based on is called J-Linkage. J-Linkage addresses the problem of simultaneous multiple model fitting within a dataset containing outliers, namely observations which do not fit any model. In our case outliers are bad ciphertexts.

We apply J-DFA to the specific case of faults injected on the state during the last round of AES and we successfully compare our results with the classical approach in different attack conditions. In particular, we relax the knowledge of the attacker and show that J-DFA still allows to find the correct solution.

This part is organized as follows. In Chapter 9 we introduce fault attacks and fault injection techniques. Then we describe the AES and present the state-of-the-art on DFA against AES. In Chapter 10 we present our attacks targeting the sequence of operations. In Chapter 11 we present our results on the impact that the knowledge on the fault model has on attacks. Finally, in Chapter 12 we present the J-DFA.

9 Preliminaries on DFA and AES

The effect of faults on electronic devices has been studied since the 1970s. May and Woods of Intel Corporation noticed how elements present in packaging material produced radioactive particles that caused faults in chips [MW79]. Specifically, they observed that the uranium and thorium residues in the packaging of the 2017-series 16-KB DRAMs emitted α particles during their radioactive decay. These particles created a charge in sensitive chip areas causing random single-bit errors. Even if these elements were present in only two or three parts per million levels in package materials, this concentration was sufficient to affect behavior of dynamic RAMs and CCDs.

Following research included the analysis of the interaction of cosmic-ray particles with computer memories [Zie79]. The effect of cosmic rays is very weak at ground level due to the earth's atmosphere, but it becomes more evident as altitude increases. Moreover, the more RAM a computer has, the higher the chance of a fault is. This has caused an increasing interest in the subject by organizations such as NASA and Boeing.

Subsequently, different fault induction methods have been discovered and considerable effort has been dedicated to improve the resistance of electronic devices operating in extreme conditions.

When the device is performing a cryptographic operation, the misbehavior caused by fault induction can be used to retrieve secret information. Indeed, if the attacker has physical access to the device, she may deliberately induce faults to make it malfunction and use the erroneous result to extract the secret stored on it.

The first application of faults to attack cryptographic systems was presented by Boneh et al. in 1997 [BDL97]. The authors show how one erroneous RSA signature allows the attacker to efficiently factor the RSA modulus with high probability, when the RSA implementation is based on the Chinese Reminder Theorem (CRT). In the same work the authors also show how to attack other RSA implementations, Rabin's signature scheme, the Fiat-Shamir identification scheme and Schnorr's identification scheme, with few bit flips.

Subsequently, the idea of applying faults to attack implementations of cryptographic algorithms was extended to other schemes. Bao et al. presented attacks against DSS and other signature schemes [BDH+97]. Biehl et al. presented two types of attacks on elliptic curve cryptosystems [BMM00] which were later refined by Ciet and Joye [CJ05]. Zheng and Matsumoto presented attacks on ElGamal signature scheme [ZM06].

The examples above all target public key systems, but following the same idea attacks have been extended to symmetric ciphers. Biham and Shamir presented an attack on DES which uses 200 erroneous ciphertexts obtained by single-bit faults [BS97]. The technique introduced against block ciphers is referred to as Differential Fault Analysis (DFA), since it exploits methods of differential cryptanalysis. Indeed, it is based on the analysis of the differences between correct and faulty ciphertexts in order to obtain information on the secret key. DFA attacks have been presented against several symmetric encryption schemes, such as DES and triple-DES [BS97, Hem04], AES [BS03, Gir03, PQ03, TM09, MSS06, PMC+11], CLEFIA [CWF07, TF08a], Camellia

[ZWXF09, ZW09, ZW10], ARIA [LGL08], SMS4 [ZW06, LG08, LGW09, LSLY11], RC4 [BGN05], Trivium [HR08, HGL09] and so on.

Any DFA consists in two main phases. The first phase involves all the steps necessary to fault injection and result collection. Namely, all measurements and analysis of the device and algorithm under attack, to identify the execution time and the location where to inject the fault. It comprises also the execution of the algorithm and the collection of results. The second phase consists in the analysis of the collected results. It mainly involves techniques from differential cryptanalysis, but also from statistics. In this thesis we focus on the second phase. We replace the first phase by software simulations.

This chapter is organized as follows. In Section 9.1 we recall some of the most common techniques used to inject faults in electronic devices, while in Section 9.2 we recall some countermeasures to thwart faults. In Section 9.3, we describe the AES algorithm and finally, in Section 9.4 we recap the state of the art on DFA against the AES.

9.1 Fault injection Techniques

There are many different fault induction methods available, allowing to obtain different effect. Some examples are:

- **Power spikes**: circuit manufacturers define upper and lower voltage thresholds within which their circuits will function correctly. A variation in power supply voltage of much more than the specified tolerance could cause the device to not work properly. So, power spikes injected during execution may cause a processor to misinterpret or skip instructions. They can be applied to mount so called *code change attacks*, whose aim can be to make conditionals to work improperly, loop counters to be altered and arbitrary instructions to be executed. They are non-invasive attacks, which do not require physical opening or chemical preparation of the chip. Practical fault attacks using power spikes are presented in [ABH+02, KQ07, SH08a].

- **Clock glitches**: similarly to power supply, a device tolerates deviations from the standard clock period within a certain range. Glitches can be intentionally injected to cause misbehavior. These may cause data being misread, when the circuit tries to read a value from the data bus before the memory had time to latch out the asked value, but they can also cause an instruction jump, when the circuit starts executing a new instruction before the microprocessor finished executing the previous one. Examples of application of clock glitches to alter the execution of an algorithm are given in [AK96, QLLL17].

- **Temperature variations**: circuits have a temperature range within which they function correctly. An adversary may vary the temperature using an alcoholic cooler until the chip exceeds the threshold bounds. When conducting temperature attacks on smartcards 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 temperature to a value where write operations work but reads do not (or the other way around), a number of attacks can be mounted [BECN+04].

- **White Light**: all electric circuits are sensitive to light due to photoelectric effects. If a circuit is exposed to intense light for a brief time period, the current induced by photons can be used to induce faults. This is an inexpensive mean of fault induction [Gir03].

- **Laser**: laser can reproduce a wide variety of faults and can be used to simulate faults induced by particle accelerators. The effect obtained is similar to white light but the advantage of a laser over white light is that it allows to precisely target a small circuit area. Examples are given in [SA02, vWWM11].

- **X-rays and ion beams**: high-energy hard X-ray sources may have enough energy per particle to interact with DRAM circuitry [GA03]. They have the advantage of allowing the implementation of fault attacks without necessarily de-packaging the chip. If applied successfully they can flip single bits in memory [Col02].

- **Eddy currents**: placing the device in an external electromagnetic field can induces eddy currents on the surface of the chip. These may influence transistors and memory cells since they may change the threshold voltage of the transistor, such that it cannot be switched anymore. Eddy currents can also be used to heat a material in a uniform way, so it can also be used to induce heat faults. Eddy currents can be used to induce faults very precisely, such that individual chosen bits can be set or reset [QS02].

9.2 Countermeasures

Since faults have been identified as a problem for cryptographic implementations, several countermeasures have been developed and deployed. They help circuits to avoid, detect and/or correct faults. Both hardware and software countermeasures can be adopted. There are indeed different levels where the countermeasure can be inserted: protocol level, cryptographic primitive level, arithmetic level or physical level. Depending on the level to be protected, different types of countermeasures can be adopted. The most commonly deployed are the following:

- **Detectors**: are used to detect variations in physical quantities. They can identify changes in the gradient of light, or can detect variations in the applied voltage and continuously check that it is within the circuit's tolerance bounds. Or still, they can be used to ensure that the clock frequency is not outside the expected range.

- **Active shields**: are metal meshes that cover the entire chip and data are continuously passing through them. If there is a disconnection or modification of this mesh, then the chip will not operate anymore. This is primarily a countermeasure against probing, but it also helps protecting against fault injection as it makes the location of specific blocks in a circuit harder.

- **Hardware redundancy**: hardware blocks can be duplicated and the consistency between the two instances can be tested by a comparator. When they do not match, an alert signal is asserted. An alternative approach considers that each hardware block is duplicated at least thrice. Then the comparator can find mismatch between results and assert the proper error. In this case it is also possible to correct the fault through a majority vote. So, it permits error detection and correction, while the basic approach allows only detection. Similar methods are based on complementary duplication. Namely, one block contains the real data and the second block contains its complementary. Multiple duplications can also be considered. The idea behind this approach is that it is very difficult to inject multiple faults with complementary effects.

- **Variable redundancy**: similar to hardware redundancy but in software.

- **Time redundancy**: consists in processing each operation twice and comparing results [AN00]. If each operation is executed more than twice, then the error may be corrected through a majority vote. A possible modification of these methods, consists in re-computing with swapped or switched operands. In the first case, one block computes the operation on original data, while its duplicate (or more than one) computes the result with the operands' little endian and big endian bits swapped. In the second case the duplicate block computes the result on operands shifted by a number of bit positions. The result is re-swapped or re-shifted and compared to the result of the first block to detect potential faults [PF82]. All these techniques can be implemented both in hardware and in software.

- **Bus and memory encryption**: let p be a hardwired keyed permutation and f a simple hardwired block-cipher. An ephemeral key k is generated upon power-on. When the microprocessor wants to write the value m at RAM address i, the system stores $v = f_k(m, i)$ at address $h_k(i)$. When the microprocessor requires the contents of address i, the system computes $h_k(i)$, retrieves v from address $h_k(i)$, decrypts it as $m = f_k^{-1}(v, i)$ and returns m to the microprocessor. This makes targeting a specific memory cell useless, since subsequent computations with identical data use different memory cells.

- **Execution Randomization**: in this case the order in which operations are executed is randomized within the algorithm. Or dummy random cycles are introduced during code processing. Hence, it becomes difficult to predict what the device is doing at any given cycle and thus when to inject the fault to alter the target operation or data.

- **Redundancy Mechanisms**: such as Hamming codes [LCC+00], checksums and error correction codes [PWGV02]. The typical example being checksums attached to each machine word in RAM or EEPROM to ensure integrity.

9.3 The Advanced Encryption Standard

The Advanced Encryption Standard (AES) is a symmetric scheme adopted by the U.S. government as a standard for the encryption of sensitive information [NIoSTN01].

Originally named Rijndael by its authors Daemen and Rijmen, the AES is a substitution-permutation network. Namely, this block cipher operates on a fixed-length block of bits with a transformation that consists of permutations and substitutions. Both the encryption and decryption algorithms take as input a 128-bit block and a key of size 128, 192 or 256 bits and output a 128-bit block. The transformation applied to the block consists of $n = 10, 12$ or 14 (w.r.t. the key size) repetitions of a round function, which consists of invertible steps, and the initial addition with the key.

The steps of the round function are AddRoundKey, SubBytes, Shiftrows and MixColumns. The final round consists of SubBytes, ShiftRows and AddRoundKey operations (namely, no MixColumns is operated).

The 128-bit block is managed as a 4×4 matrix of bytes, named the *state*. Each byte of the state is seen as an element of the finite field $\mathbb{F}_{2^8}$ defined by the Rijndael's polynomial $x^8+x^4+x^3+x+1$.

A schematic representation of the AES transformation is given in Figure 9.1. We denote

- by P the plaintext;
- by C the correct ciphertext and by C^* the faulty ciphertext;
- by S^j the temporary state result after the j-th round;
- by K the AES secret key and by K^j the j-th round key.
- by S_{ij} and K_{ij} the byte in the i-th row and j-th column of S and K respectively.

9.3.1 AddRoundKey step

Each byte of the state is xored with the corresponding byte of a 128-bit round key. Each round key is derived from the original secret key using Rijndael's key schedule (see below) and it is the same size as the state.

$$\underset{\text{state}}{\begin{array}{|c|c|c|c|}\hline b_{00} & b_{01} & b_{02} & b_{03} \\ \hline b_{10} & b_{11} & b_{12} & b_{13} \\ \hline b_{20} & b_{21} & b_{22} & b_{23} \\ \hline b_{30} & b_{31} & b_{32} & b_{33} \\ \hline \end{array}} \oplus \underset{\text{round key}}{\begin{array}{|c|c|c|c|}\hline k_{00} & k_{01} & k_{02} & k_{03} \\ \hline k_{10} & k_{11} & k_{12} & k_{13} \\ \hline k_{20} & k_{21} & k_{22} & k_{23} \\ \hline k_{30} & k_{31} & k_{32} & k_{33} \\ \hline \end{array}} = \underset{\text{new state}}{\begin{array}{|c|c|c|c|}\hline b'_{00} & b'_{01} & b'_{02} & b'_{03} \\ \hline b'_{10} & b'_{11} & b'_{12} & b'_{13} \\ \hline b'_{20} & b'_{21} & b'_{22} & b'_{23} \\ \hline b'_{30} & b'_{31} & b'_{32} & b'_{33} \\ \hline \end{array}}$$

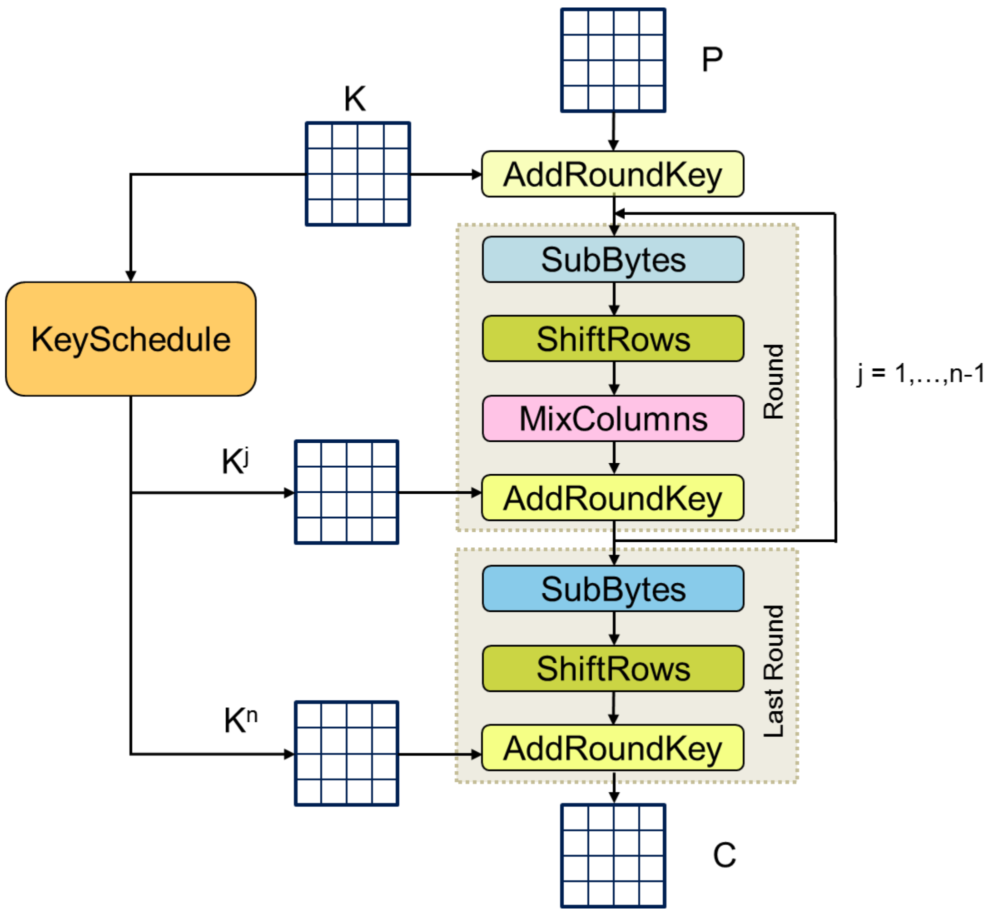


Fig. 9.1: *Schematic representation of the AES transformation.*

9.3.2 SubBytes step

This is the unique non-linear transformation of the cipher. It operates independently on each byte of the state and is composed by two transformations:

- inversion over $\mathbb{F}_{2^8} = \mathbb{F}_2[x]/x^8 + x^4 + x^3 + x + 1$,
- an affine transformation over $\mathbb{F}_2^8$.

In other words, the inverse of each byte is computed over $\mathbb{F}_2[x]/x^8+x^4+x^3+x+1$ and then, considering $\mathbb{F}_{2^8}$ as a vector space over $\mathbb{F}_2$, is transformed by the affine transformation:

$$\begin{bmatrix} 1&0&0&0&1&1&1&1\\ 1&1&0&0&0&1&1&1\\ 1&1&1&0&0&0&1&1\\ 1&1&1&1&0&0&0&1\\ 1&1&1&1&1&0&0&0\\ 0&1&1&1&1&1&0&0\\ 0&0&1&1&1&1&1&0\\ 0&0&0&1&1&1&1&1 \end{bmatrix} \begin{bmatrix} x_0\\ x_1\\ x_2\\ x_3\\ x_4\\ x_5\\ x_6\\ x_7 \end{bmatrix} + \begin{bmatrix} 1\\ 1\\ 0\\ 0\\ 0\\ 1\\ 1\\ 0 \end{bmatrix}$$

In practice this computation can be replaced by a look up table, named Rijndael's S-box (Table 9.1). Namely, instead of computing the transformation on each byte in each round, the byte can be replaced by the corresponding byte in the S-box, since the result of the transformation is fixed over $\mathbb{F}_2[x]/x^8 + x^4 + x^3 + x + 1$.

	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f
00	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
10	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
20	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
30	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
40	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
50	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
60	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
70	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
80	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
90	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
a0	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
b0	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
c0	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d0	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e0	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f0	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Table 9.1: *The S-box used in AES, in hexadecimal notation. For each byte the multiplicative inverse in $\mathbb{F}_2[x] = x^8+x^4+x^3+x+1$ is computed and then the affine transformation is applied.*

9.3.3 ShiftRows step

Bytes in each row are cyclically shifted to the left by a certain number of places. In particular, the bytes of the row i are shifted to the left by i positions.

$$\begin{array}{|c|c|c|c|}\hline b_{00} & b_{01} & b_{02} & b_{03} \\ \hline b_{10} & b_{11} & b_{12} & b_{13} \\ \hline b_{20} & b_{21} & b_{22} & b_{23} \\ \hline b_{30} & b_{31} & b_{32} & b_{33} \\ \hline \end{array} \longrightarrow \begin{array}{|c|c|c|c|}\hline b_{00} & b_{01} & b_{02} & b_{03} \\ \hline b_{11} & b_{12} & b_{13} & b_{10} \\ \hline b_{22} & b_{23} & b_{20} & b_{21} \\ \hline b_{33} & b_{30} & b_{31} & b_{32} \\ \hline \end{array}$$

9.3.4 MixColumns

The four bytes of each column are combined using a linear transformation. In particular, each column is multiplied by the matrix

$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \tag{9.1}$$

Notice that all operations are performed in the Rijndael's field.

The MixColumns step can also be seen as follows. Each column is considered as a polynomial over $\mathbb{F}_{2^8}$ and is multiplied modulo x^4+1 with the polynomial $c(x) = \{03\}\,x^3 + \{01\}\,x^2 + \{01\}\,x + \{02\}$. The coefficients of $c(x)$, as the coefficients of the matrix above, are given here in hexadecimal notation.

9.3.5 Key schedule

The key schedule is used to expand the secret key into a number of round keys that are used during the AddRoundKey step. The number of keys produced by the key schedule depends on the number of rounds executed, namely depends on the variant of AES.

A representation of the key schedule for a 128-bit key is given in Figure 9.2. It makes use of three operations that operate on a 32-bit word:

- Rotate: rotates the word so that the highest byte becomes the lowest;
- SubBytes: applies the subBytes of the AES round function to each byte of the word;
- Rcon: bitwise sums the first byte to the constant 2^{i-1}, where the constant is computed in Rijndael's field.

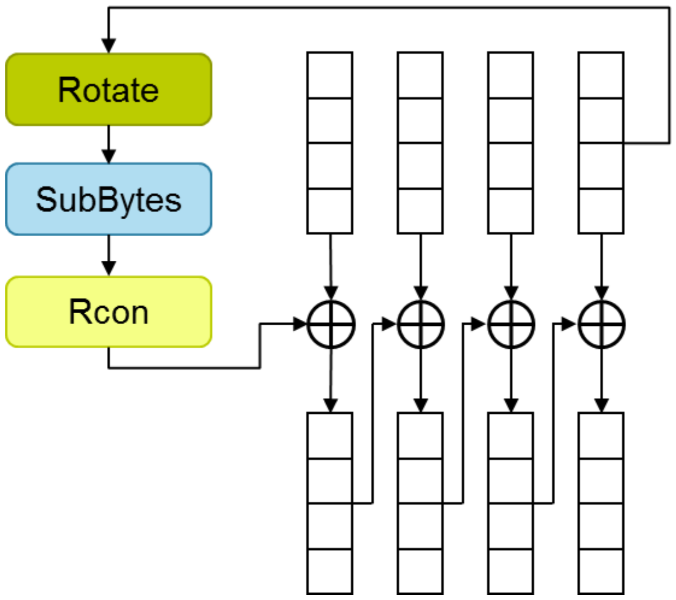


Fig. 9.2: *One round of the key schedule for AES-128. The four 32-bit words are given by the 128 bits of the key.*

9.4 Previous DFA Results on AES

Since the AES has been chosen to be the successor of the DES, several DFA attacks have been presented against it.

Most of them are based on the injection of faults in the AES state during the encryption process or on the key during the key-schedule computation. Very few examples have been presented where the fault causes the reduction of the number of round iterations.

One of the first example of DFA against AES-128 was presented by Blömer and Seifert [BS03]. Their idea can actually be applied to any cipher and it assumes that the attacker can force to 0 the value of a chosen bit. Let $\mathbf{0}$ denote the all-zero plaintext. First, an encryption without faults is executed, obtaining the ciphertext C. After the first key addition, we have $S^1 = \mathbf{0} \oplus K$. So, for each byte of the state the equality $S_{ij} = K_{ij}$ holds. Then, a second encryption is executed and a fault is injected on a selected bit at the input of the first round, so that it is set to 0. The output ciphertext C^* is collected. Comparison between correct and faulty ciphertexts allows to retrieve the value of a bit of the key. Indeed, if $C = C^*$, then setting the value of the target bit to 0 did not change the value of this bit, which means that the corresponding bit of the key is equal to 0, since $S_{00} = K_{00}$. On the contrary, C^* means that setting the value of the target bit changed the value of this bit and thus we can conclude that the corresponding bit of the key is equal to 1. By applying the same technique to the other 127 bits of the state, the attacker can recover the secret key.

Notice that if the attacker can force to 0 the value of a chosen bit, she can in principle decide to force a bit of the key and mount a similar attack in the following way. Two encryptions with the same plaintext are executed. During the first encryption no faults are induced. During the second encryption one bit of the key is forced to 0 before the first key addition. Then, the obtained faulty ciphertext is compared with the correct ciphertext output of the first encryption. If the two ciphertexts are equal then the bit of the key has not been affected by the fault injection. This means that the bit was originally 0. On the contrary, if the two ciphertexts are different, then the bit of the key was originally equal to 1.

Giraud proposed two attacks on AES-128 [Gir03]. The first attack assumes a single-bit fault on the state at the input of the ninth round and requires in average 35 pairs of correct and faulty ciphertexts to retrieve the whole key. A detailed description of this attack will be given in Section 11.1.1. The second attack exploits multiple errors during the encryption and the key schedule. In a first step the last 4 bytes of K_9 are obtained by exploiting single-byte faults introduced on K^9, just before the computation of K^{10}. Then other 4 bytes of K^9 are obtained by exploiting byte-faults introduced on K^8, just before the computation of K^9. Finally, the AES key is obtained by introducing a fault on S^8 at the input of the 9^{th} round and by using the 8 bytes of K^9 retrieved in the first two steps. In this case the secret key can be recovered using 31 pairs of correct and faulty ciphertexts.

Piret and Quisquater show that an attacker can extract a 128-bit key from eight pairs of correct and faulty outputs [PQ03]. Their attack considers a fault model where a random byte fault is injected at the beginning of the 9th round. The corresponding differential at the input of the last round has four non-zero bytes. The corresponding four bytes of the round key can be retrieved with 2 pairs of correct and faulty ciphertexts in average. A total of eight pairs allows to retrieve the whole round key. Once the round key is known, the secret key can be computed by the inverse of the key schedule. A detailed description of the attack is given in Section 11.1.2.

Tunstall et al. proposed an enhancement of the DFA method of Piret and Quisquater, using a well-placed fault of one-byte [TM09]. As with previous attack methods, the attacker induces a fault on a byte at the input of the 8th round. The corresponding differential at the input of the 9th round has four non-zero bytes, one for each column of the state. The scenario is thus similar to inject four bytes in Piret and Quisquater's method. With only one pair of correct and faulty ciphertext, the secret key is recovered. This DFA attack needs the least number of pairs of correct and faulty ciphertexts among the DFA methods which injects a fault during the encryption process.

Moradi et al. [MSS06], proposed a more generalized fault attack by considering two different fault models. In the first, the authors consider that one out of four bytes in the same column are corrupted at the input of the 9th round. The set of faults thus comprise also the faults admitted by Piret and Quisquater. In the second model the authors consider that all four targeted bytes are corrupted. For the first fault model the attack requires around four faulty ciphertexts whereas in the second fault model the attack requires around 1500 faulty ciphertexts. A description of this attack will be given in Section 11.1.3.

A significant research has been also conducted on the AES key schedule. Chen et al. [CY03], improved Giraud's attack [Gir03] and showed how the AES-128 key can be retrieved by inducing faults in 9-th round key and requires less than 30 faulty ciphertexts.

Peacham et al. [PT] proposed an attack where a fault is induced while the 9th round key is being generated. Therefore, the induced fault propagated to the 10th round key. Peacham's attack required only 12 faulty ciphertexts to retrieve the AES-128 secret key.

Takahashi et al. [TFY07] proposed a generalized attack that required only two faulty ciphertexts to reduce the number of key hypotheses for a AES- 128 secret key from 2^{128} to 2^{48}. Other variants of this attack were presented. One allows to reduce the number of key hypotheses to 2^{16} using four faulty ciphertexts, the other allows to determine the secret key, using seven faulty ciphertexts.

Kim et al. [KQ08] proposed an improved attack on AES-128 key schedule which requires only two faulty ciphertexts to reduce the key space to 2^{32}. Kim also proposed a different attack based on inducing single byte fault in the first column of 8th round key. This attack requires two faulty ciphertexts to uniquely determine the secret key [Kim12].

All the above attacks target AES-128, but a number of DFA attacks have been presented also on AES-192 and AES-256 targeting the encryption process or the key schedule.

Initially, it was assumed that the attack proposed by Piret and Quisquater could be extended to the two larger versions of the AES with little modification. However, this assumption has been shown to be wrong.

In 2009, Li et al. [Kim12] proposed a complete attack on AES-192 and AES-256. This attack requires 16 or 3000 faulty ciphertexts depending on the fault model. Subsequently, many attacks were proposed on AES-192 and AES-256 [GT, Kim10, TF10]. One of the most powerful among these attacks is an attack proposed by Kim [Kim10], which only requires two faulty ciphertexts to uniquely determine the AES-192 key and three faulty ciphertext to retrieve the AES-256 key.

Floissac et al. first proposed an attack on the AES-192 and AES-256 key schedule [FL10]. They use a single byte fault model where a fault is induced in the 10th and 12th round key for different instantiations of the block cipher. In both the cases their attack required 16 faulty ciphertexts to retrieve the secret key. This attack was improved by Kim [Kim12], who proposed an attack that requires between four and six faulty ciphertexts to uniquely determine a AES-192 secret key and four faulty ciphertexts to uniquely determine a AES-256 secret key.

The attacks listed here are only a part of the DFA attacks available in literature. It is worth noticing that all these attacks are based on a fault model which requires to alter a portion of the state or the (round) key. Very few attacks have been presented, which alter the sequence of operations within the algorithm.

Choukri and Tunstall show how to exploit a fault that reduces the number of rounds of an AES to one [CT05]. The cryptanalysis of the resulting algorithm is simple and only requires two pairs of plaintext/ciphertext. The idea is the following. Let P_1 and P_2 be two different plaintexts. If the AES algorithm is reduced to one round, then the corresponding ciphertexts differ by

$$C_1 \oplus C_2 = \text{MixColumns}(\text{SubBytes}(P_1 \oplus k) \oplus \text{SubBytes}(P_2 \oplus k)). \tag{9.2}$$

Notice that the ShiftRows function is not taken into account as it is a bytewise permutation, while the last AddRoundKey is ignored as its effect is removed by xoring the two ciphertexts together.

Equation 9.2 can be rewritten as

$$\text{MixColumns}^{-1}(C_1 \oplus C_2) = (\text{SubBytes}(P_1 \oplus k) \oplus \text{SubBytes}(P_2 \oplus k)). \tag{9.3}$$

Since P_1 and P_2 are known, this equation can be evaluated for each byte of the state considering all the possible key guesses (of course, the guess is made only on the key byte that is xored with the target byte). This can be seen as calculating a table of differentials which leads to two different hypotheses for each byte of the key. This results in an exhaustive search of 2^{16} possible key candidates.

However, Choukri and Tunstall consider an implementation of the AES in which the last round is included in the main loop procedure to facilitate their DFA on the target. If the algorithm is implemented with the last round outside the main loop, as recommended in the standard document [NIoSTN01], then reducing the number of rounds to one results in: the initial key addition, the first round and the last round. This scenario is analyzed by Park et al. in [PMC$^+$11]. The authors show that the 128-bit key can be retrieved by obtaining 10 ciphertexts from the reduced-round AES.

10

DFA against AES tampering with the instruction flow

The most known DFA attacks against AES imply fault models where the fault injection modifies an intermediate state, a round key, or the total amount of round iterations.

