SoC physical security evaluation

Thomas Trouchkine

▶ To cite this version:

Thomas Trouchkine. SoC physical security evaluation. Micro and nanotechnologies/Microelectronics. Université Grenoble Alpes [2020-..], 2021. English. NNT : 2021GRALT018 . tel-03282313

HAL Id: tel-03282313
https://theses.hal.science/tel-03282313

Submitted on 9 Jul 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITE GRENOBLE ALPES

Spécialité : **NANO ELECTRONIQUE ET NANO TECHNOLOGIES**

Arrêté ministériel : 25 mai 2016

Présentée par

Thomas TROUCHKINE

Thèse dirigée par **Jessy CLÉDIÈRE** Université Grenoble Alpes et codirigée par **Guillaume BOUFFARD**, Agence Nationale de la Sécurité des Systèmes d'Information

préparée au sein du **Laboratoire CEA/LETI**
dans **l'École Doctorale Electronique, Electrotechnique, Automatique, Taitement du Signal (EEATS)**

Évaluation de la sécurité physique des SoC

SoC physical security evaluation

Thèse soutenue publiquement le **24 mars 2021**,
devant le jury composé de :

Monsieur Jessy Clédière
CEA-E5, Université Grenoble Alpes, Directeur de thèse
Madame Karine Heydemann
MAITRE DE CONFERENCE, Laboratoire d'Informatique de Paris 6, Rapportrice
Monsieur Philippe Maurine
MAITRE DE CONFERENCE, Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier (LIRMM), Rapporteur
Madame Clémentine Maurice
CHARGE DE RECHERCHE, CNRS, CRIStAL, Université de Lille, Examinatrice
Madame Marie-Laure Potet
PROFESSEUR, ENSIMAG, Examinatrice et Présidente du Jury
Monsieur Jean-Max Dutertre
PROFESSEUR, École des Mines de Saint Étienne, Examinateur
Monsieur Patrick Schaumont
PROFESSEUR, Worcester Polytechnic Institute, Examinateur
Monsieur Lilian Bossuet
PROFESSEUR, Université Jean-Monnet, Examinateur
Monsieur Patrick Haddad
DOCTEUR INGENIEUR, ST Microelectronics, Invité
Monsieur Guillaume Bouffard
DOCTEUR, Agence Nationale de la Sécurité des Systèmes d'Information/Ecole Normale Supérieure, Invité, co-encadrant

ABSTRACT

Since the democratization of mobile devices, sensitive operations like payment, identification or healthcare, usually done using security evaluated smartcards, are handled by these devices. However, mobile devices neither are designed for security nor security evaluated. Therefore, their resistance against powerful attacks, like physical attacks is questionable.

In this thesis, we aim at evaluating the security of mobile devices against physical attacks, in particular perturbation attacks. These attacks aims at modifying the execution environment of the device to induce bugs during its computation. These bugs are called faults. These faults can compromise the security of a device by allowing the cryptanalysis of its secret or forcing an unauthorized authentication for instance.

Mobile devices are powered by modern processors, which are the heart of this work, and are never evaluated against fault attacks. However, our knowledge about fault attacks on smartcards is not relevant as the processors powering smartcards are way less complex, in terms of number of modules, technology node and optimization mechanisms, than modern processors.

Regarding this situation, we aim at providing rationals on the security of modern processors against fault attacks by defining a fault characterization method, using it on representative modern processors and analyzing classical security mechanisms against the characterized faults.

We characterized three devices, namely the BCM2837, BCM2711b0 and the Intel Core i3-6100T against fault attacks using two different injection mediums: electromagnetic perturbations and a laser. We determined that these devices, despite having different architecture and using different mediums are faulted in similar ways. Most of the time, a perturbation on these devices modify their executed instructions.

As this is a powerful fault, we also analyzed classical security mechanisms embedded in such devices. We successfully realized a differential fault analysis on the AES implementation of the OpenSSL library, which is used in every Linux based operating system. We also analyzed the Linux user authentication process involved in the `sudo` program. This work highlights the lack of tools to efficiently analyze Linux programs, which are rather complex with dynamic linking mechanisms, against fault attacks.

RÉSUMÉ

De nos jours, nos appareils mobiles sont utilisés pour réaliser des opérations sensibles telles que du paiement, de l'identification ou la gestion de services santé. Historiquement, ces opérations sont réalisées par des appareils conçus et évalués pour résister à diverses attaques: les éléments sécurisés. En revanche, les appareils mobiles sont conçus pour fournir la meilleure performance possible et ne subissent aucune évaluation de sécurité. Cet état de fait interroge sur la résistance de ces appareils face aux attaques classiques contre lesquelles se protègent les éléments sécurisés.

Parmi ces attaques, nous nous proposons, dans cette thèse, d'étudier les attaques par perturbations. Ces attaques consistent à modifier les conditions d'exécution du circuit ciblé afin d'induire des erreurs dans son fonctionnement. Ces erreurs volontaires, communément appelées fautes, permettent de créer des failles dans la cible pouvant aller jusqu'à la cryptanalyse d'un algorithme de chiffrement ou l'authentification d'un utilisateur non autorisé.

Bien que ces méthodes d'attaques soient connues et étudiées sur les éléments sécurisés, les appareils modernes reposent sur des processeurs modernes présentant des différences par rapport aux processeur des éléments sécurisés. Cela peut être le nombre de module qu'ils embarquent, leur finesse de gravure ou des optimisations.

L'impact de ces différences sur la sécurité des processeur n'a pas été étudié en prenant en compte la possibilité d'induire des fautes. C'est ce que nous réalisons dans cette thèse. Nous définissons une méthode permettant de caractériser les effets de perturbations sur un processeur moderne que nous appliquons sur trois processeurs représentatifs des appareils existants: le BCM2837, le BCM2711b0 et l'Intel Core i3-6100T. Nous avons également utilisés deux moyens de perturbation classiques: l'injection d'onde électromagnétique et l'utilisation d'un laser. L'étude de ces cibles, en variant les moyens d'injections de faute, nous a permis de déterminer qu'elles réagissent toutes de manière similaire aux différentes perturbations malgré leur différentes architectures. L'effet le plus marquant étant la modification des instructions exécutées.

Ce type de faute est très fort car il permet de modifier une partie du programme exécuté pendant son exécution. Vérifier le programme avant de l'exécuter ne protège en rien face à ce type de fautes, par exemple. C'est pourquoi nous avons également étudié la résistance des mécanismes de sécurité présents dans ces cibles face à ce type de faute. Nous avons notamment réussi à cryptanalyser l'implémentation de l'algorithme de chiffrement AES de la bibliothèque OpenSSL, très utilisé dans les systèmes utilisant Linux. Nous avons également étudié la résistance du mécanisme d'authentification des utilisateurs d'un système Linux en regardant le programme `sudo`. Cette étude nous a, en particulier, révélé que la communauté manque

d'outils efficace pour analyser ce type de programmes face aux fautes. En effet, les programmes s'exécutent dans un environnement Linux bénéficient de nombreux mécanismes liés au noyau Linux qui rendent l'exécution d'un programme difficile à étudier.

So what if you fail once or twice ?
I don't even know how many thousand times I failed to control my rage.

— Dragon Sin of Wrath, Meliodas (The Seven Deadly Sins)

REMERCIEMENTS

S'il est bien une chose que tous les efforts du monde ne peuvent égaler, c'est l'inspiration, l'aide et les conseils des personnes que nous croisons au cours de notre vie.

Et parce que cette thèse présente une partie des efforts que j'ai fourni durant ces trois dernières années, je tiens à dédier ce chapitre à ces personnes qui m'ont donné de quoi arriver jusqu'ici. Il est difficile de mettre un mot, ou une définition sur ce que j'ai reçu, mais je sais que cela m'a énormément aidé.

Avant toute chose, je tiens à remercier mon jury de thèse, Mme. Karine Heydemann, M. Philippe Maurine, Mme. Clémentine Maurice, Mme. Marie-Laure Potet, M. Jean-Max Dutertre, Mr. Patrick Schaumont, M. Lilian Bossuet et M. Patrick Haddad, qui, non seulement d'accepter de juger de mon travail, m'ont régulièrement aidé à travers des critiques et des échanges sur mes travaux qui ont souvent porté leurs fruits.

Je remercie également mon directeur de thèse, M. Jessy Clédière, pour son aide et ses conseils tout au long de la thèse malgré la distance qui nous sépare.

Et je remercie très chaleureusement mon encadrant, et collègue, et désormais ami, M. Guillaume Bouffard. Le remercier pour son encadrement ne lui rend pas justice tant son investissement dépasse celui d'un encadrant, nous avons pu échanger, partager et nous entraider tellement souvent que je ne peux tout énumérer ici. Il me tarde de pouvoir retourner partager des bières en terrasse avec toi !

D'autres remerciements, très forts, se doivent d'aller à mes parents Emmanuelle et François. Leur soutien ne se matérialise pas par des conseils sur la sécurité des processeurs ou sur une critique de la méthode de caractérisations des fautes mais par des choses bien plus indescriptibles pour moi. Ils ont été, sont et seront un soutien qui apporte bien plus que ce que je ne peux décrire. Que ce soit un foyer familial quand le moral est en baisse ou un stock de papier de toilette en pleine pandémie.

Une dédicace toute spéciale va à ma petite sœur, Julie. Encore une fois, décrire la nature exact de l'aide qu'elle m'apporte est très difficile, elle a ce magnifique défaut d'être incapable de voir mes défauts, mes erreurs et mes échecs et sera toujours une bonne raison de faire en sorte que je n'ai rien de tout ça.

Ma prochaine pensée va à l'ensemble des professeurs que j'ai eu durant ma scolarité. J'ai eu l'occasion de m'essayer à l'exercice de l'enseignement et j'ai bien compris que certains m'ont apporté plus

que cela. Je pense en particulier à Livia, Khalid et Francesco, mes professeurs de collège lorsque j'étais en Roumanie, sans savoir pourquoi, c'est à partir de cette période que j'ai commencé à vraiment m'intéresser aux sciences. Je tiens également à remercier mes professeurs de lycée pour m'avoir poussé sur la voie vers les écoles d'ingénieurs. Je remercie également, justement, ces professeurs que j'ai rencontré, en école d'ingénieur, à Gardanne, et qui m'ont donné ou dévoilé cette fibre pour la micro-électronique et m'ont un peu poussé vers la thèse, je pense en particulier à M. Jean-Baptiste Rigaud.

J'en profite pour remercier mes camarades de promotion, avec qui j'ai passé des moments uniques que ce soit en cours, en soirée ou en association. Je remercie en passant l'équipe du mandat 9 de M-GaTE que j'ai eu la chance de présider pendant un an et avec lesquels j'ai beaucoup appris.

Naturellement, mes prochains remerciements vont envers mes amis, que je ne remercierai jamais assez et qui m'apportent énormément. Je pense aux Maissois, Benoît, Gaëlle et leur fille Maëlyne (*aka.* Choupette !) dont j'ai la chance d'être le parrain. Vous formez une famille exceptionnelle et je suis heureux d'en être un membre invité ! Je remercie également mes camarades judokas, Rodolphe et Sylvie, dont l'exigence et la douceur s'équilibrent parfaitement, Kathleen qui a hérité des deux, Yann et Sylvain, les frangins qui croquent la vie à pleine dent, Jérôme qui parcourt le monde, Rémi à la volonté infaillible avec qui j'ai eu la chance de passer les *kata*, André qui nous a formé pour ces *kata*, la famille Mota, l'équipe cadet: Killian, Sylvain, Morgan et Alexandre, la famille Reytier: Patrick et Brigitte, Fabien, Morgane (*aka. Meuh*) ainsi que Kévin, Kelly et Maiwen qui m'ont offert un vrai refuge avec des barbecues, des soirées et des séances de *wakeboard* ou de voile mémorables. Vous m'avez énormément appris, tant sur l'importance des amis et d'une équipe, que sur l'exigence personnelle ainsi que l'investissement physique et psychologique pour progresser, et évidemment la volonté de se battre malgré ce qui nous fait face. Tout ceci m'a servi, pour la réalisation des travaux présenté dans cette thèse, mais pas seulement. Parmi les Maissois qu'il reste, je tiens à remercier mes camarades de soirées, où nous avons tantôt montrer nos talents au *Beer-Pong* tantôt refait le monde, merci Florian, Jérémy B. et Juliette (le couple parfait !), Jérémy T., Adeline et Marjolaine, ma super voisine avec qui je courrais après le bus pour aller au lycée.

Une pensée particulière va envers Jean, qui m'a appris à jouer de la guitare mais qui au final m'a partagé bien plus que de la technicité sur un instrument. C'est une vrai philosophie de vie, mise à l'épreuve par la maladie, qui m'a été transmise et que je veux honorer.

Parmi les pensées particulières, il y en a pour ces amis de longue date, ces relations qui survivent à la distance et au temps, je pense en particulier à Thomas L., Antoine et Thibaut Tai. Savoir que vous êtes quelque part, plus ou moins dans le coin, et qu'on partagera de nouveaux de bons moments ensemble est toujours bon pour le moral. J'ai hâte d'entendre tout ce que vous avez à me raconter !

Et puisque que nous en sommes aux bons amis, je tiens à remercier des bandes particulières. Les Maissois et les judokas, que j'ai déjà eu l'occasion de remercier. Merci aux 6 doigts de la main (ordre de chevaliers encore meilleurs que les 7 pêchés capitaux), Arnaud le chanteur, Florian S. le sportif, Camille la motarde, Pierre (*aka.* Caillou) le futur astronaute breton ! et Paul (*aka.* Skøll) le viking, pour votre bonne humeur, votre soutien ainsi que les *Skypéro* ! Mention spéciale à Marie et Fanny, qui accompagnent deux d'entre eux dans la vie et qui me grillent (trop) facilement sur *Among Us*. Merci également à la *Dream Team*, David, le chevalier au cœur pur, Guillaume, que j'ai déjà eu l'occasion de mousser et Louiza, la plus grande adoratrice de plantes et de chats au monde, vous êtes des collègues exceptionnels.

Je tiens à remercier également les *Nakamas*, qui m'ont fait découvrir le *crossfit* et dont la bonne humeur n'égale que l'énergie, que ce soit pour enchaîner les *burpess*, à la *box* ou dans mon salon, ou pour dévorer des barbecues et des raclettes. Mention spéciale aux coachs: Jérémy, Mickael et Édouard qui sont des exemples à suivre dans bien des domaines.

Enfin merci à la Légion Jedi, une superbe guilde avec laquelle j'ai occupé de nombreuses heures de confinement et de couvre-feu à rigoler tout en sauvant la galaxie !

Et pour finir, je remercie également l'ensemble de mes collègues de l'ANSSI, et en particulier ceux du laboratoire de sécurité des composants, pour leur accueil, leur soutien, leur bonne humeur, les bières le soir et les discussions polémiques. Merci Eliane (*the boss*), Patrick, Guenaël, Emmanuel, Karim, Ryad, Adrian, Guillaume, Boris, David, Louiza, Julien et Yoan.