In this chapter, we present four DFA attacks that exploit faults causing a misbehavior of the process during the final round. In particular, we assume that one of the steps composing the AES final round function (i.e. the AddRoundKey, the SubBytes and the ShiftRows operations) is not executed or that the MixColumns operation is executed also in the final round.

We want to emphasize that the attacks we present in this chapter are currently theoretical. The purpose of these results is to demonstrate that errors induced on the sequence of operations are a concern for implementations of cryptographic algorithms. Thus countermeasures against fault attacks should cover and protect also those parts of the design that are in charge of controlling the instruction flow.

The chapter is structured as follows. In Section 10.1, we describe how the instruction flow can be affected both in software and hardware implementations. In Section 10.2, we present our attacks against the four steps composing the AES round function. Finally, in Section 10.3, we discuss how the effectiveness of some countermeasures against faults is affected by our attacks.

10.1 Tampering with the instruction flow

Most of the known DFA attacks require to corrupt a small portion of the message (or the key) at a given cycle. Then, depending on the injection technique, the target of the attack is either the storage for the data or the computing units while processing the data.

But effective fault attacks can also be mounted by tampering with the sequence of the instructions executed by the cryptographic algorithm.

A practical attack of such a kind has been demonstrated against a software implementation of AES in [CT05]. The authors were able to skip the instruction that drives the repetition of the rounds, effectively obtaining the AES internal state after only one round. In particular, a software implementation of AES on a smart card is considered where the round steps are sequentially executed and a jump condition manages the round counter. With a single glitch on the power supplied to the smart card, they are able to skip exactly the “conditional jump”, with the effect of reducing the number of rounds to one.

A first step for the attack consists in the characterization of the device by determining the configuration that causes the proper glitch. This can be achieved by conducting several experiments where the clock period, the applied voltage and the duration of the glitch are varied. In a second phase, the portion of the code where to inject a fault must be determined. This can be achieved by examining the executed code in detail and by estimating the length of interesting instructions in terms of clock cycles. A third phase consists in determining when the target instruction is executed by the device. By measuring the current consumption of the smart card, it is possible to observe a pattern that repeats itself nine times and a shorter final pattern due to the absence of the MixColumns operation in the final round.

Once the right position and the size of the glitch have been found, the power supply is interrupted or lowered. This results in operations to be skipped.

For a detailed description of the experiment on a specific device, we refer to [CT05], where concrete parameter settings and time costs are provided.

Another example of attacks based on tampering the instruction flow is provided in [SH08b], where the target of the attack is a software implementation of RSA. In this latter case the injected faults aim at skipping specific operations within the algorithm, similarly to the attacks we introduce in the next sections.

These works demonstrate that attacks based on the alteration of the sequence of instructions (even by just skipping a single instruction) are indeed a concern for software implementations of cryptographic algorithms. But the same reasoning can be applied to hardware implementations, too.

10.1.1 Hardware implementations

A hardware implementation of the AES algorithm can be split into three main parts, with different logical functionality: the storage for the data, the function that updates the data, and the controller.

The storage for the data holds the message to process and all the intermediate values during the computation. It also contains the key. It can be implemented by registers or RAM cells. In the specific case of the AES algorithm, it stores the secret key (including all the round keys) and all the intermediate states during the rounds. Usually, there is no need to keep all these intermediate values together and hence the result of a round replaces the previous state within the same memory cells.

The function that updates the data refers to all the combinational logic that is needed to compute the next values starting from the current data. It implements a basic functionality that is applied several times to the data in order to compute the final result. About AES, such combinational logic implements the round function (SubBytes, ShiftRows, MixColumns, KeyAddition) and the key schedule; both are applied 10, 12, 14 times to the data, depending on the key size.

The controller represents all the additional gates that manage the execution and allow to properly drive the data through the computation. It is composed by a mix of registers and combinational logic, and in the case of AES is everything else that does not fall in the two previous classes (e.g. the counter for the rounds).

The presented logical split is also taken into account when fault attacks are applied to a hardware implementation of AES. Since most of the known attacks require to corrupt a small portion of the message (or the key) at a given cycle, the target of the attack is either the storage for the data or the update function, depending on the injection technique applied.

But from the technology point of view, there is no real distinction between registers storing data and registers storing the internal state of the controller (e.g. round counter vs AES state). The same is true for the gates implementing the round function and the ones of the control signals driving such round function. This means that any technique for fault injection which is able to alter the data is also able in principle to alter the control signals. It follows that faults on controller are also a concern and any countermeasure against fault attacks should cover and protect in the same way the whole design, including the controller.

In the specific case of AES, which is a relatively simple algorithm, it is fair to assume that the internal state of the controller requires few tens of bits to manage all the operations (e.g. round count, encryption or decryption, key size, etc.). This depends on the specific implementation, but still it is compatible with the amount of faulted bit required by most of the common attacks against AES (e.g. single bit, single byte).

A fault injected in the controller can lead to an attack in two ways. First of all, the fault can manifest as an equivalent fault on the data, where one of the several known attacks can apply [Gir03, PQ03, TM09, Muk09]. For instance, executing the wrong operation in an implementation that processes one byte at a time will produce at some point a (randomly) wrong byte into the data storage.

The second scenario is when the fault manifests as an alteration of the instruction flow. One of the few known examples of this kind are the class of attacks based on a reduced number of executed rounds.

The attacks that we present in Section 10.2 also fall in this scenario, since they are based on the alteration of the instruction flow or the control logic.

Faults that alter the sequence of instructions generate very peculiar behaviors. Our attacks can leverage such specific properties, observed on the faulty results, to improve the efficiency, compared with attacks simply considering random fault models.

10.2 DFA against the round steps

In this section, we present in theory four specific attacks that target the last round of AES-128.

The fault injection we consider in this section causes the process either to skip one of the transformations of the last round or to execute the MixColumns transformation also during the last round.

In each attack we exploit the difference between correct and faulty ciphertexts to obtain the value of the temporary state just before the attacked transformation. Then, we derive the last round key K^{10}, by using the correct ciphertext, and finally we get the secret key K, by inverting the KeySchedule transformation on K^{10}.

Notice that, similar attacks can be conducted against the decryption function. In this case, the output of the execution is the plaintext and the round key used in the last round is exactly the secret key. So, in this case the attack will compare correct and faulty plaintexts, obtaining an intermediate state what will be used to get the last round key, namely the secret key. No application of the KeySchedule is needed in this case.

Before presenting the attacks, recall that, by definition, the correct ciphertext is

$$C = ShiftRows(SubBytes(S^9)) \oplus K^{10} \tag{10.1}$$

where S^9 represents the AES state at the beginning of the last round.

10.2.1 Altering the execution of SubBytes

In this attack we consider a fault that causes the execution to skip the last SubBytes operation.

Depending on the granularity of the implementation, the AES process can handle one or more bytes per clock cycle. Therefore, the fault injection can cause the SubBytes transformation to be not executed either on a single byte or on the whole AES state (or on any combination in between).

Since all the bytes are computed independently of each other, the attack applies in any case.

In other words, if the process handles a single byte per clock cycle, then the fault injection affects only one byte of the state and we conduct the analysis on that byte to recover the corresponding byte of the key. To obtain the whole key, we need to repeat the attack for all the sixteen bytes and therefore we need to generate more pairs of correct and faulty ciphertexts.

On the contrary, if the process handles a k-byte word per clock cycle, then a single injection fault affects k bytes of the state. In this case, we can conduct the analysis on each of these bytes independently and obtain the corresponding bytes of the key.

For the sake of simplicity, we now consider that the fault injection causes the skipping of the SubBytes transformation for the whole state. This leads to a faulty ciphertext

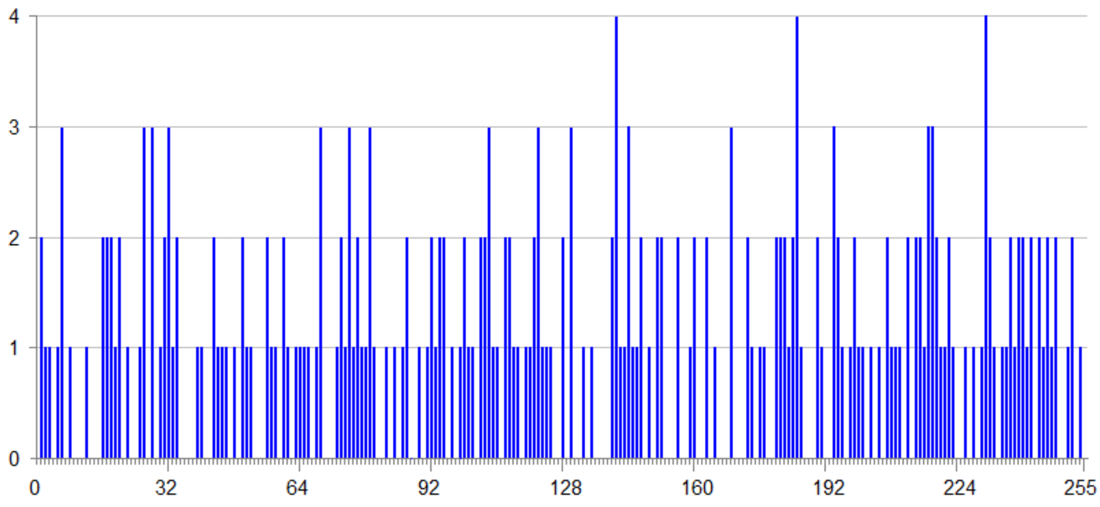


Fig. 10.1: *For any $\delta \in [0, \dots, 255]$, the number of bytes x such that $x \oplus SubBytes(x) = \delta$.*

$$C^* = ShiftRows(S^9) \oplus K^{10} \tag{10.2}$$

and to an observed difference between correct and faulty ciphertexts

$$\varDelta = C \oplus C^* = ShiftRows(SubBytes(S^9) \oplus S^9) \tag{10.3}$$

We can now conduct the following analysis on each byte of the state independently.

Let us denote by i' the position of the i-th byte of the state after applying the ShiftRows transformation. Hence, for any i, we have

$$C_{i'} = SubBytes(S_i^9) \oplus K_{i'}^{10}. \tag{10.4}$$

and

$$C_{i'}^* = S_i^9 \oplus K_{i'}^{10} \tag{10.5}$$

It follows that the observed difference between correct and faulty bytes is

$$\varDelta_{i'} = C_{i'} \oplus C_{i'}^* = SubBytes(S_i^9) \oplus S_i^9 \tag{10.6}$$

Before starting the analysis, it is possible to precompute for any byte $x \in [0, 255]$ the value:

$$\delta = x \oplus SubBytes(x) \tag{10.7}$$

It is then possible to fill out a table where for each δ the bytes x that satisfy (10.7) are given. Figure 10.1 shows the number of such x for each possible δ. We can notice that several δ's can never occur and that for the other δ's the number of occurrences is very small. In particular, in the worst case (when $\delta \in \{\texttt{0x8D}, \texttt{0xB9}, \texttt{0xE7}\}$) it is equal to 4.

It follows that, given $\varDelta_{i'}$ only a limited number of candidates for S_i^9 satisfy (10.6).

Consequently, also the set of possible candidates for $K_{i'}^{10} = C \oplus SubBytes(S_i^9)$ has been considerably reduced.

It follows that, in the worst case the search space for K^{10} has been reduced to 32 bits (2 bits for each byte) and the attacker can obtain K with an exhaustive search.

Alternatively, with other pairs of correct and faulty ciphertexts (at most three), obtained using different plaintexts and the same key K, only the right value of $K_{i'}^{10}$ is expected to appear in the set of key candidates of each execution.

In particular, if the implementation allows to skip the last SubBytes transformation for all the sixteen bytes of the state with a single fault, then two or three pairs are sufficient for obtaining the whole key. Alternatively, two or three pairs are needed for each word handled by the implementation.

Example 10.1. Suppose that the secret key of the device we want to attack is such that $K_0^{10} =$ `0xD0` and that we want to obtain it.

We choose a plaintext and execute the encryption on the device. Then, using the same plaintext, we re-execute the encryption, but inducing a fault to skip the last SubBytes operation.

Suppose that the obtained outputs are such that $C_0 =$ `0x25` and $C_0^* =$ `0xA7`. Consequently the observed difference is $\Delta_0 = C_0 \oplus C_0^* =$ `0x82` and, according to Fig. 10.1, only three candidates for S_0^9 are admitted. Specifically, the set of candidates for S_0^9 is {`0x77`, `0x7D`, `0xA6`}. It follows, that the set of candidates for K_0^{10} is {`0xD0`, `0xDA`, `0x01`}.

Now, we consider another pair of correct and faulty ciphertexts, obtained by using a different plaintext and the same fault. Suppose that the obtained values are $C_0 =$ `0xB1`, $C_0^* =$ `0x08` and $\Delta_0 =$ `0xB9`. It follows that the set of candidates for S_0^9 is {`0x1D`, `0xC1`, `0xD8`, `0xF8`} and consequently, the set of candidates for K_0^{10} is {`0x15`, `0xC9`, `0xD0`, `0xF0`}. The intersection between the two sets of key candidates gives the right value of K_0^{10}, which is exactly `0xD0`. △

Now, we would like to underline the fact that it is possible to modify the fault model, provided that it continues to leak some information on the secret. For instance, we can consider fault models where the SubBytes transformation is executed twice or is replaced by its inverse. The principles of the attack still hold. Actually, as shown below, we can still analyze an equation similar to 10.7, fill out the corresponding distribution table and observe which candidates satisfy the obtained difference between correct and faulty ciphertexts.

10.2.1.1 InvSubBytes/SubBytes

As a variant of the previous attack, we can consider the case where a fault injection causes the inverse of the SubBytes transformation to be executed instead of the SubBytes itself. Such a scenario can happen in hardware implementations that share most of the datapath between the encryption and the decryption functionalities. In these designs the entities for the direct and the inverse SubBytes are both instantiated and a multiplexer selects between the two depending on the control bit that sets encryption or decryption.

The attack is basically the same with the difference that, for any i it is

$$C_{i'}^* = InvSubBytes(S_i^9) \oplus K_{i'}^{10} \tag{10.8}$$

and

$$\Delta_{i'} = C_{i'} \oplus C_{i'}^* = SubBytes(S_i^9) \oplus InvSubBytes(S_i^9) \tag{10.9}$$

For any possible value of $\Delta_{i'}$, the number of occurrences is given in Figure 10.2 and we can observe that in the worst case (for $\delta \in$ {`0x8D`, `0x8F`}) it is 5.

Again, with two or three pairs of correct and faulty ciphertexts, the attacker can obtain the secret key.

10.2.2 Jumping the ShiftRows

Now we present a possible attack when the fault causes the skipping of the last ShiftRows operation. Similarly to Section 10.2.1, this attack applies when the fault injection affects whether a single row or the whole state, since the ShiftRows operation transforms each row independently. The only difference is in the number of faulty ciphertexts needed.

We consider the case where the whole state is affected and use the same faulty ciphertext to analyze all the rows contemporary.

Jumping the last ShiftRows operation gives

$$C^* = SubBytes(S^9) \oplus K^{10}. \tag{10.10}$$

Let us denote by s the temporary state after the last SubBytes transformation, i.e. $s = SubBytes(S^9)$. Then, the observed difference between correct and faulty ciphertexts is

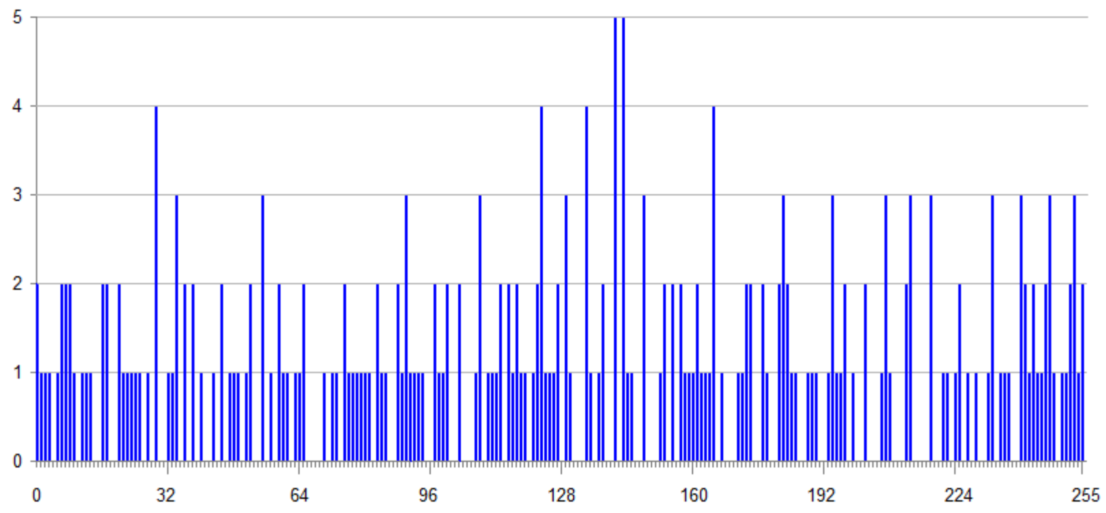


Fig. 10.2: *For any $\delta \in [0,\ldots,255]$, the number of bytes x such that $InvSubBytes(x) \oplus SubBytes(x) = \delta$.*

$$\Delta = C \oplus C* = \begin{bmatrix} 0 & 0 & 0 & 0 \\ s_5 \oplus s_1 & s_9 \oplus s_5 & s_{13} \oplus s_9 & s_1 \oplus s_{13} \\ s_{10} \oplus s_2 & s_{14} \oplus s_6 & s_2 \oplus s_{10} & s_6 \oplus s_{14} \\ s_{15} \oplus s_3 & s_3 \oplus s_7 & s_7 \oplus s_{11} & s_{11} \oplus s_{15} \end{bmatrix} \tag{10.11}$$

By the knowledge of Δ we can restrict the set of possible candidates for the state s and consequently for the last round key K^{10}. Notice that we cannot deduce any information about the first row of the state (and of the key), because the fault injection has no effect over it. On the contrary, for the other three rows we can solve the corresponding linear systems derived from equation (10.11).

- For the second row, we have

$$\begin{bmatrix} 1\,1\,0\,0 \\ 0\,1\,1\,0 \\ 0\,0\,1\,1 \\ 1\,0\,0\,1 \end{bmatrix} \begin{bmatrix} s_1 \\ s_5 \\ s_9 \\ s_{13} \end{bmatrix} = \begin{bmatrix} \Delta_1 \\ \Delta_5 \\ \Delta_9 \\ \Delta_{13} \end{bmatrix} \tag{10.12}$$

  The matrix is singular with rank 3. By guessing a byte, say s_1, the other three bytes can be derived depending on it. Hence, the set of possible candidates for the 4-uple $[s_1, s_5, s_9, s_{13}]$ has dimension 256.

- For the third row, we have

$$\begin{bmatrix} 1\,0\,1\,0 \\ 0\,1\,0\,1 \\ 1\,0\,1\,0 \\ 0\,1\,0\,1 \end{bmatrix} \begin{bmatrix} s_2 \\ s_6 \\ s_{10} \\ s_{14} \end{bmatrix} = \begin{bmatrix} \Delta_2 \\ \Delta_6 \\ \Delta_{10} \\ \Delta_{14} \end{bmatrix} \tag{10.13}$$

  The matrix is singular with rank 2. Hence, the set of possible candidates for the 4-uple $[s_2, s_6, s_{10}, s_{14}]$ has dimension 256^2.

- For the fourth row, we have

$$\begin{bmatrix} 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} s_3 \\ s_7 \\ s_{11} \\ s_{15} \end{bmatrix} = \begin{bmatrix} \Delta_3 \\ \Delta_7 \\ \Delta_{11} \\ \Delta_{15} \end{bmatrix} \tag{10.14}$$

Similarly to the case of the second row, the matrix is singular with rank 3. Hence, the set of possible candidates for the 4-uple $[s_3, s_7, s_{11}, s_{15}]$ has dimension 256.

The search space for the state s, and consequently for the last round key (since $K^{10} = s \oplus C$), has been reduced from 128 to 64 bits (32 bits for the first row, 8 bits for the second and fourth rows and 16 bits for the third row). With a 64-bit effort, the attacker can obtain the whole secret key. Alternatively, with two or three pairs of correct and faulty ciphertexts, the attacker can uniquely retrieve the value of the second, third and fourth rows. The whole key can thus be obtained with a 32-bit effort.

10.2.3 Executing the MixColumns

By definition, in the last round of AES the MixColumns transformation is not executed. By inducing a fault (for instance, by targeting the round counter), we can make the algorithm execute it, obtaining the faulty ciphertext:

$$C^* = MixColumns(ShiftRows(SubBytes(S^9))) \oplus K^{10} \tag{10.15}$$

Let us denote by s the temporary state after the last ShiftRows transformation, i.e. $s = ShiftRows(SubBytes(S^9))$. The observed difference between correct and faulty ciphertexts is

$$\Delta = C \oplus C^* = s \oplus MixColumns(s) \tag{10.16}$$

By considering Equation 9.1 and Equation 10.16, for any column of s we have:

$$\begin{bmatrix} 3 & 3 & 1 & 1 \\ 1 & 3 & 3 & 1 \\ 1 & 1 & 3 & 3 \\ 3 & 1 & 1 & 3 \end{bmatrix} \begin{bmatrix} s_i \\ s_j \\ s_k \\ s_l \end{bmatrix} = \begin{bmatrix} \Delta_i \\ \Delta_j \\ \Delta_k \\ \Delta_l \end{bmatrix} \tag{10.17}$$

The matrix is singular with rank 3. Hence, the set of possible candidates for each column of s has dimension 256. It follows that the dimension of the state space, and consequently of the key space, has been reduced to 2^{32}. With an exhaustive search, the attacker can obtain the whole key. Alternatively, with two pairs in average the attacker can uniquely determine the key.

Similarly to the previous attacks, we can consider a fault injection to affect a single column of the state. In this case, we need to conduct the attack for each column by generating different faulty ciphertexts.

10.2.4 Jumping the AddRoundKey

In this attack, we consider a fault that causes the execution to skip the last AddRoundKey operation. We can consider different scenario, depending on the granularity of the implementation. As told for previous attacks, the fault injection can affect either a single byte or the whole state or any combination in between. We can therefore apply the analysis on the affected bytes independently, since the AddRoundKey operation consists of a bitwise xor.

Suppose that the fault injection affects the whole state. This leads to a faulty ciphertext

$$C^* = ShiftRows(SubBytes(S^9)) \tag{10.18}$$

The attacker can then simply add correct and faulty ciphertexts, obtaining the last round key: $C \oplus C^* = K^{10}$. By inverting the KeySchedule on K^{10}, the secret key K is obtained.

Only one pair of correct and faulty ciphertexts is then needed to conduct the attack.

Alternatively, if the fault affects k bytes of the state, it is possible to obtain the corresponding bytes of K^{10} and the remaining bytes by generating other faulty ciphertexts. Once obtained all the bytes of K^{10} is then possible to apply the inverse of the KeySchedule and obtain the secret key.

10.3 Effectiveness against countermeasures

The presented attacks can impact the effectiveness of some countermeasures that protect AES implementations against fault attacks. In particular, countermeasures based on redundancy are affected. We briefly described such countermeasures in Section 9.2, for a detailed description we remind to [SM12, BNFR12, MSY06, VKS11].

A class of these countermeasures uses coding techniques to add redundancy on the AES computation. In most of the proposed cases error detection/correction codes are applied to the AES state and key only. This means that when an operation is skipped the encoding of the data can be still valid and thus the countermeasure fails in detecting the attack.

Some form of redundancy can be introduced with the round counter, in order to protect the "conditional jumps" and make the attack in [CT05] unfeasible. However, this kind of countermeasure will not be successful against the attacks presented in this work, even if the attacker is using the same resources and techniques of the attacker that reduces the number of rounds.

Another countermeasure is based on the duplication of parts of, or all, the algorithm. This would make the attacks more difficult, since a double fault injection would be necessary. However, such a solution has a significant impact on the performance of the algorithm.

Also the inclusion of random delays in the algorithm makes the attack more difficult, since the detection of target instructions becomes more involved. This results in a more difficult achievement of successful attacks, but it is still possible to design the attack in order to ignore the random effects.

An additional method for protecting against fault attacks consists in using sensors on the microcontrollers to detect fault injections. But, different sensors should be used for different fault injection techniques and this can be an excessively expensive solution for general purpose microcontrollers.

11

The role of the fault model in DFA

Most DFA attacks rely on a specific fault model implied by the attacker. For instance, [PQ03] requires that a single byte is faulted during the second-last round, while [Gir03] requires that a single bit is flipped in the last round.

The chosen fault model impacts the efficiency of the attack: the largest the model is, the bigger the number of required faulty ciphertexts is. Depending on the target cryptosystem and the specific attack, the knowledge of the fault model by the attacker can fundamentally impact the success rate of the attack.

In this chapter we investigate these two aspects in practice. To this end we examine the impact on some DFA attacks against the AES algorithm. We chose three well known attacks that target the intermediate state of AES, but the analysis can be extended to other attacks sharing the same framework.

The chapter is structured as follows. In Section 11.1 we describe three well known DFA attacks against AES-128, from an information theory point of view. In Section 11.2 we show how these attacks become less efficient when the fault model is enlarged. Then, in Section 11.3 we show how the probability to mount a successful attack decreases if the attacker does not know for certain the effect of the fault injection technique she applies. Finally, in Section 11.4 we focus on a specific example and show how the knowledge of the fault model affects the attack effectiveness.

11.1 Information theory of DFA against AES

Most DFA against AES assume that the intermediate state is perturbed in some way by physically faulting the execution. Then the corresponding faulty ciphertext is compared with the correct ciphertext to obtain information on the secret key.

From an information theoretical point of view, the information about the injected fault translates in an equivalent amount of information about the secret key. In particular, the smallest the model is (namely, the smallest the set of acceptable faults is), the smallest the set of key candidates is. Hence, it is fundamental for the attacker to be able to distinguish which kind of fault has occurred.

As introduced in [LGSO10], each attack has a different level of efficiency (i.e. number of experiments to obtain the entire key) and this is directly linked to the underlying fault model considered for the attack.

Following this perspective, all the attacks can be studied and compared taking into account the amount of information on the secret key that the faults provide. Here we give a quick overview of some of the main fault attacks against AES-128 in literature highlighting their properties from the information theory point of view. The analysis can be similarly extended to other attacks.

11.1.1 Giraud's attack

The first DFA against the AES algorithm has been presented by Giraud in [Gir03].

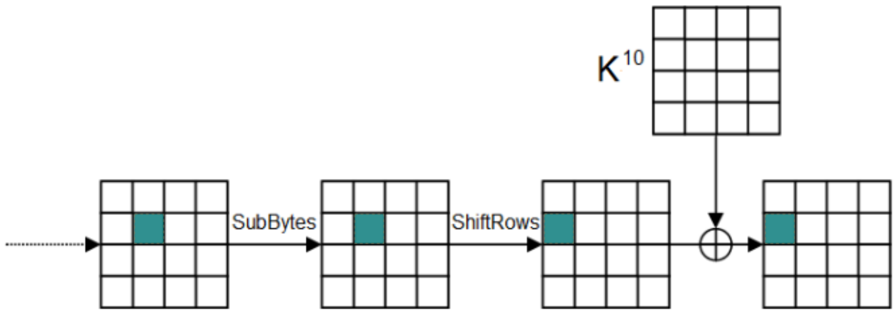


Fig. 11.1: *Fault's propagation in Giraud's attack.*

The considered fault model is a single bit flip just before the SubBytes operation of the last round. This fault results in one faulty byte in the ciphertext, as shown in Fig. 11.1. Due to the AES structure, it is easy for the attacker to identify the affected byte of the state of AES by comparing the correct and faulty ciphertexts. The attacker can then compute the corresponding byte of the last RoundKey by an exhaustive search in the following way.

Let us denote by s the byte where the fault occurs at the beginning of the last round, by c and c^* the corresponding bytes in C and C^* respectively, by k the corresponding byte of the last RoundKey K^{10}.

By definition of AES, we have

$$c = \text{SubBytes}(s) \oplus k \tag{11.1}$$

and

$$c^* = \text{SubBytes}(s \oplus \varepsilon) \oplus k, \tag{11.2}$$

where ε denotes the injected fault and, since ε corresponds to a single-bit flip,

$$\varepsilon \in \{\texttt{0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80}\}. \tag{11.3}$$

For all possible values $\tilde{k}$ of the key byte, the attacker computes

$$\begin{aligned} \tilde{s} &= \text{SubBytes}^{-1}(c \oplus \tilde{k}) \\ \tilde{s}^* &= \text{SubBytes}^{-1}(c^* \oplus \tilde{k}) \end{aligned} \tag{11.4}$$

and checks whether $\tilde{s} \oplus \tilde{s}^*$ satisfies the fault model or not, i.e.

$$\tilde{s} \oplus \tilde{s}^* \in \{\texttt{0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80}\}.$$

Namely, the attacker checks if $\tilde{s}$ and $\tilde{s}^*$ differ by one bit. If it is the case, the value $\tilde{k}$ is a possible candidate for k. Otherwise it is discarded.

After testing all values, the set of possible candidates contains in average 8 elements. In other words, the attack discards all the candidates of the byte of the key which have a corresponding fault that is not included among the possible ones considered by the model and reduces in average the set of candidates from 2^8 possible to 2^3 compatible with the model. This means that each experiment provides $\log_2 \frac{2^8}{2^3} = 5$ bits of information about the secret key.

Two experiments in average are enough to uniquely identify the correct value of the byte of the key. Namely, with another pair of correct and faulty ciphertexts, with fault induced in the same byte, the attacker obtains another set of candidates for k and the intersection of this set with the first one gives the correct value of k.