CONTENTS

i INTRODUCTION 1
1 ABOUT CYBERSECURITY 3
1.1 Cybersecurity problematic 4
1.2 Secure Devices Evaluation 4
1.3 ANSSI's role in France 6
1.4 This thesis 6
2 STUDY CONTEXT 9
2.1 Introduction 10
2.2 Secure Elements 10
2.2.1 Architecture 11
2.2.2 Central Processing Unit (CPU) 12
2.2.3 Packaging 13
2.3 Systems On Chip 13
2.3.1 Architecture 15
2.3.2 CPU 17
2.3.3 Packaging 20
2.4 Multi-application system security 21
2.5 Conclusion 23
3 PHYSICAL ATTACKS 25
3.1 Side-channel attacks 26
3.1.1 Micro-Architectural Attacks 28
3.1.2 Side-Channel countermeasures 28
3.2 Invasive attacks 28
3.2.1 Reverse engineering 29
3.2.2 Focused Ion Beam 29
4 PERTURBATION ATTACKS 31
4.1 Genesis 33
4.2 Inducing a fault: injection mediums 34
4.2.1 Design considerations 34
4.2.2 Clock glitches 36
4.2.3 Voltage glitches 36
4.2.4 Temperature manipulation 37
4.2.5 Electromagnetic perturbations 37
4.2.6 Optical perturbations 39
4.2.7 Body biasing 41
4.2.8 X-Rays 41
4.2.9 Software induced perturbations 41
4.3 Characterizing a fault 44
4.3.1 Fault models 44
4.3.2 Fault analysis 46
4.4 Exploiting a fault 47
4.4.1 Fault attacks against multi-application systems 47
4.5 Countermeasures 56
4.5.1 Space redundancy 57
4.5.2 Operation duplication 57

4.5.3 Infection countermeasure 58
4.5.4 Code hardening 58
4.5.5 Sensors 60
4.6 Conclusion on perturbation attacks 60

ii CONTRIBUTION 63
5 FAULT EFFECT CHARACTERIZATION ON SYSTEMS ON CHIP 65
5.1 SoC modeling 66
5.2 Attacker model 66
5.3 Experimental method 67
5.3.1 Top-down approach 67
5.3.2 Target setup 68
5.4 Determining the faulted element 73
5.5 Conclusion 75
6 EXPERIMENTAL WORK 77
6.1 Practical work setup 78
6.1.1 Attack benchs 78
6.1.2 Evaluated devices 81
6.1.3 Tools 85
6.2 BCM2837 characterization 91
6.2.1 Hot-spots maps 92
6.2.2 Analyzer results 94
6.2.3 Micro-architectural analysis using a test program 100
6.2.4 Micro-architectural analysis on a baremetal setup with JTAG 104
6.2.5 Conclusion on the BCM2837 characterization 107
6.3 BCM2711b0 characterization 108
6.3.1 Hot-spots maps 108
6.3.2 Analyzer results 109
6.3.3 Conclusion on the BCM2711b0 characterization 113
6.4 Intel Core i3 characterization 114
6.4.1 Hot-spots maps 114
6.4.2 Analyzer results 116
6.4.3 Conclusion on the Intel Core i3-6100T 119
6.5 Characterization conclusion 119
7 FAULT MODEL EXPLOITABILITY 121
7.1 DFA on the OpenSSL AES implementation 123
7.1.1 Source code location 123
7.1.2 Static analysis 124
7.1.3 Fault attack on the OpenSSL AES 126
7.1.4 DFA software 128
7.1.5 Conclusion on the OpenSSL AES DFA 132
7.2 Baremetal AES PFA [159] 132
7.2.1 PFA Result 132
7.2.2 Conclusion on the PFA on our baremetal AES 134
7.3 sudo authentication 134
7.3.1 Attack model 135
7.3.2 sudo analysis 135
7.3.3 Code analysis 141
7.3.4 Program setup 142

7.3.5 Side channel analysis 144
7.3.6 Exploitation 145
7.3.7 Conclusion on the forced authentication 147
7.4 Analysis tools 147
7.4.1 The function analyzer 148
7.4.2 Fault simulator 149
7.5 Evaluation conclusion 151

iii conclusion 153
8 conclusion 155

iv appendix 159
a bcm2837 maps per fault model 161
b fault analyzer interface 163
c gdb analysis of sudo with debug symbols 167
d sudoers_policy_check() disassembly 169
e openssl aes aes_encrypt() round function (dissassembled) 171
e.1 OpenSSL AES round computation (Disassembled) 171

bibliography 173

LIST OF FIGURES

Figure 1 Evaluation process actors (example with Common Criteria (CC)) 5
Figure 2 Smartcard in a payment environment 10
Figure 3 Secure element architecture 11
Figure 4 Secure element CPU architecture 12
Figure 5 Secure element packaging 13
Figure 6 ISO7816 pins 13
Figure 7 Smartphone environment 14
Figure 8 System on Chip architecture 15
Figure 9 Cache incoherence after a module updated a data in its dedicated cache memory 16
Figure 10 Modern CPU architecture 17
Figure 11 Out-of-order execution principle 19
Figure 12 Package on package 20
Figure 13 BGA grid on a PCIe chip 21
Figure 14 Multi-application system security dependencies 23
Figure 15 Current in a Complementary MOS (CMOS) logical inverter gate for different inputs. 27
Figure 16 Perturbation attack analysis and evaluation process 32
Figure 17 *Stimuli* able to perturb a digital device 34
Figure 18 Timing constraints on the input for a DFF to behave correctly 35
Figure 19 Timing constraint in digital devices 35
Figure 20 Clock glitch effect on a D Flip-Flop (DFF) 36
Figure 21 How ElectroMagnetic (EM) sampling faults occur [71] 38
Figure 22 ST Microelectronics M27C256B Erasable Programmable Read-Only Memory (EPROM) 39
Figure 23 Laser effect on a CMOS logical inverter outputting a logical zero 40
Figure 24 One-bit DRAM memory cell 42
Figure 25 DRAM line activation and copy in the row buffer 43
Figure 26 PMS supplying the power voltage and clock to cores and module of a SoC 43
Figure 27 Fault propagation through digital devices abstraction layers with some fault effects as examples. Inspired from [94]. 45
Figure 28 Fault injection characterization state of the art. 46
Figure 29 Differential Fault Analysis (DFA) Principle 52
Figure 30 Propagation of a faulted byte before the last `MixColumns` operation in the Advanced Encryption Standard (AES) 53
Figure 31 Memory partioning principle 54

Figure 32 Boot loader stage for bypassing secure boot using a fault attack [101] 55
Figure 33 Operation duplication possible implementations 57
Figure 34 Software countermeasures integration in the compilation process. 59
Figure 35 Fault effect characterization overview 67
Figure 36 General program organization 68
Figure 37 Succession of states during the execution of a program containing only non-changing state instructions 69
Figure 38 Generic attack bench organization and interactions 78
Figure 39 ANSSI's EM bench with Intel Core i3 DUT 80
Figure 40 Pulse generated by the AvTech (100 V input) 80
Figure 41 Pulse generated by the AvTech zoomed on the first peak (200 V input) 81
Figure 42 Raspberry Pi 3 model B board 82
Figure 43 BCM2837 infrared backside layout image 83
Figure 44 Open BCM2837 with the chip in its package 83
Figure 45 Raspberry Pi 4 board 84
Figure 46 BCM2711b0 infrared backside layout image 84
Figure 47 Intel Core i3 SoC 85
Figure 48 Bench manager software general organization 85
Figure 49 Example of an experiment process 86
Figure 50 Fault analyzer software principle 87
Figure 51 Hot spots of the BCM2837 regarding EM perturbation 93
Figure 52 Input voltage amplitude effect on BCM2837 during EM perturbation 93
Figure 53 `and r8,r8` faulted values distribution on BCM2837 using EM perturbation 94
Figure 54 `orr r5,r5` faulted values distribution on BCM2837 using EM perturbation 95
Figure 55 Probability of observed registers to be faulted for both experiments on BCM2837 using EM perturbation 95
Figure 56 Probability of observing the different fault models for both experiments on BCM2837 using EM perturbation 96
Figure 57 Data processing instruction encoding on ARM 99
Figure 58 Pipeline execute stage architecture with ARM instruction (figure 57) corresponding bits 102
Figure 59 Probability of observing the different fault models for `orr r3,r3` experiment on BCM2837 with a baremetal setup using EM perturbation 105
Figure 60 BCM2711b0 hot spots leading to faults using laser perturbation 108
Figure 61 `orr r5,r5` faulted values distribution on BCM2711b0 using laser perturbation 109

Figure 62 Probability of observed registers to be faulted for `orr r5,r5` experiment on BCM2711b0 using laser perturbation 110
Figure 63 Probability of observing the different fault models for `orr r5,r5` experiment on BCM2711b0 using laser perturbation 110
Figure 64 BCM2711b0 hot spots leading to the different observed fault models 112
Figure 65 BCM2711b0 hot spots leading to the different observed fault models (alternative version) 113
Figure 66 Intel Core i3 hot spots leading to reboots using EM perturbation 115
Figure 67 Input voltage amplitude effect on Intel Core i3 during EM perturbation 115
Figure 68 `mov rbx,rbx` faulted value distribution on Intel Core i3 using EM perturbation 116
Figure 69 `orr rbx,rbx` faulted value distribution on Intel Core i3 using EM perturbation 117
Figure 70 Probability of observing the different fault models for both experiments on Intel Core i3 using EM perturbation 118
Figure 71 Linux dynamic linker interventions during program execution 122
Figure 72 Impact of the delay on the probability while faulting an OpenSSL AES encryption using EM perturbation on BCM2837. 126
Figure 73 Impact of the delay on the number of faulted diagonals while faulting an OpenSSL AES encryption using EM perturbation on BCM2837. 127
Figure 74 Distribution of the first byte for 10 000 faulted ciphers on our baremetal AES on the BCM2837. 133
Figure 75 `sudo` calling architecture and dependencies regarding the user authentication process. 141
Figure 76 Target program execution flow 143
Figure 77 EM activity of the CPU during a user authentication on BCM2711b0 145
Figure 78 Analyzer and simulator software architecture. 148
Figure 79 `sudoers_policy_check` function partial disassembly printed in a shell. 150
Figure 80 Fault simulation process. 151
Figure 81 BCM2837 hot spots leading to the different observed fault models 161
Figure 82 Fault analyzer graphical interface 164

Figure 83 `sudoers_policy_check()` Tikz disassembly. 169

LIST OF TABLES

Table 1 Target initial values for fault characterization considering ten registers 73
Table 2 Data fault models 74
Table 3 Instruction fault models 75
Table 4 `orr r5,r5` corruptions regarding the “Or with other obs” fault model 97
Table 5 `orr r5,r5` corruptions regarding the “Or with two other obs” fault model 98
Table 6 `orr r5,r5` corruptions regarding the “Other obs value” fault model 98
Table 7 `and r8,r8` corruptions regarding the “Other obs value” fault model 98
Table 8 Binary values of the observed opcodes. 100
Table 9 Fault distribution on `cmp` test code 101
Table 10 `orr r5,r5` corruptions regarding the “Other obs value” fault model on the BCM2711b0 perturbed with a laser 113
Table 11 `mov rbx,rbx` corruptions regarding the “Other obs value” fault model 118
Table 12 `orr rbx,rbx` corruptions regarding the “Other with other obs” fault model 119
Table 13 Forbidden values for every byte of the observed ciphers. 133
Table 14 Possible key bytes for the correct guess $y_1 =$ `0x30` with the correct key in red. 134
Table 15 Library paths 140

LISTINGS

Figure 4.1 Step counter countermeasure on a block of I instructions 60
Figure 5.1 Example of a test program for fault characterization 70
Figure 5.2 Example of a test program for fault characterization on memory accesses 70
Figure 5.3 Example of a test program for instruction skip/repetition characterization 71

Figure 5.4 Example of a test program for instruction repetition with replacement characterization 71
Figure 6.1 Example of params.py file for the fault analyzer 87
Figure 6.2 Example of values for which the fault analyzer could not determine a fault model. 90
Figure 6.3 Types of fault models as defined in fault_models.py 91
Figure 6.4 Or with other observed fault model test function as implemented in fault_models.py 91
Figure 6.5 Fault class as implemented in fault.py 92
Figure 6.6 Two first instruction fault models as implemented in fault_models.py 92
Figure 6.7 BCM2837 immediate value test program 101
Figure 6.8 BCM2837 memory test code 103
Figure 6.9 BCM2837 baremetal test program 105
Figure 6.10 BCM2837 assembly code of the loop test program 106
Figure 6.11 BCM2837 with baremetal setup with correct identity memory mapping 107
Figure 6.12 BCM2837 with baremetal setup with faulted memory mapping 107
Figure 7.1 OpenSSL AES pre-computed tables (partial) 124
Figure 7.2 OpenSSL AES round computation (C) 125
Figure 7.3 DFA program main loop 129
Figure 7.4 Example usage and result of the DFA program on the first diagonal 130
Figure 7.5 AES correct key as implemented in the DFA program 130
Figure 7.6 Faulted ciphers used in our DFA (ciphers_diag0.txt) 130
Figure 7.7 Commands to identify the strcmp calls of the sudo program using gdb. 137
Figure 7.8 gdb output of a strcmp call comparing two hashes. 138
Figure 7.9 gdb output of a strcmp call comparing two hashes with libraries compiled with debug symbols. A bigger representation is available in appendix C. 139
Figure 7.10 sudo binary policy return value checking in main function from sudo.c 141
Figure 7.11 C code for the child process in the sudo evaluation setup. 143
Figure 7.12 Part of the C code for the parent process communicating with the child process. 144
Figure B.1 Fault analyzer command line interface (List of experiments). Experiments with a star (*) are already analyzed. 163
Figure B.2 Fault analyzer command line interface (List of results) 163
Figure B.3 Fault analyzer command line interface (Result) 164
Figure B.4 main of the fault analyzer 165

Figure C.1 `gdb` output of a `strcmp` call comparing two hashes with libraries compiled with debug symbols. 167

LIST OF EQUATIONS

Figure 4.1 Setup timing constraint on clock cycle in a digital device. 35
Figure 4.2 Hold timing constraint on clock cycle in a digital device. 36
Figure 4.3 Fault masking attack model. 40
Figure 4.4 Computation of a cipher using RSA-CRT 48
Figure 4.5 Factorization of RSA modulus in a faulted CRT implementation 48
Figure 4.6 Factorization of RSA modulus in a faulted CRT implementation (Lenstra's improvement) 49
Figure 4.7 Relation between the faulty signature obtained by skipping the last squaring and the correct signature in the square and multiply algorithm. 49
Figure 4.8 Relation between the faulty signature obtained by skipping the squaring in the ith and $(i-1)^{th}$ iteration of the loop in the square and multiply algorithm. 50
Figure 4.9 Bézout's identity 50
Figure 4.10 AES last round *SubBytes* and *AddRoundKey* operations in the presence of a corrupted SBox 51
Figure 4.11 Relation between the forbidden value and the secret key 51
Figure 4.12 Relation between the faulty ciphertext, the correct ciphertext and the possible faulty outputs of the *MixColumns* operation during a DFA on AES 52
Figure 4.13 Relation between the faulty ciphertext and the correct cipher text in the case of the skipping of the last *AddRoundKey* operation in AES 53
Figure 4.14 Relation between the virtual address and the physical address in Linux based OS 54
Figure 4.15 Infection countermeasure 58

Figure 5.1 Difference between all registers initial values property 72
Figure 5.2 No arithmetical links between registers initial values property 72
Figure 5.3 Registers initial values construction 72
Figure 5.4 State computation in a CPU 73
Figure 5.5 Faulted state computation in a CPU 74
Figure 5.6 Data fault model 74

Figure 5.7 Instruction fault model 74
Figure 7.1 Pre-computation of the Te tables in the OpenSSL AES 124

ACRONYMS

se Secure Element

soc System on Chip

fi Fault Injection

emfi ElectroMagnetic Fault Injection

emp EM Pulse

tee Trusted Execution Environment

iot Internet of Things

pc Program Counter

dut Device Under Test

iot Internet of Things

cpu Central Processing Unit

os Operating System

bga Ball Grid Array

dfa Differential Fault Analysis

pmic Power Management Integrated Circuit

mmu Memory Management Unit

io Input/Output

alu Arithmetical and Logical Unit

gpio General Purpose Input/Output

em ElectroMagnetic

lfi Laser Fault Injection

dpa Differential Power Analysis

cpa Correlation Power Analysis

mos Metal Oxide Semiconductor

nmos N-doping MOS

pmos P-doping MOS

cmos Complementary MOS

aes Advance Encryption Standard

sca Side-Channel Analysis/Attack

mia Mutual Information Analysis

lra Linear Regression Attack

tpm Trusted Platform Module

scp Secure Channel Protocol

iot Internet of Things

rsa Rivest, Shamir, and Adelman

ecdsa Elliptic Curve Digital Signature Algorithm

aes Advanced Encryption Standard

fa Fault Attack

vlsi Very Large Scale Integration

ram Random Access Memory

dram Dynamic RAM

des Data Encryption Standard

ecc Elliptic Curve Cryptography

mcu Micro-Controller Unit

fbbi Forward Body Biasing Injection

dff D Flip-Flop

rom Read-Only Memory

eprom Erasable Programmable Read-Only Memory

eeprom Electrically EPROM

uv Ultra Violet

ir Infra Red

puf Physically Unclonable Function

gpu Graphics Processing Unit

rdma Remote DMA

pms Power Management Subsystem

pll Phase Locked Loop

fpga Field Programmable Gate Array

isa Instruction Set Architecture

cfg Control Flow Graph

crt Chinese Reminder Theorem

pfa Persistent Fault Analysis

sbox Substitution Box

jcvm JavaCard Virtual Machine

bcv ByteCode Verifier

mpu Memory Protection Unit

pte Page Table Entry

ddr4 DDR version 4

sram Synchronous Random Access Memory

rop Return-Oriented Programming

dos Deny Of Service

ahb Advanced High-performance Bus

axi Advanced eXtensible Interface

trng True Random Number Generator

pin Personal Identification Number

gsm Global System for Mobile communication

vpu Video Processing Unit

npu Neural Processing Unit

adb Android Debug Bridge

pcb Printed Circuit Board

spi Serial Peripheral Interface

scssi Service Central de la Sécurité des Systèmes d'Information

cnet Centre National d'Étude des Télécommunications

nfc Near Field Communication

jtag Join Test Action Group

mab Micro-Architectural Block

hdmi High-Definition Multimedia Interface

tlb Translation Lookaside Buffer

cc Common Criteria

cspn Certificat de Sécurité de Premier Niveau

anssi Agence Nationale de la Sécurité des Systèmes d'Information

eu European Union

GDPR General Data Protection Regulation

RTL Register Transfert Level

FIB Focused Ion Beam

BSZ Beschleunigte Sicherheitszertifizierung

BSPA Baseline Security Product Assessment

ITSEF Information Technology Security Evaluation Facility

VNA Vector Network Analyzer

Part I

INTRODUCTION

It's good to know where you come from...

1 ABOUT CYBERSECURITY

Even though we should learn from those who came before us, we must also forge our own path.

— Avatar Korra (The Legend Of Korra)

ABSTRACT

This chapter presents the impact of new digital technologies development since the start of the 2000's regarding security. It introduces the need of secure and evaluated device for providing some critical services needed in our society. This highlights the need of product evaluation and the importance of defense agencies, in France. It also places the work presented in this thesis regarding this security evaluation problematic applied on modern devices.

Contents

1.1 Cybersecurity problematic 4
1.2 Secure Devices Evaluation 4
1.3 ANSSI's role in France 6
1.4 This thesis 6

1.1 CYBERSECURITY PROBLEMATIC

Since the democratization of the Internet and the development of many services relying on digital devices and remote communication, several threats against people, companies and states data have been identified. Indeed, a lot of, and sometimes critical, services rely on the correct behavior of digital systems. Therefore, these systems security is important to assess.

Currently, there are two main sensitive elements to protect: the data privacy and the service's continuous operation. Regarding data privacy, since the development of important companies such as Google, Amazon, Facebook, Apple and Microsoft (also known as the GAFAM), the storing and processing of data belonging to people is a burning issue. Recently, the European Union (EU) has taken an important initiative with the General Data Protection Regulation (GDPR)[1]. The GDPR aims at giving people more control over their personal data collected by companies. Companies are imposed to respect and apply owner data's choices such as accessing them or deleting them all.

Regarding online services, as some of them provide sensitive operations, they must be secured against attacks to be sure they remain on online. Sensitive actors are banks, the Internet providers, the transport companies, the energy provider companies, the state with all its ministries and agencies, *etc*. Regarding all the services they manage, the threats are multiple. For instance, a more and more present one is the ransomware attack. According to a recent interview of the general director of the Agence Nationale de la Sécurité des Systèmes d'Information (ANSSI) (National Security Agency of France) by the French Senate[2], the number of ransomware attacks has grown with a factor between three and four in a year. To prevent ransomware, the solution is to provide secure infrastructures to organizations. Ultimately, securing an infrastructure requires secure and trusted softwares and devices.

To identify these products, several public and private security evaluation schemes have appeared. These schemes aim at evaluating the security of products against state of the art attacks. Then, the usage of these products ensures a security level corresponding to their evaluation level regarding the state of the art at the moment of their evaluation.

1.2 SECURE DEVICES EVALUATION

The evaluation of secure products is an important activity for providing secure systems to companies, governments and society in general. There exists several certification schemes: some private (EMVCo, FIPS, ISO, GlobalPlatform, *etc*) and some public (Common Criteria (CC) in Europe, Certificat de Sécurité de Premier Niveau (CSPN) in France, Beschleunigte Sicherheitszertifizierung (BSZ) in Germany, LINCE

1 https://ec.europa.eu/justice/smedataprotect/index_en.htm

2 https://www.usine-digitale.fr/article/ransomware-covid-19-espionnage-l-\
anssi-fait-un-etat-des-lieux-de-la-cybersecurite.N1024629

in Spain and Baseline Security Product Assessment (BSPA) in Netherland for instance).

An evaluation process aims at assessing the conformity of a product regarding a security reference. This security reference is challenged *via* criteria and a method. All existing evaluation schemes propose their own security reference, criteria and methods.

The evaluation of a product involves several actors presented in figure 1.

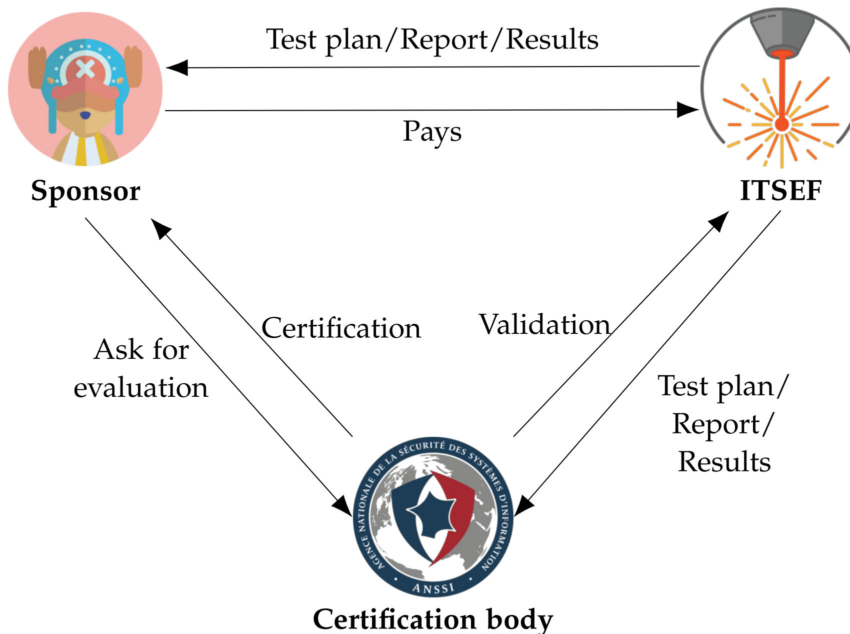


Figure 1: Evaluation process actors (example with CC)

This figure shows the main actors involved in the CC evaluation process:

- the sponsor, can be a company or an organization for instance, wants to realize the evaluation of a product. Depending on the evaluation level, the sponsor must provides different information about the evaluated product such as the source code for instance. Sometimes, the sponsor is the product developer but it is not necessary.
- the Information Technology Security Evaluation Facility (ITSEF) is the security laboratory in charge of the product evaluation. Depending on the target (hardware or software) different skills may be required. These ITSEFs are regularly challenged by the certification body to assess they are able to provide a state of the art aware evaluation.
- the certification body aims at assessing that the evaluator work is relevant and matches with the evaluation methodology and the state of the art regarding the security level asked by the sponsor. Also, this organization delivers the certification. In France, the certification body is the National Center for Certification which is a part of the ANSSI.

The evaluation of a product is a long process (from six months up to one year) which involves the definition of a target of evaluation, a test plan and a report which presents the test results and concludes about the conformity of the product regarding the targeted security reference.

The certification body ensures that the evaluators are up-to-date with security state of the art and validating the tests and results, it is important that it keeps a high expertise level on all the security topics and remains aware of efficient methods to evaluate a product. For these reasons, the ANSSI certification body is also assisted by seven laboratories all dedicated to a technical field of cybersecurity: cryptography, software and hardware architecture, software applications, exploration and detection, network and protocols, wireless communications and hardware.

1.3 ANSSI'S ROLE IN FRANCE

The ANSSI is the cybersecurity part of the French National Security agency, reporting directly to the prime minister of France. It aims at improving French citizens, companies and government's security against cyber-attacks. To fulfill this role, it carries several missions: educate people (professionals, military and civil) about the existing risks, proposing good practices (both in technology usage and development), provide a reaction in case of attacks, emulate the research and the development of secure technologies with universities and companies, keep a state of the art of the existing attacks and realize product certification.

Despite these missions remain unchanged since the agency's creation in 2009[3], the technologies and environment have changed a lot. One of the last changes is the democratization of mobile devices, such as smartphones for instance. Nowadays, a smartphone enable to run various applications easily, however, many of these applications manipulates sensitive data.

Moreover, more and more companies are willing to evaluate such devices. The problem is that these devices are not only dedicated to security but also provide a lot of services contrary to security oriented design devices that are usually evaluated. This design difference makes the study of multi-purposes (in other words, able to run various and non trusted applications) devices security a complex task. Also, due to their complexity, the state of the art about their security is shallow and there are no identified methods to evaluate it.

1.4 THIS THESIS

As a member of the ANSSI's hardware security lab, my thesis work aims at anticipating the requirement of security evaluation regarding mobile devices by giving answers to the following questions: how to evaluate the security of a modern system against physical attacks ?

3 https://www.legifrance.gouv.fr/jorf/id/JORFTEXT000020828212

Currently, there is no evaluation scheme that proposes an evaluation method for modern devices. Our aim is to use what already exists on currently evaluated devices and determine the effort to do to evaluate more complex devices.

Therefore, this work focuses on several questions. The first one is: what are the impacting differences, from a security point of view, between a modern device and a Secure Element (SE) we already evaluate ? This is discussed in chapter 2. From this analysis, we want to identify what are the known threats, from a hardware point of view, against SEs and if they are suitable for attacking a modern device. The state-of-the-art of the existing threats is discussed in chapter 3 and more specifically perturbations in chapter 4. The assessment that perturbations are effective against modern devices is discussed both in chapter 4 for the already existing perturbation methods and in chapter 6 for our own experiments.

After having assessed that perturbations are actually effective on modern devices, we want to determine if it is possible to effectively characterize and understand the perturbation effects on modern devices at various levels and despite their complexity. To answer this problem, we propose a characterization method we define in chapter 5 and that we apply in chapter 6 on several targets.

This lead to our final question: is it possible that the characterized faults obtained by perturbing the device are suitable to attack security mechanisms embedded in modern devices ? This supposes the knowledge of these mechanisms and a way to confront their normal behavior with perturbations. All of this is discussed in chapter 7.

2 STUDY CONTEXT

Sorry does not make noodles.

— M. San Ping (Kung Fu Panda)

ABSTRACT

This chapter presents secure elements and system on chip. Secure elements are historically the devices used for security while system on chip are a new kind of versatile devices performance oriented but more and more used for sensitive applications. This chapter focuses on the similarities and the differences between these devices to highlight the security concerns of system on chips.

Contents

2.1 Introduction 10
2.2 Secure Elements 10
2.2.1 Architecture 11
2.2.2 CPU 12
2.2.3 Packaging 13
2.3 Systems On Chip 13
2.3.1 Architecture 15
2.3.2 CPU 17
2.3.3 Packaging 20
2.4 Multi-application system security 21
2.5 Conclusion 23

2.1 INTRODUCTION

Nowadays, sensitive applications are handled by two kind of devices. SEs which are the historical, highly-secured and evaluated devices and System on Chips (SoCs) which are the new generation of devices, connected, low-energy, high-performance, versatile, *etc*.

The apparition of SoCs for sensitive applications raise security questions. This chapter focuses on the environmental, architectural and structural differences between SEs and SoCs to determine where these security concerns come from and what should be taken into account if one wants to evaluate a SoC security as it is done with SEs.

2.2 SECURE ELEMENTS

SEs are the historical digital devices, its design is security oriented and it is therefore dedicated to sensitive operations [1]. It started to be widely used once integrated in smartcards. For instance, during payments, the smartcard is a cornerstone element which ensures security properties as shown on figure 2. These security properties are provided by the SEs powering the smartcard and the reader.

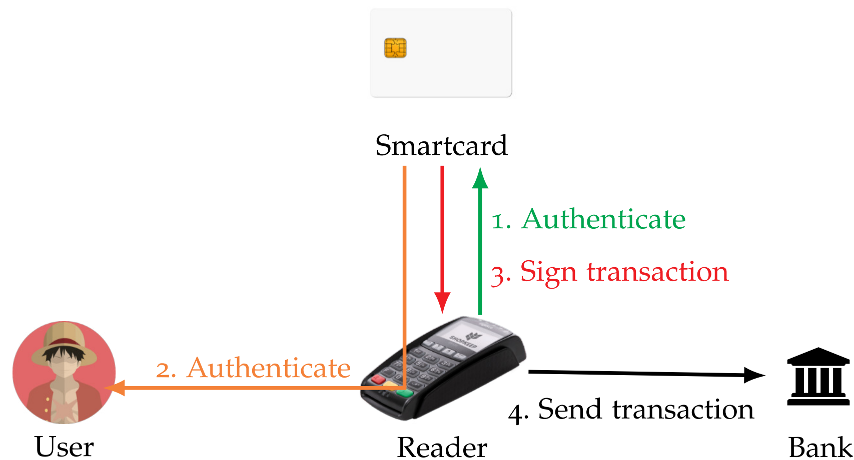


Figure 2: Smartcard in a payment environment

In the first step of the payment process, the reader will authenticate the smartcard by verifying its public key certificate which is signed by its issuer, most of the time a bank.

Once the smartcard is authenticated, the card will authenticate the user *via* the reader. This step is critical as it involves a secret for securing the transaction. Various methods exist for authenticating a user and the most used is the secret Personal Identification Number (PIN) code verification even if biometric verification is more and more used. By authenticating itself, the user accepts the transaction proposed by the reader.

Once both the smartcard and the user are authenticated, the reader realizes some checks such as the maximum transaction value or bank security rules. Then the transaction is accepted and the smartcard can

sign it with its private key. This is also a critical step as anyone who knows the private key can sign transactions.

Finally, once the transaction is signed, it is sent to the bank to be effective. This step does not involve the smartcard.

Regarding this high level view, the critical elements a SE must protect are the private key used to sign the transactions and the PIN code (or any other asset) used for authenticating the card owner. To ensure a secure element is effectively able to protect these assets, there are evaluation processes which aim at verifying the security of such devices. As completely securing a device, including doing the exhaustive verification of a device security is very complex task, SEs adopt a relatively simple architecture compared with more generic digital devices.

2.2.1 *Architecture*

The architecture of a SE is designed for security. However, depending on the use case, it might be able to execute multiple applications. Having multiple applications running on the same device introduces security concerns, involving in particular the sharing of the memory. To face this problem, SEs integrate a Memory Protection Unit (MPU) which manage the access to the memory. Also, they are powered by a virtual machine (usually the JavaCard Virtual Machine (JCVM)) which ensures the memory partitioning between applications. The global architecture of a SE is presented in figure 3.

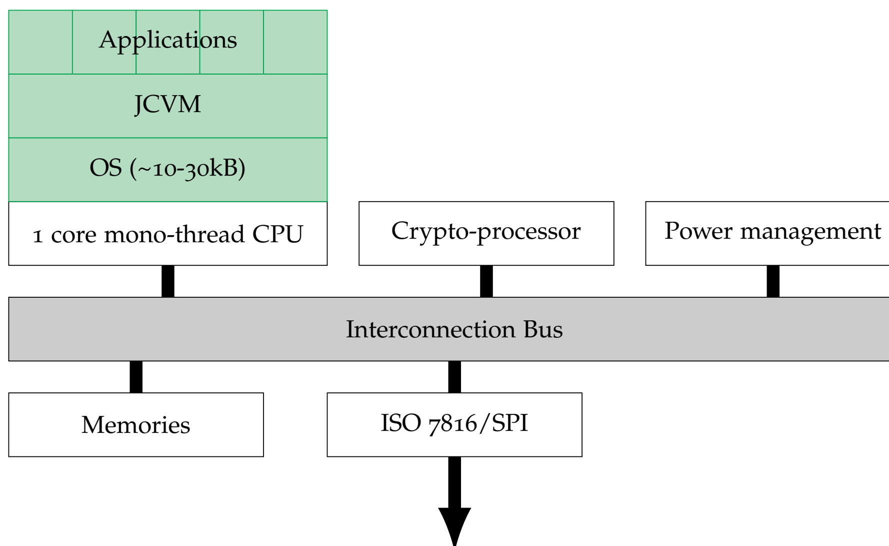


Figure 3: Secure element architecture

It shows that a SE is powered by a very simple CPU composed of only one core and the MPU. The core is usually based on the ARM architecture and is presented more in depth in section 2.2.2. For cryptographic operations, the CPU is helped by a crypto-processor which implements heavily protected cryptographic algorithms and a True Random Number Generator (TRNG). The memories are the Random Access Memory (RAM), a Read-Only Memory (ROM) or a Flash mem-

ory. The communication with the external world is done *via* a dedicated and standard interface, the most common one is the ISO7816 protocol but the Serial Peripheral Interface (SPI) is sometimes used. All these elements are connected via an interconnection bus, usually the bus is an ARM one such as the Advanced High-performance Bus (AHB) or Advanced eXtensible Interface (AXI) buses. Finally, all the hardware is powered by an external source for both the supply voltage and the clock, the reader supplies both of them.

On the software side, the SE is runs a small Operating System (OS) on which a virtual machine is added. Several technologies exist for it but the most used is the JCVM. The JCVM implements the Java Card specification [2] and interfaces with the components of the SE, in particular, the crypto-processor.

2.2.2 *CPU*

The CPU is essential in digital devices as it executes the programs and manipulates the data. Even if, on SEs, there is a dedicated crypto-processor implementing cryptographic algorithms, the CPU must ensure the good execution of the JCVM and all applications. Therefore, it is important to understand his designed. The architecture of a SE's CPU is shown in figure 4.

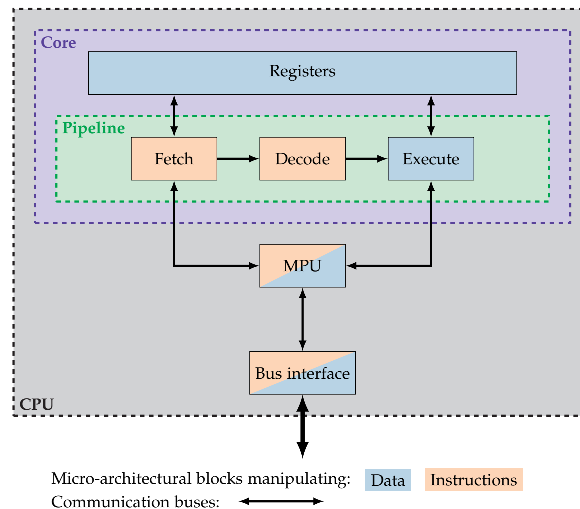


Figure 4: Secure element CPU architecture

The CPU of a SE is very simple: it is composed of a core which embed the pipeline that fetches and decodes instructions before executing them. The core itself is composed of internal registers, also an interruption handler and a debug interface are present but they are not represented in the figure 4.

The second element of the CPU is the memory interface which is composed of a MPU and a bus interface making the link between the CPU and the interconnection bus.