Actually, two iterations each one providing 5 bits of information are more than the needed. In theory a second experiment providing just 3 bits would be enough. But in practice this is not the case and the attacker is forced to the actual experiment. Since each experiment provides

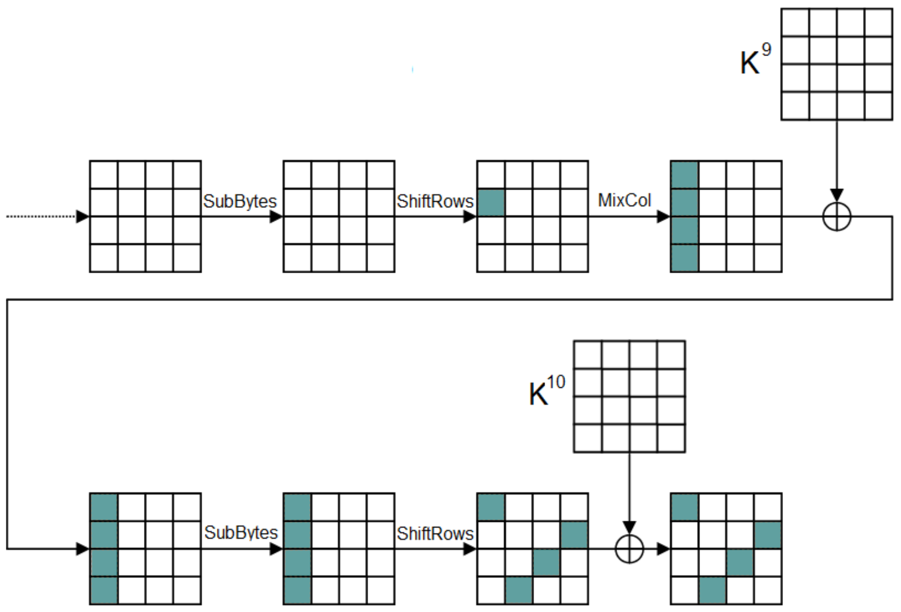


Fig. 11.2: *Fault's diffusion caused by the MixColumns operation in Piret's attack.*

information only on one byte of the key, the extra information ($2^5/2^3$) cannot be used among different bytes, and then some of such information is "wasted", from the pure efficiency point of view.

The process must be repeated independently for each of the 16 bytes of the key. At the end, the result is that in average 32 experiments are needed, each one providing 5 bits of information on the key space, which is 128 bits wide.

Note that the analysis allows to retrieve the last RoundKey and that the secret key can be obtained simply by applying the inverse KeySchedule operation on it.

11.1.2 Piret and Quisquater's attack

After [Gir03] the attacks have been extended in order to consider faults injected at different positions in the computation and affecting more data in the state. The attack in [PQ03] is based on the perturbation of the AES state between the MixColumns of the 8th round and the MixColumns of the 9th round, where a single-byte fault is injected. As shown in Fig. 11.2, a single-byte fault results in four faulty bytes in the ciphertext, due to the MixColumns operation. By comparing correct and faulty ciphertexts, it is easy for the attacker to identify the column of the state where the fault occurred (but not the exact byte). The corresponding four bytes of the last RoundKey can be computed by an exhaustive search in the following way.

Let us denote by s the column of the state where the fault occurred, by c and c^* the corresponding 4-tuple of bytes in C and C^* respectively, by k^9 and k^{10} the corresponding 4-tuple of bytes of the RoundKey K^9 and K^{10} respectively.

By definition of AES, we have

$$\begin{aligned} c &= \text{SubBytes}(\text{MixColumns}(s) \oplus k^9) \oplus k^{10} \\ c^* &= \text{SubBytes}(\text{MixColumns}(s \oplus \varepsilon) \oplus k^9) \oplus k^{10} \end{aligned} \tag{11.5}$$

where ε denotes the injected fault and

$$\varepsilon \in \left\{ \begin{bmatrix} \varepsilon_0 \\ \varepsilon_1 \\ \varepsilon_2 \\ \varepsilon_3 \end{bmatrix} \mid \exists !\, j \text{ s.t. } \varepsilon_j \neq 0 \text{ and } \varepsilon_i = 0\ \forall i \neq j \right\}. \tag{11.6}$$

Thanks to the linearity of the MixColumns and the AddRoundKey operations, we can write (11.5) as

$$\begin{aligned} c &= \text{SubBytes}(s') \oplus k^{10} \\ c^* &= \text{SubBytes}(s' \oplus \varepsilon') \oplus k^{10} \end{aligned} \tag{11.7}$$

where $s' = \text{MixColumns}(s) \oplus k^9$ and $\varepsilon' = \text{MixColumns}(\varepsilon)$. Note that s' simply corresponds to the state at the beginning of the last round and that

$$\varepsilon' \in \left\{ \text{MixColumns} \begin{bmatrix} \varepsilon_0 \\ \varepsilon_1 \\ \varepsilon_2 \\ \varepsilon_3 \end{bmatrix} \mid \exists!\ j \text{ s.t. } \varepsilon_j \neq 0 \text{ and } \varepsilon_i = 0\ \forall i \neq j \right\}. \tag{11.8}$$

For all possible values $\tilde{k}$ of the 4-tuple k^{10}, the attacker computes

$$\begin{aligned} \tilde{s} &= \text{SubBytes}^{-1}(c \oplus \tilde{k}) \\ \tilde{s}^* &= \text{SubBytes}^{-1}(c^* \oplus \tilde{k}) \end{aligned} \tag{11.9}$$

and checks whether $\tilde{s} \oplus \tilde{s}^*$ satisfies the fault model or not. Namely, the attacker checks if the difference between $\tilde{s}$ and $\tilde{s}^*$ satisfies (11.8). If it is the case, the value $\tilde{k}$ is a possible candidate for k.

After testing all values, the set of possible candidates contains in average 2^{10} elements. That is, the set of candidates is reduced from 2^{32} to 2^{10}. This means that each experiment provides $\log_2 \frac{2^{32}}{4 \cdot 255} \sim 22$ bits of information, and then two experiments in average are enough to uniquely determine four bytes of the key.

Similarly to the previous case, the extra information cannot be reused among columns. In total the attack requires in average 8 experiments to obtain the entire key.

11.1.3 Moradi, Shalmani and Salmasizadeh's attack

In [MSS06] two attacks are presented, where the fault is injected in the same position of the attack in [PQ03]. In fact, both the attacks consider faults affecting a single column of the state between the MixColumns of the 8th round and the MixColumns of the 9th round, resulting in four faulty bytes, as in [PQ03]. The difference between the two attacks presented in [MSS06] and the attack of [PQ03] is the considered fault model. The first attack in [MSS06] considers the set of any fault affecting up to three bytes (i.e. either one, two or three bytes). The second attack in [MSS06] considers the model where all the four bytes are faulted. In Fig. 11.3 we give an example of the fault propagation.

As in the previous section, let s be the column of the state where the fault occurred, c and c^* the corresponding 4-tuple of bytes in C and C^* respectively and k^9 and k^{10} the corresponding 4-tuple of bytes of the RoundKey K^9 and K^{10} respectively.

As in Equation 11.5, we have

$$\begin{aligned} c &= \text{SubBytes}(\text{MixColumns}(s) \oplus k^9) \oplus k^{10} \\ c^* &= \text{SubBytes}(\text{MixColumns}(s \oplus \varepsilon) \oplus k^9) \oplus k^{10} \end{aligned} \tag{11.10}$$

where ε denotes the injected fault.

For the first attack of [MSS06], it holds

$$\varepsilon \in \left\{ \begin{bmatrix} \varepsilon_0 \\ \varepsilon_1 \\ \varepsilon_2 \\ \varepsilon_3 \end{bmatrix} \mid \exists j \text{ s.t. } \varepsilon_j = 0 \right\}. \tag{11.11}$$

The set of considered faults, which includes also the ones considered in [PQ03], is thus composed by $255 \cdot 4 + 255 \cdot 255 \cdot 6 + 255 \cdot 255 \cdot 255 \cdot 4$ possibilities, that is about 2^{26} in total. The

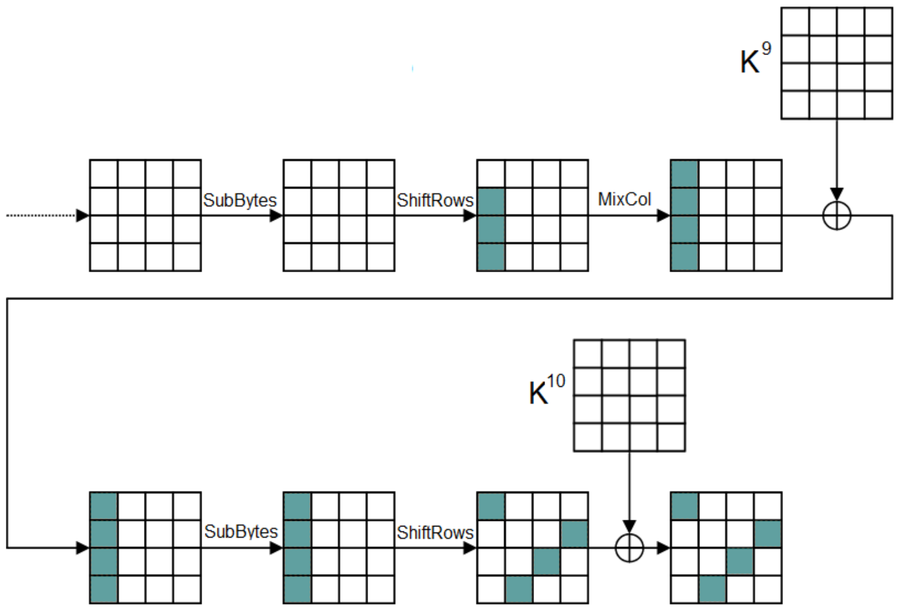


Fig. 11.3: *Fault's diffusion in Moradi's attack.*

corresponding information brought by each experiment on four bytes of the key is $2^{32}/2^{26} = 6$ bits. In average 6 experiments are needed to obtain four bytes of the key and 24 for the whole key.

The second attack in [MSS06] considers the model where all the four bytes are faulted. Namely,

$$\varepsilon \in \left\{ \begin{bmatrix} \varepsilon_0 \\ \varepsilon_1 \\ \varepsilon_2 \\ \varepsilon_3 \end{bmatrix} \mid \varepsilon_j \neq 0 \ \forall\ j \right\}. \tag{11.12}$$

Hence the model includes about $255^4 = 2^{31.98}$ faults. In this case each experiment provides in average about 0.2 bits of information. It means that in average 1495 experiments are needed in order to uniquely obtain four bytes of the key.

11.1.4 Recap on attacks

Table 11.1 summarizes for each of the listed attacks the information on the secret key of AES-128 that the faults provide to the attacker and the average number of experiments to retrieve the entire key.

Note that each experiment provides information about a part of the key. For instance, in [Gir03] it provides information about a byte and in [PQ03] about four bytes. The extra information provided by experiments targeting the same word of the key cannot be reused on other words and then some of such information is "wasted", from the pure efficiency point of view.

This is the reason why the value in the last column of Tab. 11.1 does not correspond simply to 128 divided by the value of the second column.

Attack	Bits of information per experiment	Number of experiments
[Gir03]	5	32
[PQ03]	10	8
[MSS06] 1	6	24
[MSS06] 2	0.2	5980

Table 11.1: *Information provided by each experiment and number of experiments to obtain the entire key.*

11.2 How the fault model affects the attack

Any of the known DFA against AES relies on a specific fault model in order to infer information about the secret key.

As shown in Section 11.1, the amount of information that the fault model provides is $\log_2\left(\frac{\#\text{all faults}}{\#\text{admitted faults}}\right)$ and this information allows to reduce the space of key candidates. It follows that by enlarging the fault model (i.e. by increasing the number of admitted faults) the amount of information decreases and the number of candidates kept valid in each experiment increases. Namely, the space of key candidates gets restricted more slowly and more experiments are necessary to obtain the correct key.

In this section we will show this fact by considering the attacks presented in Section 16.1. We will also show that in the extreme case, where all faults are admitted by the model, the fault does not provide any information and then it does not allow to discard any candidate, making impossible the attack.

11.2.1 Application to Giraud's attack

As described in Section 11.1.1, the fault model in [Gir03] is a single-bit flip at the beginning of the last round, which results in one faulty byte in the ciphertext.

This means that the faults admitted by the model are only eight among all the possible ones that can occur on a byte. That is, each experiment provides $\log_2 \frac{2^8-1}{2^3} \sim 5$ bits of information about the corresponding byte of the key, which allows to reduce in average the set of candidates from 2^8 to 2^3. Two experiments in average are enough to uniquely identify the correct value.

Now we want to show that by admitting more faults in the model, the amount of information decreases.

Consider the model complementary to the model of [Gir03]. Namely, the possible faults are those affecting at least two bits in a byte. The number of admitted faults is then $255 - 8 = 247 \sim 2^{7.95}$ and each experiment provides $\log_2 \frac{255}{247} \sim 0.05$ bits of information. It follows that each experiment allows to reduce the search space from 2^8 to $2^{7.95}$ and that 160 experiments in average are necessary to uniquely identify the correct value of a byte of the key.

It is clear that by admitting more faults in the model, the number of key candidates compatible with the model increases and only few candidates can be discarded during the analysis. But, what is important is that it is still possible to do it. The only consequence is that a bigger number of experiments is necessary.

On the contrary, if the fault model includes all possible faults on the byte, i.e. both the faults of [Gir03] and of its complementary variant, no information is provided to the attacker. In fact, the number of admitted faults in this case is 255 and by an information theoretical point of view the information provided is $\log_2 \frac{255}{255} = 0$. What happens during the analysis is that any candidate of the key is never discarded, since the difference $\tilde{s} \oplus \tilde{s}^*$ always satisfies the model.

11.2.2 Application to Piret and Quisquater's attack

The attack in [PQ03] is based on a single-byte fault injection before the MixColumns of the 9^{th}, resulting in four faulty bytes in the ciphertext, due to the MixColumns operation. This means that the faults admitted by the model are only $4 \cdot 255$ among all the possible ones that can occur on a column of the state (i.e. $2^{32}-1$). That is, each experiment provides $\log_2 \frac{2^{32}-1}{4 \cdot 255} \sim 22$ bits of information about four bytes of the key, which allows to reduce in average the set of candidates from 2^{32} to 2^{10}. Two experiments in average are enough to univocally identify the correct value.

Now, consider the complementary model. Namely, the possible faults are those affecting at least two bytes in a column. The number of admitted faults is then $(2^{32}-1)-(4 \cdot 255) \sim 2^{31.99}$ and each experiment provides $\sim \log_2 \frac{2^{32}-1}{2^{31.99}} \sim 0.01$ bits of information. It follows that each experiment allows to reduce the search space from 2^{32} to $2^{31.99}$ and that more than 3000 experiments are necessary to univocally identify the correct value of four bytes of the key.

As in the previous case, it is still possible to discard some candidates for the key during the analysis, even if they are very few, with the consequence that a bigger number of experiments is necessary. On the contrary, if the fault model includes both the faults of [PQ03] and of its complementary variant (i.e. all the $2^{32}-1$ possible ones), no information is provided to the attacker and any candidate of the key is never discarded, since it always satisfies the fault model.

11.2.3 Application to Moradi, Shalmani and Salmasizadeh's attack

The two attacks described in [MSS06] assume two complementary fault models. In the first one, the model includes faults affecting up to three bytes (i.e. about 2^{26}) and in the second one the model includes faults affecting all four bytes (i.e. 255^4). Both the attacks allow to retrieve four bytes of the key, the only difference being the number of required experiments.

If the model admits faults of both variants, no information will be provided to the attacker. In fact, the number of admitted faults in this case is $2^{32}-1$ and by an information theoretical point of view the information provided is $\log_2 \frac{2^{32}-1}{2^{32}-1} = 0$. In fact, during the analysis, any candidate of the key is never discarded, since it always satisfies the model. So, even if the sum of the two models covers of all possible faults that may occur on a 4-tuple of bytes, for the attacker is fundamental to distinguish whether a faulty ciphertext derives from the first or second model.

11.3 How the knowledge of the model affects the attack

In section 11.2, we have shown that by relaxing the fault model (i.e. admitting more possible faults) the number of key candidates kept valid in each experiment increases. In the extreme case, where all faults are admitted by the model, no information is provided to the attacker, making impossible the attack.

This highlights the fact that the knowledge of the fault model directly affects the effectiveness of the attack, until the point to remove the basis for a successful attack. In other words, if the attacker explicitly knows the fault model (e.g. a single-bit flip), then she is able to conduct the attack, alternatively she is not able to reduce the search space. Actually, no knowledge about the fault model corresponds to the scenario where the model includes all possible faults, which does not provide any information.

It is worth noting that, in order to have a successful attack, the knowledge on the fault model does not have to explicitly list the set of allowed faults. It can be any information that reduces the combinations of cases that happen in the experiments, among all the possible ones. For instance, such information could be the fact that exactly the same fault is injected in several executions, even without knowing the actual value ε of the fault. Otherwise, it could be the fact that the fault lowers the Hamming weight of the intermediate values. Several different fault models can be defined in practice, depending on the target implementation and the way faults are injected.

What is important to notice is that, by just observing the pair of correct and faulty ciphertexts, the attacker is not able to distinguish which kind of fault has occurred, unless he has some information on the injection technique. Namely, unless he knows that some physical action on the device has only well-defined effects. This depends on both the injection technique and the architecture of the target implementation. For instance, some techniques may always change bits from 1 to 0 and never from 0 to 1 (or vice versa). While, in an implementation that manages 8 bits of data per cycle it may be likely to confine the fault to a single byte only.

By considering the attacks in [Gir03,PQ03,MSS06], we show in this section how the knowledge of the fault model affects the effectiveness of the attack. In particular, we will show that if the attacker is not able to distinguish whether the injected fault belongs to the model, he has a very low probability to find the correct key. Moreover, we will show that not only the attacker has low probability to find the key, but it is likely that he will find the wrong one. In fact, it is commonly believed that applying the attack to pairs of ciphertexts some of which do not satisfy the model (in this case, we say that the faulty ciphertext is bad, alternatively that it is good) will result in no solution for the key [PQ03]. But we will show that the probability to find a wrong key is actually not negligible and thus the attacker cannot distinguish that some of the collected pairs are bad.

11.3.1 Giraud

The fault model in [Gir03] is a single-bit flip at the beginning of the last round, which results in one faulty byte in the ciphertext. What the attacker observes is just the single faulty byte in the ciphertext and theoretically that fault could have been produced by any fault on the corresponding byte of the state. That means, the faults admitted by the model are only eight among all the possible ones that can occur on a byte.

If the attacker has no knowledge about the fault injection, namely if she is not able to distinguish if the injected fault is a single-bit flip, the attack will not be successful with a very high probability.

Let's consider the case where all the 2^8-1 possible faults on a byte of the intermediate state are equally probable. If the attacker observes a faulty ciphertext as in Fig. 11.1, the probability that the injected fault is a single-bit fault is $\frac{2^3}{2^8-1} \simeq \frac{1}{2^5}$. Since the attack requires two faulty ciphertexts, the probability of injecting two single-bit faults is $\sim \frac{1}{2^{10}}$. This is exactly the probability to correctly conduct the attack and get the correct key.

At first sight, it may seems that if the attacker considers two bad faulty ciphertexts, or one good and one bad, she simply obtains no solution. But, in practice, she can obtain a solution (which is wrong) with a considerable probability.

In support of our thesis, we conducted the following test. We chose a pair of message and key and executed the encryption. Then with the same message and key, we executed 255 encryptions simulating a fault injection on a fixed byte with the fault assuming all 255 possible values. As message and key, we chose a pair that gives all zero as output and such that the last round key is also all zero. The target byte is the byte in the first row and first column. For all the 255 pairs of correct and faulty bytes, we conducted the analysis and retrieved the key byte candidates. Of course, the correct key byte is in the key set of the experiments corresponding to fault $\varepsilon \in \{0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80\}$ and never appears in the key set of any other experiment. But, we can also observe that each key candidate appears in the key set of eight pairs in average. Hence, by considering two pairs of correct and faulty ciphertexts, the probability the analysis conducts to a given common key is $\frac{2^3}{2^8-1} \cdot \frac{2^3}{2^8-1} \sim \frac{1}{2^{10}}$. It follows that the probability of obtaining a solution is $\sim \frac{2^8}{2^{10}}$ and the probability that this solution is wrong is $\frac{2^8-1}{2^{10}}$.

On the other hand, the probability to get no solution is $1 - \frac{2^8-1}{2^{10}} \sim \frac{3}{4}$. Even if the probability to get no solution is high, it does not help the attacker in the analysis. Actually, the fact that, by considering a pair of ciphertexts, the attacker gets no solution, does not allow him to label the two ciphertexts as bad ciphertexts. In fact, one of them can be a good ciphertext, but the attacker is not able to distinguish which one.

Making a recap, one out of 4 experiments gives a solution. Only one out of 2^{10} gives the correct solution. But, 2^8 out of these 2^{10} give wrong solutions.

11.3.2 Piret and Quisquater

The attack in [PQ03] considers a single-byte fault injected between the MixColumns of the 8^{th} and 9^{th} round, resulting in four faulty bytes in the ciphertext, due to the MixColumns operation. In Fig. 11.2, we give an example of the attack where the fault is injected in the first byte of the first column.

According to the attack, the resulting pattern of four faulty bytes can be obtained by injecting the fault in any of the four bytes of the first column. But the same pattern can be obtained by injecting whatever fault in the first column. This means that, by faulting from one through four bytes, the attacker will obtain the same faulty pattern in the ciphertext. Namely, there are $2^{32} - 1$ possible faults resulting in the same pattern and the probability that the injected fault is a single-byte fault is $\frac{4 \cdot 255}{2^{32}-1} \sim \frac{1}{2^{22}}$.

Since the attack requires two faulty ciphertexts, the probability of injecting two single byte faults is $\sim \frac{1}{2^{44}}$. This is exactly the probability to correctly conduct the attack and get the correct key.

If the attacker considers two faulty ciphertexts, the probability the analysis conducts to a common key is $\frac{4 \cdot 255}{2^{32}-1} \cdot \frac{4 \cdot 255}{2^{32}-1} \sim \frac{1}{2^{44}}$. It follows that the probability of obtaining a solution is $\sim \frac{2^{32}}{2^{44}}$ and the probability that this solution is wrong (i.e. the two ciphertexts are both bad or one good and one bad) is $\frac{2^{32}-1}{2^{44}} \sim \frac{1}{2^{12}}$. Whereas, the probability to get no solution is $1 - \frac{1}{2^{12}}$, but it does not help the attacker in the analysis.

Making a recap, one out of 2^{12} couples of pairs of correct and faulty ciphertexts gives a solution. Only one out of 2^{44} gives the correct solution. But, 2^{32} out of these 2^{44} give wrong solutions.

11.3.3 Moradi, Shalmani and Salmasizadeh

In the first attack in [MSS06], the model includes faults affecting up to three bytes of a column of the intermediate state (i.e. about 2^{26}) and in the second one the model includes faults affecting all four bytes (i.e. 255^4).

As in the previous case, the attacker observes four faulty bytes, whose pattern can be obtained by injecting whatever fault in a column of the intermediate state. Let's consider the probabilities of a successful attack for both the scenarios.

- **First model**: The probability that the injected fault belongs to the first model is $\frac{2^{26}}{2^{32}-1} \simeq \frac{1}{2^6}$. Since the attack requires in average six faulty ciphertexts, the probability of injecting six faults belonging to the model is $\sim \frac{1}{2^{36}}$ and this is exactly the probability to correctly conduct the attack and get the correct key.
  If the attacker considers six faulty ciphertexts, the probability the analysis conducts to a common key is $\frac{1}{2^{36}}$. The probability of obtaining a solution is $2^{32} \cdot \frac{1}{2^{36}}$ and the probability that this solution is wrong is $\frac{2^{32}-1}{2^{36}} \sim \frac{1}{2^4}$. Whereas, the probability to get no solution is $1 - \frac{1}{2^4}$.
  Making a recap, one out of 2^4 experiments gives a solution. Only one out of 2^{36} gives the correct solution. But, 2^{32} out of these 2^{36} give wrong solutions.
- **Second model**: The probability that the injected fault belongs to the second model is $\frac{255^4}{2^{32}-1} \sim 0.98$. Since the attack requires on average 1495 faulty ciphertexts, the probability of injecting all faults belonging to the model is very close to 0. This makes the attack quite impractical. In the next section we will analyze this case more in details.

11.3.4 Recap on probabilities

The idea presented in this section can be extended to any differential fault attack against AES, obtaining the corresponding probabilities. In Tab. 11.2 we recap the probabilities related to the attacks we have previously described.

We can observe that by relaxing the fault model (namely, by admitting a larger set of faults), the probability to inject an admitted fault increases, hence also the probability to find the correct key increases. But, this increase is quite insignificant compared to the increase of the probability to find a wrong solution. Actually, by relaxing the model, the number of required pairs of faulty and correct ciphertexts increases and, as a consequence, the probability of picking all good pairs decreases. On the other hand, given a large number of pairs, the probability that they have a common solution increases, making an exhaustive search infeasible.

Reference attack	Size of the fault model	Number of exp.	Prob. to find the correct key	Prob. to find a wrong key	Prob. to find no solution
Giraud	2^3	2	$\left(\frac{2^3}{2^8-1}\right)^2 \sim \frac{1}{2^{10}}$	$\sim \frac{2^8-1}{2^{10}} \sim \frac{1}{2^2}$	$\sim 1 - \frac{1}{2^2}$
Piret & Quisquater	$255 \cdot 4 \simeq 2^{10}$	2	$\sim \left(\frac{2^{10}}{2^{32}}\right)^2 \sim \frac{1}{2^{44}}$	$\sim \frac{2^{32}}{2^{44}} \sim \frac{1}{2^{12}}$	$\sim 1 - \frac{1}{2^{12}}$
Moradi et al. 1	$\sim 2^{26}$	6	$\sim \left(\frac{2^{26}}{2^{32}}\right)^6 \sim \frac{1}{2^{36}}$	$\sim \frac{2^{32}}{2^{36}} \sim \frac{1}{2^4}$	$\sim 1 - \frac{1}{2^4}$
Moradi et al. 2	$255^4 \sim 2^{31.98}$	1495	$\sim \left(\frac{2^{31.98}}{2^{32}}\right)^{1495} \sim \frac{1}{2^{33.77}}$	$\sim \frac{2^{32}}{2^{33.77}} \sim 0.3$	~ 0.7

Table 11.2: *Probabilities to find correct, wrong or no key, in average, assuming all the possible faults are uniformly distributed. Analysis on different fault models: on 1 byte for Giraud, on 4 bytes for the other attacks.*

11.4 A concrete example

In this section we want to apply the concepts introduced in previous sections in order to understand how the knowledge of the fault model affects the attack in a concrete example. We will focus on the second attack presented in [MSS06].

In order to mount an attack, the attacker has to either select a priori the fault model or to have any technique able to identify which fault is occurring. In this concrete example, we select the fault model presented in [MSS06] because it is based on a pretty relaxed fault model and then it requires the lowest precision.

Indeed, the attack we are considering assumes all four bytes of the column affected by the fault (namely, 255^4 faults are admitted out of the $2^{32}-1$ possible ones) and it requires in average 1495 experiments to obtain four bytes of the key.

If the attacker is able to guarantee that all the injected faults belong to the selected model, the analysis will conduct to the correct key for sure. In other words, the correct key will remain in the set of candidates with a probability of 1 until the last pair is considered.

On the contrary, as explained in sec. 11.3, if the attacker has no information on the fault injection the analysis will conduct to a wrong solution or to no solution with high probability.

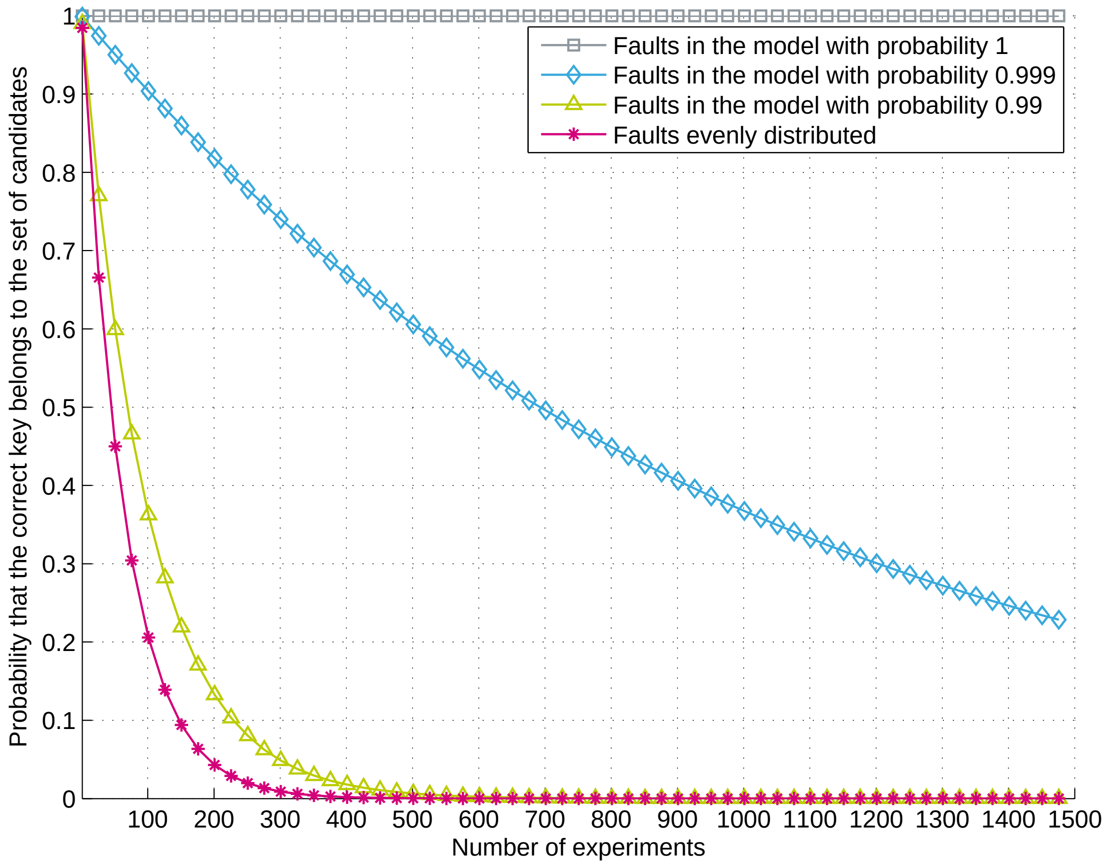


Fig. 11.4: *Probability to keep the correct key among the candidates during Moradi's attack.*

This may happen if the injection technique generates either uniformly distributed faults with no bias, or faults not belonging to the selected model. Actually, during the analysis, the correct key will be discarded as soon as a bad pair is considered. As the number of required pairs increases, the probability of picking all good pairs decreases and, as a consequence, the probability to discard the correct key becomes very high.