2.2.3 *Packaging*

The packaging of a SE is very depending of its usage as it defines the form factor of the token it will power. However, most of the time, the SE is embedded in a smartcard as shown in figure 5.

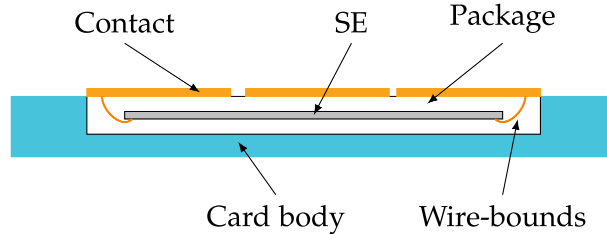


Figure 5: Secure element packaging

This packaging is simple: the SE only presents the Inputs/Outputs (IOs) corresponding to the ISO7816-3 protocol and these IOs are connected to the contacts of the card via wire-bounds. The corresponding pins are shown in figure 6.

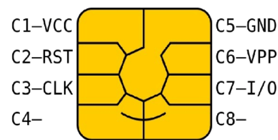


Figure 6: ISO7816 pins

The smartcard connection is composed of eight pins. They are not all used. The `VCC` pin is connected to the power supply. Indeed, smartcard are powered *via* an external source of energy. The same way the `CLK` pin provides the clock and the `GND` is the ground. The card can be reset using the `RST` pin and all the wired communications go through the `I/O` pin. The `VPP` entry is not used anymore and is also named `SPU` (for Standard or Proprietary Used), however, its usage is not specified.

Smartcards rely on external sources of power and clock to operate. This specificity can be used by an attacker to insert glitches in the card. We will discuss this kind of attacks in chapter 4. However, to prevent them, recent implementations integrate internal clocks and power management integrated circuits..

2.3 SYSTEMS ON CHIP

SoCs are a recent kind of chips which aim at providing the same services as a complete computer but integrated in a single chip. The idea is to integrate them in mobile and constraint devices such as smartphones or Internet of Things (IoT) devices. The first popular device of this kind was the iPhone which has been commercialized by Apple in 2007.

From this moment, the importance of smartphones in our society grew to become a part of citizens daily life. With this growth and the development of technologies, in particular the IoT which aims

at connecting everything and everyone, a lot of services, sometime sensitive ones, have been added to these devices. The consequence is that smartphones are interacting with a lot of elements as shown in figure 7.

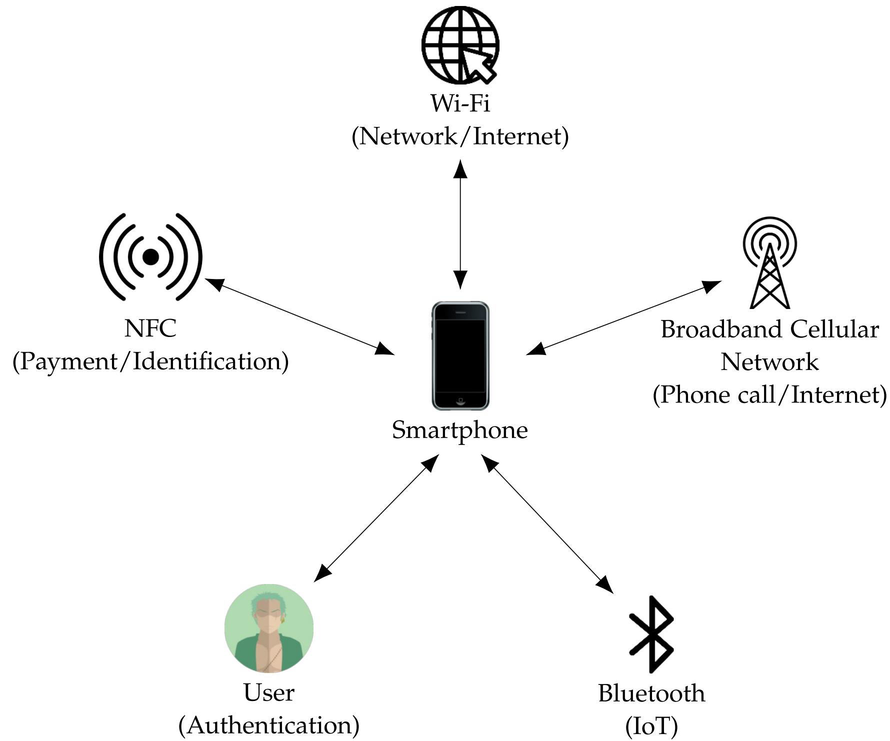


Figure 7: Smartphone environment

This figure shows all the interactions a modern smartphone can manage. It must be able to authenticate its user, using a PIN code, a password or even biometric identification. It can be connected with various IoT devices. This requires a lot of security as it can be connected to home systems such as smart door locks for instance. It handles phone call and therefore must secure the communications on the Global System for Mobile communication (GSM) network. It can identify itself and realizes payments via its Near Field Communication (NFC) interface. And, last but not least, it is connected to the internet and provide various applications such as banking, insurance, taxes management, social networks, video streaming, *etc*.

Presenting all the possible use cases of such a device is not the aim of this section, however, one can easily see that this kind of device can be used for many things and sometimes, critical ones. For instance, the smartphone can replace the smartcard in figure 2. It communicates with the reader via NFC, the reader can authenticate the smartphone (or at least the application running on the smartphone), then the smartphone authenticates the user as shown in figure 7 and it can finally sign the transaction.

However, despite this use case is possible, some questions remains:

- how much the SoC powering the smartphone can be trusted to protect the keys it stores compared to a SE? This question assesses the lack of security evaluation on SoCs.

- Therefore, does the reader authenticate the smartphone itself or the application? This is very different from the smartcard case where the reader authenticates the card which authenticates the executed application. Authenticating an application running on a not trusted platform might not give the expected level of security without further study.

- And finally, if the reader only authenticate the application, what are the consequences in terms of security?

This is a quick analysis regarding a specific and well know use case already managed with SEs, however one can see that such a complete and complex device can raise many security concerns.

To answers these questions, it is important to deeply analyze how the SoCs powering such devices are made and how much do they differ from SEs.

2.3.1 *Architecture*

A SoC architecture is quite similar to SEs one on the principle but differ on the number of integrated modules, on the complexity of their interfaces and on the software architecture. The figure 8 shows the general architecture of such device. This is a simplified model but one must know that every SoC differs and can integrate several modules in addition to those presented in this figure.

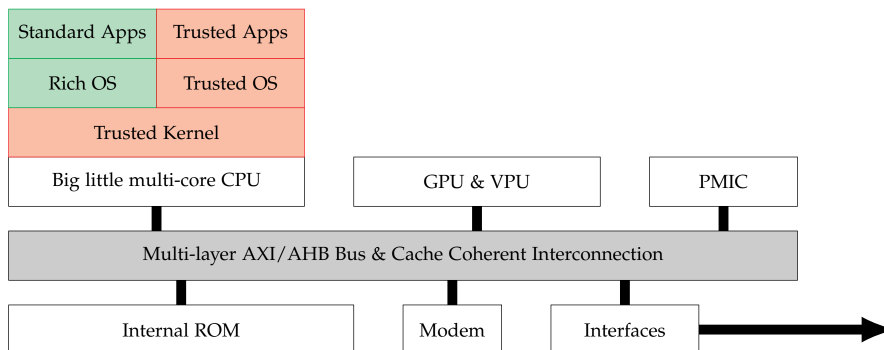


Figure 8: System on Chip architecture

In terms of hardware, these devices are always composed of a complete CPU with a big little architecture. The big little architecture consists in mixing the cores integrated in the CPU. For instance, in a eight cores CPU, four of them are "little" cores which are a bit slow but does not consume a lot of energy while the other four cores are "big" ones which are used when the device needs an important computation power whereas they have a more important energy consumption. This able SoCs to adapt their performances regarding their need in computation power and to save energy.

This is why, the CPU also integrates a Power Management Integrated Circuit (PMIC). This circuit is dedicated to the energy management and supply the voltage power and clock frequency to every other modules in the SoC. It allows a real time energy management.

For parallel computation and the video processing, the chip integrate a Graphics Processing Unit (GPU) and a Video Processing Unit (VPU). The GPU is usually the bigger module as it can embed eight to sixteen cores. In the same way, more and more devices are integrating Neural Processing Units (NPUs) for neural networks computation.

As shown in figure 7, SoCs have multiple interfaces. The wireless ones, *i.e.* the Bluetooth, the broadband cellular network and the Wi-Fi are managed by a modem. A chip is dedicated to NFC communication. Also, there are many wired connections, to communicate with the external memory, the other components on the motherboard, or for debug purposes *via* the Android Debug Bridge (ADB) for instance.

In terms of memory, such device only embed a ROM, which usually is a flash memory. Their RAM is an external memory. The reason is that these devices usually work with 2 GB to 16 GB of memory. Memories with such storing capacity require a lot of space and therefore are integrated in a dedicated chip.

For the connection of the modules, the chip integrates a multi-layer interconnection bus. Usually an ARM AXI or AHB bus. Also, this bus must handle the cache coherency, as almost every module (at least the CPU and the GPU) have a cache memory. This cache memory mirrors a subpart of the RAM in the modules, therefore, the interconnection bus must ensure that all the data in the caches of all the modules are updated as shown in figure 9.

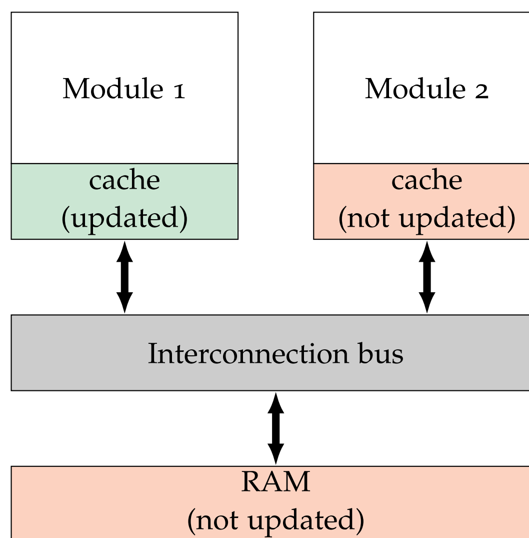


Figure 9: Cache incoherence after a module updated a data in its dedicated cache memory

This figure shows a situation where the module 1 just updated a data in the memory. So the data is modified in its dedicated cache, however, this modification was not propagated to the RAM or the other modules cache yet. In this kind of situation, the module 2 can access the same data, however, it is not updated in its cache, creating an incoherence. It is one of the interconnection bus roles to ensure that the data the module 2 wants to read is correctly updated in its cache.

Regarding the software part, SoCs aim at providing a secure architecture based on Trusted Execution Environments (TEEs). TEE are specific OSs which are trusted and can only execute signed and trusted applications. Along them, there is a richer OS (usually Linux, Android or iOS) which is less trusted but can load and execute any application. In practice, the TEE executes in a specific mode of a CPU core. In other words, there is a core which can switch from the normal mode to the secure mode and the trusted OS only executes on this core in secure mode. The consequence of this architecture is that, compared with SEs, the SoCs CPU handles all the critical operations and the execution of the TEE, making it a critical component for SoCs security.

2.3.2 *CPU*

CPUs integrated in SoCs are considered as modern CPUs. They are designed for to give the highest computational power while reducing their energy consumption. In these purposes, they implement a complex architecture with a lot of optimizations, such as the cache memory. This section will present their general architecture and some optimization mechanisms which make them very different from SEs CPUs.

2.3.2.1 *Architecture*

The architecture of a modern CPU is quite similar to the SE one on the concept but very different in the realization as presented in figure 10.

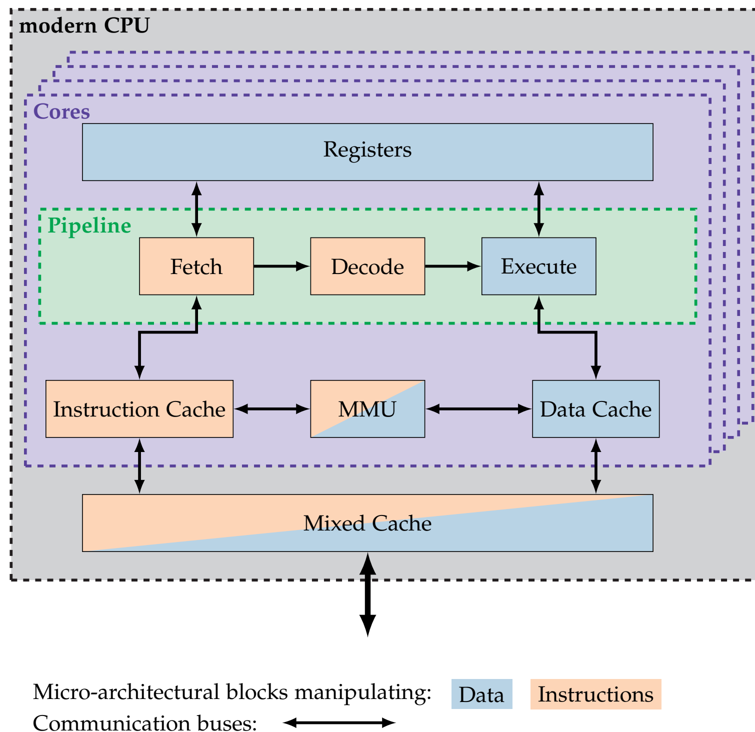


Figure 10: Modern CPU architecture

It is composed of several (between two and height) cores. The pipeline they implement can vary from a design to another, however, the three main functions (fetch, decode and execute) are always

present. The pipeline implementation varies more on the optimizations it integrates.

Along the pipeline, the cores have registers and an internal cache memory (also call L1 cache). The cores are designed with an Harvard architecture: the instructions and the data are stored in different cache memories and are carried on different buses. The last element is the Memory Management Unit (MMU), which provides a virtualization of the memory in addition to the memory protection usually provided by a MPU. This virtualization helps the kernel to optimize the usage of the RAM. Of course, the cores also integrate an interruption handler and a debug interface that are not represented in the figure 10.

Outside the cores, there is the second level of cache (L2 cache). This cache stores both the instructions and the data, which corresponds to a Von Neumann architecture. A modern CPU can therefore be considered with a mixed architecture. Also, the L2 cache is connected to the interconnection bus for external communication.

Along all these elements, modern CPUs usually implement optimization mechanisms which aim at increasing its computational power. These optimizations are important to take into account as they can raise security concerns [3].

2.3.2.2 *Optimizations*

Modern CPUs optimizations are an important feature as they improve the device average execution time of a program by reducing memory access time, anticipating instruction execution, *etc*. However, they can raise some security concerns which were highlighted with the Spectre [4] and Meltdown [3] attacks.

CACHE MEMORY. The cache memory is an optimization mechanism which aims at reducing the memory access time. As mentioned above, cache memories are integrated in the computational units and mirror a subpart of the RAM. Therefore, when a memory access is done and the data is in the cache, the access time is slightly reduced compared with an access to the RAM. However, this mechanism adds some problems like the cache coherency presented in figure 9.

OUT-OF-ORDER EXECUTION. The out-of-order execution was presented in 1967 [5]. This optimization mechanism aims at optimizing the usage of Arithmetical and Logical Units (ALUs) of cores. The idea is to fetch several instructions, separate them depending on their data dependency, execute all of them at the same time on the different ALUs, store the result in shadow registers and copy these shadow registers in the regular registers sequentially as shown in figure 11. Cores with such optimization are named hyperscalar cores.

The figure 11 shows the principle of the out-of-order execution. The instructions are fetched (1), then an organization buffer splits them depending if there is data dependencies between them (2), creating packs of instructions. These packs are queued in the instructions handler which distribute them in the different ALUs when they are

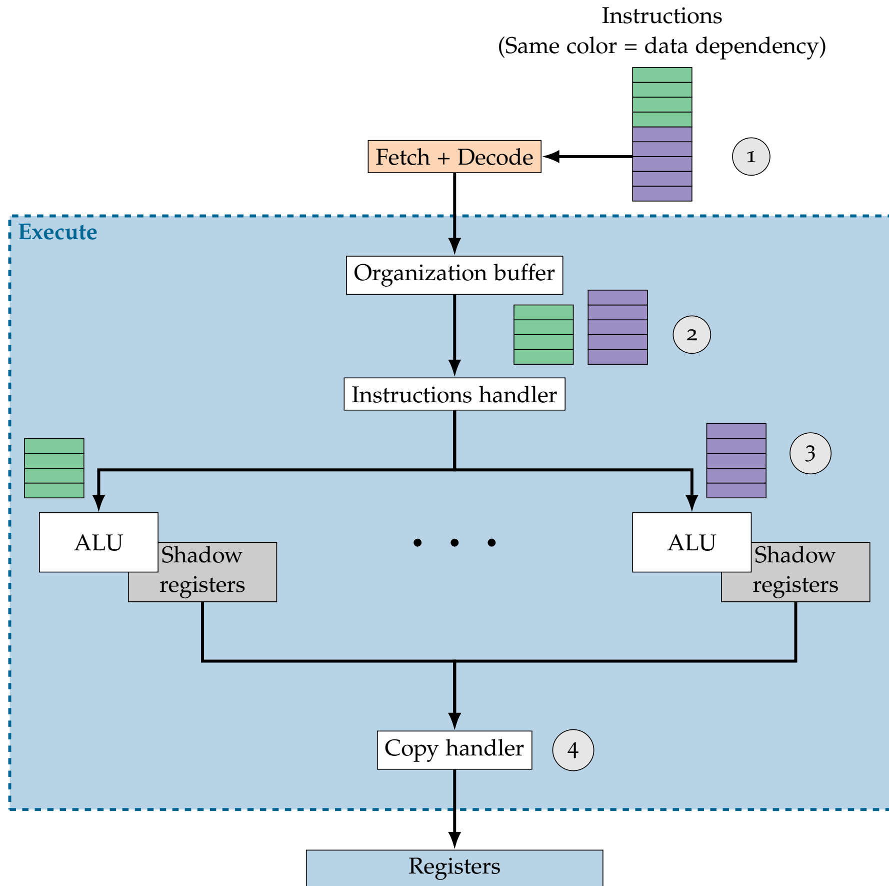


Figure 11: Out-of-order execution principle

available (3). Every ALU works as a classical one except that it does not interact with the core registers but a copy of them named shadow registers. Finally, when the computation is over, the copy handler will update the CPU registers with the shadow registers (4).

In practice, the ALUs used in the execute stage are different, there are some dedicated to logical and arithmetical operations, some for floating point arithmetic, some for memory access, *etc*. In the end, the organization buffer and the instruction must also organize the execution regarding the available ALUs. Usually, a modern core has between three and nine concurrent ALUs.

The consequence of this optimization is that instructions are not executed sequentially but in parallel, only their result is updated sequentially keeping the Instruction Set Architecture (ISA) abstraction.

BRANCH PREDICTION. The branch prediction aims at avoiding the pipeline clear when it is filled with instructions that are not supposed to be executed. This can happen when there is a conditional jump in the program, the pipeline fetches the instructions sequentially, however, when a branch instruction is executed, the target instructions might be not the ones directly following the branch. In this situation, the pipeline must be emptied to avoid the execution of non desired instructions and refilled with the correct ones.

A solution to reduce the time loss due to the empty and refill of the pipeline is to anticipate the instructions targeted by a branch and to fetch them. This anticipation is named branch prediction and is integrated in every modern core.

Combined with out-of-order execution, the branch prediction leads to speculative execution. Speculative execution is the anticipate computation of instructions. Indeed, as the out-of-order execution allows the parallel execution of instructions, sometimes the test condition of a branch and its possible paths are executed at the same time. In this case, the paths are executed speculatively and only the semantically correct one is kept after the test condition.

As mentioned with Spectre and Meltdown [3, 4] speculative execution recently arose security concerns as in critical softwares some paths that should not be executed unless the test condition say so can leak information *via* side-channels.

2.3.3 *Packaging*

The last difference to consider between SEs and SoCs is the packaging. As the SoCs are usually embedded in mobile devices and are larger than SEs, their packaging is designed to be as compact as possible. Also, as mentioned above, they do not integrate a RAM which is external but must be as close as possible to the chip to reduce memory accesses delay.

All these constraints led to a new kind of packaging presented in figure 12: the package on package.

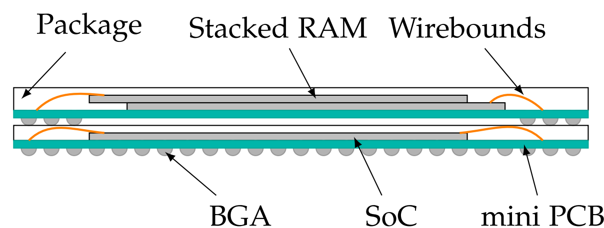


Figure 12: Package on package

The package on package design is organized as two chips stacked one above the other. Usually the bottom chip is composed of the SoC and the above chip embeds the RAM. Therefore, the surface of the whole system is minimized while the RAM remains close to the SoC.

In terms of connection, two types are used, a mini Printed Circuit Board (PCB) or wirebounds, both route the output of the chip to a Ball Grid Array (BGA). This BGA corresponds to the connections that are soldered on the motherboard. The package containing the RAM is also soldered to the SoC via a BGA connection.

Despite being very compact, BGAs present a major drawback which is the reworking. Indeed, chips using BGAs usually output hundreds of connections as shown in figure 13.

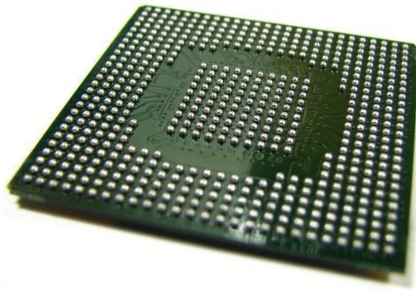


Figure 13: BGA grid on a PCIe chip

The consequence is that un-soldering and re-soldering such chip is a complicated task which requires a dedicated machine and a high level of knowledge. Moreover, the layout is not standard and therefore being able to connect to or test a specific IO is very time consuming with a high risk of breaking the device connections.

The last consequence of this packaging is that the SoC is completely enclosed in a complex package, with the RAM on the top. Therefore, accessing it is complicated and regarding security concerns, some attacks require a physical access to the chip.

Considering this kind of attacks such package seems to harden them. However, despite the chip needs a RAM to work properly, such devices can partially run without it. The reason is that the RAM needs to be initialized and therefore the program doing it does not work with the RAM but with the cache memory. This program is critical as it is involved in the secure boot (presented in the section 4.4.1.3) of the chip and can therefore be targeted even if the RAM is removed, freeing the access to the die.

2.4 MULTI-APPLICATION SYSTEM SECURITY

Before looking at a system security, it is important to understand on which mechanisms the security rely, in particular considering multi-application systems.

A system is the union of a device with a software layout (usually an OS). A multi-application system is a system that is able to execute simultaneously multiple applications by ensuring the following properties:

- **application integrity**: this property ensures the executed applications are not corrupted during their loading.
- **application correct execution**: this property ensures that no corruption occurs during the execution of an application.
- **application data confidentiality**: this property ensures that the data belonging to an application are not manipulable (*i.e.* readable, writable nor executable) by another application executed on the system.

These security properties ensure that every application will be executed as expected and with a protected and dedicated memory de-

spite they physically share one. They rely on different security mechanisms which are the following:

- **memory partitioning**: this mechanism allows the system to dedicate a part of the memory for every application. Applications cannot access to a memory part which does not belong to them.
- **secure boot**: this mechanism allows the device to check that every piece of software composing the system are the legitimate ones.
- **cryptography**: this mechanism allows the system to ensure the authenticity and confidentiality of data.

These mechanisms are necessary to have a trusted system. A part of the device, named root of trust, is *de facto* trusted by design and will ensure *via* cryptography mechanisms and the secure boot that the OS can be trusted. Once the OS is trusted, it will setup the memory partitioning. However, all these mechanisms must also be trusted and therefore they rely on critical mechanisms of the device which are the following:

- **execution flow**: this mechanism ensures that every steps of an implementation are correctly executed and in the right order.
- **data integrity**: this mechanism ensures that there is no error in the reading or writing of data in the device or its memory.

These mechanisms are what a device must warranty for its correct operation despite security concerns. However, these mechanisms are the foundations for any security features the system must provide. The figure 14 shows the dependencies between all these features.

In a device, the base of the security is the data integrity, which is (in this case) not a security feature but a required property for a device to behave correctly. If the data integrity is granted, then it is possible to have a correct execution flow because this mechanism is requires the correctness of the instruction pointer and, in some cases, other data.

With a device ensuring data integrity and the correct execution flow, it is possible to implement algorithms, in particular ones dedicated to security. The base of the security mechanisms is the cryptography. It can either be implemented in hardware or in software and they rely on a secret key that must be securely stored, in the root of trust. When the cryptography is implemented and we have a root of trust, it is possible to implement a secure boot. The secure boot will propagate the trust from the root of trust to the OS powering the device.

On multi-application systems, the OS will configure the memory management mechanism (MPU or MMU) of the device to ensure the memory partitioning. These micro-architectural elements might be faulted but, no previous published work demonstrated its feasibility. From this step, applications can be executed by the system and all the three security properties will be ensured.

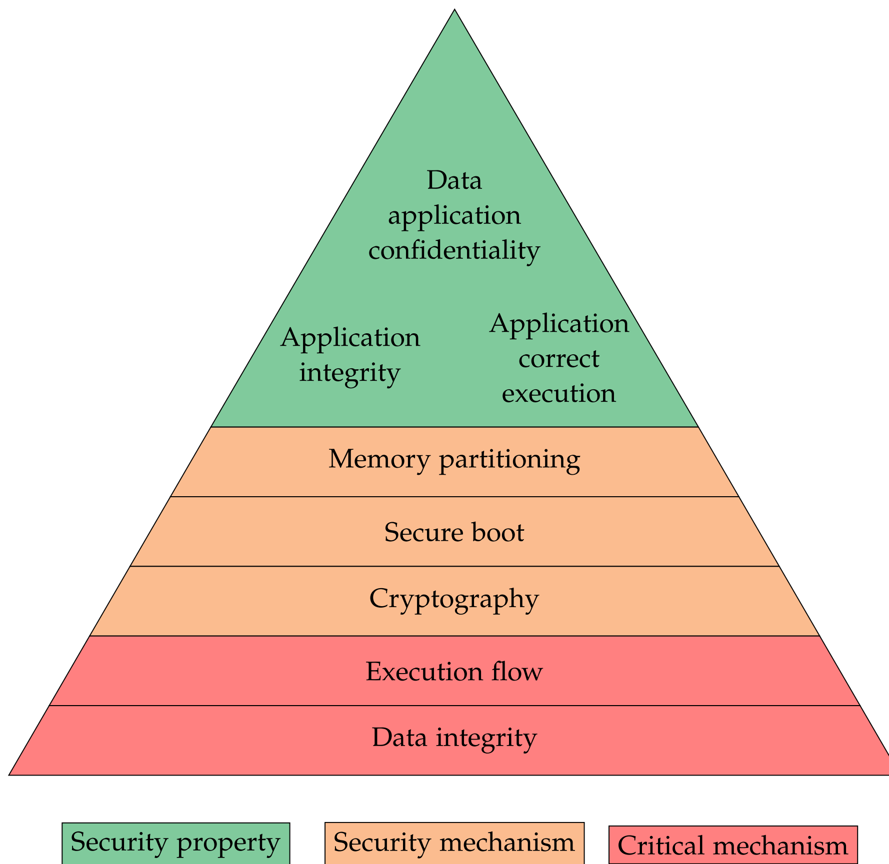


Figure 14: Multi-application system security dependencies

2.5 CONCLUSION

SEs are the historical and most used devices for security applications and sensitive operations. However, new devices are more and more used to fill this role: the SoCs.

SoCs are very versatile, multi-applications, multi-interfaces and performance oriented devices. Due to their versatility, they are used (among other) for sensitive applications such as identification, payment, *etc*.

However, their youth and their complexity compared with SEs rise questions about their security level. Indeed, their complex architecture, their multiple use cases, their ability to execute un-trusted applications and their permanent connection make their security evaluation a complicated task. Moreover, as mentioned above, they are designed for optimizing their computational power and energy consumption not for ensuring the same security level than SEs.

Regarding these differences, and because SoCs are more and more present in our society, this thesis focuses on evaluating the security of such devices.

3 PHYSICAL ATTACKS

You don't stand a single chance to win unless you fight.

— Mikasa Ackerman (Attack On Titan)

ABSTRACT

This chapter presents a brief state of the art of physical attacks. It mainly focus on side channel attacks and invasive techniques. Regarding the perturbation attacks, a they are a important part of this thesis work, they are detailed in chapter 4.

Contents

3.1 Side-channel attacks **26**
- 3.1.1 Micro-Architectural Attacks 28
- 3.1.2 Side-Channel countermeasures 28

3.2 Invasive attacks **28**
- 3.2.1 Reverse engineering 29
- 3.2.2 Focused Ion Beam 29

In cybersecurity, physical attacks are a special kind of attacks which consists in using the physical environment of the target to either retrieve information or break security mechanisms. All these attacks are tested during a secure element evaluation and they are divided in different categories: The Side-Channel Analysis/Attacks (SCAs), presented in section 3.1, aim at retrieving information about a device behavior or its secrets by observing physical values that may be correlated with this information. The perturbation attacks, presented in chapter 4, which aim at producing non-definitive bugs in the device to lower its security. And finally, the invasive attacks, presented in section 3.2, are aiming at modifying a circuit to either help its analysis or perturbation. These attacks can also be used to reverse the device layout, *i.e.* determining the layout with all layers and connections from the device itself.

3.1 SIDE-CHANNEL ATTACKS

SCAs exploit information leakage during an algorithm execution. The leakage comes from the algorithm implementation and was usually not considered during the algorithm design. For this reason, these attacks are sometimes called implementation attacks. However, because these attacks are powerful and model leakages [6] have been proposed, SCAs are more and more considered in the design phase of algorithms, in particular for cryptographic applications.

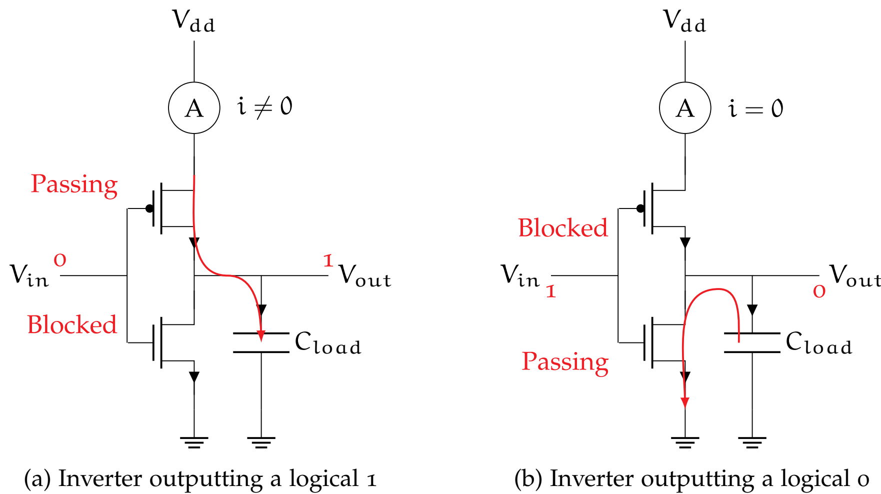
The first public work about SCA was presented in 1996 [7] and is about observing the execution time of cryptographic implementations to recover their manipulated secrets. Even if execution time is an old leakage source and one of the simplest to exploit, it's also very hard to develop a constant time application if the device executing it is not specifically developed for this purpose or the developers make a mistake. This has been demonstrated with various attacks, especially recent ones targeting Trusted Platform Modules (TPMs) [8], modern processors [3, 4] or the Secure Channel Protocol (SCP)02 [9] widely used in smartcards. Actually, implementing secure-to-timing-analysis applications is so complicated, especially in a performance oriented system, that a whole research field called Micro-Architectural Attacks is dedicated to identify and protect against them on modern computers. The most famous attacks of this kind are Spectre [4] and Meltdown [3] which were presented in 2018 and started the discovery of several CPUs vulnerabilities [10, 11, 12].

After the first timing attack, several works were done to improve the efficiency of SCAs. The Differential Power Analysis (DPA) was introduced in 1999 [13]. This method consists in comparing the power consumption of a device manipulating a secret with simulated traces for different values of this secret. Finding minimal distance between the real trace and the simulated ones allows to recover the secret.

DPAs were improved in 2002 [14] with the introduction of template attacks. A template attack consists in profiling a fully mastered copy of the target device to build a profile which can be used to attack the

real target using only few traces. This method was, for instance, used to break a 3DES smartcard implementation [15].

In 2004, the Correlation Power Analysis (CPA) was introduced [6] with the concept of leakage model. The CPA remains one of the most famous attacks method and was successfully used to attack various systems like smartcards [16], smartphones [17] or IoT devices [18]. The core idea is to observe the physical leakage as a function of the manipulated secret. The first used leakage models are the hamming weight and the hamming distance. These models suit well the power consumption and electromagnetic emanations because, in the CMOS technology, the power consumption is directly linked to the logical gates output as presented in figure 15.



(a) Inverter outputting a logical 1

(b) Inverter outputting a logical 0

Figure 15: Current in a CMOS logical inverter gate for different inputs.

Figure 15 shows that depending on the output of a logical inverter, there is a current which is pulled or not from v_{dd}. This current is used to charge a C_{load} capacity which models the transistors inherent capacity and commutation time. The consequence is that the power consumption $V_{dd} \times i$ is directly linked to the manipulated data. The principle is the same for electromagnetic emanations as they are proportional to the derivative of the current and this remains the case for any kind of CMOS logical gate.

Later on, several improvements to SCAs were proposed with the Mutual Information Analysis (MIA), the Linear Regression Attack (LRA), the modeling of physical leakage and the usage of information theory [19, 20].

Finally, the last important step for SCAs was the introduction of the usage of deep learning algorithms for improving template attacks [21]. Despite these attacks are quite young, they have already been used to successfully break Rivest, Shamir, and Adelman (RSA) [22], AES [23] and Elliptic Curve Digital Signature Algorithm (ECDSA) [24] implementations.

3.1.1 *Micro-Architectural Attacks*

Micro-Architectural Attacks are a special kind of SCAs which focus on finding timing leakage in modern CPU architectures. These leakages can be useful for various things: CPU architecture reverse engineering [25], key extraction and data spying [12] or unauthorized communication (also know as covert-channels) [26].