In particular, let's suppose that the injection technique generates all the possible faults on the four target bytes with the same probability, but the attacker wrongly selects the second model in [MSS06]. Namely, the attacker has no true information about the target device and no true information about the effect of the fault injection technique she uses. With these hypotheses, the faults are uniformly distributed and this is the worst-case scenario for conducting the attack. It follows that the probability that a good fault occurs is $\frac{255^4}{2^{32}-1} \sim 0.98$. By considering only a pair of correct and faulty ciphertexts, the attacker gets $2^{31.9}$ candidates for the key and the correct value is among them with a probability of 0.98. By considering other pairs, say k pairs, the probability to keep the correct key in the set of candidates corresponds to 0.98^k. Namely, the probability quickly decreases. When we consider 45 pairs, the attacker still has 2^{31} candidates, but the probability of the correct key being in the set is already decreased to 0.49. At the end of the analysis, the correct key is obtained with a probability of $7 \cdot 10^{-13}$, which is the probability that all the 1495 considered pairs are good ones.

The two scenarios we have just described (i.e. total knowledge and no knowledge about the injected faults) are two extreme situations. In-between there are scenarios where the fault injection technique produces faults belonging to the model more frequently than faults excluded by the model.

For instance, let's suppose that one among 100 injections falls out of the model. Namely, when the attacker injects a fault, the fault belongs to the model with a probability equal to 99% (to be compared with 98% in case of uniformly distributed faults). It follows that the probability to obtain the correct key increases a little, but it is still very low.

Now, let's suppose to increase the bias on the injection technique: only one among 1000 injections falls out of the model. The probability to keep the correct key value among the candidates considerably increases. Indeed, after 1495 experiments it is about 20%.

Figure 11.4 depicts the different scenarios we have presented. It is easy to see that the closer the probability gets to 1, which corresponds to total knowledge, the higher the probability of finding the correct key is. But, even with a fault model with 99% accuracy the effectiveness of the attack is heavily lowered.

12

J-DFA: a novel approach to differential fault analysis

We have seen in previous chapter, how DFA attacks usually require some kind of knowledge by the attacker on the effect of the faults on the target device, which in practice turns to be a poorly reliable information typically affected by uncertainty.

Depending on the target cryptosystem and the specific attack, the a-priori knowledge of the fault model by the attacker can fundamentally impact the effectiveness of the attack.

As explained in Section 11.1, from the information theory point of view, every fault provides information about the secret key. Such information depends on the precision of the injection technique but also on the knowledge that the attacker has on the induced effect. All DFA attacks work when all the specific instances of executed faults perfectly match the a-priori knowledge of the attacker about the possible effects of the injections. In practice, however, some more general situations should occur. Firstly, the attacker can be forced to consider a wider set of possible effects of the fault injections. Due to this uncertainty, the efficiency of the attack lowers as the information provided by each fault lowers too. In this case the attack still works, but the number of required faulty ciphertexts increases. Moreover, it is possible that the effects of some faults fall out of the considered set of models. In this case the attack either terminates with a wrong key or with no solution, because of the wrong a-priori hypothesis.

In this chapter we introduce a novel DFA approach, based on the application of a clustering technique named *J-Linkage*, with the aim of increasing the robustness of the attack and softening the requirements on the a-priori knowledge by the attacker. J-Linkage [TF08b] is a clustering technique which addresses the problem of simultaneous multiple model fitting within a dataset containing outliers, namely observations which do not fit any model. Originally proposed for geometric model fitting in Computer Vision, J-Linkage is used here to derive a new DFA approach, called J-DFA. Thanks to the inherent properties of the J-Linkage technique, J-DFA will result in a versatile tool that not only can be easily used to quickly replicate many classical DFA attacks, but also produces reliable solutions in a wider range of practical attack scenarios. In order to show the effectiveness of the proposed approach, we apply J-DFA to the specific case of faults injected in the last round of AES and we successfully compare our results with the classical approach in different attack conditions.

The chapter is organized as follows. In Section 12.1 we briefly illustrate common DFA attacks against last AES round with particular attention to the details of a class of them which will be used as example. In Section 12.2 we provide the key features of the J-Linkage clustering technique. In Section 12.3 we describe our J-DFA technique and we explain how to map a specific known DFA attack on the J-Linkage settings. In Section 16.3 we show the results of several experiments and we highlight the benefits introduced by the new approach.

12.1 DFA against last round of AES

As a reference, we generalize the attack of [Gir03], described it in Section 11.1.1, obtaining a family of classical attacks based on the injection of the fault at the beginning of the last AES round. This class of attacks will be used throughout the chapter to detail the application of our approach and to compare the results with well-known attacks.

The considered fault model includes any possible alteration of the AES state just before the SubBytes operation of the last round. Since the MixColumns operation is not performed in the last round, each byte of the AES state affected by the fault can be considered independently. For the sake of simplicity the following description focuses on the case where a single byte is affected per injection, without loss of generality. Indeed, due to the AES structure, it is easy to deduce the number of perturbed bytes by simply observing the pair of correct and faulty ciphertexts. Therefore, in case of multiple affected bytes, the attacker can either consider separately the bytes in the second stage of the attack, or just ignore the observations involving more than a single byte.

The attack presented by Giraud is an instance of this class of attacks, where the specific fault model considered for the injection is a single bit flip in the AES state at the beginning of the last round. Note however that the same attack procedure can be trivially extended to different fault models, like 2-bit flips, 3-bit flips, etc.

According to Giraud's attack, the attacker can compute the corresponding byte of the last RoundKey by an exhaustive search in the following way.

Let (C, C^*) be an experiment, namely a pair of respectively correct and faulty ciphertexts generated on the same plaintext using the same key. Let us denote by s the byte where the fault occurs at the beginning of the last round, by c and c^* the corresponding bytes in C and C^* respectively, and by k the corresponding byte of the last RoundKey K^{10}. By definition of AES, we have

$$\begin{aligned} c &= \mathrm{SubBytes}(s) \oplus k \\ c^* &= \mathrm{SubBytes}(s \oplus \varepsilon) \oplus k \end{aligned} \tag{12.1}$$

where ε denotes the injected fault.

In general, ε can belong to any subset E of the 255 possible faults that can be induced on a byte. In the specific case described in [Gir03], ε belongs to

$$E = \{0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80\}.$$

For all possible values $\tilde{k}$ of the key byte, the attacker computes

$$\begin{aligned} \tilde{s} &= \mathrm{SubBytes}^{-1}(c \oplus \tilde{k}) \\ \tilde{s}^* &= \mathrm{SubBytes}^{-1}(c^* \oplus \tilde{k}) \end{aligned} \tag{12.2}$$

and subsequently

$$\tilde{s} \oplus \tilde{s}^* = \mathrm{SubBytes}^{-1}(c \oplus \tilde{k}) \oplus \mathrm{SubBytes}^{-1}(c^* \oplus \tilde{k}) = \varepsilon. \tag{12.3}$$

For different experiments, the attacker checks whether ε satisfies the fault model or not, i.e. if it belongs to the set E. If this is the case, the value $\tilde{k}$ is a possible candidate for the key byte k, otherwise the specific $\tilde{k}$ is not compatible with the assumed model and it is discarded.

Observe that in a sense, each experiment (C, C^*) induces a relationship between faults ε and corresponding key candidates $\tilde{k}$ described by

$$f_{(C,C^*)}(\tilde{k}) = \mathrm{SubBytes}^{-1}(c \oplus \tilde{k}) \oplus \mathrm{SubBytes}^{-1}(c^* \oplus \tilde{k}). \tag{12.4}$$

After testing all values, the set of possible key byte candidates is downsized with respect to the initial set of 256 possible elements. In other words, the attack discards all the candidates of the byte of the key which correspond to a fault that is not included among the faults considered by the model. The size of the resulting set of candidates depends on the size of the fault model. In the case of the model described in [Gir03], on average only 8 candidates are left.

With a second pair of correct and faulty ciphertexts, with fault induced in the same byte, the attacker obtains another set of candidates for k and the intersection of this set with the first one contains the correct value for the key. In general, the procedure must be iterated until only a single candidate for the byte of the key is left in the intersection. In particular, the more precise the fault injection is, the less experiments are necessary and vice versa.

This process must be repeated independently for each of the 16 bytes of the key. Note that the analysis allows to retrieve the last RoundKey and that the secret key can be obtained by simply applying the inverse KeySchedule operation on it.

It is clear from the description that the knowledge of the fault model by the attacker is of fundamental importance for the success of the attack. In particular it is worth underlining the fact that including all the 255 possible faults in the set E is not a viable option for the attacker. In fact such model leads in never discarding key candidates and then never converging to a solution for the byte of the key. Therefore, the attacker is forced to reduce the set of considered faults, by characterizing the injection technique on the specific target device, in the same way the choice done in [Gir03] has been motivated. Such knowledge on the fault model must be obtained a-priori by the attacker and it cannot be derived from observations involving the unknown secret key.

In practice, the most critical aspect of the classical DFA approach, is the fundamental need of guaranteeing that all the considered experiments have been generated by faults belonging to the fault model assumed by the attacker. The presence of few (even a single) experiments that fall out of the model compromises the overall success of the attack. This is due to the fact that each single experiment has the power of discarding the correct candidate for the key, and this condition cannot be recovered by other experiments. In real setups it is hard for the attacker to completely prevent the existence of such bad experiments.

12.2 J-Linkage: an overview

J-Linkage is a clustering technique that is purposely drawn to resolve the problem of simultaneous multiple model estimation within a dataset containing outliers. J-linkage was originally proposed for *geometric* model fitting in Computer Vision, and in this thesis we apply it to DFA. For more details about J-Linkage see [TF08b] in which this topic is well explained in detail.

This method follows a two steps *first-represent-then-clusterize* approach: at first data are represented in a conceptual space by the votes they grant to a set of putative model hypotheses, then a greedy bottom up clustering is performed in order to obtain a segmentation of the data.

12.2.1 Conceptual representation

Let $X = \{x_1, \dots, x_n\}$ be a set of n data and consider a set of m putative models. Consider the $n \times m$ matrix P whose (i, j)-th entry is defined as

$$P(i,j) = \begin{cases} 1 & \text{if } x_i \text{ is explained by the } j\text{-th model} \\ 0 & \text{otherwise.} \end{cases} \tag{12.5}$$

P is called *preference matrix* and depicts vector-wise the preference of data with respects to hypothesized models, in particular the row $P_i \in \{0,1\}^m$ provides a conceptual representation of x_i as the characteristic function on the preferred models.

This representation is extended in a straightforward manner to subsets of data. Let $U \subseteq X$, U is portrayed as the vector of all the common preferences among all the data belonging to it:

$$\bigwedge_{x_i \in U} P_i, \tag{12.6}$$

where $\wedge$ indicates the component-wise logical conjunction on the rows of P corresponding to data in U, in other words a subset is represented by the intersection of the model preferred by its elements.

12.2.2 Clustering

The clustering algorithm proceeds in a bottom-up manner exploiting the representation provided by the matrix P. At first every data is put in its own cluster. The distance between clusters is computed as the *Jaccard distance* between the respective conceptual representations. The Jaccard distance is used in statistics to measure the dissimilarity of sample sets. For two sets A and B it is defined as

$$d_J(A,B) = \frac{|A \cup B| - |A \cap B|}{|A \cup B|} \tag{12.7}$$

and it ranges from 0 to 1.

In our scenario, the Jaccard distance measures the degree of agreement between the votes of two clusters. When it is 0 it means identical votes, while 1 means orthogonal rows of the preference matrix.

Starting from singletons, each sweep of the algorithm merges the two clusters with the smallest Jaccard distance. The cut off value is 1. The clustering procedure can be summarized as follows:

1. Put each datum in its own cluster.
2. Define the conceptual representation of a cluster using (12.6).
3. Among all current clusters, pick the two clusters with the smallest Jaccard distance.
4. Replace these two clusters with the union of the original ones.
5. Repeat from step (3) while the smallest Jaccard distance is lower than 1.

The outcome of this procedure is a segmentation of the data in disjoint clusters U_i such that $X = \bigcup_i U_i$ and if $i \neq j$ $U_i \cap U_j = \emptyset$. It is worth noting that the number of clusters is automatically detected by this algorithm, and this is certainly a remarkable propriety, since the majority of other multi model fitting techniques require this information as input parameter. Moreover this preference approach is robust to outliers, observations whose preferences deviate significantly from the rest of the data, that can in fact be recognized as small clusters.

12.3 J-DFA: DFA based on J-linkage

In this section we describe how we apply J-linkage to DFA with the aim of softening the requirement on the a-priori knowledge needed by the attacker to exploit the faults in practice. The main idea of our approach is to map the stage of differential cryptanalysis of DFA attacks into the problem of fitting multiple models to data corrupted by outliers. We call the resulting method J-DFA.

Like all classical DFA attacks, J-DFA is performed in two stages. The first stage consists in actively manipulating the target device in order to corrupt the computations and collect a set X of experimental data. Each data $x \in X$ is an experiment, i.e. a pair of correct and faulty ciphertexts (C, C^*) generated on the same plaintext using the same key. Let F denote the set of all the possible effects of the fault that may happen as a consequence of the fault injection (e.g. single byte faults). In practice, among all the possible faults, only a subset of them occurs (e.g. single bit in a byte). We denote with $E \subseteq F$ such subset. The faults belonging to E vary depending on the specific technique used for the fault injection and on the target device.

This stage works exactly in the same way as for classical DFA and it may include a step where the meaningful experimental data are extracted among all the experiments. Of course, such a step only allows to extract experiments that produce a peculiar pattern which can be identified by comparing the correct and faulty ciphertext. For instance, a single affected byte at the beginning of the last round results in a single faulted byte in the ciphertext. Therefore, such data can be easily distinguished from faults injected in rounds earlier than the last one. Still, this step does not exclude outliers (as defined, faults not belonging to the model considered by the attacker) that cannot be identified by simply observing the corrupted ciphertexts.

The second stage is the application of techniques of differential cryptanalysis in order to translate the information obtained on the ciphertexts into information on (a portion of) the secret key k. In this second stage the J-Linkage tool is introduced.

In general, DFA is based on the fact that for each possible fault, every experiment (C, C^*) is *compatible* only with a (small) set of values of the involved portion of the key. In the classical view, this fact is used to remove inconsistent values from the set of the possible candidates for that portion of the key.

The J-DFA approach aims at relaxing this hypothesis, by replacing the concept of compatible-incompatible key values (with regards to a specific experiment) with the concept of key values *voted* by a specific experiment.

In the J-DFA view, each experiment can be represented in a conceptual space as characteristic function of the pair(s) (fault, key) = (ε, k) preferred by that experiment. Such pairs represent putative models for the J-Linkage clustering technique. The clustering algorithm aggregates together experiments with similar preferences. At the end the experiments are split in clusters, where each cluster refers to one (or more) specific pairs (ε, k). Highly-populated clusters include experiments with similar votes, while experiments that are poorly compatible with others are left in lowly-populated clusters.

In a successful J-DFA the correct candidate for the portion of the key is the one corresponding to the most-populated clusters and thus the one that has been voted the most.

12.3.1 J-DFA steps

The J-DFA technique involves a sequence of four distinct steps which are represented in Figure 12.1:

1. Data mapping,
2. Conceptual representation,
3. Clustering,
4. Key ranking.

12.3.1.1 Data mapping

The attacker selects a hypothesized fault model $H \subseteq F$ including only the faults considered likely to occur. This selection is based on the assumption derived from the injection technique and the target device.

From the set H, the whole *space of hypotheses* is generated, that is $H \times K$, the Cartesian product between H and K. K represents the set of all the possible values for the portion of the key involved in the attack (i.e. 256 values if one byte of the key is involved).

The space of the hypotheses thus includes all the pairs (ε, k) that are considered possible by the attacker. Such pairs correspond to putative models within the J-Linkage framework.

It is worth noting that most of the classical DFA only consider the case where the fault model is known a-priori, namely $H = E$. Instead in J-DFA we are not making such assumption.

Each experiment $x = (C, C^*)$ induces a relationship between faults ε and key values $k \in K$ which we denote as f_x. Namely, $f_x(k) = \varepsilon$ for some k and ε.

12.3.1.2 J-Linkage conceptual representation

The preferences are built by assuming that a model (ε, k) is preferred by an experiment x if the f_x associates ε and the key candidate k, namely $f_x(k) = \varepsilon$.

The preference matrix is implemented as a matrix with $m = |H| \times |K|$ columns and $n = |X|$ rows. Recall that each column represents a model (ε, k) whereas each row indicates the preferred models by each experiment x.

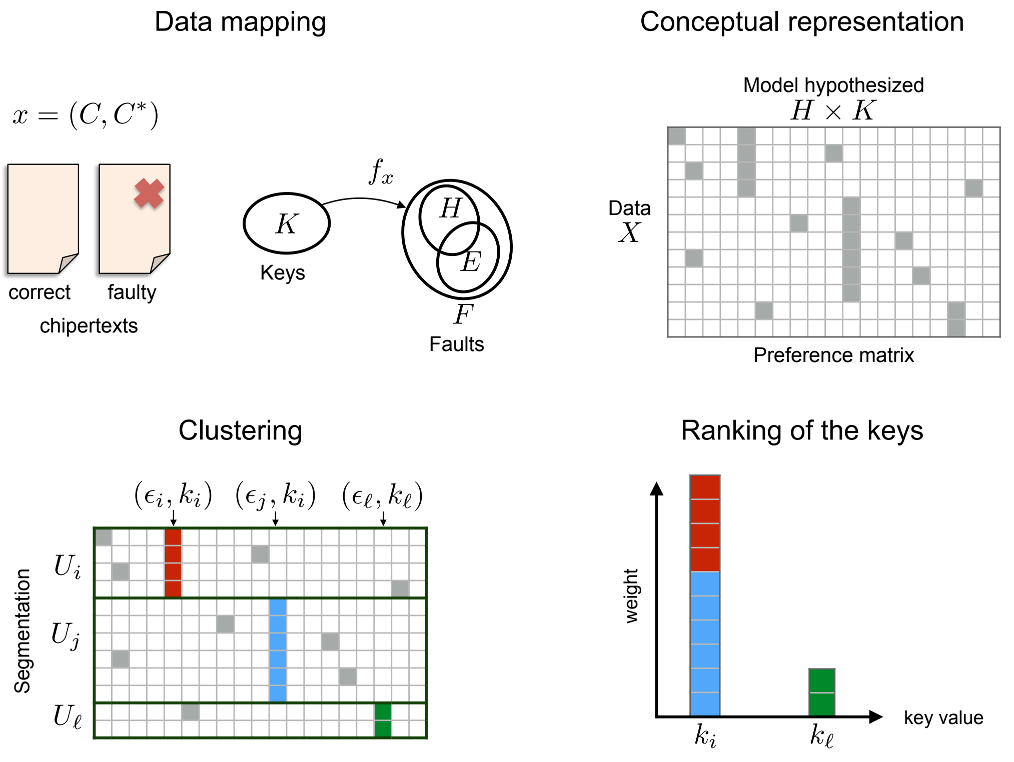


Fig. 12.1: *J-DFA in a nutshell.* ***Data mapping****: An experiment $x = (C, C^*) \in X$ defines a map f_x between the possible key values K and the set of possible faults F. E represents the faults that really occur in the experiments, H consists in the faults hypothesized by the attacker.* ***Conceptual representation****: The preference matrix is built, representing every datum by the votes (gray cells) it grants to the set of putative models $(\varepsilon, k) \in H \times K$.* ***Clustering****: J-Linkage segments the preference matrix in clusters U_i, U_j, U_ℓ (data are arranged such that consecutive data belong to the same cluster for sake of visualization only). It is hence possible to extract the most preferred models per cluster $(\varepsilon_i, k_i), (\varepsilon_j, k_j), (\varepsilon_\ell, k_\ell)$. Note that the same key may appear as preferred by several clusters.* ***Ranking of the keys****: Finally, votes are aggregated with respect to keys and the most preferred one is retained. (Figure best viewed in color)*

12.3.1.3 J-Linkage clustering

Experiments are split in clusters by J-Linkage. Each cluster U_i is representative of one or more models (ε, k).

Experiments belonging to the same cluster means that they all share at least one preference (i.e. model). Vice versa experiments split in different clusters means that they did not have a common preference explaining all of them.

Note that even if the experiments can be split in different clusters representative of different pairs (ε, k), still these models can share a common key (but referring to different faults).

More formally we say that a cluster U_i is k-*compatible*, if there exists at least one hypothesized fault model ε associated with k by all the experiments in U_i.

12.3.1.4 Ranking of keys

The candidates of the key $k \in K$ are ranked based on the size of the clusters U_i obtained from the previous step, by defining for each k a weight

$$w(k) = \sum_{U_i\ k-\text{compatible}} |U_i|. \tag{12.8}$$

The recovered key κ is the one with highest weight, i.e.

$$\kappa = \arg\max w. \tag{12.9}$$

12.3.2 Robustness of J-DFA

Under the correct hypothesis on the fault model, J-DFA guarantees that the correct key is among the preferences of the most populated clusters. Similarly to classical DFA, when a sufficient number of experiments is provided to J-DFA, the most preferred candidate is the correct key. However, differently from classical DFA, the clustering approach of J-Linkage makes J-DFA a *robust* technique and this is the main rationale behind the interest in J-Linkage applied to DFA.

We now explain this concept more in detail. First of all, J-DFA is robust against *outliers*. In the context of fault attacks an outlier can be defined as an experiment which has been produced by a fault that does not belong to the fault model H assumed by the attacker. As seen in Chapter 11, the effectiveness of classical DFA is heavily compromised in presence of outliers, leading to either no solutions or a wrong solution for the key. In classical DFA the fault model H cannot be simply set equal to F to avoid the presence of any outlier, because such condition would never converge to a solution.

Instead, J-DFA nicely manages the outliers and it leads to the correct solution provided that it is fed with enough coherent experiments. Furthermore J-DFA manages the case with $H = F$ (i.e. all the possible faults are valid) without any special adaptation. In this way, J-DFA is also robust in the choice of the fault model H. Finally, J-Linkage does not require to know in advance the number of models for the faults that the specific injection technique in place generates. This means that in the extreme case in which the attacker has no knowledge a-priori, the fault model can simply be assumed to $H = F$. Even in this scenario, which is very interesting in practice, J-DFA successfully leads to the correct solution. We will detail the benefits of this property of J-DFA in the next section.

12.4 Evaluation results

In this section we show the results of the application of the J-DFA technique under different attack conditions. All the presented attacks focus on faults injected at the beginning of the last round of AES. As explained in section 12.1, for the sake of simplicity we consider experiments affecting only a single byte at a time. Different conditions for the attack means that case by case we evaluate different hypothesized fault models H assumed by the attacker, while keeping fixed the set of actual faults E.

Since J-DFA relies on the same approaches as those of classical DFA for what regards the injection stage, we focus on the cryptanalysis stage for our evaluation. For this reason, similarly to many DFA works described in literature, we simply generated the experiments through simulations. Namely, we used a software implementation of an unprotected AES modified in order to be able to induce the desired fault at the beginning of the last round. In this way we easily collected a large number of couples (C, C^*), to feed J-DFA.

For the same reason we do not consider any countermeasure against faults in our analysis. In fact, most countermeasures aim at making hard for the attacker to successfully apply the first stage of the DFA, which consists in collecting the observations. Examples of such kind of countermeasures are shielding, sensors, redundancy, including multiple executions of the same operation (see [KSV13]). Still novel DFA attacks are of interest because such countermeasures are not able to cover every possible progress in the injection techniques. Like many previous works on DFA, we focus only on the differential cryptanalysis stage and take for granted that the attacker is able to obtain some observations, since J-DFA applies to the second stage of the attack.

We first show the results of J-DFA applied to the very same conditions described in [Gir03]. Then we extend the analysis to different practical attack scenarios.

12.4.1 J-DFA with profiling

In order to show in detail how J-DFA can be concretely used, we first apply our approach in the classical DFA scenario. DFA attacks assume that the attacker has some kind of a-priori knowledge of the effects produced by the injection technique on the target device. This translates in the fact that the fault model H considered by the attacker perfectly matches the set of faults E that occurs in practice, namely $H = E$.

12.4.1.1 Giraud's Attack

As a running example, the J-DFA technique is applied to Giraud's DFA against AES using the same fault model assumed in [Gir03]. In this case the relationship that binds fault values ε to key values $k \in K$ is the same used in [Gir03] and f_x coincides with the function reported in Eq. (12.4).

The fault model is limited to all the possible single-bit-flips on a byte, i.e.

$$H = \{\texttt{0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80}\}. \tag{12.10}$$

The space of hypothesis provided to J-DFA is then generated by $H \times K$ and consists of $8 \cdot 256$ available models, one for each combination of (ε, k), with $\varepsilon \in H$ and $k \in K = \{\texttt{00,01,...,FE,FF}\}$.

The preference matrix consists in a matrix with $m = 8 \cdot 256$ columns, one for each different model. Each experiment $x = (C, C^*)$ is represented as a row in the matrix, where the i-th value in the row is set to 1 for each column i of a model compatible with the experiment, and set to 0 otherwise. In other words, an experiment x votes for a model (ε, k) if $f_x(k) = \varepsilon$.

In order to validate the effectiveness of J-DFA, we applied the attack as described above, by feeding it with one experiment at a time, until a single candidate for the key is found. We repeated the test 100 times on different experiment datasets, obtaining in all the cases the correct candidate k for the key. On average, 2.1 experiments are necessary to obtain only a single candidate. This result confirms the value provided in [Gir03] and shows that J-DFA is as effective as classical DFA when applied in the same conditions.

This setup, like for the original DFA, is not computationally intensive. When few experiments are used (i.e. < 20), our implementation of J-DFA performs each attack in a negligible time (i.e. < 1 s). Furthermore, when $H = E$ and the number of experiments is comparable with (or lower than) the size of the set H, the attack does not really benefit of the clustering (since likely each cluster will be populated only by a single experiment).

12.4.1.2 Extended fault models

Although [Gir03] only explicitly considers the fault model described in (12.10), the same attack can be trivially extended to other fault models injected at the beginning of the last round of AES. In the same way J-DFA applies by simply changing the set of hypothesis H accordingly to such extended models. Once setup J-DFA, several attacks with different fault models can be easily performed, since they all share the same mapping function that binds faults and key values. Indeed, such map only depends on the point of injection of the fault, namely the beginning of the last round. With this regards, J-DFA can be conveniently used as a tool for the analysis of classical DFA in different conditions. In order to further validate the J-DFA, we tested it under some representative conditions:

Case	Hypothesized fault model ($H = E$)	$\|E\|$	Number of experiments
1	$\{\texttt{0x01}\}$	1	1.9
[Gir03]	Giraud's fault model of Eq. (12.10)	8	2.1
2	$\{\texttt{0x01}, \texttt{0x02}, \ldots, \texttt{0x0F}\}$	15	2.3
3	HammingWeight(ε) $= \{1, 2\}$	36	13.4
4	$\{\texttt{0x01}, \texttt{0x02}, \ldots, \ldots, \texttt{0x7F}\}$	127	210.3

Table 12.1: *Average number of experiments (over 100 trials) required in order to obtain the correct key value under different fault models.*

1. A particular scenario could be the case in which the attacker is able to inject always the fault in a fixed position, for instance a bit flip on the least significant bit. In this case the attacker has a perfect a-priori knowledge of the fault model, which is $H = E = \{\texttt{0x01}\}$. The hypothesis space consists of the 256 possible values of the key associated with that single possible fault $\varepsilon = \texttt{0x01}$.
2. Similarly, different scenarios could consider the attacker able to affect by fault only a fixed portion of the byte. In case the fault affects only the least significant half of the byte, the fault model would include all the 15 possible combinations of 4 bits from `0x01` up to `0x0F`, where `0x00` is excluded because it represents no-fault.
3. Otherwise, the case in which the attacker is able to fault a single bit or a couple of bits. In this case, the fault model is represented by all the combinations of 8 bits with Hamming weight 1 (i.e. 8 faults, the same assumed by [Gir03]) or Hamming weight 2 (i.e. 28 faults). In total the fault model would include 36 different faults.
4. Under even different attack conditions the injection could affect all the bits except the most significant bit, the fault model would comprise all the 127 possible combinations of 7 bits (again the value `0x00` is excluded).

J-DFA has been applied in all the attack conditions listed above, 100 times each. Table 12.1 reports how many experiments are needed on average to obtain the correct value κ as single candidate.

It is worth noting that all these tests assume some a-priori knowledge in order to have $H = E$. This is the assumption made in classical DFA. Thus J-DFA can be considered as a tool to quickly replicate classical attacks. Indeed, the same number of experiments would be required to mount classical DFA in the different scenarios considered above.