The first works about side-channel micro-architectural attacks were focusing on how the timing difference due to the cache memory can leak information about executed cryptographic algorithms [27]. However, to reach higher and higher performances, manufacturers implemented several optimization mechanisms which introduce timing leakage at various level of the micro-architecture. The most famous are the speculative execution [28], the out-of-order execution [29, 30], the return stack buffers [31] and the instruction prefetching [32]. Even if these attacks are architecture specific and complicated to achieve because they mostly rely on race conditions, they also present the threat to be achievable over the network [33].

3.1.2 *Side-Channel countermeasures*

There are two principal ways to protect against SCAs; avoiding leakages and hiding information. The best case is to completely avoid leakages from the implementation, however, this is nearly infeasible for power consumption and electromagnetic emanations. For timing leakage, nowadays every cryptographic algorithm must have a constant time implementation to be secure against them. This presents drawbacks in terms of execution speed.

In the cases leakages can't be avoid, the defense strategy consists in hiding the relevant information. Leakage channels are naturally noised but not enough to prevent effective correlations. Therefore, the idea consists in adding noise by generating randomness and masking the relevant information with it. There are several masking methods but it is interesting to note that some of them are proven secure against a reasonable level of SCAs [34].

3.2 INVASIVE ATTACKS

Invasive attacks are a kind of attack that aim at modifying the circuit in a irreversible way to ease its observation or perturbation. Usually, invasive attacks consists in opening the device to expose the die. This reduces the noise in the leakage and the package absorption while doing a perturbation or an observation. More details about this kind of attack is given in the section about optical perturbations (section 4.2.6) as these attacks need an unobstructed access to the die.

3.2.1 *Reverse engineering*

A interesting invasive attack category consists in reverse engineering devices. Indeed, as the devices are built with layers (an active layer with transistors and several connection layers). Being able to have an imaging of the different layers able to build the device schematic from its die. This is powerful for analyzing an implementation or for reading hard-coded values. This technique is named the de-layering. However, its main drawback is that this technique completely destroy the target circuit.

To protect devices against this kind of attacks, the most used solution is to obfuscate the hardware. There are several techniques which can operate at different level of the circuit design process. First techniques were working directly on the netlist [35, 36], then several techniques appeared to work on the different descriptions of a circuit: at the Register Transfert Level (RTL) [37], at the logic level [38, 39] and directly on the layout [40]. Also, obfuscation techniques may rely on Physically Unclonable Functions (PUFs) [41].

3.2.2 *Focused Ion Beam*

Focused Ion Beams (FIBs) are a tool used by circuit manufacturers to dope, repair and debug circuits among other usages [42]. As their name suggests it, they throw ions to a semiconductor and modify its structure. This able to create, destroy wires in the circuit or change the doping of a semi-conductor.

In terms of security, as this able to modify a circuit in a permanent way, it is possible increase the leakage of certain data, modify a component behavior (like avoiding a counter to increment) or deactivate a countermeasure. Despite, the use of a FIB is expensive as it requires both the equipment and a dedicated expert, being able to modify a circuit is a strong hypothesis regarding an attacker.

To protect against these attacks, manufacturers integrate shields and sensors in their devices. These protections aim at detecting any intrusion or modification on the die. However, they only work when the device is turned on, which means that while the attacker keep the device off it can do any modification on the device. More detail about them is given in section 4.5.5 as they are also used for protecting against perturbations, which require the device to be turned on when being used.

4 PERTURBATION ATTACKS

Laziness is the mother of all bad habits.
But ultimately she is a mother and we should respect her.

— Nara Shikamaru (Naruto)

ABSTRACT

This chapter shows an overview of perturbation attacks. From their genesis, it describe a large overview of research topics related to this problematic: the injection mediums, the fault models, the fault analysis, some famous attacks using faults and the countermeasures.

Contents

4.1 Genesis 33
4.2 Inducing a fault: injection mediums 34
4.2.1 Design considerations 34
4.2.2 Clock glitches 36
4.2.3 Voltage glitches 36
4.2.4 Temperature manipulation 37
4.2.5 Electromagnetic perturbations 37
4.2.6 Optical perturbations 39
4.2.7 Body biasing 41
4.2.8 X-Rays 41
4.2.9 Software induced perturbations 41
4.3 Characterizing a fault 44
4.3.1 Fault models 44
4.3.2 Fault analysis 46
4.4 Exploiting a fault 47
4.4.1 Fault attacks against multi-application systems 47
4.5 Countermeasures 56
4.5.1 Space redundancy 57
4.5.2 Operation duplication 57
4.5.3 Infection countermeasure 58
4.5.4 Code hardening 58
4.5.5 Sensors 60
4.6 Conclusion on perturbation attacks 60

Perturbation attacks intend to stress a target device by pushing it beyond its nominal execution point to force unintended behavior. In some cases, such behavior can create security breaches in the system. Usually, knowing that a perturbation able to break a security feature is enough for most attacks. However, having a model of the possible perturbations, understand their impact on the hardware and on the software, *etc*, is important to build efficient countermeasures and evaluate the impact of such attacks.

For these reasons, perturbation attacks became an important research field in hardware security and their analysis and evaluation on devices is a time consuming and complex process summarized in figure 16.

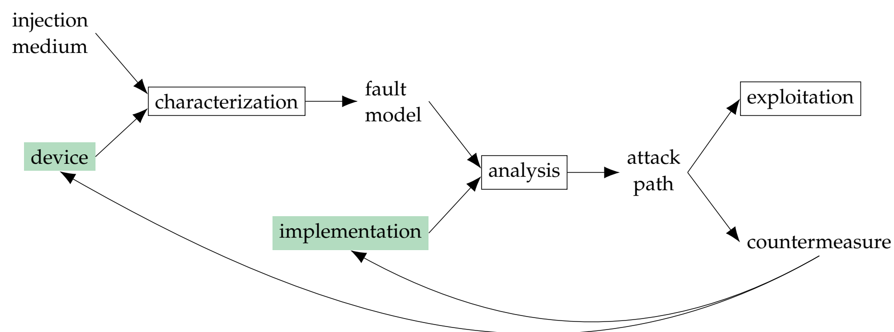


Figure 16: Perturbation attack analysis and evaluation process

This figure presents the analysis and evaluation process for a system against Fault Attacks (FAs). The aim is to build a system composed of a device and implementations resistant to FAs. To assess this resistance several steps are needed.

The first step is the fault characterization, it consists in determining in which ways the device is perturbed regarding different injection mediums. These mediums are the different techniques that can be used to induce a fault in the device and are presented in section 4.2. The result of this characterization is the fault model which is a representation of the fault effects on the device. There are numerous fault models and being able to determine it may be a complex process, this is presented in section 4.3.1.

The next step is the implementation analysis regarding the fault model. This consists in simulating or emulating the possible fault models determined on the device on an implementation as presented in section 4.3.2. As many fault models can be considered and regarding the complexity of the implementation, testing every fault at any stage of the implementation is too time consuming, making this step very tricky, in particular on complex implementations. The result is an attack path which can be exploited for confirming its relevance. The exploitation allows to evaluate the difficulty of doing the attack and to measure its impact. Some of them are presented in section 4.4 with a presentation of a system security model.

The last step is building efficient countermeasures. Regarding the injection medium, the fault model or the implementation different countermeasures can be adopted and integrated to the device or the implementation. Some FA countermeasures are presented in section 4.5. Once a countermeasure is designed and integrated in the system, a new characterization and/or analysis can be done to evaluate its efficiency. This loop can be repeated as many times as needed to reach the desired security level.

This chapter will present an overview of the public work about perturbation attacks. It aims at providing a clear and large overview of the challenges the securing of systems highlight and how they are settled by the community.

4.1 GENESIS

Despite bugs were known to appear in digital devices, the first public observation of a perturbed device was due to an unintentional perturbation on a satellite in 1975 [43]. In this case, memory errors were observed due to "galactic cosmic rays".

From this observation and in order to prevent perturbations on in-production satellites, a lot of characterization work on circuit has been done to characterize and prevent the effects of cosmic rays in Metal Oxide Semiconductor (MOS) memory cells [44], Very Large Scale Integration (VLSI) circuits [45] and Dynamic RAMs (DRAMs) [46].

The functional analysis of digital devices kept going by studying the impact of different perturbations. The effects of high temperature [47], alpha-particules [48], heavy ions [49, 50] and lasers [51, 52] were analyzed to simulate the space environment on semiconductor devices.

In parallel of device safety, cryptographic researcher teams, aware of the presence of faults in devices, were working on cryptanalysing algorithms in the presence of faults. The seminal work was published in a memo about the RSA signature generation [53].

Later on, a lot of algorithms were analyzed: the RSA and Rabin signature [54], secret key algorithms like Data Encryption Standard (DES) [55], public key signature algorithms (RSA, ElGamal, Schnorr and DSA) [56] and Elliptic Curve Cryptography (ECC) [57].

At this point, around 2000's, the perturbation of devices was done using various mediums and cryptographic attacks based on faults were only theoretical and these two disciplines did not already match, at least publicly.

In the 90's, the French security agency, the *Service Central de la Sécurité des Systèmes d'Information (SCSSI)* and the French national telecommunication center, the *Centre National d'Étude des Télécommunications (CNET)* were already evaluating secure devices against FAs, mostly using light and lasers.

The first public work was published in 2002 with Skorobogatov [58]. In his work, he presents the usage of a laser, usually used to simulate

cosmic ray, to intentionally perturb a Micro-Controller Unit (MCU). In 2005, in his thesis, he introduces the semi-invasive attacks [59].

From this moment, research on FA was focused on several points: the analysis of various injection mediums, the exploitation of faults, the simulation of faults in circuits and programs and the building of countermeasures.

4.2 INDUCING A FAULT: INJECTION MEDIUMS

Perturbation attacks rely on the apparition of faults during a system execution. During an attack, these faults are intentionally created within the device. To behave correctly, digital devices have to fulfill physical constraints. These constraints are directly linked to their design and, if they are respected in their nominal operation point, it is possible to break these constraints conditions by inducing *stimuli* in the device as presented in figure 17.

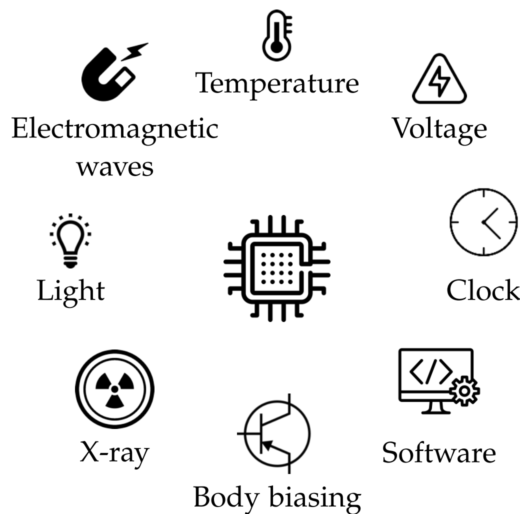


Figure 17: *Stimuli* able to perturb a digital device

Most of these *stimuli* are physical like the voltage, the temperature, the EM waves, the light, the X-rays, the clock and the body biasing. They are used to push some components of a device at their limit of operation, inducing faults in them.

While perturbing a device, three behaviors can create a fault, a timing violation [60], a sampling error [61] or a transient current [62]. This section presents the design of a synchronous circuit and how these injection mediums actually induce faults in them.

4.2.1 *Design considerations*

Before understanding how faults are induced into a device, it is important to understand how they are designed and how this design is constrained.

Timing faults comes from the instability of DFFs, also known as registers. Indeed, in synchronous circuits, registers sample the output data of the logic synchronized with the chip frequency. But to sample

data correctly, two constraints must be respected: the input data of the DFF must be stable for at least t_{setup} seconds before the clock rising edge and t_{hold} seconds after the clock rising edge as shown on figure 18.

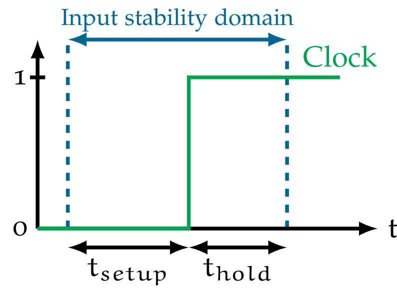


Figure 18: Timing constraints on the input for a DFF to behave correctly

These constraints impact circuit designs as the propagation time of the logic between registers must fit with these timings as shown on figure 19.

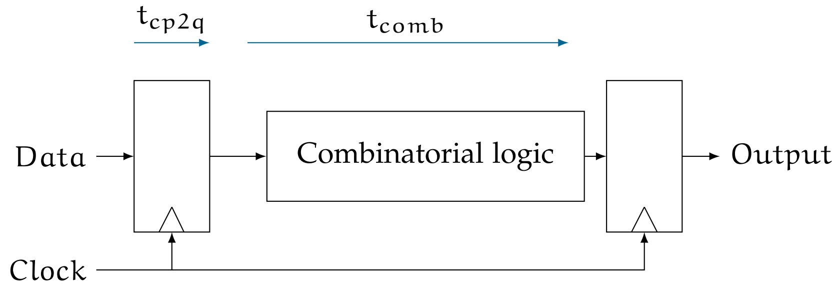


Figure 19: Timing constraint in digital devices

This figure shows how digital devices are designed. The function of the device is decomposed in small logical segments placed between two DFFs. When a clock rising edge happens, the data in the DFFs are output in the logic after t_{cp2q} seconds, then hold for the rest of the clock cycle. The signal propagates through the logic during t_{comb} seconds before reaching the second DFF. Regarding this design and the constraints on the DFFs, the clock frequency of a device must fill the constraint presented in equation (1). It summarizes the fact

$$t_{clock} \geqslant t_{cp2q} + t_{comb} + t_{setup} \tag{1}$$

Equation: Setup timing constraint on clock cycle in a digital device.

that the clock cycle t_{clock} must be high enough so the setup timing is respected. This is done by determining the logical path between two DFFs with the highest propagation time t_{comb}. This path is named the critical path. In practice, it is that critical path that constraints the clock cycle t_{cycle} and is the most sensitive to faults.

Regarding the hold timing, the constraint is that the input of the second DFF must remain stable for at least t_{hold} seconds. Giving the constraint presented in equation (2).

$$t_{cp2q} + t_{comb} \geqslant t_{hold} \quad (2)$$

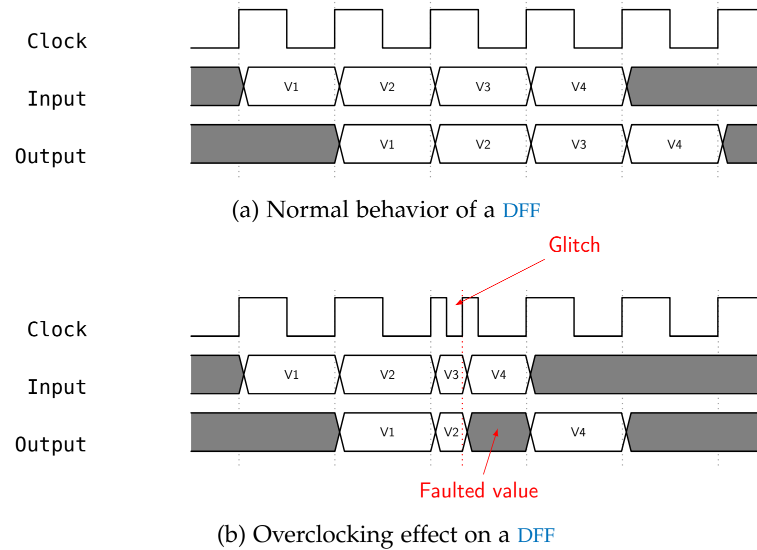
Equation: Hold timing constraint on clock cycle in a digital device.

This equation summarizes that the propagation in the logical cannot be too fast otherwise the hold timing will not be respected.

The non-respect of the setup or hold constraints is a well known source of bugs, in particular due to variations in the manufacturing process of devices which can modify the critical path [63].

4.2.2 *Clock glitches*

Clock glitches are local variation of the target device clock frequency. They aim at breaking the constraint presented in equation (1) and therefore introduce a fault in DFFs. This behavior is presented in figure 20.



(a) Normal behavior of a DFF

(b) Overclocking effect on a DFF

Figure 20: Clock glitch effect on a DFF

This figure shows how a setup timing violation can be done using a clock glitch. Due to this glitch, the value V3 is not held long enough to ensure that the value is correctly sampled by the DFF.

This injection medium was used to attack smartcards [64] as they are powered by an external clock. It was also recently used in a special kind of perturbation attack named *ClkScrew* [65] in which the SoC corrupted PMIC intentionally modifies the cores frequency on the fly.

4.2.3 *Voltage glitches*

Voltage glitches intent to create timing errors by modifying the propagation time in the circuit logic. Indeed, regarding equations (1) and (2) increasing or reducing the logic propagation time t_{comb} can induce either a setup timing violation or a hold timing violation. Overpow-

ering reduces the propagation time and underpowering increases the propagation time in standard cells [66].

This medium is used for attacking devices with external power supply such as smartcards [67] or IoT devices [68].

4.2.4 *Temperature manipulation*

Overheating works in a similar way as voltage perturbation, it creates timings errors by modifying the propagation time of the logic. However, compared with voltage glitches it presents many drawbacks. Due to thermal inertia, realizing a "temperature glitch" is too long compared with devices computation speed. Also, depending on the process technology, overheating may increase or reduce the propagation time in standard cells [66].