12.4.2 J-DFA without profiling

Besides being able to replicate classical DFA, J-DFA becomes particularly interesting in cases where the a-priori knowledge of the fault model is poor or even completely absent. Indeed, thanks to the inherent robustness of the J-Linkage clustering technique introduced in section 12.2, J-DFA can be applied in a wider range of practical conditions compared to the classical attacks. In particular, the tool converges to the correct key even if the fault model H does not perfectly match the actual set of injected faults E, provided that $H \cap E \neq \emptyset$ and that enough experiments are available.

In order to show the robustness of J-DFA in practice, we performed several attacks where the set of injected faults is fixed for all the tests while the fault model varies. For all the tests the injected faults are the 8 single-bit-flip considered in [Gir03], i.e. E is defined as in Eq. (12.10). The set of faults assumed by the attacker, namely H, varies starting from a single fault (i.e. $H = \{\texttt{0x01}\}$), up to covering all the possible 255 faults on a single byte. A total of 255 different conditions are tested, differing each other for the amount of faults included in the fault model. Note that for the purpose of the tests, the faults in H are selected in order to maximize the intersection $E \cap H$.

The setup of the tests just described leads to 3 different kinds of conditions:

Hypothesized fault model ($H \subset E$)	AVG number of required experiments
{0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40}	2.2
{0x01, 0x02, 0x04, 0x08, 0x10, 0x20}	2.5
{0x01, 0x02, 0x04, 0x08, 0x10}	3.5
{0x01, 0x02, 0x04, 0x08}	3.6
{0x01, 0x02, 0x04}	4.8
{0x01, 0x02}	6.3
{0x01}	9.5

Table 12.2: *Average number of experiments (over 100 trials) required in order to obtain the correct key value under different hypothesized fault models belonging to the case $H \subset E$.*

Hypothesized fault model ($H \supset E$)	AVG number of required experiments
{0x01, 0x02, . . . , 0x10}	2.5
{0x01, 0x02, . . . , . . . , 0x20}	4.1
{0x01, 0x02, . . . , . . . , . . . , 0x40}	8.4
{0x01, 0x02, . . . , . . . , . . . , . . . , 0x60}	10.7
{0x01, 0x02, . . . , . . . , . . . , . . . , . . . , 0x80}	11.5
{0x01, 0x02, . . . , . . . , . . . , . . . , . . . , . . . , 0xA0}	13.4
{0x01, 0x02, . . . , . . . , . . . , . . . , . . . , . . . , . . . , 0xFF}	16.9

Table 12.3: *Average number of experiments (over 100 trials) required in order to obtain the correct key value under different hypothesized fault models belonging to the case $H \supset E$.*

(i) $H = E$. This condition represents the case described in [Gir03] and explored in section 12.4.1. The attacker perfectly knows a-priori the set of possible faults injected in practice.
(ii) $H \subset E$. This condition represents the case in which the attacker underestimates the faults that occur in practice and therefore she assumes a fault model that includes only some of the actual faults, but not all.
(iii) $H \supset E$. This condition represents the case in which the attacker overestimates the faults that occur in practice and therefore the fault model includes some faults that never occur in practice.

We do not treat the case $H \cap E = \emptyset$ since in this case no meaningful information can be extracted from the attack and then the method fails. The more general condition $H \cap E \neq \emptyset$ follows from (ii) or (iii). The first case has already been explored in section 12.4.1. Tables 12.2 and 12.3 show the amount of experiments needed on average to get from J-DFA the correct key κ as single candidate for different sizes of the hypothesized fault model H (belonging to the case $H \subset E$ or $H \supset E$ respectively). Each value is averaged over 100 trials on different experiment datasets.

The results show that J-DFA converges to the correct solution in all conditions. In particular it successfully works even in case the whole set of possible faults is assumed in the fault model (i.e. $H = F$, described in the last row of Table 12.3). This case is particularly interesting in practice because it does not require any a-priori knowledge by the attacker on the effects of the injection technique. We recall that the classical approach of DFA cannot manage such condition. In practical scenario the attacker could successfully attack an AES implementation without the need of any characterization of the injection technique. She can generate several experiments (couples of correct and faulty ciphertexts). Then she extracts only the ones involving a single byte of the ciphertext (where likely the injection technique did affect only a single byte in the last round), independently from the kind of faults. Finally, she applies J-DFA with $H = F$ and obtains the correct key.

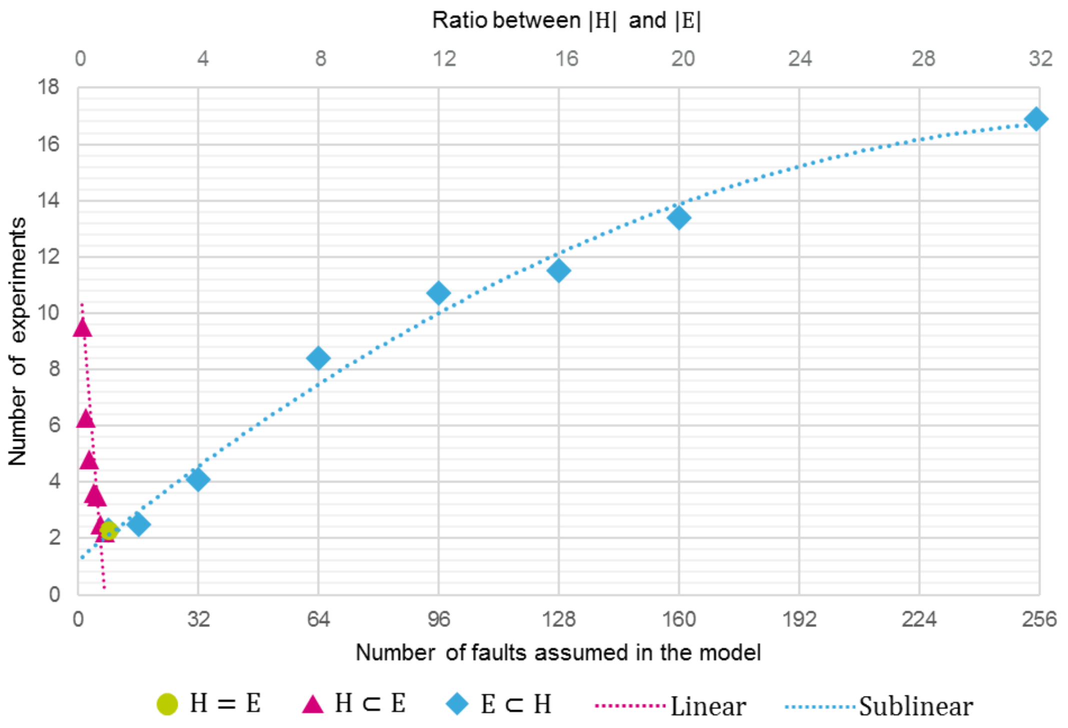


Fig. 12.2: *Efficiency of the attack when the model H and the reality E differ. On the horizontal axis we depict both the dimension of H (below) and the ratio between the dimension of H and E (above).*

The results of the tests also show that the most efficient (in term of number of experiments) condition for the attacker is the case where she perfectly predicts the set of faults that can occur. And this is the condition implicitly assumed in most of classical DFA described in literature.

Another information that the results exhibit is how the efficiency of the attack decreases when the model H and the reality E differ. In Figure 12.2 we depict the results of our experiments and we can see that there are two trends:

- When $H \subset E$ the amount of needed experiments grows linearly, due to the fact that among all the experiments provided to J-DFA some of them are generated by faults not included in H and then they do not provide information about the key. Of course, the more occurring faults do not belong to H, the less experiments are meaningful among all.
- When $H \supset E$ the amount of needed experiments grows sub-linearly, due to the fact that even if all the experiments do provide some information about the correct key, such level of information per experiment decreases.

This means that in practice the best choice for the attacker is to place herself in the case $H = E$, but this requires perfect knowledge on the effects of the injection technique. When the attacker has some uncertainty about the injection, it is preferable to overestimate H rather than underestimate it. And since in a practical scenario the attacker may not know in advance the number of different occurring faults, a larger H is a safer choice, up to the point to simply use $H = F$ and do not rely on any prediction on the fault model.

It is worth noting that, due to its robustness, J-DFA requires different amounts of experiments in different conditions, but always leads to the correct solution.

Instead, classical DFA in case of $H \subset E$ tends to produce no solution (i.e. none of the key candidates are compatible) or a wrong solution (i.e. a key candidate is found but is not the correct key). This effect becomes stronger (i.e. more probable) when the difference between H and E increases.

When $H \supset E$, classical DFA are still able to converge to the correct key provided that enough experiments are available. But they can suddenly reject the correct value of the key when outliers come into the picture.

It is fundamental for a successful DFA to have the guarantee that none of the experiments falls out of the assumed fault model H. In practical scenarios it can be hard to ensure such condition, since often different faults have different probabilities to occur, but rarely the attacker is completely certain about the effects of the faults. And we recall the fact that these outliers cannot be discarded by simply observing the couple (C, C^*).

The usual way to manage such practical condition, besides having a strong a-priori knowledge, is to enlarge the fault model H as much as possible. The concrete issue with this approach in classical DFA is twofold. First, a wide fault model H requires a number of experiments that grows with the size of H, which increases the probability to include an outlier. Second, if any outlier is present among the experiments, with a wide fault model becomes more probable to obtain a wrong key candidate rather than just converge to no solution. And since the attacker can only test the correctness of the whole 16 bytes of the key and not each byte separately, even few wrong candidates per byte can lead to an unfeasible search of the correct key.

This explains why is so desirable the property of robustness to the outliers that J-DFA brings in practical attack scenarios.

Besides the most preferred candidates for the key, J-DFA also identifies the most preferred models (ε, k). This means that it is possible to understand from J-DFA the actual set of faults E that occurred in the experiments used for the attack. Such information may be used to enhance the overall efficiency of the attack, for instance by getting E while attacking the first byte with a wide fault model and then set the refined model (i.e. $H = E$) for the other bytes (assuming that the injection technique affects all the bytes in the same way). Otherwise that property may be exploited to use J-DFA as an analysis tool (rather than for attacks), to characterize the fault models for different injection techniques.

12.4.2.1 Worst case scenario

Like the classical DFA, also J-DFA becomes ineffective when all the faults occur (i.e. $E = F$) with the same probability. This is due to the fact that the injection of a fault does not provide any kind of information and consequently, the attack is useless.

That said, if there is even a small bias in the effects of the injection technique, for instance at least one of the faults is slightly less probable than the others, then J-DFA can be still applied. In this demanding scenario, however, the computational overload becomes considerable and even using a large amount of experiments, the results can still be affected by uncertainty. This because all the experiments are compatible with high probability with all the faults but one, and consequently the greedy segmentation used by J-Linkage fails in finding few predominant keys.

As a reference, we applied J-DFA in the case where all the possible effects occur with the same frequency except one, namely $\varepsilon = \texttt{0xFF}$ never occurs. We set the space of the hypotheses $H = E$, then assuming that the attacker knows a-priori which is the fault that does not occur. Our implementation of J-DFA fed with 4000 experiments, takes about 23 hours to provide the solution, and it returns 16 candidates for the key that are compatible with all the experiments. Figure 12.3 shows the preferences obtained for each candidate of the key. There are 16 peaks, which include the correct key κ, that have similar weights (much higher than all the others). The test on 4000 experiments is shown here to provide an indication about how heavy becomes the computation to converge to a single solution. With less than 4000 experiments, the compatible candidates are still much more than 16. Instead we did not try by further increasing the number of experiments because we already consider 23 hours of computation a substantial effort. Rather than pursuing in that way, we look for an alternative approach to tackle this extreme case.

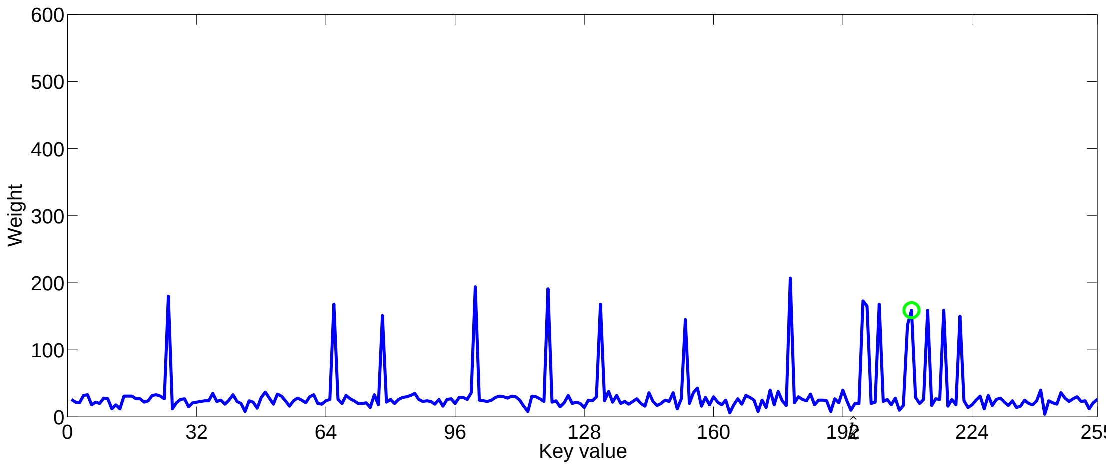


Fig. 12.3: *Preferences (weights) obtained for each candidate of the key, using 4000 experiments. The correct key κ is dotted in green.*

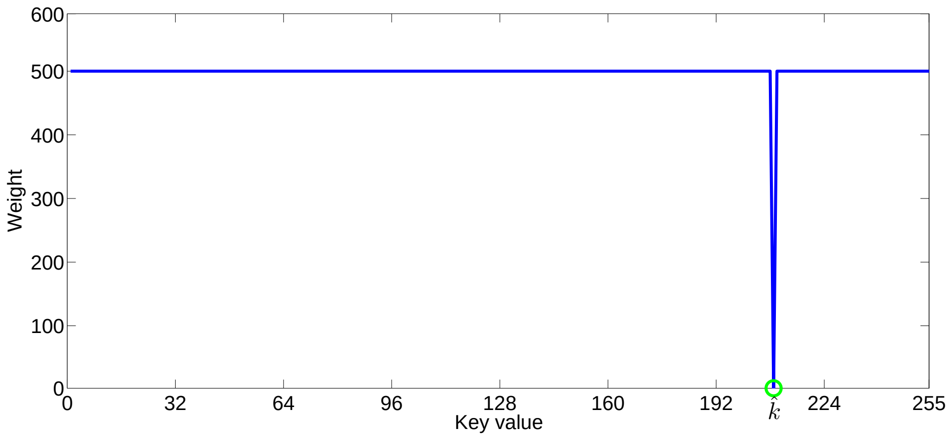


Fig. 12.4: *Preferences (weights) obtained for each candidate of the key using $\tilde{P}$ as preference matrix instead of P, using 1000 experiments. The correct key κ is dotted in green.*

Therefore, in this case, rather than considering the preferences of the experiments, it is convenient to reverse the point of view and consider the negation of the preference matrix:

$$\tilde{P}(i,j) = \begin{cases} 0 & \text{if } x_i \text{ is explained by the } j\text{-th model} \\ 1 & \text{otherwise.} \end{cases} . \tag{12.11}$$

The rationale is that by feeding J-Linkage with the negated preferences, the tool will lead to a wrong set of extracted keys which collects the votes of the majority of data, but the correct key can be singled out as the one that does not receive any vote. We propose to apply J-DFA on $\tilde{P}$ and to change Equation (12.9) in

$$\kappa = \arg\min w. \tag{12.12}$$

It is worth noting that in this complemented case we are using J-Linkage in a non-conventional way, from the perspective of the clustering techniques. In fact, it is highly probable that J-Linkage will not separate data, but rather it will put all the experiments in a single cluster. Still the procedure is meaningful for the fault attack application, because the correct key is among the models that are not preferred by any cluster.

We applied J-DFA on $\tilde{P}$ in the same conditions of the previous test: $H = E$ including all the possible faults with the same frequency except one. In this case we fed J-DFA with 1000 experiments, which required only 3200 seconds of computation.

Figure 12.4 shows the preferences for each candidate. As explained before, the result must be interpreted differently; in fact, we expect the correct candidate to be among the least preferred keys. There are 255 candidates with high level of preference and only a single candidate which has a much lower level of preference equals to zero. This candidate coincides with the correct key κ, suggesting that it is more efficient to derive information about the secret key by observing models that are *not compatible* with the experiments, rather than the preferred ones.

The test reveals that using $\tilde{P}$ is a viable solution to successfully tackle some conditions that are usually hard for the attack.

13

Conclusions

In this part we presented different results regarding differential fault analysis on the AES algorithm.

We have seen that most of the known attacks based on faults against AES specifically target the processed data, i.e. the message or the key. But the memory that stores the intermediate data is not the only portion of the device where faults can occur. Actually, a small number of attacks in literature is based on the alteration of the instruction flow, specifically on the reduction of the number of rounds to one or two.

By extending the idea of targeting the instruction flow, instead of the data, we presented in Chapter 10 some new attacks against AES that exploit misbehaviors of the algorithm execution. In particular, we have shown how a differential fault analysis can be conducted when the main operations that compose the AES round function are corrupted, skipped or repeated during the final round.

The simplest case is when the AddRoundKey operation is jumped. In this case the difference between the correct and faulty ciphertexts corresponds to the last round key for those bytes affected by the fault. Such bytes of the key can thus be immediately retrieved. In the worst case where the fault affects only one byte at a time, sixteen pairs of correct and faulty ciphertexts are required to recover the whole key.

When the operation SubBytes is skipped, or its inverse is executed in its place, the search space for each key byte is reduced to two bits in the worst case. With two pairs of correct and faulty ciphertexts the byte of the key can be uniquely determined. If the fault injection affects multiple bytes contemporary, then a pair of correct and faulty ciphertexts can be used to perform DFA on each affected byte independently. It follows that in case the fault injection affects all bytes, then the whole key can be retrieved with only two pairs of correct and faulty ciphertexts in average. Similar results are obtained when the inverse of the SubBytes operation is executed at its place.

When the ShiftRows operation is jumped on a given row, except the first one, the search space for the corresponding four bytes of the key can be reduced from 2^{32} to a number that depends on the target row. If the target row is the second or the fourth, then with a pair of correct and faulty ciphertexts the search space can be reduced to 2^8. With two pairs in average, the 4-tuple of key bytes can be uniquely retrieved. When the target row is the third, the search space is reduced to 2^{16} with a single pair of correct and faulty ciphertexts. Two pairs or three pairs would allow to uniquely determine the 4-tuple of key bytes. Since the ShiftRows operation acts as the identity for the first row, it is not possible to reduce the search space for the corresponding bytes of the key. So once the other three rows of the key are recovered, the attacker can retrieve the secret key with an exhaustive search on 2^{32} possible values.

If the MixColumns operation is executed also during the last round, the search space for each key column can be reduced from 2^{32} to 2^8 for each experiment. Two pairs in average are enough to uniquely recover the key column. If the fault affects one column at time, then 8 pairs in average are required to retrieve the whole key.

Even if these attacks are presented only theoretically, they highlight the importance of protecting also the instruction flow within the cryptographic algorithm. In fact, most of the countermeasures against fault attacks aim at protecting the internal state or the round keys.

In Chapter 11 we provided an analysis of DFA against AES that highlights the role of the fault model considered by the attacker. As shown in section 11.4, the knowledge that the attacker has on the injected fault is fundamental. Even with a fault model with 99% accuracy the effectiveness of the attack is heavily lowered. The (un)knowledge of the fault by the attacker can at the end remove the basis for an attack based on faults.

With large (relaxed) fault models each experiment is less prone to fall out of the fault model. But they require a sequence of several experiments; in the long run, even a single experiment with a wrong fault model leads to a wrong key (or no solution). Vice versa, with small fault models just few experiments are enough, but in this case there is high probability that the fault will fall out of the model.

Even very relaxed fault models still assume that few effects can never be reached with the specific injection technique on the specific device. This is for instance the case of the attacks presented in [MSS06]. The first attack is based on the knowledge that is impossible by injecting the fault to affect 4 bytes of the state at the same time. This small piece of information leads to the possibility to build the attack. Even by combining the two attacks (with two complementary fault models) the attacker cannot tackle the case where any fault can evenly occur. Still, for each experiment the attacker must know which one of the 2 attacks has to be applied. There the implicit piece of information that the attacker must know is the ability to distinguish experiments that fall in the first fault model from experiments that fall in the second one. And the simple observation of the ciphertexts alone does not provide such information. This implies that the attacker must have another way to get such information, otherwise she will not be able to obtain the secret key.

As highlighted, in order to make a fault attack successful in practice, the attacker must have at least a small piece of information provided by the perturbation of the computation. The worst case for the attack is when the injected faults lead to uniformly distributed effects and the attacker has no way to refine the model. Even a small biasing in the effect that can be recognized by the attacker is enough to make the basis for a successful attack. With this regard, some algorithms are easier to tackle compared with AES. DES for instance propagates the effect of the fault between the two halves of the state in different ways and such difference biases the effect on the ciphertext. The attacker is then able to classify all the experiments by just observing ciphertexts and to identify whether the experiments belong to the fault model or not. In case of AES the knowledge on the fault can be gained by the attacker by either understanding the specific implementation under attack or by characterizing the injection technique in order to build a-priori a good fault model.

Following this perspective, in Chapter 12 we have presented J-DFA: a novel approach for DFA which exploits a robust clustering algorithm tailored to fault analysis.

We argue that the benefit yielded by J-DFA is twofold. First, it is a versatile tool that can be easily used to quickly replicate many classical DFA attacks unified in a common framework. A peculiar result of J-DFA is that, besides the preferred candidate for the key, it also provides the preferred models for the fault. This is a quite remarkable ability because it furnishes precious information which can be used to analyze, compare and characterize different specific injection techniques on different devices. The second benefit is that, thanks to its robustness, J-DFA produces reliable solutions in a wider range of practical scenarios, even if the a-priori knowledge of the attacker is poor or completely absent.

Even if we deal only with faults injected in the last round of AES, our approach could be extended to different positions for the faults (e.g. the attack described in [PQ03]) or even different algorithms, taking advantage of the generality of the J-Linkage conceptual representation. From the theoretical point of view, the only step of the procedure that requires to be adapted is the mapping, which must represent the different injection position or the different algorithm. In practice, attacking a different step of AES (e.g. the second-last round) may lead to a huge number of models to be considered. One possibility to tackle this, could be to exploit the fact that J-DFA is able to converge to a solution even if the fault model does not include all the faults that occur, then intentionally keeping the fault model limited.

However, it is worth noting that in a practical scenario, considering an injection point different than the last round (e.g. the second-last round) is of interest only when the fault cannot be injected in the last round (e.g. a countermeasure protecting only the last round but not the second-last).

Due to the way we constructed J-DFA and its inherent robustness to outliers, the set of fault models can even include and mix faults generated by injections in different stages of the execution (i.e. mixing different mapping), such as faults coming from the last round and from the second-last round.

Part III

Partial key exposure attacks against RSA

14

Introduction

Partial key exposure attacks, introduced by Boneh, Durfee and Frankel in 1998 [BDF98], exploit the knowledge of part of the secret key bits of the RSA algorithm to fully recover the key itself and break the system. The feasibility of such attacks and the number of bits required to mount them depends on the parameter settings, such as the dimension of the public and private exponents. Boneh, Durfee and Frankel, for instance, show that for small public exponents e only a quarter of the least significant bits of the secret exponent is sufficient to factor the RSA modulus and thus recover the secret key. Their method is linear in $e \log_2 e$ and thus feasible as long as e is small enough.

After the seminal work of Boneh et al., several results have been presented. They mainly try to address the question whether similar attacks can be mounted for larger values of e, but also analyze different use cases, such as RSA implementation based on the Chinese Remainder Theorem (RSA-CRT) [QC82] or moduli obtained as the product of prime powers.

All partial key exposure attacks rely on Coppersmith's methods to find small solutions of modular polynomials. This method has been originally presented for univariate modular polynomials in [Cop96b] and subsequently extended to bivariate equations, enabling the factorization of an RSA modulus given half of the bits of one of its prime factors [Cop96a]. These methods make extensive use of lattice theory and translate the problem of finding modular polynomial roots into lattice reduction problems.

The high interest for this family of attacks is mainly motivated by the study of side channel attacks. In such attacks, introduced in 1996 by Paul Kocher [Koc96], some physical information (such as power consumption, electromagnetic emission, acoustic emission, etc.) is extracted by examining the device executing the cryptographic computation and is then used to recover bits of the secret key. Since this procedure may be hard and time consuming, it is very useful to limit the number of bits that the attacker needs to recover. Here is where partial key exposure attacks come in to place. Indeed, it is not necessary for the attacker to recover the whole bits of the key by side channel information, but it is sufficient to recover a part of them and then the whole key can be reconstructed with the application of partial key exposure attacks.

The importance of partial key exposure attacks is given also by the consideration that in many cases some countermeasures can be adopted by implementers to thwart side channel attacks and such countermeasures let the attacker obtain information on only a part of the secret exponent. Indeed, a common countermeasure used for RSA is exponent blinding, originally introduced in [Koc96] but often attributed to [Cor99]. This countermeasure has the feature to change the private exponent at each computation, thus not permitting the use of multiple traces, as required for DPA. This results in the need of using a single trace to discover the secret key.

The question as to whether a partial key exposure attack could be applied in this setting was answered in [JL12]. The authors presented two techniques to recover the full exponent, knowing enough most significant bits or least significant bits of it, leaving the open question as up to which extent it is possible to apply partial key exposure when exponent blinding is applied to the Chinese Remainder Theorem variant of RSA.

In this part of the thesis, we present new methods for partial key exposure attacks when the exponent blinding countermeasure is applied, improving the results of [JL12] for common RSA settings and providing novel attacks for the CRT variant. In both cases, we present results where part of the most or least significant bits are known.

As for RSA, we first provide a technique based on the knowledge of the least significant bits that is more efficient than [JL12], since it requires reducing a lattice basis of lower dimension. Then, we present a method based on the knowledge of the MSBs that reduces the number of required bits compared to [JL12] and moreover we make it to not rely on a common heuristic assumption. For the CRT variant, we present novel attacks since this particular case has never been analyzed before. For all presented methods, we provide experimental results using moduli of 2048 or 3072-bit length.

This part is organized as follows. In Chapter 9, we recall the RSA algorithm and the parameter settings commonly chose for real applications, which are the ones we perform our analysis on. Then we present some previous results on partial key exposure attacks and we give a brief introduction about lattices and Coppersmith's method. In Chapter 16 we present our attacks on RSA and CRT-RSA with exponent blinding and provide experimental results. Conclusions are finally given in Chapter 17.

15

Preliminaries on partial key exposure attacks and RSA

All partial key exposure attacks follow a common methodology, which translates the problem of finding roots of modular polynomials into a lattice reduction problem. Different attacks have been presented since the seminal work of Boneh, Durfee and Frankel [BDF98], each presenting its ad-hoc method to build the desired lattice.

This chapter is mainly focused on introducing the general strategy used in partial key exposure attacks together with an overview of the RSA algorithm and the state of the art on the subject. The chapter is organized as follows. In Section 15.1 we first described the RSA algorithm and then recall the most common parameter settings. In Section 15.2 we give an overview of previous results on partial key exposure attacks against RSA. Finally, in Section 15.3 we briefly introduce lattices and then describe the general strategy of Coppersmith method.

15.1 RSA Applications

A pair of private and public key for RSA is generated as follows.

At first, two large distinct primes p and q are chosen at random. Then the modulus $N = p \cdot q$ is defined together with $\phi(N) = (p-1)(q-1)$, where ϕ denotes the Euler's totient function. Then an integer e is chosen such that $1 < e < \phi(N)$ and $\gcd(e, \phi(N)) = 1$ (i.e., e and $\phi(N)$ are coprime). Finally, the multiplicative inverse of e modulo $\phi(N)$, denoted by d, is computed. Namely, e and d satisfy

$$ed \equiv 1 \bmod \phi(N) \ . \tag{15.1}$$

The pair (N, e) is released as the public key, whereas d is the private key. Notice that also p, q and $\phi(N)$ are kept private, otherwise they can be used to calculate d.

To encrypt a message, it is first turned into an integer m, such that $0 \leq m < N$, and then a modular exponentiation by e is performed. Namely, the ciphertext is computed as $c = m^e \pmod N$. To decrypt the ciphertext c, an exponentiation by d is performed: $m = c^d \pmod N$.

The correctness of the algorithm relies on the Euler's theorem, which states that for each a that is coprime with N the equation $a^{\phi(N)} \equiv 1 \bmod N$ holds. Thus, the following equalities show that m is correctly retrieved:

$$m^{ed} \equiv m^{1+k\phi(N)} \equiv m\left(m^{\phi(N)}\right)^k \equiv m(1)^k \equiv m \bmod N \ . \tag{15.2}$$

The first equality holds since $ed \equiv 1 \bmod \phi(N)$, that implies $ed = 1 + k\phi(N)$ for some integer k.

15.1.1 CRT-RSA

In order to speed up the exponentiation computation, some RSA implementations make use of a technique based on the Chinese Remainder Theorem (CRT) [QC82]. In particular, one can use exponents

$$d_p = d \bmod (p-1) \quad \text{and} \quad d_q = d \bmod (q-1)$$

to compute

$$m_1 = c^{d_p} \bmod p \quad \text{and} \quad m_2 = c^{d_q} \bmod q \; .$$

Then, the value

$$h = q^{-1}(m_1 - m_2) \bmod p$$

is computed and the message retrieved as $m = m_2 + hq$.

15.1.2 Exponent Blinding

The side-channel countermeasure considered in this work is the exponent blinding, introduced by Kocher [Koc96]. It consists of adding a random multiple of $\phi(N)$ to d. In particular, RSA exponentiation is computed by using the new exponent $d^* = d + \ell\phi(N)$, for some $\ell > 0$ randomly chosen at each execution. The correctness of RSA is still valid, since

$$m^{ed^*} \equiv m^{ed+e\ell\phi(N)} \equiv m^{1+(k+e\ell)\phi(N)} \equiv m\left(m^{\phi(N)}\right)^{k+e\ell} \equiv m(1)^{k+e\ell} \equiv m \bmod N \; . \tag{15.3}$$

CRT-RSA can be protected with exponent blinding, too. Thus, exponentiation is computed by using $d_p^* = d_p + \ell_1(p-1)$ and $d_q^* = d_q + \ell_2(q-1)$, for some $\ell_1, \ell_2 > 0$ randomly chosen.