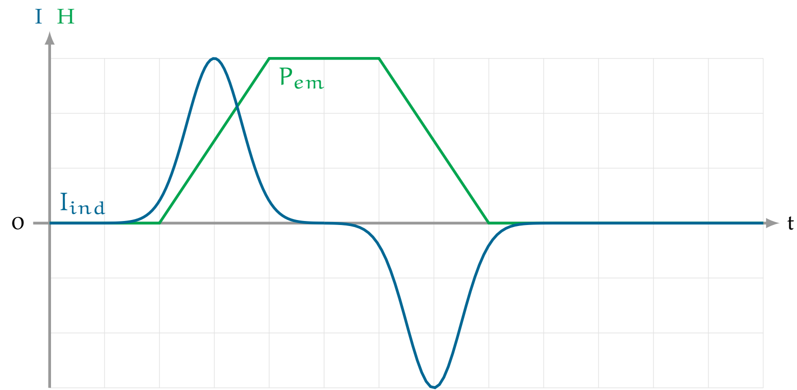
Despite these drawbacks, heating faults were successfully used to attack a RSA implementation on a micro-controller [69].

4.2.5 *Electromagnetic perturbations*

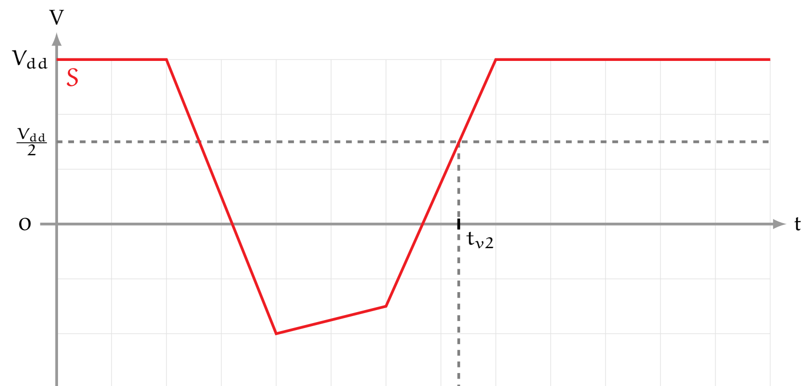
EM perturbations create sampling errors in the chip DFFs [70]. Sampling errors appears when an EM perturbation modifies the behavior of a DFF when its commuting. In other word, EM perturbations create errors in DFF outputs on clock rising edges.

The sampling faults come from a bad sampling of the DFF input (D) during an EM perturbation [71]. The phenomenon which leads to the sampling fault is summarized in figure 21.

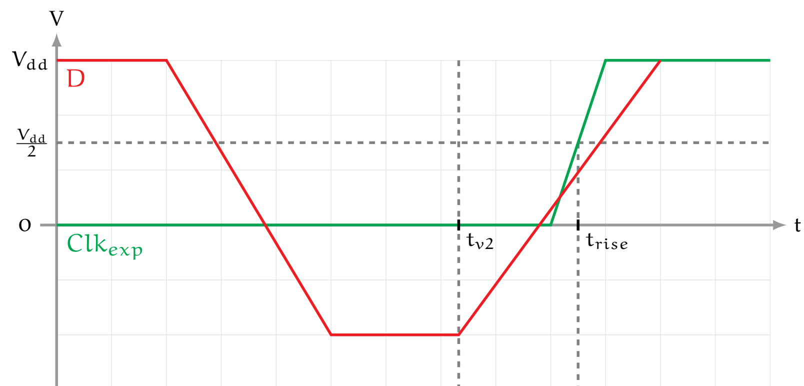
This figure presents how the signals of a DFF behave against an EM perturbation. These signals are the supply voltage (S), the clock (Clk), the input (D) and the output (Q). Figure 21a shows the injected EM Pulse (EMP) (P_{em}) and the corresponding induced current (I_{ind}) which will appear in the chip. On the EMP rising edge, this current will induce a supply voltage drop as depicted in figure 21b.



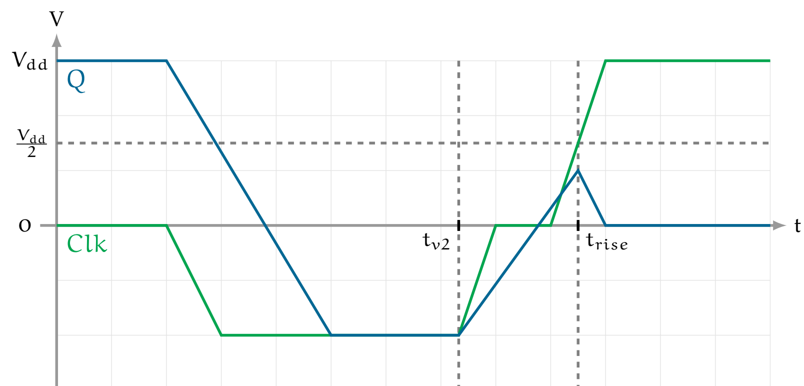
(a) Electromagnetic pulse and induced current (absolute units)



(b) Supply voltage drop and restoration due to electromagnetic pulse



(c) DFF input (D) drop and restoration due to electromagnetic pulse and non glitched clock



(d) DFF output (Q) drop and restoration due to electromagnetic pulse and glitched clock

Figure 21: How EM sampling faults occur [71]

On the falling edge, a reverse current will appear which will restore the supply voltage level. The important thing to note in figure 21b is that since the supply voltage level is below $V_{dd}/2$ the chip is stalled as all transistor are *de facto* blocked.

A consequence of this stalling is that all signals (D, Clk and Q) are stuck to 0 as represented in figures 21c and 21d. Once the supply voltage cross $V_{dd}/2$ (at t_{v2} s), all the signals are restoring. It is during this restoring phase that the sampling fault can appear.

The figure 21c shows the non glitched clock Clk_{exp} with the glitched DFF input. The sampling error happens when the clock signal arises before the input value D has reached $V_{dd}/2$ as in this case a logical 0 will be sampled instead of a logical 1.

The figure 21d presents a case where the clock signal restored quickly and the rising edge happen while the input value is still below $V_{dd}/2$ (at t_{rise} s). In this case, the sampled value is a logical 0 and the output Q is therefore constrained to 0 V instead of V_{dd} causing the error.

This behavior shows that EM perturbations must be well synchronized with the clock rising edges to produce faults. Also, this demonstrates that EM perturbations can be used to force bits from 1 to 0 quite easily. Despite these constraints, EM perturbations is a very used injection medium, mainly for its ease of use as most of the time no target preparation is needed and it has a good accuracy [72]. Finally, EM perturbation successes attacking various systems as MCU cache memories [73] and modern processor secure boot [74] for instance.

4.2.6 *Optical perturbations*

Optical perturbations are used to induce currents in a chip. Indeed, as transistors are basically a silicon crystal, they have optical properties and are sensitive to light. This property was very useful back when the only memories available were EPROMs. These particular memories can store data which can be erased using Ultra Violet (UV) light (figure 22). The CMOS transistor optical properties are still used for camera photo-diodes [75] which are basically transistors without a gate.

Figure 22: ST Microelectronics M27C256B EPROM

Also, as mentioned in section 4.1, Infra Red (IR) lasers are used to simulate cosmic rays effects on CMOS transistors. Both UV [76] and IR [77] are used in hardware security.

Optical perturbations induce currents in CMOS logical gates which can lead to bit flips as presented on a CMOS inverter in figure 23.

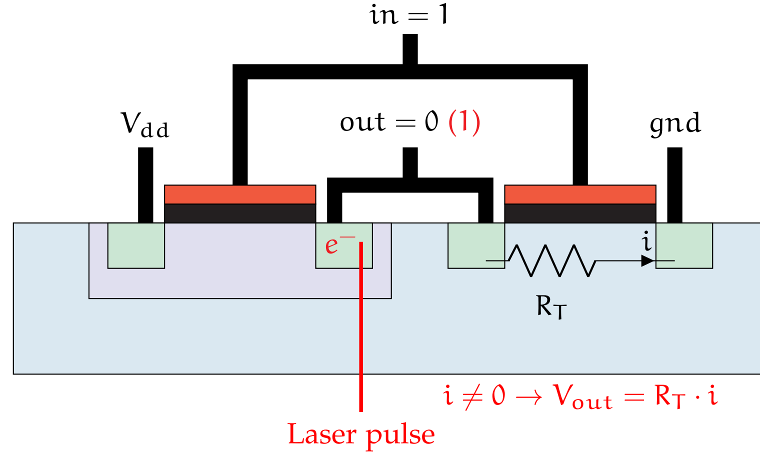


Figure 23: Laser effect on a CMOS logical inverter outputting a logical zero

This figure shows the effect of a laser pulse on CMOS logical inverter which outputs a logical 0. Considering the gate is in its stationary state, as the input is a logical 1, the P-doping MOS (PMOS) transistor (on the left) is blocked and the N-doping MOS (NMOS) transistor (on the right) is passing and equivalent to a resistor R_T. In this situation, the out signal is connected to the ground and there is no current i in R_T. A fault can be obtained by illuminating the transistor drain with a laser pulse. The photons coming from the laser will disperse their energy in the crystal and create electrons. Because it is their only way, these electrons will migrate to the ground, this migration will result in a current i through the resistance R_T and therefore a voltage $V_{out} = R_T \cdot i$ between the output of the gate and the ground will appear. If enough electrons are produced, this voltage will exceed $V_{dd}/2$ and the gate output will therefore correspond to a logical 1 which is the wrong value.

This remains available if the gate outputs a logical 1, in this case, the PMOS is passing and the NMOS is blocked. Therefore electrons will migrate to the supply voltage and this will create a voltage which oppose V_{dd} and forces the output to a logical 0.

As this can be applied to any CMOS gates with more or less difficulties, optical fault attacks able to flip bits in devices. Attacks relying on these faults are called optical fault masking [78] because bit flips can be modeled using a logical mask XORed with the target value as presented in equation (3).

$$\tilde{v} = v \oplus e \tag{3}$$

Equation: Fault masking attack model.

This equation models bit flips using a logical mask e also called the error, which is XORed to a target value v to obtain a faulted value $\tilde{v}$. The applied mask is usually bit-wide or byte-wide.

In addition to their modeling, Laser Fault Injections (LFIs) also present the best repeatability and accuracy among fault injection mediums. For these reasons, it is one of the most used medium, in particular in high-level certification processes. It was used to successfully attack memories [77], cryptographic implementations [79], PUFs [80] and smartphone secure-boot [81].However, considering modern SoCs, the transistor technology is usually around $10\,\mathrm{nm}$ while the size of a laser spot is usually $1\,\mu\mathrm{m}$ which means that contrary to what is depicted on figure 23, we usually target a many transistors.

4.2.7 *Body biasing*

Body biasing is a technique which consists in biasing the substrate of transistors in a circuit. This technique is usually used in low power devices to modulate the transistor threshold [82]. The bias is usually around a few mV.

In hardware security, Forward Body Biasing Injection (FBBI) consists in glitching the transistor substrate with an important voltage ($60\,\mathrm{V}$ during $8\,\mu\mathrm{s}$ [83]). This glitch provokes a threshold modification in transistors blocking them all, introducing faults in the circuit [84].

4.2.8 *X-Rays*

X-rays are the youngest injection medium used in hardware security [85]. However, as mentioned in section 4.1, cosmic rays (and in particular X-rays) are known to perturb digital devices for a long time.

X-rays can empty the charges of floating gates of transistors and therefore force zeros in flash memories or Electrically EPROM (EEPROM). They also can modify the behavior of transistors, forcing NMOS as passing and PMOS as blocked. This able circuit edition but the phenomenon is reversible by heating the circuit to $150\,^{\circ}\mathrm{C}$ during $1\,\mathrm{h}$. As they can be focused to a nano-scale (and therefore target a single transistor), X-rays are the most accurate injection medium but a synchrotron is required for such accuracy [85].

4.2.9 *Software induced perturbations*

Recently, some software induced perturbations have been presented. These perturbations rely either on a software overusing a component like in the rowhammer attack (section 4.2.9.1) or on a corrupted energy manager like in the ClkScrew attack (section 4.2.9.2).

4.2.9.1 *Rowhammer attacks*

Rowhammer attacks are a special kind of perturbation attacks because they are triggered *via* software. They rely on the rowhammer bug to force bit-flips in the memory of a digital device [86]. The rowhammer bug is based on a natural phenomenon, the discharging of capacitor over the time. In DRAMs, data is stored in the form of charges in capacitor as shown in figure 24.

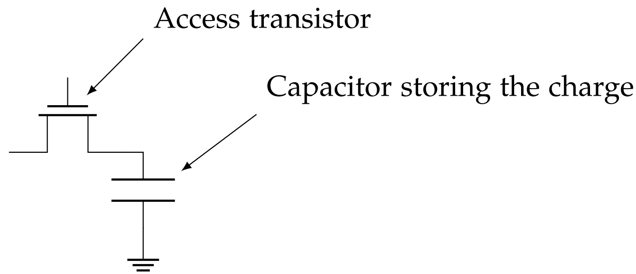


Figure 24: One-bit DRAM memory cell

These capacitors naturally discharge over time and to not lose information, DRAMs have a refresh mechanism to recharge them. Usually, this refresh happens every 64 ms. Rowhammer attacks consists in accelerating the discharge of capacitors in DRAMs so they are fully discharged when the refresh happen. As the presence of a charge can either be considered as a logical 1 (in the case of true-cells) or as a logical 0 (in the case of anti-cells), this mechanism can be used to flip bits in the memory.

Accelerating the discharge of a memory cell is done by keeping activated the memory lines around it. Indeed, DRAMs are organized as a matrix, when a data is read, the line storing it is activated and copied in the memory row buffer and the needed data is read from this buffer as presented in figure 25.

Activating a line accelerate the discharging of its adjacent lines capacitors. By remaining a line activated it is therefore possible to discharge the capacitors of its adjacent lines faster than the refresh period. Usually the rowhammer attack is realized by activating the two lines surrounding the targeted one. A line is kept activated by continuously reading a data in it (500 000 accesses in less than 64 ms). This high repetition of reads at a specific line in the DRAM have given the name rowhammer to the attack.

Rowhammer attacks were used to successfully attack various systems with different OS and architectures. The first one was presented by the Google team Project Zero in which they achieved a privilege escalation under Linux on an Intel processor [87]. Latter on, various rowhammer attacks were published: a remote attack using JavaScript [88], an attack on an ARM device powered by Android [89], one using the integrated GPU of a SoC to fast access the DRAM [90] and one using TCP/IP and the Remote DMA (RDMA) to hammer the DRAM from the network [91].

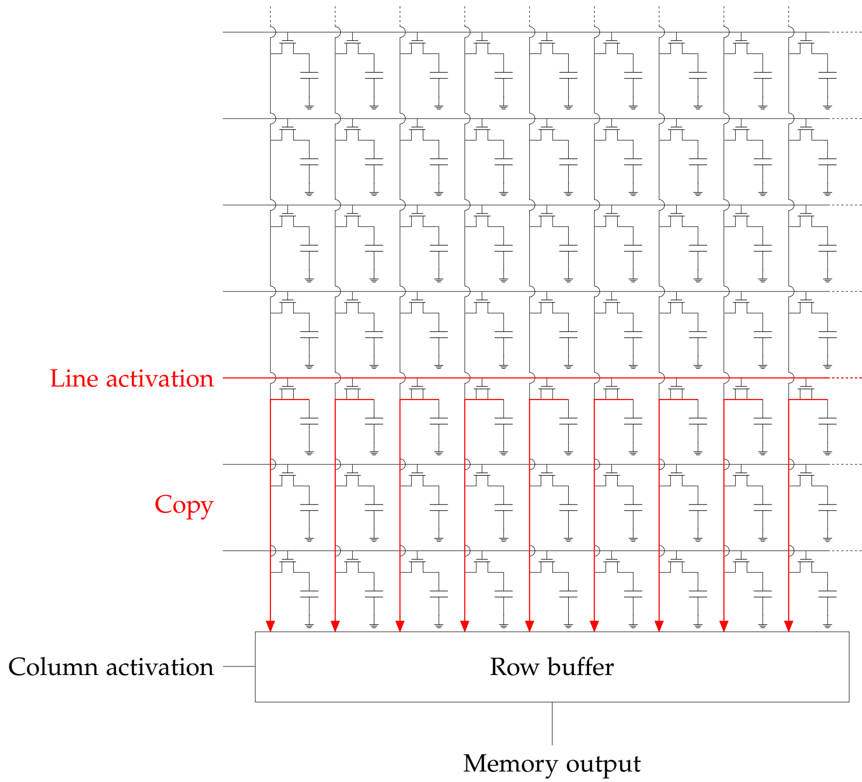


Figure 25: DRAM line activation and copy in the row buffer

The last attack using rowhammer is a side-channel attack. Indeed, the more capacitors in the target line are charged, the more probable a bit-flip will happen using rowhammer. This probability variation leaks information about the content of the target line which is therefore used as a side-channel for reading a memory using rowhammer in the RAMbleed attack [10].