15.1.3 Common Parameters Setting

The modulus N has prime factors p and q that for security purposes are chosen of equal bit-size. We assume wlog that $p > q$, that implies

$$q < \sqrt{N} < p < 2q < 2\sqrt{N}$$

and

$$\sqrt{N} < p + q < 3\sqrt{N} \; .$$

It is common practice to choose the modulus N as 1024, 2048 or 3072-bit long.

The most common value for the public exponent e is $2^{16} + 1$. This is also the default value for the public exponent in the OpenSSL library. Other common values are 3 and 17. NIST mandates that e satisfies $2^{16} < e < 2^{256}$ [KSD13]. Therefore, to be as generic as possible but still adhering to realistic scenarios, we will consider in our analysis $3 \leq e < 2^{256}$. We will then provide experimental results for the most common case $e = 2^{16} + 1$.

The exponent d is commonly chosen to be full size, namely as large as $\phi(N)$. In order to speed-up the decryption process, someone suggests using smaller d. However, this choice may lead to security problems as Wiener's attack [Wie90]. Therefore, it is usually avoided.

The dimension of the random factor ℓ used in the exponent blinding countermeasure is a tradeoff between security and efficiency. If ℓ is 32-bit long or smaller, it allows some combination of brute-forcing and side-channel as in [FKM$^+$06], where a brute-force on ℓ is required. Thus, it is a safer choice to use ℓ with bit-size 64. A larger dimension would make the decryption process less efficient.

In our analysis, to maintain generality, we will consider $0 \leq \ell < 2^{128}$ and in our experiments we will test bit-sizes of 0, 10, 32, 64 and 100. Our methods never require the capability of brute-forcing the values of k or ℓ, sometimes needed in other works.

To recap, in this work we will consider both RSA and CRT-RSA implementations that make use of the exponent blinding countermeasure. Our RSA settings will consider moduli of 1024, 2048 and 3072 bits, public exponent such that $3 \leq e < 2^{256}$, private exponent of full size and a randomization factor up to 128 bits. To derive theoretical bounds in next sections, we prefer to express the restrictions on e and ℓ with respect to the modulus N. In general, we translate them to the less restrictive conditions: $\ell < 2N^{\frac{1}{8}}$ and $e < 2N^{\frac{1}{4}}$. When necessary, we will consider more restrictive bounds. We will run experiments by considering the widely used public exponent $e = 2^{16} + 1$ and random values ℓ of different bit-size from 0 to 100. The modulus N will be 2048 or 3072-bit long, but note that our attacks are effective also for other sizes.

15.2 Partial key exposure attacks

Partial key exposure attacks were introduced by Boneh, Durfee and Frankel in 1998 [BDF98]. In their work, the authors presented several attacks on RSA based on the knowledge of the least significant bits of the private exponent or of the most significant bits of the private exponent [BDF98]. When the LSBs are known they show that a quarter of the private exponent is sufficient to break the system if the public exponent is relatively small, i.e. smaller than $N^{\frac{1}{4}}$. When the MSBs are known, the number of bits that the adversary needs to know depends on his knowledge of e. Supposing that $N^{\frac{1}{4}} < e < N^{\frac{1}{2}}$ and its factorization is known, then at most half of the bits of d is required. The smaller e is, the smaller the number of required bits is. Indeed, when e is close to $N^{\frac{1}{4}}$ then only a quarter of bits of d is sufficient to mount the attack. When the factorization for e is not known and $e < N^{\frac{1}{2}}$, at least half of the bits of d is required. Unlike the previous case, the smaller e is, the bigger the number of required bits is.

In 2003, Blömer and May presented partial key exposure attacks considering larger values of the public exponent e. They show that for $N^{\frac{1}{2}} < e < N^{\frac{3}{4}}$ the number of MSBs of d required to mount the attack increases as e grows. For instance, when e is close to $N^{\frac{1}{2}}$ then half of the bits of d suffice to mount the attack, whereas for e close to $N^{\frac{2}{3}}$ the fraction of required bits is bigger than 80%. When LSBs of the private exponent are known, they provide results for all exponents $e < N^{\frac{7}{8}}$. When e is close to $N^{\frac{1}{2}}$, about 90% of the bits are required, whereas when e is close to $N^{\frac{7}{8}}$, almost all the bits must be known. In this work, Blömer and May provide also results for CRT-RSA. In the case of known LSBs, for low public exponents e (i.e. $e = poly(\log N)$), half of the LSBs of d_p suffice to mount the attack. In the case of known MSBs for $e < N^{\frac{1}{4}}$ again only half of the MSBs of d_p are required.

In [LZL14], Lu et. al. extended the attack for CRT-RSA up to $e < N^{\frac{3}{8}}$ and d_p of full-size. The bigger e is, the bigger the number of required bits of d_p is, for both the MSBs and LSBs cases.

In [EJMdW05], Ernst et al. provided new results in the case e or d is full-size and the other is relatively small. For instance, when d is close to $N^{\frac{1}{3}}$ then a quarter of its MSBs or of its LSBs are sufficient to mount the attack.

All these works consider the private exponent d smaller than N. In 2012, Joye and Lepoint [JL12] analyzed RSA implementations using larger exponents d, which is the scenario of exponent blinding. We will give more details about their results in the next chapter when comparing our approach with their one.

15.3 General Strategy

Partial key exposure attacks rely on Coppersmith's method for finding roots of modular polynomials and multivariate polynomials. This method makes significant use of lattices and lattice reduction algorithms.

We give here a brief introduction to lattices and to the general strategy used in partial key exposure attacks and thus also in our attacks.

15.3.1 Lattices

We start with the definition of lattice in a real vector space, from Janusz [Jan96].

Definition 15.1. *Let V be a n-dimensional R-vector space. Given $\{b_1, \dots, b_n\}$ linearly independent vectors of V, the free abelian group $L = \mathbb{Z}b_1 + \cdots + \mathbb{Z}b_n$ is called a* (full rank) integer lattice *in V. The set $B = \{b_1, \dots, b_n\}$ is the basis of the lattice.*

In other words, the integer lattice spanned by B is the set of all integer linear combinations of vectors of B. Namely, the set $L(B) = \{\sum_i x_i b_i \ : \ x_i \in \mathbb{Z}\}$.

The $(n \times n)$-matrix consisting of the row vectors $b_1, \ldots, b_n$ is called basis matrix and is denoted by $\mathbf{B}$. Every lattice has an infinite number of lattice bases. It follows from the following proposition.

Proposition 15.1. *If B_1 and B_2 are two lattice bases of L, then there exists an unimodular*[1] *n-dimensional matrix $\boldsymbol{U}$ s.t. $\boldsymbol{B}_1 = \boldsymbol{U} \times \boldsymbol{B}_2$.*

Hence a basis is obtained from another through a unimodular transformation. It follows that $\det(\mathbf{B})$ is an invariant of L, namely it is independent of the choice of the basis, and we refer to it as $\det(L)$. The dimension of the lattice is $\dim(L) = n$.

Informally, we say that some bases of L are *good* and some are *bad*. A basis is good, if its vectors are reasonably short and nearly orthogonal.

For any basis $B = \{b_1, \ldots, b_n\}$, it holds that $\prod_{i=1}^{n} \|b_i\| \geq \det(L)$. Thus, good bases come closer to equality than bad ones.

The goal of lattice reduction algorithms is to find a basis with good vectors. The LLL algorithm [LLL82] produces in polynomial time a set of reduced basis vectors whose norm is bounded by the following theorem.

Theorem 15.1 (Lenstra-Lenstra-Lovász). *Let L be a lattice of dimension n. The LLL-algorithm outputs in polynomial time reduced basis vectors v_i, $1 \leq i \leq n$, satisfying*

$$\|v_1\| \leq \|v_2\| \leq \ldots \leq \|v_i\| \leq 2^{\frac{n(n-1)}{4(n+1-i)}} \det L^{\frac{1}{n+1-i}} \ .$$

15.3.2 General Strategy

In [Cop96b], Don Coppersmith presents a rigorous method to find small roots of univariate modular polynomials. The method is based on LLL and can be extended to polynomials in more variables, but only heuristically.

In this work we use the following reformulation of Coppersmith's theorem due to Howgrave-Graham [How97].

Theorem 15.2 (Howgrave-Graham). *Let $f(x_1, \ldots, x_k)$ be a polynomial in k variables with n monomials. Let m be a positive integer. Suppose that*

1. $f(r_1, \ldots, r_k) = 0 \bmod b^m$ where $|r_i| < X_i \ \forall\ i$, for some X_i;
2. $\|f(x_1X_1, \ldots, x_kX_k)\| < \dfrac{b^m}{\sqrt{n}}$, with $X_1, \ldots, X_k$ bounds on the root as in item 1.

Then $f(r_1, \ldots, r_k) = 0$ holds over the integers.

The general strategy is the following. Starting from an RSA equation we construct a multivariate polynomial $f_b(x_1, \ldots, x_k)$ modulo an integer b, such that its root $(r_1, \ldots, r_k)$ contains secret values. Our goal is to find this root, even if no classic root finding method is known for modular polynomials. So, we construct k polynomials $f_1, \ldots, f_k$ satisfying the two conditions of Theorem 15.2 so that such polynomials will have the same root $(r_1, \ldots, r_k)$ over $\mathbb{Z}$. Finally, we compute the common roots of these polynomials and recover the secret values.

To generate such polynomials, we apply the following strategy. Starting from f_b we construct auxiliary polynomials $g_i(x_1, \ldots, x_k)$ that all satisfy condition 1 of Howgrave-Graham's Theorem. Since every integer linear combination of these polynomials also satisfies condition 1, we look for linear combinations that also satisfy condition 2. Such combinations will be the polynomials $f_1, \ldots, f_k$.

To find such polynomials $f_1, \ldots, f_k$, we build a lattice $L(B)$ where the basis B is composed by the coefficient vectors of the polynomials $g_i(x_1X_1, \ldots, x_kX_k)$ (with $X_1, \ldots, X_k$ bounds on the root as in Theorem 15.2).

[1] $\mathbf{U}$ is unimodular if it has integer entries and $\det(U) = \pm 1$.

By using the LLL-lattice reduction algorithm, we obtain a reduced basis for the lattice L as in Theorem 15.1. The first k vectors of the reduced basis have norm smaller than $\frac{b^m}{\sqrt{n}}$, if:

$$2^{\frac{n(n-1)}{4(n+1-k)}} \det L^{\frac{1}{n+1-k}} < \frac{b^m}{\sqrt{n}} .$$

We may let terms that do not depend on N contribute to an error term ϵ and consider the simplified condition

$$\det L \leq b^{m(n+1-k)} . \tag{15.4}$$

If this condition holds, then we can use the first k reduced-basis vectors to construct the polynomials $f_1, \dots, f_k$ satisfying the second condition of Theorem 15.2.

Subsequently, in order to compute $(r_1, \dots, r_k)$, we do the following.

If $k = 1$, then we consider the polynomial $F = f_1(x_1)$ and apply a classic roots finding algorithm for univariate polynomials over the integers.

If $k > 1$, we use the resultant computation to construct k univariate polynomials $F_i(x_i)$ from $f_1, \dots, f_k$ and apply a classic roots finding algorithm for each of them. The effectiveness of this last method relies on the following heuristic assumption.

Assumption 1 *The resultant computation for the polynomials f_i described above yields a non-zero polynomial.*

This assumption is fundamental and widely used for many partial key exposure attacks in literature [JL12, LZL14, BM03, BDF98, EJMdW05]. None of our experiments has ever failed to yield a non-zero polynomial and hence to mount the attack.

In this work we will make use of a seminal result due to Coppersmith, based on the strategy described above. We present here a more general variant of it, due to May [May03], together with a sketch of its proof to illustrate how we will construct lattices for our experiments.

Theorem 15.3. *Let $N = pq$ with $p > q$. Let k be an unknown integer that is not a multiple of q. Suppose we know an approximation $\widetilde{kp}$ of kp with $|kp - \widetilde{kp}| \leq 2N^{\frac{1}{4}}$. Then we can factor N in time polynomial in* $\log N$.

Proof (Sketch of proof). Define the univariate polynomial

$$f_p(x) = x + \widetilde{kp}$$

with root $x_0 = kp - \widetilde{kp}$ modulo p.

Divide the interval $[-2N^{\frac{1}{4}}, 2N^{\frac{1}{4}}]$ into 8 subintervals of size $\frac{1}{2}N^{\frac{1}{4}}$ centered at some x_i. For each subinterval consider the polynomial $f_p(x - x_i)$ and find its roots r such that $|r| \leq \frac{1}{4}N^{\frac{1}{4}}$. Among all these roots of all these polynomials there is also x_0. So, for each $f_p(x - x_i)$ set $X = \frac{1}{4}N^{\frac{1}{4}}$. Fix $m = \lceil \log N/4 \rceil$ and set $t = m$.

Define the auxiliary polynomials

$$\begin{aligned} g_{i,j}(x) &= x^j N^i f^{m-i} \text{ for } i = 0, \dots, m-1;\ j = 0 ; \\ h_i(x) &= x^i f^m(x) \text{ for } i = 0, \dots, t-1 . \end{aligned}$$

and construct the lattice spanned by the vectors $g_{i,j}(xX)$ and $h_i(xX)$.

By applying the LLL-algorithm to L, a reduced basis is obtained. From the shortest vector construct the polynomial $f_i(x)$. Among its roots over the integers, there are also the roots of $f_p(x - x_i)$. Compute the roots of $f_i(x)$ by using a classic roots-finding algorithm. Construct the set R of all integer roots of the polynomials $f_i(x)$. The set R will contain also the root x_0.

Thus, $f(x_0) = kp$ can be computed and, since k is not a multiple of q, the computation of $\gcd(N, kp)$ gives p.

Recall that the LLL-algorithm is polynomial in the dimension of the matrix basis and in the bit-size of its entries. Since the dimension of the lattice is $m + t = \lceil \log N/2 \rceil$ and the bit-size of its entries is bounded by a polynomial in $(m \log N)$, every step of the proof can be done in polynomial time. □

16

New attacks on RSA with exponent blinding

16.1 Attacks on RSA

In this section we present two attacks on RSA implementations, one given the most significant bits of the private exponent and the other one given its least significant bits. We assume that the private exponent d is full-size and that it is masked by a random multiple ℓ of $\phi(N)$. Thus, exponentiation is performed by using the exponent $d^* = d + \ell\phi(N)$ for some $\ell \geq 0$. When $\ell = 0$ clearly $d^* = d$, that means that no countermeasure is applied.

16.1.1 Partial Information on LSBs of d^*

In this section, we assume that the attacker is able to recover the least significant bits of the secret d^*. We write $d^* = d_1 \cdot M + d_0$, where d_0 represents the fraction of d^* known to the attacker while d_1 represents the unknown part. For instance, if the attacker knows the m LSB of d^*, then $M = 2^m$.

To prove our result, we generalize the method used in [BM03], by introducing the new factor ℓ.

Theorem 16.1. *Let (N, e) be an RSA public key with $e = N^\alpha \leq 2N^{\frac{1}{4}}$ and $d^* = d + \ell\phi(N)$, for some $\ell = N^\sigma \leq 2N^{\frac{1}{8}}$. Suppose we are given d_0 and M satisfying $d_0 = d^* \mod M$ with*

$$M \geq N^{\frac{1}{3}\sqrt{1+6(\alpha+\sigma)}+\frac{1}{6}(1+6\sigma)+\varepsilon} ,$$

for some $\varepsilon > 0$. Then, under Assumption 1, we can find the factorization of N in time polynomial in $\log N$.

Proof. We start from the RSA equation

$$ed - 1 = k\phi(N) .$$

Since $d^* = d + \ell\phi(N)$, we obtain the equation

$$ed^* - 1 = (k + e\ell)\phi(N) .$$

Let $k^* = k + e\ell$, so that $ed^* - 1 = k^*\phi(N)$.
By writing $d^* = d_1 M + d_0$ and considering that $\phi(N) = N - (p + q - 1)$, we get

$$k^* N - k^*(p + q - 1) - ed_0 + 1 = eMd_1 .$$

It follows that the bivariate polynomial

$$f_{eM}(x, y) = xN - xy - ed_0 + 1$$

has root $(x_0, y_0) = (k^*, p+q-1)$ modulo eM.
In order to bound x_0, notice that

$$k^* = \frac{ed^* - 1}{\phi(N)} < e\left(\frac{d + \ell\phi(N)}{\phi(N)}\right) < e(1+\ell) \leq 2N^{\alpha+\sigma} .$$

In addition, recall that $p + q \leq 3N^{\frac{1}{2}}$.
We can set the bounds $X = 2N^{\alpha+\sigma}$ and $Y = 3N^{\frac{1}{2}}$ so that $x_0 \leq X$ and $y_0 \leq Y$.
To construct the lattice, we consider the following auxiliary polynomials

$$\begin{aligned} g_{i,j}(x,y) &= x^i (eM)^i f_{eM}^{m-i} \text{ for } i = 0,\dots,m;\ j = 0,\dots,i; \\ h_{i,j}(x,y) &= y^j (eM)^i f_{eM}^{m-i} \text{ for } i = 0,\dots,m;\ j = 1,\dots,t , \end{aligned}$$

for some integers m and t, where $t = \tau m$ has to be optimized.
All integer linear combinations of these polynomials have the root (x_0, y_0) modulo $(eM)^m$, since they all have a term $(eM)^i f_{eM}^{m-i}$. So, the first condition of Theorem 15.2 is satisfied. In order to satisfy the second condition, we must find a short vector in the lattice spanned by $g_{i,j}(xX, yY)$ and $h_{i,j}(xX, yY)$. In particular, this vector shall have a norm smaller than $\frac{(eM)^m}{\sqrt{\dim L}}$.
The second condition of Theorem 15.2 is satisfied when inequality (15.4) holds, i.e. if

$$\det L \leq (eM)^{m(n-1)} . \tag{16.1}$$

An easy computation shows that $n = \left(\tau + \frac{1}{2}\right) m^2$ and that

$$\det L(M) = \left((eMX)^{3\tau+2} Y^{3\tau^2+3\tau+1}\right)^{\frac{1}{6}m^3(1+o(1))} .$$

Considering the bounds $X = 2N^{\alpha+\sigma}$ and $Y = 3N^{\frac{1}{2}}$, we obtain the condition

$$\left((eM2N^{\alpha+\sigma})^{3\tau+2}(3N^{\frac{1}{2}})^{3\tau^2+3\tau+1}\right)^{\frac{1}{6}m^3(1+o(1))} \leq (eM)^{m(n-1)}$$

that reduces to

$$N^{\frac{m^3}{6}\left((\alpha+\sigma)(3\tau+2)+\frac{1}{2}(3\tau^2+3\tau+1)\right)(1+o(1))} \leq (eM)^{m(n-1)-\frac{m^3}{6}(3\tau+2)(1+o(1))} .$$

We know that $eM \geq N^{\alpha\frac{1}{3}\sqrt{1+6(\alpha+\sigma)}+\frac{1}{6}(1+6\sigma)+\varepsilon}$, so the above condition is satisfied if

$$9\tau^2 + 6(\alpha+\sigma+\tau) - 2\sqrt{1+6(\alpha+\sigma)}(1+3\tau) + 2 \leq 0 .$$

The left-hand side is minimized, for

$$\tau = \frac{1}{3}\left(\sqrt{1+6(\alpha+\sigma)} - 1\right) .$$

Thus, for this choice of τ condition 16.1 is satisfied so we can successfully apply the LLL-algorithm.

From the LLL-reduced basis, we construct two polynomials $f_1(x,y)$, $f_2(x,y)$ with the common root (x_0, y_0) over the integers. By the heuristic assumption, the resultant $res_x(f_1, f_2)$ is not zero and we can find $y_0 = p + q - 1$ using standard root finding algorithms. This gives us the factorization of N.

To conclude the proof, we need to show that every step of the method can be done in time polynomial in log(N). The LLL-algorithm runs in polynomial time, since the basis matrix B has constant dimension (fixed by m) and its entries are bounded by a polynomial in N. Additionally, $res_x(f_1, f_2)$ has constant degree and coefficients bounded by a polynomial in N. Thus, every step can be done in polynomial time. □

We would like to make two considerations. The first is that when $\sigma = 0$, we get the same result of [BM03]. Indeed, our method is a generalization of it. The second is that we obtain the same bound of [JL12], but our approach is more effective in practice. As we will show in Section 16.3.1, we are able to get closer to the theoretical bound by using smaller lattices.

16.1.2 Partial Information on MSBs of d^*

In this section, we prove that if the attacker knows a sufficiently large number of most significant bits of the protected exponent, then she can factor N. To prove this result, we show how the partial knowledge on d^* can be used to construct an approximation of p that allows to apply Theorem 15.3.

The advantage of this approach compared to [JL12] is that it does not rely on the heuristic assumption 1 and yields to a better bound.

Theorem 16.2. *Let (N, e) be an RSA public key with $e = N^\alpha$ and $d^* = d + \ell\phi(N)$ for some $\ell = N^\sigma$ with $\sigma > 0$ and $N^{\alpha+\sigma} < 2N^{\frac{3}{8}}$. Suppose that $|p-q| \geq cN^{\frac{1}{2}}$, for some $c \leq \frac{1}{2}$, and suppose we are given an approximation $\widetilde{d^*}$ of d^* such that*

$$|d^* - \widetilde{d^*}| \leq cN^{\frac{1}{4}+\sigma} .$$

Then we can find the factorization of N in time polynomial in $\log N$.

Notice that, like in Theorem 16.1, we have $ed^* - 1 = k^*\phi(N)$ with $k^* = k + e\ell$. In order to prove Theorem 16.2 we need first to prove the following lemma.

Lemma 16.1. *With $N^{\alpha+\sigma} < 2N^{\frac{3}{8}}$, given $\widetilde{d^*}$ such that $|d^* - \widetilde{d^*}| \leq \frac{1}{4}N^{1-\alpha}$ then the approximation $\widetilde{k^*} := \left\lceil \frac{e\widetilde{d^*}-1}{N+1} \right\rceil$ of k^* is exact.*

Proof. This proof follows the same strategy used in the proof of Theorem 6 of [BM03]. Note that

$$\begin{aligned} |k^* - \widetilde{k^*}| &< \left| \frac{ed^*-1}{\phi(N)} - \frac{e\widetilde{d^*}-1}{N+1} \right| \\ &< \left| \frac{(ed^*-1)(N+1) - (e\widetilde{d^*}-1)(N+1-(p+q))}{\phi(N)(N+1)} \right| . \end{aligned}$$

Then, given that $\phi(N) > N/2$, $p+q \leq 3N^{\frac{1}{2}}$, $N^2 + N > N^2$ and $d^* < 2N^{1+\sigma}$, we obtain

$$\begin{aligned} |k^* - \widetilde{k^*}| &< \left| \frac{e(d^* - \widetilde{d^*})}{\phi(N)} \right| + \left| \frac{(p+q)(e\widetilde{d^*}-1)}{\phi(N)(N+1)} \right| \\ &< \left| \frac{\frac{1}{4}N^\alpha N^{1-\alpha}}{\frac{N}{2}} \right| + \left| \frac{6N^{\frac{3}{2}+\alpha+1+\sigma}}{\frac{N}{2}(N+1)} \right| \\ &< \frac{1}{2} + 12N^{-\frac{1}{2}+\frac{3}{8}} < \frac{1}{2} + \frac{12}{N^{\frac{1}{8}}} . \end{aligned}$$

With RSA parameters, we have $12 \ll N^{1/8}$, so we can safely assume $|k^* - \widetilde{k^*}| < 1$. But the difference between two integers is an integer, thus we can conclude that it is zero, therefore $\widetilde{k^*} = k^*$. □

It is worth to observe two facts: first, the bound $|d^* - \widetilde{d^*}| \leq \frac{1}{4}N^{1-\alpha}$ requires the attacker to get the $(\log_2(N^{\sigma+\alpha}) + 2)$ most significant bits of d^*, a result which holds even for $\sigma = 0$ (i.e. $d^* = d$); second, the assumption $N^{\alpha+\sigma} < 2N^{\frac{3}{8}}$ of Lemma 16.1 always holds for our choice of RSA parameters.

We can now prove Theorem 16.2.

Proof (Proof of theorem 16.2). We begin by applying Lemma 16.1 to obtain the value of k^*. The condition $|d^* - \widetilde{d^*}| \leq \frac{1}{4}N^{1-\alpha}$ of the lemma is always satisfied by our choices of RSA parameters because $\frac{1}{2}N^{\frac{1}{4}+\sigma} \ll \frac{1}{4}N^{1-\alpha}$, since $N^\sigma < 2N^{\frac{1}{8}}$ and $N^\alpha < 2N^{\frac{1}{4}}$.
We can define an approximation $\tilde{s}$ of $s = p + q$ as

$$\tilde{s} := 1 + N - \frac{e\widetilde{d^*} - 1}{k^*} \ .$$

Reminding that k^*, with the assumption of $\sigma > 0$, is lower bounded by $N^{\alpha+\sigma}$, we obtain

$$|s - \tilde{s}| = \left| \frac{e}{k^*} \left(d^* - \widetilde{d^*} \right) \right| \leq \frac{N^\alpha}{N^{\alpha+\sigma}} c N^{\frac{1}{4}+\sigma} \leq c N^{\frac{1}{4}} \ .$$

We use $\tilde{s}$ to define

$$\tilde{p} := \frac{1}{2} \left(\tilde{s} + \sqrt{\tilde{s}^2 - 4N} \right)$$

as an approximation of p.
Without loss of generality, following Appendix B of [BDF98], we now assume that $\tilde{s} \geq s$, so that $\tilde{p} \geq p$.
Observe that

$$\begin{aligned} \tilde{p} - p &= \frac{1}{2}(\tilde{s} - s) + \frac{1}{2} \left(\sqrt{\tilde{s}^2 - 4N} - \sqrt{s^2 - 4N} \right) \\ &= \frac{1}{2}(\tilde{s} - s) + \frac{(\tilde{s} + s)(\tilde{s} - s)}{2 \left(\sqrt{\tilde{s}^2 - 4N} + \sqrt{s^2 - 4N} \right)} \ . \end{aligned}$$

Since $\tilde{s} \geq s$, we have $\tilde{s}^2 - 4N \geq s^2 - 4N = (p - q)^2$ and $|p - q| \geq cN^{\frac{1}{2}}$ with $c \leq \frac{1}{2}$.
Noting that $\tilde{s} \leq s + cN^{\frac{1}{4}}$, we have

$$\tilde{s} + s \leq 2s + cN^{\frac{1}{4}} \leq 2(p + q) + N^{\frac{1}{4}} \leq 6N^{\frac{1}{2}} + N^{\frac{1}{4}} \leq 7N^{\frac{1}{2}} \ .$$

It follows that

$$\begin{aligned} \tilde{p} - p &\leq \frac{1}{2}(\tilde{s} - s) + \frac{(\tilde{s} + s)(\tilde{s} - s)}{4(p - q)} \\ &\leq \frac{1}{2} cN^{\frac{1}{4}} + \frac{(7N^{\frac{1}{2}})(cN^{\frac{1}{4}})}{4cN^{\frac{1}{2}}} \leq \frac{1}{4} N^{\frac{1}{4}} + \frac{7}{4} N^{\frac{1}{4}} \leq 2N^{\frac{1}{4}} \ . \end{aligned}$$

Since the approximation $\tilde{p}$ satisfies the hypothesis of Theorem 15.3 with $k = 1$, we can find the factorization of N in time polynomial in $\log N$. □

From Theorem 16.2 we can recover the minimum number of known MSBs required. In accordance to previous sections we define this quantity as $\log_2 M$ where M is defined as

$$M = \frac{d^*}{|d^* - \widetilde{d^*}|} = \frac{2N^{1+\sigma}}{cN^{\frac{1}{4}+\sigma}} = \frac{2}{c} N^{\frac{3}{4}} \geq 4N^{\frac{3}{4}} \ . \tag{16.2}$$

It is important to underline that this bound is not affected by the size of α and σ as long as the condition of Lemma 16.1 holds. In fact, while it might seem counter-intuitive, the presence of the countermeasure (i.e. $\sigma > 0$) improves the theoretical bound $|d - \tilde{d}| \leq cN^{\frac{1}{4}-\alpha}$ of Theorem 3.3 of [BDF98]. However, this difference was not shown in the experimental results, probably due to low value of α when $e = 2^{16} + 1$.

Also note that Theorem (16.2) provides a significant improvement over the bound of [JL12]. In fact, for $\alpha + \sigma \leq \frac{1}{2}$ (which is always true in our setting), their bound is $|d^* - \widetilde{d^*}| \leq N^{\alpha+\sigma}$, which would require knowledge of $\log_2(N^{1-\alpha})$ bits.

16.1.2.1 Considerations on c

It can be noted from equation (16.2) that the required number of bits to be recovered depends on c which is unknown to the attacker. It's easy to show that c is closely related to $\frac{1}{2^{i+1}}$ where i is the number of most significant bits that p and q have in common. While it is true that attacker has no a priori knowledge of c and thus can't a priori know how many bits she needs to recover before being able to apply Theorem 16.2, it is also true that she can get its exact value after recovering the required minimum bits $\log_2(4N^{\frac{3}{4}})$. In fact, she can compute $\tilde{p}$ and $\tilde{q} = \frac{N}{\tilde{p}}$ and retrieve c which is lower bounded by NIST in the condition $|p - q| > 2^{\log_2(N)/2-100}$ so that $\log_2(4N^{\frac{3}{4}})$ are always enough to compute it.

16.1.2.2 Attack using both MSBs and LSBs of $\mathbf{d}^*$

We want to briefly analyze also the case where the attacker might be able to detect bits in different positions of d^*. In this scenario, the attacker could obtain enough most significant bits to satisfy Lemma 16.1 and obtain $\frac{1}{4}\log_2 N$ least significant bits to recover half of the bits of p and factor N, as shown in [BDF98]. Thus, the knowledge of only $(\log_2(N^{\frac{1}{4}+\sigma+\alpha})+2+\epsilon)$ bits and the resolution of an univariate equation are required. We don't describe the attack in detail because, once k^* is recovered applying Lemma 16.1, it reduces to the method of [BDF98]. Thus, we remind the reader to it. In Section 16.3, we will provide experimental results.

16.2 Attacks on CRT-RSA

In this section we present two attacks on CRT-RSA implementations, where we target exponentiation by d_p^*. One is based on the knowledge of the most significant bits of the CRT private exponent and one is based on the knowledge of its least significant bits. We assume that the private exponent d_p is full-size (with respect to p) and that it is masked by a random multiple ℓ of $(p-1)$, for some $\ell \geq 0$. When $\ell = 0$ clearly $d_p^* = d_p$, that means that no countermeasure is applied.

16.2.1 Partial Information on LSBs of $\mathbf{d}_\mathbf{p}^*$

Assuming that the attacker is able to recover the least significant bits of the secret d_p^*, we can write $d_p^* = d_1 \cdot M + d_0$ where d_0 is known while d_1 is unknown. The integer M is a power of two and represents the bound on the known part.

To prove our result, we use a method presented by Herrmann and May to find the solutions of a bivariate linear equation modulo p [HM08].

Theorem 16.3. *Let (N,e) be an RSA public key with $e = N^\alpha$. Let $d_p = d \bmod p-1$ and let $d_p^* = d + \ell(p-1)$ for some $\ell = N^\sigma$ with $\sigma \geq 0$. Suppose that $N^{\alpha+\sigma} \leq N^{\frac{1}{\sqrt{2}}-\frac{1}{2}}$ and that we are given d_0 and M satisfying $d_0 = d_p^* \bmod M$ with*

$$M \geq N^{1-\frac{1}{\sqrt{2}}+\alpha+2\sigma+\varepsilon} ,$$

for some $\epsilon > 0$. Then, under Assumption 1, we can find the factorization of N (in time polynomial in $\log N$).

Proof. We start from the equation

$$ed_p - 1 = k_p(p-1) .$$

Since $d_p^* = d_p + \ell(p-1)$, we obtain

$$ed_p^* - 1 = (k_p + e\ell)(p-1) .$$

Let k_p^* denote $k_p + e\ell$. By writing $d_p^* = d_1 M + d_0$, we obtain the following equation

$$eMd_1 + k_p^* + ed_0 - 1 = k_p^* p .$$

It follows that the bivariate polynomial

$$f_p(x,y) = eMx + y + ed_0 - 1$$

has root $(x_0,y_0) = (d_1,k_p^*)$ modulo p.
In order to bound y_0, notice that

$$k_p^* = \frac{ed_p^* - 1}{(p-1)} < e\left(\frac{d_p + \ell(p-1)}{(p-1)}\right) < e(1+\ell) \leq 2N^{\alpha+\sigma} .$$

Additionally, recall that $d_1 = \frac{d_p^*}{M} - d_0$.

We can set bounds $X = N^{\frac{1}{\sqrt{2}} - \frac{1}{2} - \alpha - \sigma}$ and $Y = 2N^{\alpha+\sigma}$ so that $x_0 \leq X$ and $y_0 \leq Y$.

To construct the lattice, we consider the following auxiliary polynomials:

$$\bar{f} = x + Ry + R(ed_0 - 1) \text{ where } R = (eM)^{-1} \bmod N ;$$
$$g_{k,i} = y^i \bar{f}^k N^{\max\{t-k,0\}},\ k = 0, \ldots, m; i = 0, \ldots, m-k .$$

for some integers m and t, where $t = \tau m$ has to be optimized.

All integer linear combinations of these polynomials share the root (x_0, y_0) modulo p^t. Thus, the first condition of Theorem 15.2 is satisfied. In order to satisfy the second condition, we have to find a short vector in the lattice L, spanned by $g_{k,i}(xX, yY)$. In particular, this vector shall have a norm smaller than $\frac{p^t}{\sqrt{\dim L}}$.

The second condition of Theorem 15.2 is satisfied when equation (15.4) holds, i.e. when

$$\det L \leq N^{\frac{1}{2}\tau m(n-1)} . \tag{16.3}$$

A straightforward computation shows that $n = \frac{1}{2}(m^2 + 3m + 2)$ and that

$$\det L(M) = (XY)^{\frac{1}{6}(m^3+3m^2+2m)} N^{\frac{1}{6}m\tau(m\tau+1)(4+3m-m\tau)} .$$

Thus, condition (16.3) becomes

$$(XY)^{\frac{1}{6}(m^3+3m^2+2m)} \leq N^{\frac{1}{4}\tau m(m^2+3m) - \frac{1}{6}m\tau(m\tau+1)(4+3m-m\tau)}$$

that reduces to

$$XY \leq N^{\frac{1}{2}(3\tau + 2\tau^3 - 6\tau^2)} .$$

Since $XY = 2N^{\frac{1}{\sqrt{2}} - \frac{1}{2}}$, the above condition is satisfied if

$$\frac{1}{\sqrt{2}} - \frac{1}{2} - \frac{1}{2}(3\tau + 2\tau^3 - 6\tau^2) \leq 0 .$$

The left-hand side is minimized for $\tau = 1 - \frac{1}{\sqrt{2}}$. For this choice of τ condition (16.3) is satisfied, so we can successfully apply the LLL-algorithm and then find the root (d_1, k_p^*). From this values, we can obtain $p - 1$ and then the factorization of N.

To conclude the proof, we need to show that every step of the method can be done in time polynomial in log(N). The LLL-algorithm is polynomial in the dimension of the matrix, that is $\mathcal{O}(m^2)$, and in the bit-size of its entries, that are $\mathcal{O}(m \log N)$. Additionally, $res_y(f_1, f_2)$ has constant degree and coefficients bounded by a polynomial in N. Thus, every step can be done in polynomial time. □

16.2.2 Partial Information on MSBs of $\mathbf{d_p^*}$

In this section, we prove that if the attacker knows a sufficiently large number of most significant bits of the protected exponent d_p^*, then she can factor N.

To prove this result, we show how the partial knowledge on d_p^* can be used to construct an approximation of a multiple of p that allows to apply Theorem 15.3.

Theorem 16.4. *Let (N, e) be an RSA public key with $e = N^\alpha$. Let $d_p = d \bmod p - 1$ and let $d_p^* = d_p + \ell(p-1)$, for some $\ell = N^\sigma$ with $\sigma \geq 0$. Suppose that $N^{\alpha+\sigma} \leq \frac{1}{2}N^{\frac{1}{4}}$ and that we are given an approximation $\widetilde{d_p^*}$ of d_p^* such that*

$$|d_p^* - \widetilde{d_p^*}| \leq N^{\frac{1}{4} - \alpha} .$$

Then, we can find the factorization of N in time polynomial in $\log N$.

Proof. We start from equation

$$ed_p^* - 1 = k_p^*(p-1)$$

with $k_p^* = k_p + \ell e$.
Note that $k_p^* \leq 2N^{\alpha+\sigma} < \frac{1}{2}N^{\frac{1}{2}}$ implies that q can't divide k_p^*.
We compute an approximation

$$\widetilde{k_p^* p} := e\widetilde{d_p^*} - 1$$

of $k_p^* p$, up to an additive error of at most

$$\begin{aligned} |k_p^* p - \widetilde{k_p^* p}| &= |ed_p^* - 1 + k_p^* - e\widetilde{d_p^*} + 1| \\ &= |e(d_p^* - \widetilde{d_p^*}) + k| \leq N^{\frac{1}{4}} + 2N^{\alpha+\sigma} \leq 2N^{\frac{1}{4}} \ . \end{aligned}$$

Since the approximation $\widetilde{k_p^* p}$ satisfies the hypothesis of Theorem 15.3, we can find the factorization of N in time polynomial in $\log N$. □

The bound of Theorem 16.4 implies that an attacker has to know at least $\log_2 M$ bits, where

$$M = \frac{d_p^*}{|d_p^* - \widetilde{d_p^*}|} = \frac{2N^{\frac{1}{2}+\sigma}}{N^{\frac{1}{4}-\alpha}} = 2N^{\frac{1}{4}+\alpha+\sigma} \ . \tag{16.4}$$

This bound holds when the condition $N^{\alpha+\sigma} \leq \frac{1}{2}N^{\frac{1}{4}}$ holds, which is not always the case in our settings. For instance, a RSA modulus of 1024 bit with $\log_2 e = 256$ and $log_2 \ell = 128$ will have $N^{\alpha+\sigma} \leq 2N^{\frac{3}{8}}$. For these cases we are unaware of successful applications of Coppersmith's method.

In [LZL14] Section 4 it is presented a novel technique for the CRT case with better bound but with the requirement to have d_p not full size. This requirement also implies that no countermeasure is applied.

16.3 Experimental Results

In our experiments we target RSA applications with 2048 or 3072-bit modulus N and public exponent $e = 2^{16} + 1$, since this is the most common choice made for real implementations. In addition we assume that a random multiple ℓ of $\phi(N)$ (or of $(p-1)$ for CRT-RSA applications) is added to the private exponent d (respectively d_p).

For each dimension of ℓ, we first report the theoretical bound on the minimum number of bits of the secret key that the attacker needs to know to recover it entirely. These values are derived from theorems we have proved in previous sections. Then, we report the average minimum number of bits that we really needed in our tests. In fact, theoretical bounds are reached when the lattice dimension goes to infinity. In general, the smaller is the number of known bits, the bigger the lattice shall be. To concretely mount an attack, one needs to construct a lattice whose dimension is such that the LLL-algorithm runs in practical time.

Recall that the running time of LLL-algorithm depends on the lattice dimension and on the dimension of the entries of its matrix-basis. Since the dimension of the entries depends on the bounds X_i and on the modular polynomial used, the LLL-algorithm may have different running times for the same lattice dimension. We decided to fix an upper bound on the dimension of the lattices we constructed. We chose the threshold 80 as a tradeoff between efficiency and effectiveness of our attacks. Indeed, this choice allows us to get closer to the theoretical bounds as opposed to smaller dimensions. On the other hand, 80 is small enough to make the LLL-algorithm running in practical time. We fixed the same threshold for all attacks in order to compare their effectiveness when using the same lattice dimension.

We implemented our methods with the SAGE computer-algebra system [S$^+$14] and run it on a 3GHz Intel Core i5. Except for the CRT-MSB case, where we used only 20 experiments, for all other attacks we ran 100 experiments generating different key pairs and different values of ℓ. We report the average values obtained from these experiments.

16.3.1 Results with known LSBs of d^*

Experimental results are presented in Table 16.1. For generating lattices, we used $m = 11$ and $t = \tau m$, where τ is defined in the proof of Theorem 16.1. Notice that τ is always very small resulting in $t = 0$ for each experiment. Thus, the dimension of the lattice is always equal to 78.

$\log_2 \ell$	$\log_2 N = 2048$				$\log_2 N = 3072$			
	theo. bound	exp. bound	dim(L)	LLL	theo. bound	exp. bound	dim(L)	LLL
0	1040	1043	78	18 s	1552	1556	78	23 s
10	1060	1063	78	19 s	1572	1577	78	30 s
32	1103	1106	78	22 s	1615	1620	78	40 s
64	1164	1171	78	50 s	1678	1684	78	58 s
100	1232	1243	78	70 s	1747	1756	78	80 s

Table 16.1: *Experimental results for partial key exposure attack given least significant bits of the secret exponent $d^* = d + \ell\phi(N)$. The public exponent is $e = 2^{16} + 1$.*

The difference between theoretical and experimental bounds is of very few bits and the LLL-algorithm's running time is small.

It is worth to say that for $\ell = 0$ and small e, the attack in [BDF98] is more effective than our attack. Indeed the $n/4$ least significant bits of d are sufficient to factor N. However, their attack requires a brute-force search on k, that in our case is allowed only when $e + e\ell$ is small. Thus, with the introduction of exponent blinding, or for larger dimension of e, their method can't be applied, because the brute force-search becomes impractical.

In Table 16.2 we report experimental results to compare our approach and the approach of [JL12] for the same scenario. Specifically, we consider 1000-bit modulus N, public exponent $e = 2^{16} + 1$ and $\ell \in \{10, 100, 200, 300\}$ as used in [JL12]. In our analysis we use a bivariate polynomial instead of a trivariate polynomial, thus we perform a single resultant computation, instead of three. The theoretical bound we obtain is the same of [JL12], but our approach allows us to get closer to it as shown in Table 16.2. Moreover, we do it by using smaller lattices.

16.3.2 Results with known MSBs of d^*

In Table 16.3 we present our results. Since this method uses a univariate polynomial, it is possible, in theory, to match the theoretical limit, although the lattice dimension would make LLL highly impractical. By imposing the threshold for the maximum dimension of the lattice equal to 80, the LLL-algorithm's running time is about 2 hours. For constructing such a lattice, we used $m = 40$ and $t = 40$.

The experiments confirmed the independence of the bound with respect to the dimension of the random integer ℓ.

Unfortunately, in this case we cannot compare our approach with the approach of [JL12], because they didn't provide any experimental result respecting our assumptions. In fact, they use very large values of ℓ, namely 500, 600 or 700-bit long for a modulus N of size 1000 bits. These settings do not satisfy our requirement of Lemma 16.1 for $N^{\alpha+\sigma} \leq 2N^{\frac{3}{8}}$. In any case, our approach improves their bound, as said in section 16.1.2.

Results using both MSBs and LSBs.

As said in section 16.1.2, it is possible to mount an attack knowing both MSBs and LSBs of d^*. A univariate polynomial is constructed and its root is found by constructing a lattice as in the proof of Theorem 15.3. In Table 16.4 we provide some experimental results for this method.

$\log_2 \ell$	Approach of [JL12]				Our approach			
	theo. bound	exp. bound	dim(L)	LLL	theo. bound	exp. bound	dim(L)	LLL
10	535	580	16	1 s	535	540	10	1 s
100	700	760	16	1 s	700	720	10	1 s
200	871	960	16	1 s	871	920	10	1 s
300	1033	1160	16	1 s	1033	1120	10	1 s

Table 16.2: *Comparison between the approach of [JL12] and our approach for partial key exposure attack given least significant bits of the secret exponent* $d^* = d + \ell\phi(N)$. *The modulus* N *is 1000-bit long and* $e = 2^{16} + 1$.

$\log_2 \ell$	$\log_2 N = 2048$				$\log_2 N = 3072$			
	theo. bound	exp. bound	dim(L)	LLL	theo. bound	exp. bound	dim(L)	LLL
0	1555	1555	80	112 m	2323	2331	80	203 m
10	1538	1555	80	112 m	2306	2331	80	203 m
32	1538	1555	80	112 m	2306	2331	80	203 m
64	1538	1555	80	112 m	2306	2331	80	203 m
100	1538	1555	80	112 m	2306	2331	80	203 m

Table 16.3: *Experimental results for partial key exposure attack given most significant bits of the secret exponent* $d^* = d + \ell\phi(N)$. *The public exponent is* $e = 2^{16} + 1$.

$\log_2 \ell$	$\log_2 N = 2048$				$\log_2 N = 3072$			
	theo. bound	exp. bound	dim(L)	LLL	theo. bound	exp. bound	dim(L)	LLL
0	17+514	17+526	80	2h27m	17+770	17+789	80	4h50m
10	27+514	27+526	80	2h27m	27+770	27+789	80	4h50m
32	49+514	49+526	80	2h27m	49+770	49+789	80	4h50m
64	81+514	81+526	80	2h27m	81+770	81+789	80	4h50m
100	117+514	117+526	80	2h27m	117+770	117+789	80	4h50m

Table 16.4: *Experimental results for partial key exposure attack given most and least significant bits of the secret* $d^* = d + \ell\phi(N)$. *The public exponent is* $e = 2^{16} + 1$.

16.3.3 Results with known LSBs of $\mathbf{d_p^*}$

In Table 16.5 we present our results, obtained by generating lattices using $m = 3$ an $t = 11$. To get closer to the theoretical bound, the lattice dimension should be significantly increased. But this makes the LLL-algorithm's running time highly impractical. By setting the threshold 80 for the lattice dimension, the LLL-algorithm's running time is about 13 minutes.

In this case, the difference between theoretical and experimental bounds is about 80 bits for 2014-bit N and about 100 for 3072-bit N. Given a smaller number of leaked bits one can still mount the attack by constructing bigger lattices, but the computation will need more time to end. For example, by setting $t = 5$ and $m = 18$ it is sufficient to obtain 50 (or 70) bits more than the theoretical bound to solve. But the corresponding lattice dimension is 190, which makes the LLL-algorithm end in about one day. By setting $t = 7$ and $m = 24$ it is sufficient to obtain 40 (or 60) bits more than the theoretical bound to solve. But the lattice dimension is around 500 and we think that the LLL-algorithm would be highly impractical in this case.

Notice that for $\ell = 0$ and small e, Blömer and May show that a quarter of d_p is sufficient to the attacker to factor N [BM03]. To prove their result, they use a brute-force search on k_p, that is allowed only when $e + e\ell$ is small. Thus, for $e = 2^{16} + 1$ and $\ell = 0$ their method is better

$\log_2 \ell$	$\log_2 N = 2048$				$\log_2 N = 3072$			
	theo. bound	exp. bound	dim(L)	LLL	theo. bound	exp. bound	dim(L)	LLL
0	617	691	80	8 m	917	1019	80	14 m
10	637	712	80	10 m	937	1041	80	16 m
32	681	758	80	13 m	981	1087	80	21 m
64	745	822	80	17 m	1045	1154	80	28 m
100	817	894	80	18 m	1117	1227	80	34 m

Table 16.5: *Experimental results for partial key exposure attack against CRT-RSA, given least significant bits of the secret exponent* $d_p^* = d_p + \ell(p-1)$. *The public exponent is* $e = 2^{16} + 1$.

$\log_2 \ell$	$\log_2 N = 2048$				$\log_2 N = 3072$			
	theo. bound	exp. bound	dim(L)	LLL	theo. bound	exp. bound	dim(L)	LLL
0	528	540	80	3h03m	783	803	80	08h54m
10	537	550	80	3h59m	793	813	80	12h17m
32	560	573	80	4h23m	815	835	80	13h46m
64	591	604	80	4h52m	848	868	80	15h59m
100	628	640	80	6h13m	884	903	80	22h25m

Table 16.6: *Experimental results for partial key exposure attack given most significant bits of the CRT secret exponent* $d_p^* = d + \ell(p-1)$. *The public exponent is* $e = 2^{16} + 1$.

than our method, since a smaller number of leaked bits are sufficient to factor N. But, for larger dimension of e and when $\ell > 0$ their method is no more effective because the brute force-search becomes unfeasible.

16.3.4 Results with known MSBs of d_p^*

Also in this case we imposed the threshold 80 for the lattice dimension, which allowed us to run the LLL-algorithm in practical time. We constructed lattices by using $m = 40$ and $t = 40$.

In Table 16.6 we report the theoretical and experimental number of leaked bits, the lattice dimension and the running time of LLL-algorithm.

As opposite to the case based on LSBs, this method is the most effective also for $\ell = 0$. Indeed, our method is a generalization of [BM03], thus for $\ell = 0$ we obtain their original result which is the most effective method in literature for this scenario.

17 Conclusions

We presented some methods to mount partial key exposure attacks on RSA with exponent blinding. We investigated both RSA and CRT-RSA, focusing on practical settings for the exponents and the blinding factor ℓ. In particular, we focused on public exponent e such that $3 \leq e < 2^{256}$, combining the upper bound provided by NIST with the frequent value of 3. Additionally, we focused on full size private exponents and $\ell < 2^{128}$, as commonly used in real implementations.

We derived sufficient conditions to successfully mount partial key exposure attacks in different scenarios and validated them providing numerical experiments, using N of size 2048 or 3072 and $e = 2^{16} + 1$, which is the most commonly used setting in real implementations.

As for RSA, we improved the results of [JL12] with the aim of reducing the number of bits to be recovered by the adversary through side-channel. In particular, when least significant bits are exposed, our approach allows to get closer to the theoretical bound by using smaller lattices, as shown in Table 16.2. Whereas, when most significant bits are exposed, we presented a method that does not rely on the heuristic assumption and that provides better bounds, as shown in Section 16.1.2.

Additionally, we provided novel results for the particular case where the adversary is able to recover non-consecutive portions of the private information.

As for CRT-RSA with exponent blinding, we provided novel results for both scenarios when either least or most significant bits are exposed.

In Table 17.1, we recap the numerical results we obtained from our experiments. For each dimension of ℓ we provide the minimum number of bits of the protected exponent that is sufficient to the attacker to successfully break the system.

With the only exception of the RSA attack based on most significant bits (columns named MSB), the number of known bits depends on the bit-size of the blinding factor ℓ.

When attacking RSA using the least significant bits, the ratio of exponent bits required is between 50.6% and 57.8% and increases with the blinding factor size. When the attack is based on the most significant bits, the ratio decreases with the blinding factor from 75.9% to 73.5%. When both most and least significant bits are used, the ration ranges between 26.2% and 29.9%.

When attacking CRT-RSA with the least significant bits, the ratio of exponent bits goes from 33.1% to 41.6%. While, if the most significant bits are used, it ranges between 26.1% and 29.8%.

$\log_2 \ell$	$\log_2 N = 2048$					$\log_2 N = 3072$				
	LSB	MSB	MSB+LSB	CRT LSB	CRT MSB	LSB	MSB	MSB+LSB	CRT LSB	CRT MSB
0	1043	1555	17+526	691	540	1556	2331	17+789	1019	803
10	1063	1555	27+526	712	550	1577	2331	27+789	1041	813
32	1106	1555	49+526	758	573	1620	2331	49+789	1087	835
64	1171	1555	81+526	822	604	1684	2331	81+789	1154	868
100	1243	1555	117+526	894	640	1756	2331	117+789	1227	903

Table 17.1: *The number of bits that the attacker needs to know to successfully mount partial key exposure attacks. The public exponent is* $e = 2^{16} + 1$. *The private exponent is blinded using the random factor* ℓ.

18

Closing remarks and future directions

In this thesis, we have presented several contributions to cryptanalysis of primitives and their implementations.

In Part I, we have presented new techniques to efficiently scan the space of low-weight differential trail cores in bit-oriented ciphers. We have proposed to represent difference patterns as ordered lists of units in order to arrange them in a tree. The task of generating trails with weight below some target weight thus starts with the traversal of the tree. In fact, a difference pattern fully specifies a 2-round trail core, from which longer trail cores can be built. Conveniently defining units may allow to tightly bound the weight of all descendants for any node. This in turn allows to efficiently prune the tree as soon as a node with weight above the limit is encountered.

We have applied these techniques to efficiently generate 2-round trail cores in the four largest Keccak-f permutations. We have then used the generated cores to cover the space of all 3-round trail cores below weight 45. To this end, we have presented efficient techniques for trail extension which exploit the properties of the mixing layer and the non-linear layer. Covering such a space has allowed us to cover the space of all 6-round trail cores below weight 91. It resulted that such space is empty. Therefore, we can state that a 6-round trail in Keccak-$f[b]$, with $b \in \{200, 400, 800, 1600\}$, has weight at least 92.

Hence, with these new techniques, we are able to scan the space of trails with weight per round of 15. This space is orders of magnitude larger than previously best result published on Keccak-$f[1600]$ that reached 12, which in turn is orders of magnitude larger than any published results achieved with standard tools, that reached at most 9.

Our search has also allowed us to provide new and improved bounds for the minimum weight of differential trails on 3, 4 and 5 rounds. These results have been beneficial in the design process of Kravatte, when choosing its number of rounds.

Along with lower bounds on the weight of trails, this analysis has provided new insights in the structural properties of Keccak-f, as how trail weights scale with the width and the number of rounds.

There are different directions that we can take for future works. The first is to further improve the search for differential trails of Keccak-f. Pushing the limit weight above 45, like 48 or 51, may indeed reveal which parts of the approach are the new bottlenecks, as it happened with the approach of [DV12] when we pushed the target weight beyond 36. Additionally, it would be interesting to target 4-round trails. For such trails a minimum is not known, but only a bound, for the three largest variants of Keccak-f.

A second direction is to apply the generic approach to linear trails of Keccak-f. For Keccak-f, the propagation of differential trails and linear trails is very similar. With some adaptations, we can cover also the whole space of 3-round linear trails up to some weight. It would be interesting to see which target weight can be reached with the same computational effort required for differential trails.

Still another direction could be to apply the generic approach of the tree traversal to other cryptographic primitives. While our first aim was to improve lower bounds for the weight of trails in Keccak-f, actually some of the new techniques we have developed happen to be more generically applicable to primitives with bit-oriented mixing layers and a relatively lightweight non-linear layer. Some examples might be Ascon [DEMS16], Noekeon [DPVR00] and BaseKing [DPA00].

In Part II we have presented several results regarding differential fault analysis against AES. First, we have investigated whether it is possible to define DFA attacks that exploit faults tampering with the instruction flow. In fact, almost all DFA attacks in literature leverage on faults on data and very few on reducing the number of rounds. We believe that also the instruction flow is a possible target for DFA, since it has been proved that single instructions can be jumped when properly faulting the computation, as with power spikes and clock glitches. Hence, we provide four new attacks that exploit faults causing a misbehavior of the AES algorithm during its last round. As future work, it would be interesting to extend such attacks to previous rounds and study how the computational complexity of the attack increases.

Subsequently, we have analyzed the role of the fault model considered by the attacker. We have seen that the knowledge that the attacker has on the effect of the fault injection technique significantly impacts the effectiveness of the attack. Even with a fault model with 99% accuracy the effectiveness of the attack is heavily lowered. The (un)knowledge of the fault by the attacker can at the end remove the basis for an attack based on faults. With large (relaxed) fault models each experiment is less prone to fall out of the fault model. But this implies a larger amount of needed experiments. In the long run, even a single experiment with a wrong fault model leads to a wrong key (or no solution). Vice versa, with small fault models just few experiments are enough, but in this case there is high probability that the fault will fall out of the model.

Following this perspective, we have presented a novel approach for DFA which exploits a robust clustering algorithm tailored to fault analysis. This new method, called J-DFA, can be easily used to quickly replicate many classical DFA attacks unified in a common framework. Moreover, thanks to its robustness, J-DFA produces reliable solutions in a wider range of practical scenarios, even if the a-priori knowledge of the attacker is poor or completely absent.

As future direction, it would be interesting to apply the J-DFA to study the case where faults are injected in the 8th or 9th round. Up to now in fact, we dealt only with faults injected in the last round of AES. It might be also interesting to consider even different algorithms, taking advantage of the generality of the J-Linkage conceptual representation.

In the last part of the thesis, i.e. Part III, we have presented some methods to mount partial key exposure attacks on RSA with exponent blinding. We investigated both RSA and CRT-RSA versions when the least significant bits or the most significant bits are exposed. We have first derived sufficient conditions to successfully mount partial key exposure attacks and then we have validated them providing numerical experiments.

As for RSA, we improved previous attacks with the aim of reducing the number of bits to be recovered through side-channel in the online stage or to reduce the complexity of the mathematical analysis in the offline stage. In the case of CRT-RSA with exponent blinding, we provided novel results for both scenarios when either least or most significant bits are exposed.

The complexity of the attack depends on the size of a lattice that must be reduced through basis reduction algorithms. It would be interesting, for a future work, to further investigate the lattice construction with the goal of achieving smaller dimensions and thus improve the efficiency of the analysis.

References

[ABH$^+$02] Christian Aumüller, Peter Bier, Peter Hofreiter, Wieland Fischer, and Jean-Pierre Seifert. Fault attacks on RSA with CRT: concrete results and practical countermeasures. *IACR Cryptology ePrint Archive*, 2002:73, 2002.

[AK96] Ross Anderson and Markus Kuhn. Tamper resistance: A cautionary note. In *Proceedings of the 2Nd Conference on Proceedings of the Second USENIX Workshop on Electronic Commerce - Volume 2*, WOEC'96, pages 1–1, Berkeley, CA, USA, 1996. USENIX Association.

[AN00] Lorena Anghel and Michael Nicolaidis. Cost reduction and evaluation of a temporary faults detecting technique. In Ivo Bolsens, editor, *2000 Design, Automation and Test in Europe (DATE 2000), 27-30 March 2000, Paris, France*, pages 591–598. IEEE Computer Society / ACM, 2000.

[BDF98] Dan Boneh, Glenn Durfee, and Yair Frankel. An attack on RSA given a small fraction of the private key bits. In Kazuo Ohta and Dingyi Pei, editors, *Advances in Cryptology - ASIACRYPT 1998, Proceedings*, volume 1514 of *LNCS*, pages 25–34. Springer, 1998.

[BDH$^+$97] Feng Bao, Robert H. Deng, Yongfei Han, Albert B. Jeng, A. Desai Narasimhalu, and Teow-Hin Ngair. Breaking public key cryptosystems on tamper resistant devices in the presence of transient faults. In Bruce Christianson, Bruno Crispo, T. Mark A. Lomas, and Michael Roe, editors, *Security Protocols, 5th International Workshop, Paris, France, April 7-9, 1997, Proceedings*, volume 1361 of *Lecture Notes in Computer Science*, pages 115–124. Springer, 1997.

[BDK$^+$] Julien Brouchier, Nora Dabbous, Tom Kean, Carol Marsh, and David Naccache. Thermocommunication. *IACR Cryptology ePrint Archive, Report 2009/002.*

[BDL97] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the importance of checking cryptographic protocols for faults (extended abstract). In Walter Fumy, editor, *EUROCRYPT*, volume 1233 of *Lecture Notes in Computer Science*, pages 37–51. Springer, 1997.

[BDP$^+$14] G. Bertoni, J. Daemen, M. Peeters, G. Van Assche, and R. Van Keer. CAESAR submission: Ketje v1, March 2014. `http://ketje.noekeon.org/`.

[BDP$^+$15] G. Bertoni, J. Daemen, M. Peeters, G. Van Assche, and R. Van Keer. CAESAR submission: Keyak v2, August 2015. `http://keyak.noekeon.org/`.

[BDP$^+$16] Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. Farfalle: parallel permutation-based cryptography. *IACR Cryptology ePrint Archive*, 2016:1188, 2016.

[BDPV11a] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. On alignment in Keccak. In *ECRYPT II Hash Workshop 2011*, 2011.

[BDPV11b] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. The Keccak reference, January 2011. `http://keccak.noekeon.org/`.

[BDPV15] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. KeccakTools software, October 2015. `http://keccak.noekeon.org/`.

[BECN$^+$04] Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. The sorcerer's apprentice guide to fault attacks. *IACR Cryptology ePrint Archive*, 2004:100, 2004.

[BFL10] C. Bouillaguet, P.-A. Fouque, and G. Leurent. Security analysis of SIMD. In A. Biryukov, G. Gong, and D. R. Stinson, editors, *Selected Areas in Cryptography - 17th International Workshop, SAC 2010, Waterloo, Ontario, Canada, August 12-13, 2010, Revised Selected Papers*, volume 6544 of *Lecture Notes in Computer Science*, pages 351–368. Springer, 2010.

[BGK$^+$08] Luca Breveglieri, Shay Gueron, Israel Koren, David Naccache, and Jean-Pierre Seifert, editors. *Fifth International Workshop on Fault Diagnosis and Tolerance in Cryptography, 2008, FDTC 2008, Washington, DC, USA, 10 August 2008.* IEEE Computer Society, 2008.

[BGK$^+$11] Luca Breveglieri, Sylvain Guilley, Israel Koren, David Naccache, and Junko Takahashi, editors. *2011 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2011, Tokyo, Japan, September 29, 2011.* IEEE, 2011.

[BGN05] Eli Biham, Louis Granboulan, and Phong Q. Nguyen. Impossible fault analysis of RC4 and differential fault analysis of RC4. In Henri Gilbert and Helena Handschuh, editors, *Fast Software Encryption: 12th International Workshop, FSE 2005, Paris, France, February 21-23, 2005, Revised Selected Papers*, volume 3557 of *Lecture Notes in Computer Science*, pages 359–367. Springer, 2005.

[BGV11] Josep Balasch, Benedikt Gierlichs, and Ingrid Verbauwhede. An in-depth and black-box characterization of the effects of clock glitches on 8-bit mcus. In Breveglieri et al. [BGK$^+$11], pages 105–114.

[BK94] Eli Biham and Paul C. Kocher. A known plaintext attack on the PKZIP stream cipher. In Preneel [Pre95], pages 144–153.

[Ble98] Daniel Bleichenbacher. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1. In Krawczyk [Kra98], pages 1–12.

[BM03] Johannes Blömer and Alexander May. New partial key exposure attacks on RSA. In Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003, Proceedings*, volume 2729 of *LNCS*, pages 27–43. Springer, 2003.

[BMM00] Ingrid Biehl, Bernd Meyer, and Volker Müller. Differential fault attacks on elliptic curve cryptosystems. In Mihir Bellare, editor, *Advances in Cryptology - CRYPTO 2000, 20th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 2000, Proceedings*, volume 1880 of *Lecture Notes in Computer Science*, pages 131–146. Springer, 2000.

[BNFR12] Kaouthar Bousselam, Giorgio Di Natale, Marie-Lise Flottes, and Bruno Rouzeyre. On countermeasures against fault attacks on the advanced encryption standard. In Joye and Tunstall [JT12], pages 89–108.

[BS90] Eli Biham and Adi Shamir. Differential cryptanalysis of des-like cryptosystems. In Alfred Menezes and Scott A. Vanstone, editors, *Advances in Cryptology - CRYPTO '90, 10th Annual International Cryptology Conference, Santa Barbara, California, USA, August 11-15, 1990, Proceedings*, volume 537 of *Lecture Notes in Computer Science*, pages 2–21. Springer, 1990.

[BS91] Eli Biham and Adi Shamir. Differential cryptanalysis of des-like cryptosystems. *J. Cryptology*, 4(1):3–72, 1991.

[BS92] Eli Biham and Adi Shamir. Differential cryptanalysis of the full 16-round DES. In Ernest F. Brickell, editor, *Advances in Cryptology - CRYPTO '92, 12th Annual International Cryptology Conference, Santa Barbara, California, USA, August 16-20, 1992, Proceedings*, volume 740 of *Lecture Notes in Computer Science*, pages 487–496. Springer, 1992.

[BS97] Eli Biham and Adi Shamir. Differential fault analysis of secret key cryptosystems. In Burton S. Kaliski Jr., editor, *CRYPTO*, volume 1294 of *Lecture Notes in Computer Science*, pages 513–525. Springer, 1997.

[BS03] Johannes Blömer and Jean-Pierre Seifert. Fault based cryptanalysis of the advanced encryption standard (aes). In Rebecca N. Wright, editor, *Financial Cryptography*, volume 2742 of *Lecture Notes in Computer Science*, pages 162–181. Springer, 2003.

[Can12] Anne Canteaut, editor. *Fast Software Encryption - 19th International Workshop, FSE 2012, Washington, DC, USA, March 19-21, 2012. Revised Selected Papers*, volume 7549 of *Lecture Notes in Computer Science*. Springer, 2012.

[CJ98] Florent Chabaud and Antoine Joux. Differential collisions in SHA-0. In Krawczyk [Kra98], pages 56–71.

[CJ05] Mathieu Ciet and Marc Joye. Elliptic curve cryptosystems in the presence of permanent and transient faults. *Des. Codes Cryptography*, 36(1):33–43, 2005.

[CLO07] David A. Cox, John Little, and Donal O'Shea. *Ideals, Varieties, and Algorithms: An Introduction to Computational Algebraic Geometry and Commutative Algebra, 3/e (Undergraduate Texts in Mathematics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[Col02] Jim Colvin. Functional failure analysis by induced stimulus. In *ISTFA 2002 Proceedings*, pages 623–630. Ed. ASM International, 2002.

[Cop96a] Don Coppersmith. Finding a small root of a bivariate integer equation; factoring with high bits known. In Maurer [Mau96], pages 178–189.

[Cop96b] Don Coppersmith. Finding a small root of a univariate modular equation. In Maurer [Mau96], pages 155–165.

[Cor99] Jean-Sébastien Coron. Resistance against differential power analysis for elliptic curve cryptosystems. In Çetin Kaya Koç and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 1999, Proceedings*, volume 1717 of *LNCS*, pages 292–302. Springer, 1999.

[CR06] Christophe De Cannière and Christian Rechberger. Finding SHA-1 characteristics: General results and applications. In Xuejia Lai and Kefei Chen, editors, *Advances in Cryptology - ASIACRYPT 2006, 12th International Conference on the Theory and Application of Cryptology and Information Security, Shanghai, China, December 3-7, 2006, Proceedings*, volume 4284 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2006.

[Cra05] Ronald Cramer, editor. *Advances in Cryptology - EUROCRYPT 2005, 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Aarhus, Denmark, May 22-26, 2005, Proceedings*, volume 3494 of *Lecture Notes in Computer Science*. Springer, 2005.

[CT05] Hamid Choukri and Michael Tunstall. Round reduction using faults. http://www.geocities.ws/mike.tunstall/papers/CT05.pdf?, 2005.

[CWF07] Hua Chen, Wenling Wu, and Dengguo Feng. Differential fault analysis on clefia. In Sihan Qing, Hideki Imai, and Guilin Wang, editors, *ICICS*, volume 4861 of *Lecture Notes in Computer Science*, pages 284–295. Springer, 2007.

[CY03] Chien-Ning Chen and Sung-Ming Yen. *Differential Fault Analysis on AES Key Schedule and Some Countermeasures*, pages 118–129. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[Dae95] J. Daemen. *Cipher and hash function design strategies based on linear and differential cryptanalysis, PhD Thesis.* K.U.Leuven, 1995.

[dBB93] Bert den Boer and Antoon Bosselaers. Collisions for the compressin function of MD5. In Tor Helleseth, editor, *Advances in Cryptology - EUROCRYPT '93, Workshop on the Theory and Application of of Cryptographic Techniques, Lofthus, Norway, May 23-27, 1993, Proceedings*, volume 765 of *Lecture Notes in Computer Science*, pages 293–304. Springer, 1993.

[DEMS16] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. Ascon v1.2. Submission to the CAESAR competition: `http://competitions.cr.yp.to/round3/asconv12.pdf`, 2016.

[Dob98] Hans Dobbertin. Cryptanalysis of MD4. *J. Cryptology*, 11(4):253–271, 1998.

[DPA00] Joan Daemen, Michaël Peeters, and Gilles Van Assche. Bitslice ciphers and power analysis attacks. In Bruce Schneier, editor, *Fast Software Encryption, 7th International Workshop, FSE 2000, New York, NY, USA, April 10-12, 2000, Proceedings*, volume 1978 of *Lecture Notes in Computer Science*, pages 134–149. Springer, 2000.

[DPVR00] J. Daemen, M. Peeters, G. Van Assche, and V. Rijmen. Nessie proposal: the block cipher NOEKEON. Nessie submission, 2000. `http://gro.noekeon.org/`.

[DR02] J. Daemen and V. Rijmen. *The design of Rijndael — AES, The Advanced Encryption Standard*. Springer-Verlag, 2002.

[DV12] J. Daemen and G. Van Assche. Differential propagation analysis of Keccak. In Canteaut [Can12], pages 422–441.

[EJMdW05] Matthias Ernst, Ellen Jochemsz, Alexander May, and Benne de Weger. Partial key exposure attacks on RSA up to full size exponents. In Cramer [Cra05], pages 371–386.

[FH08] Julie Ferrigno and Martin HlavÃ¡Ëc. When aes blinks: introducing optical side channel, 2008.

[FKM$^+$06] Pierre-Alain Fouque, Sébastien Kunz-Jacques, Gwenaëlle Martinet, Frédéric Muller, and Frédéric Valette. Power attack on small RSA public exponent. In Goubin and Matsui [GM06], pages 339–353.

[FL10] N. Floissac and Y. L'Hyver. From aes-128 to aes-192 and aes-256, how to adapt differential fault analysis attacks. *IACR Cryptology ePrint Archive*, 2010:14, 2010.

[FLN07] Pierre-Alain Fouque, Gaëtan Leurent, and Phong Q. Nguyen. Automatic search of differential path in MD4. *IACR Cryptology ePrint Archive*, 2007:206, 2007.

[GA03] Sudhakar Govindavajhala and Andrew W. Appel. Using memory errors to attack a virtual machine. In *2003 IEEE Symposium on Security and Privacy (S&P 2003), 11-14 May 2003, Berkeley, CA, USA*, pages 154–165. IEEE Computer Society, 2003.

[Gir03] Christophe Giraud. Dfa on aes. *IACR Cryptology ePrint Archive*, 2003:8, 2003.

[GM06] Louis Goubin and Mitsuru Matsui, editors. *Cryptographic Hardware and Embedded Systems - CHES 2006, 8th International Workshop, Yokohama, Japan, October 10-13, 2006, Proceedings*, volume 4249 of *Lecture Notes in Computer Science*. Springer, 2006.

[GT] Christophe Giraud and Adrian Thillard. Piret and quisquater's dfa on aes revisited. *Cryptology ePrint Archive, Report 2010/440 (2010).*

[Hem04] Ludger Hemme. A differential fault attack against early rounds of (triple-)des. In Marc Joye and Jean-Jacques Quisquater, editors, *CHES*, volume 3156 of *Lecture Notes in Computer Science*, pages 254–267. Springer, 2004.

[HGL09] Yupu Hu, Juntao Gao, and Qing Liu. Floating fault analysis of trivium under weaker assumptions. *IACR Cryptology ePrint Archive*, 2009:169, 2009.

[HM08] Mathias Herrmann and Alexander May. Solving linear equations modulo divisors: On factoring given any bits. In Josef Pieprzyk, editor, *Advances in Cryptology - ASIACRYPT 2008, Proceedings*, volume 5350 of *LNCS*, pages 406–424. Springer, 2008.

[How97] Nick Howgrave-Graham. Finding small roots of univariate modular equations revisited. In Michael Darnell, editor, *Cryptography and Coding, 6th IMA International Conference 1997, Proceedings*, volume 1355 of *LNCS*, pages 131–142. Springer, 1997.

[HR08] Michal Hojsík and Bohuslav Rudolf. Floating fault analysis of trivium. In Dipanwita Roy Chowdhury, Vincent Rijmen, and Abhijit Das, editors, *Progress in Cryptology - INDOCRYPT 2008, 9th International Conference on Cryptology in India, Kharagpur, India, December 14-17, 2008. Proceedings*, volume 5365 of *Lecture Notes in Computer Science*, pages 239–250. Springer, 2008.

[Jan96] G. J. Janusz, editor. *Algebraic Number Fields.* American Mathematical Society, 1996.

[JL12] Marc Joye and Tancrède Lepoint. Partial key exposure on RSA with private exponents larger than N. In Mark Dermot Ryan, Ben Smyth, and Guilin Wang, editors, *Information Security Practice and Experience - ISPEC 2012, Proceedings*, volume 7232 of *LNCS*, pages 369–380. Springer, 2012.

[JT12] Marc Joye and Michael Tunstall, editors. *Fault Analysis in Cryptography.* Information Security and Cryptography. Springer, 2012.

[Kim10] Chong Hee Kim. Differential fault analysis against AES-192 and AES-256 with minimal faults. In Luca Breveglieri, Marc Joye, Israel Koren, David Naccache, and Ingrid Verbauwhede, editors, *2010 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2010, Santa Barbara, California, USA, 21 August 2010*, pages 3–9. IEEE Computer Society, 2010.

[Kim12] Chong Hee Kim. Improved differential fault analysis on AES key schedule. *IEEE Trans. Information Forensics and Security*, 7(1):41–50, 2012.

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael J. Wiener, editor, *Advances in Cryptology - CRYPTO 1999, Proceedings*, volume 1666 of *LNCS*, pages 388–397. Springer, 1999.

[KK99] Oliver Kömmerling and Markus G. Kuhn. Design principles for tamper-resistant smartcard processors. https://www.cl.cam.ac.uk/ mgk25/sc99-tamper.pdf, 1999.

[Knu94] Lars R. Knudsen. Truncated and higher order differentials. In Preneel [Pre95], pages 196–211.

[Koc96] Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In Neal Koblitz, editor, *Advances in Cryptology - CRYPTO 1996, Proceedings*, volume 1109 of *LNCS*, pages 104–113. Springer, 1996.

[KQ07] Chong Hee Kim and Jean-Jacques Quisquater. Fault attacks for CRT based RSA: new attacks, new results, and new countermeasures. In *Information Security Theory and Practices. Smart Cards, Mobile and Ubiquitous Computing Systems, First IFIP TC6 / WG 8.8 / WG 11.2 International Workshop, WISTP 2007, Heraklion, Crete, Greece, May 9-11, 2007, Proceedings*, pages 215–228, 2007.

[KQ08] Chong Hee Kim and Jean-Jacques Quisquater. New differential fault analysis on AES key schedule: Two faults are enough. In Gilles Grimaud and François-Xavier Standaert, editors, *Smart Card Research and Advanced Applications, 8th IFIP WG 8.8/11.2 International Conference, CARDIS 2008, London, UK, September 8-11, 2008. Proceedings*, volume 5189 of *Lecture Notes in Computer Science*, pages 48–60. Springer, 2008.

[KR00] Lars R. Knudsen and Vincent Rijmen. Ciphertext-only attack on akelarre. *Cryptologia*, 24(2):135–147, 2000.

[Kra98] Hugo Krawczyk, editor. *Advances in Cryptology - CRYPTO '98, 18th Annual International Cryptology Conference, Santa Barbara, California, USA, August 23-27, 1998, Proceedings*, volume 1462 of *Lecture Notes in Computer Science*. Springer, 1998.

[KSD13] Cameron F. Kerry, Acting Secretary, and Charles R. Director. FIPS PUB 186-4 FEDERAL INFORMATION PROCESSING STANDARDS PUBLICATION Digital Signature Standard (DSS). 2013.

[KSV13] Dusko Karaklajic, JÃ¶rn-Marc Schmidt, and Ingrid Verbauwhede. Hardware designer's guide to fault attacks. *IEEE Trans. VLSI Syst.*, pages 2295–2306, 2013.

[LCC$^+$00] F. Lima, E. Costa, L. Carro, M. Lubaszewski, R. Reis, S. Rezgui, and R. Velazco. Designing and testing a radiation hardened 8051-like micro-controller. In *3rd Military and Aerospace Applications of Programmable Devices and Technologies International Conference, 2000*, 2000.

[LG08] Wei Li and Dawu Gu. Differential fault analysis on the sms4 cipher by inducing faults to the key schedule. *Journal on Communications*, 29(10):135â142, 2008.

[LGL08] Wei Li, Dawu Gu, and Juanru Li. Differential fault analysis on the ARIA algorithm. *Inf. Sci.*, 178(19):3727–3737, 2008.

[LGSO10] Yang Li, Shigeto Gomisawa, Kazuo Sakiyama, and Kazuo Ohta. An information theoretic perspective on the differential fault analysis against aes. *IACR Cryptology ePrint Archive*, 2010:32, 2010.

[LGW09] Wei Li, Dawu Gu, and Yi Wang. Differential fault analysis on the contracting UFN structure, with application to SMS4 and macguffin. *Journal of Systems and Software*, 82(2):346–354, 2009.

[LLL82] A.K. Lenstra, H.W.jun. Lenstra, and Lászlo Lovász. Factoring polynomials with rational coefficients. *Math. Ann.*, 261:515–534, 1982.

[LSLY11] Ruilin Li, Bing Sun, Chao Li, and Jianxiong You. Differential fault analysis on SMS4 using a single fault. *Inf. Process. Lett.*, 111(4):156–163, 2011.

[LZL14] Yao Lu, Rui Zhang, and Dongdai Lin. New partial key exposure attacks on CRT-RSA with large public exponents. In Ioana Boureanu, Philippe Owesarski, and Serge Vaudenay, editors, *Applied Cryptography and Network Security - ACNS 2014, Proceedings*, volume 8479 of *LNCS*, pages 151–162. Springer, 2014.

[Mat94] M. Matsui. On correlation between the order of S-boxes and the strength of DES. In A. De Santis, editor, *Advances in Cryptology - EUROCRYPT '94, Workshop on the Theory and Application of Cryptographic Techniques, Perugia, Italy, May 9-12, 1994, Proceedings*, volume 950 of *Lecture Notes in Computer Science*, pages 366–375. Springer, 1994.

[Mau96] Ueli M. Maurer, editor. *Advances in Cryptology - EUROCRYPT 1996, Proceeding*, volume 1070 of *LNCS*. Springer, 1996.

[May03] Alexander May. *New RSA vulnerabilities using lattice reduction methods.* PhD thesis, University of Paderborn, 2003.

[MNS11] Florian Mendel, Tomislav Nad, and Martin Schläffer. Finding SHA-2 characteristics: Searching through a minefield of contradictions. In Dong Hoon Lee and Xiaoyun Wang, editors, *Advances in Cryptology - ASIACRYPT 2011 - 17th International Conference on the Theory and Application of Cryptology and Information Security, Seoul, South Korea, December 4-8, 2011. Proceedings*, volume 7073 of *Lecture Notes in Computer Science*, pages 288–307. Springer, 2011.

[MNS12] Florian Mendel, Tomislav Nad, and Martin Schläffer. Collision attacks on the reduced dual-stream hash function RIPEMD-128. In Canteaut [Can12], pages 226–243.

[MP13] N. Mouha and B. Preneel. Towards finding optimal differential characteristics for ARX: Application to Salsa20. *IACR Cryptology ePrint Archive*, 2013:328, 2013.

[MS01] Itsik Mantin and Adi Shamir. A practical attack on broadcast RC4. In Mitsuru Matsui, editor, *Fast Software Encryption, 8th International Workshop, FSE 2001 Yokohama, Japan, April 2-4, 2001, Revised Papers*, volume 2355 of *Lecture Notes in Computer Science*, pages 152–164. Springer, 2001.

[MSS06] Amir Moradi, Mohammad T. Manzuri Shalmani, and Mahmoud Salmasizadeh. A generalized method of differential fault attack against aes cryptosystem. In Goubin and Matsui [GM06], pages 91–100.

[MSY06] Tal Malkin, François-Xavier Standaert, and Moti Yung. A comparative cost/security analysis of fault attack countermeasures. In Luca Breveglieri, Israel Koren, David Naccache, and Jean-Pierre Seifert, editors, *FDTC*, volume 4236 of *Lecture Notes in Computer Science*, pages 159–172. Springer, 2006.

[Muk09] Debdeep Mukhopadhyay. An improved fault based attack of the advanced encryption standard. In Bart Preneel, editor, *AFRICACRYPT*, volume 5580 of *Lecture Notes in Computer Science*, pages 421–434. Springer, 2009.

[MW79] T. C. May and M. H. Woods. Alpha-particle-induced soft errors in dynamic memories. *IEEE Transactions on Electron Devices*, 26(1):2–9, Jan 1979.

[MWGP11] N. Mouha, Q. Wang, D. Gu, and B. Preneel. Differential and linear cryptanalysis using mixed-integer linear programming. In C. Wu, M. Yung, and D. Lin, editors, *Information Security and Cryptology - 7th International Conference, Inscrypt 2011, Beijing, China, November 30 - December 3, 2011. Revised Selected Papers*, volume 7537 of *Lecture Notes in Computer Science*, pages 57–76. Springer, 2011.

[NIoSTN01] National Institute of Standard and Technology (NIST). Advanced Encryption Standard (FIPS PUB 197), November 2001. `http://www.csrc.nist.gov/publications/fips/fips197/fips-197.pdf`.

[PF82] Janak H. Patel and Leona Y. Fung. Concurrent error detection in alu's by recomputing with shifted operands. *IEEE Trans. Computers*, 31(7):589–595, 1982.

[PMC$^+$11] JeaHoon Park, SangJae Moon, DooHo Choi, YouSung Kang, and JaeCheol Ha. Differential fault analysis for round-reduced aes by fault injection. In *ETRI Journal*, volume 33, pages 434–442, 2011.

[PQ03] Gilles Piret and Jean-Jacques Quisquater. A differential fault attack technique against spn structures, with application to the aes and khazad. In Colin D. Walter, Çetin Kaya Koç, and Christof Paar, editors, *CHES*, volume 2779 of *Lecture Notes in Computer Science*, pages 77–88. Springer, 2003.

[Pre95] Bart Preneel, editor. *Fast Software Encryption: Second International Workshop. Leuven, Belgium, 14-16 December 1994, Proceedings*, volume 1008 of *Lecture Notes in Computer Science*. Springer, 1995.

[PT] D. Peacham and B. Thomas. A dfa attack against the aes key schedule siventure white paper 001, 26 october (2006).

[PWGV02] Matthias Pflanz, K. Walther, Christian Galke, and Heinrich Theodor Vierhaus. On-line error detection and correction in storage elements with cross-parity check. In *8th IEEE International On-Line Testing Workshop (IOLTW 2002), 8-10 July 2002, Isle of Bendor, France*, pages 69–73. IEEE Computer Society, 2002.

[QC82] J.-J. Quisquater and C. Couvreur. Fast decipherment algorithm for rsa public-key cryptosystem. *Electronic Letters*, 18:905–907, 1982.

[QLLL17] Yifei Qiao1, Zhaojun Lu2, Hailong Liu3, and Zhenglin Liu. Clock glitch fault injection attacks on an fpga aes implementation. *Journal of Electrotechnology, Electrical Engineering and Management*, 1:73, 2017.

[QS02] Jean-Jacques Quisquater and David Samyde. Eddy current for Magnetic Analysis with Active Sensor. In *Esmart 2002, Nice, France*, 9 2002.

[S$^+$14] W. A. Stein et al. *Sage Mathematics Software (Version 6.2)*. The Sage Development Team, 2014. `http://www.sagemath.org`.

[SA02] Sergei P. Skorobogatov and Ross J. Anderson. Optical fault induction attacks. In Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar, editors, *CHES*, volume 2523 of *Lecture Notes in Computer Science*, pages 2–12. Springer, 2002.

[SH08a] Jörn-Marc Schmidt and Christoph Herbst. A practical fault attack on square and multiply. In Breveglieri et al. [BGK$^+$08], pages 53–58.

[SH08b] Jörn-Marc Schmidt and Christoph Herbst. A practical fault attack on square and multiply. In Breveglieri et al. [BGK$^+$08], pages 53–58.

[SHW$^+$14] S. Sun, L. Hu, M. Wang, P. Wang, K. Qiao, X. Ma, D. Shi, and L. Song. Towards finding the best characteristics of some bit-oriented block ciphers and automatic enumeration of (related-key) differential and linear characteristics with predefined properties. *IACR Cryptology ePrint Archive*, 2014:747, 2014.

[SIR04] Adam Stubblefield, John Ioannidis, and Aviel D. Rubin. A key recovery attack on the 802.11b wired equivalent privacy protocol (WEP). *ACM Trans. Inf. Syst. Secur.*, 7(2):319–332, 2004.

[SLdW12] Marc Stevens, Arjen K. Lenstra, and Benne de Weger. Chosen-prefix collisions for MD5 and applications. *IJACT*, 2(4):322–359, 2012.

[SM12] Jörn-Marc Schmidt and Marcel Medwed. Countermeasures for symmetric key ciphers. In Joye and Tunstall [JT12], pages 73–87.

[SO06] Martin Schläffer and Elisabeth Oswald. Searching for differential paths in MD4. In Matthew J. B. Robshaw, editor, *Fast Software Encryption, 13th International Workshop, FSE 2006, Graz, Austria, March 15-17, 2006, Revised Selected Papers*, volume 4047 of *Lecture Notes in Computer Science*, pages 242–261. Springer, 2006.

[ST] Adi Shamir and Eran Tromer. Acoustic cryptanalysis.

[TF08a] Junko Takahashi and Toshinori Fukunaga. Improved differential fault analysis on clefia. In Breveglieri et al. [BGK+08], pages 25–34.

[TF08b] Roberto Toldo and Andrea Fusiello. Robust multiple structures estimation with j-linkage. In *Proceedings of the 10th European Conference on Computer Vision: Part I*, ECCV '08, pages 537–547, Berlin, Heidelberg, 2008. Springer-Verlag.

[TF10] Junko Takahashi and Toshinori Fukunaga. Differential fault analysis on AES with 192 and 256-bit keys. *IACR Cryptology ePrint Archive*, 2010:23, 2010.

[TFY07] Junko Takahashi, Toshinori Fukunaga, and Kimihiro Yamakoshi. DFA mechanism on the AES key schedule. In Luca Breveglieri, Shay Gueron, Israel Koren, David Naccache, and Jean-Pierre Seifert, editors, *Fourth International Workshop on Fault Diagnosis and Tolerance in Cryptography, 2007, FDTC 2007: Vienna, Austria, 10 September 2007*, pages 62–74. IEEE Computer Society, 2007.

[TM09] Michael Tunstall and Debdeep Mukhopadhyay. Differential fault analysis of the advanced encryption standard using a single fault. *IACR Cryptology ePrint Archive*, 2009:575, 2009.

[VKS11] Ingrid Verbauwhede, Dusko Karaklajic, and Jörn-Marc Schmidt. The fault attack jungle - a classification model to guide you. In Breveglieri et al. [BGK+11], pages 3–8.

[vWWM11] Jasper G. J. van Woudenberg, Marc F. Witteman, and Federico Menarini. Practical optical fault injection on secure microcontrollers. In Breveglieri et al. [BGK+11], pages 91–99.

[Wie90] Michael J. Wiener. Cryptanalysis of short rsa secret exponents. *IEEE Transactions on Information Theory*, 36:553–558, 1990.

[WLF+05] Xiaoyun Wang, Xuejia Lai, Dengguo Feng, Hui Chen, and Xiuyuan Yu. Cryptanalysis of the hash functions MD4 and RIPEMD. In Cramer [Cra05], pages 1–18.

[WY05] Xiaoyun Wang and Hongbo Yu. How to break MD5 and other hash functions. In Cramer [Cra05], pages 19–35.

[WYY05] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Finding collisions in the full SHA-1. In Victor Shoup, editor, *Advances in Cryptology - CRYPTO 2005: 25th Annual International Cryptology Conference, Santa Barbara, California, USA, August 14-18, 2005, Proceedings*, volume 3621 of *Lecture Notes in Computer Science*, pages 17–36. Springer, 2005.

[Zie79] J. F. Ziegler. Science. *IEEE Transactions on Electron Devices*, 206:776–788, 1979.

[ZM06] Yuliang Zheng and Tsutomu Matsumoto. Breaking smart card implementations of elgamal signature and its variants. 2006.

[ZW06] Lei Zhang and Wenling Wu. Differential fault analysis on sms4. *Chinese Journal of Computers*, 29(9):1596–1602, 2006.

[ZW09] Xin-jie Zhao and Tao Wang. An improved differential fault attack on camellia. *IACR Cryptology ePrint Archive*, 2009:585, 2009.

[ZW10] Xin-jie Zhao and Tao Wang. Further improved differential fault analysis on camellia by exploring fault width and depth. *IACR Cryptology ePrint Archive*, 2010:26, 2010.

[ZWXF09] Yongbin Zhou, Wengling Wu, Nannan Xu, and Dengguo Feng. Differential fault attack on camellia. *Chinese Journal of Electronics*, 18(1):13–19, 2009.