4.2.9.2 *ClkScrew attacks*

The ClkScrew attack targets highly integrated SoCs. For optimizing their performance on consumption ratio, these SoCs integrate a Power Management Subsystem (PMS). This PMS supply the power voltage and the clock to every other modules as presented in figure 26.

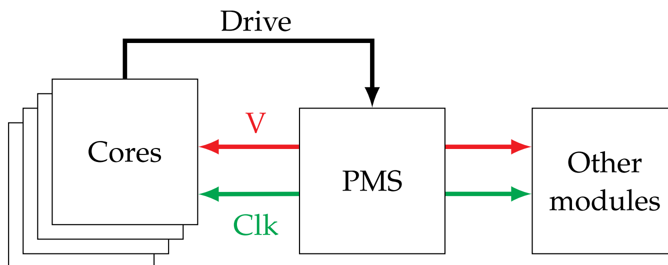


Figure 26: PMS supplying the power voltage and clock to cores and module of a SoC

The PMS is composed of a PMIC and Phase Locked Loops (PLLs) which supply the SoC modules with power voltage and a clock and is driven by the cores in kernel mode.

On modern devices, such as smartphones, the cores have various work domains which correspond to different power voltage and clock frequency pairs. The idea of the ClkScrew attack is to force a core doing a sensitive operation in an unstable domain (voltage/frequency couple) which will provoke timing errors and, therefore, faults.

There are two attacks using this path: the ClkScrew attack targeting the ARM TrustZone TEE of a Nexus 6 smartphone to load non-trusted applications in it [65] and the PluderVolt attack targeting the Intel secure enclave (SGX) [92]. As these attacks are recent, many questions remains about the security of integrated modules and how to isolate them from this kind of attacks.

4.3 CHARACTERIZING A FAULT

Being able to induce a fault in a working device is not enough to realize an attack. As all faults are not exploitable, obtaining an exploitable fault and actually exploiting it may be a trickier business than injecting a fault in a working device.

To simplify their exploitation, faults are characterized on the target, this characterization is a very time consuming process as many parameters are involved (the location of the perturbation, the intensity, the timing, *etc*). The characterization gives a fault model, this fault model is the set of the possible faults associated to their probability of occurrence [93]. The fault model is then used to analyze the target security mechanisms. In some cases, the fault can be exploited to attack the mechanism.

4.3.1 *Fault models*

Determining the fault models is very important for measuring the impact of a fault and build efficient protections against it. However, as digital devices have a complex architecture with many abstraction layers, the impact of a fault may be interpreted differently depending on the considered layer as shown on the figure 27.

This figure shows how a fault can propagate through the different abstraction layers of a device. In section 4.2, we have seen how perturbations can be created inside transistors and standard cells. This correspond to the logical level. However, in some cases, it is not possible to have information about the deep behavior of the target and therefore a more abstracted fault model is determined. Furthermore, anticipating the fault effect at a specific layer from the underlying layer is not straightforward due to the scaling factor between them.

For these reasons, some works have focused on determining what faults can be achieved regarding the different abstraction layers, with different injections medium on various targets. A summary of these works is presented in figure 28.

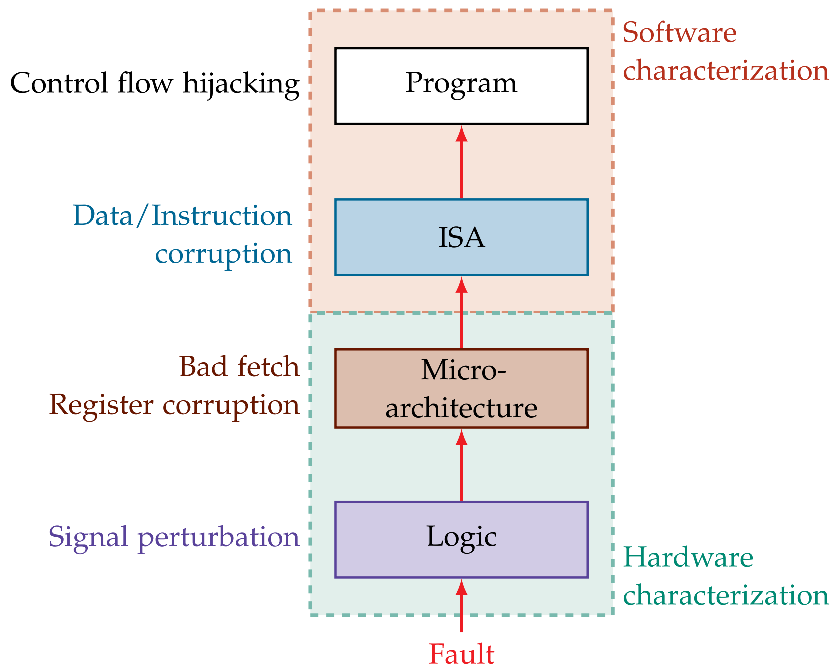


Figure 27: Fault propagation through digital devices abstraction layers with some fault effects as examples. Inspired from [94].

This figure presents the different observed fault models in various conditions. The logical level was discussed in section 4.2 and is therefore not represented. Three types of target are considered here, the MCU CPUs, the Field Programmable Gate Arrays (FPGAs) and the modern CPUs. MCUs are the most studied as they are widely used in secure and IoT devices, FPGAs are rarely used and focus on deep characterization. Modern CPUs are quite new and more complex, therefore they are less studied than MCUs CPUs.

At the micro-architectural level, several components were observed to be sensitive to faults. In [73], the authors observed that an EM perturbation can corrupt the cache fetching to replay instructions by skipping others. In general, memory elements are sensitive to perturbation, especially the flash memory which was successfully corrupted using both EMPs [103, 104] and LFIs [105, 106] on various targets. These attacks were able to corrupt both executed code and data. Another way to fault data is to target the data bus as demonstrated in [102] using EMP. Clock and voltage glitches are mainly perturbing the pipeline [67] as it is mostly where the critical paths are, this was deeply studied in a LEON-3 implemented on a FPGA in [108].

Regarding the ISA level, the main targets are either the instructions or the data [67]. Both effects can be obtained using various injection mediums on all types of targets, including modern CPUs [101, 107]. Faults on data are quite easy to analyze as the masking fault model presented in section 4.2.6, which consists in `XOR`ing the data with a mask, is usually used. Regarding instructions, there are multiple ways to analyze the fault. The masking fault model can be used by considering the instruction as a data [104]. Sometimes, it is possible to be more precise by considering the type of instruction. On a data pro-

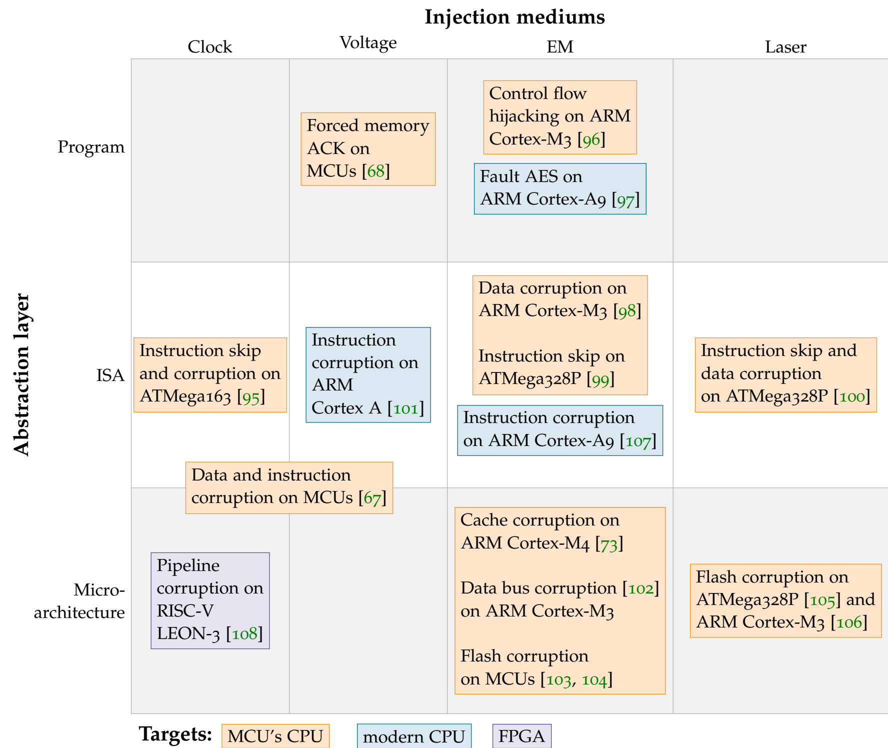


Figure 28: Fault injection characterization state of the art.

cessing instruction faults usually affect the operands [107] while on a branch instruction, it is the address that is usually corrupted [100].

Finally, program level characterization works are usually preliminary works which are used as a proof of concept. For instance, in [68], the authors present a voltage glitch method which consists in shaping the voltage glitch using a genetic algorithm [109]. The work presented in [107] is the first to focus on modern CPUs by targeting an A series ARM core and [96] focus on software vulnerabilities in IoT devices, such as a fault triggered backdoor for instance.

4.3.2 *Fault analysis*

The fault analysis is an important step as it able to identify attack paths regarding a program and fault models, to measure the impact of potential attacks and to build adapted countermeasures.

As they rely on fault models, fault analysis can also be separated in categories depending on the abstraction level we are working on. Fault analysis can be compared to a code review which integrate the possibility to have runtime faults. There are two strategies: a static analysis or a dynamic analysis. As faults are used to perturb a target during its execution, most works about analyzing implementations against faults use the dynamic approach.

As the fault models can be numerous and happen at anytime during the target execution, the number of cases to test is absolutely gigantic. To answer this issue, many works propose tools to automatize the analysis by simulating the perturbation of implementations.

At the logical level, PAFI [110] simulates Verilog code to analyze the susceptibility of a design against faults. In the fault analysis literature, the micro-architectural layer is named system level. Several technologies are used to build analysis at this level like SystemC [111] or QEMU [112].

Several works for fault analysis at the ISA level has been done. The main reason is that as many assembly languages exist with their own specificities, building an efficient generic analysis tool is complicated. Also, several methods were proposed to improve the efficiency of the analysis. In [113], the authors focus on the JavaCard bytecode while [114] proposes a formal verification of countermeasures using a SMT-solver.

Finally, in order to realize more generic fault analysis, several works have focus on doing the analysis at the program level, *i.e.* on the source code [115] or the Control Flow Graph (CFG) [116, 117] of the program.

4.4 EXPLOITING A FAULT

The exploitation of faults is a complicated task. Even if the fault model and the fault analysis helps in being confident of the success of an exploitation, some un-modeled constraints may harden the exploitation. This section presents the general approach of famous attacks and proposes a classification based on the system security model introduced in figure 14 in section 2.4.

4.4.1 *Fault attacks against multi-application systems*

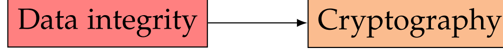
A multi-application system can be attacked in many ways, however, in hardware and embedded systems security, a system is considered broken if a security property or a security mechanism presented in figure 14 cannot be trusted.

Regarding FAs, the perturbation will affect either the data integrity or the execution flow then propagate itself. Depending of the case, different security mechanisms or properties can be broken. The following section presents the most famous attacks regarding these mechanisms and properties.

4.4.1.1 *Fault attacks targeting cryptography*

Cryptography is historically the most attacked mechanism. Indeed, the first security concerns were about the resilience of these algorithms, therefore a lot of work has been driven to evaluate the impact of fault attacks on cryptographic algorithms implemented in SEs. This section presents some famous exploits on implemented cryptography.

RSA-CRT MODULUS FACTORIZATION (BELLCORE ATTACK)



The Bellcore attack is a fault cryptanalysis attack which aims at factorizing the modulus N of a RSA public key [54]. This attack can only be applied on the Chinese Reminder Theorem (CRT) implementation of RSA, however this implementation is the most common one because of its efficiency.

The attack was introduced in [54] and successfully exploited in [118]. The attack consists in faulting the computation of intermediates values used in the CRT. As the public modulus N is the product of two primes p and q (equation (4)), a message x can be signed using a secret s as presented in equation (5). Factorizing N allows to recover the secret.

The CRT gives the possibility to compute E in equation (5) by precomputing E_1 and E_2 such as in equation (6). Therefore, using a and b with the properties depicted in equations (7) and (8), E can be computed as the linear combination of E_1 and E_2 as shown in equation (9).

$$N = p \cdot q \tag{4}$$
$$E = x^s \mod N \tag{5}$$
$$E_1 = x^s \mod p,\ E_2 = x^s \mod q \tag{6}$$
$$a = 1 \mod p,\ a = 0 \mod q \tag{7}$$
$$b = 0 \mod p,\ b = 1 \mod q \tag{8}$$
$$E = a \cdot E_1 + b \cdot E_2 \mod N \tag{9}$$

Equation: Computation of a cipher using RSA-CRT

The Bellcore attack relies on a fault that happen during the computation of E_1 or E_2. For the sake of simplicity, we suppose that E_1 is faulted. Therefore the computation of E will give a faulted value $\tilde{E}$ as presented in equation (10). Once an attacker knows E, $\tilde{E}$ and N, a factor of N can be computed as presented in equation (11)

$$\tilde{E} = a \cdot \tilde{E_1} + b \cdot E_2 \mod N \tag{10}$$
$$gcd(E - \tilde{E}, N) = gcd(a \cdot (E_1 - \tilde{E_1}), N) = q \tag{11}$$

Equation: Factorization of RSA modulus in a faulted CRT implementation

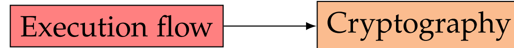
In [54], the authors also present the Lenstra's improvement which requires the knowledge of the message x and the public exponent e instead of the signature E. In these conditions, the modulus can be factorized using the same fault on E_1 as shown in equation (12).

This attack is very powerful and demonstrates how a specific implementation can weaken the security of a cryptographic algorithm against FAs.

$$\gcd(x - \tilde{E}^e, N) = q \tag{12}$$

Equation: Factorization of RSA modulus in a faulted CRT implementation (Lenstra's improvement)

ATTACKING LOGARITHM EXPONENTIATION SQUARE AND MULTIPLY IMPLEMENTATIONS



The square and multiply algorithm is the most used algorithm for computing the logarithm exponentiation. This algorithm iterates over all the bits of the private exponent $s = \{s_{t-1}, ..., s_0\}$, squares the internal state R and if the current bit is set to 1, it multiplies the message as presented in algorithm 1. In the case of a signature, this allows to compute $E = x^s \mod N$.

Algorithm 1: Left to right square and multiply algorithm for logarithm exponentiation

Result: $E = x^s \mod N$
1 $R = 1$;
2 **for** $(i == t-1; i \geqslant 0; i--)$ **do**
3 $\quad R \leftarrow R \cdot R \mod N$;
4 $\quad$**if** $s_i == 1$ **then**
5 $\quad\quad R \leftarrow R \cdot m \mod N$;
6 $\quad$**end**
7 **end**

Fault attacks on this algorithm rely on the corruption of the squaring operation (line 3). [54] introduces an attack which is based on a corruption of the squaring while [119] presents and successfully realizes an attack based on the skipping of one squaring during the signature. I present the attack using the skipping fault model.

Skipping the last squaring gives a faulty signature x_0, this signature's difference with the real one x depends on the value of the first bit of the secret exponent s_0 as shown in equation (13).

$$x = \begin{cases} (x_0)^2 & \text{if } s_0 = 0 \\ (x_0)^2 \cdot m & \text{if } s_0 = 1 \end{cases} \tag{13}$$

Equation: Relation between the faulty signature obtained by skipping the last squaring and the correct signature in the square and multiply algorithm.

Once a bit has been recover, it is possible to recover the next bit by comparing the faulty signature x_i with x_{i-1}. Indeed, between two

iterations of the loop line 2, the internal state R_i has only two possible values regarding the previous state as shown in equation (14).

Skipping the squaring in the ith loop iteration simplifies this relation as depicted in equation (15). Finally, after the whole signature computation, the faulty signature x_i obtained by skipping the ith round squaring is directly linked to the previously observed faulty signature (*i.e.*, the one observed by skipping the $(i-1)^{th}$ round squaring) as presented in equation (16).

This attack allows to recover the complete private exponent by injecting a fault for every bit to recover.

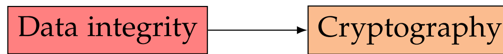
$$R_i = \begin{cases} (R_{i-1})^2 & \text{if } s_i = 0 \\ (R_{i-1})^2 \cdot m & \text{if } s_i = 1 \end{cases} \tag{14}$$

$$R_i = \begin{cases} R_{i-1} & \text{if } s_i = 0 \\ R_{i-1} \cdot m & \text{if } s_i = 1 \end{cases} \tag{15}$$

$$x_i = \begin{cases} x_{i-1} & \text{if } s_i = 0 \\ x_{i-1} \cdot m^{2^{i-1}} & \text{if } s_i = 1 \end{cases} \tag{16}$$

Equation: Relation between the faulty signature obtained by skipping the squaring in the ith and $(i-1)^{th}$ iteration of the loop in the square and multiply algorithm.

RSA MODULUS CORRUPTION



In the RSA algorithm, the integrity of the modulus N is very sensitive. Indeed, because the private key is based on its number of coprime $\Phi(N)$, any attacker able to compute this value can forge, using the extended euclidean algorithm for instance, a legitimate private key d corresponding to a known public key e as depicted in equation (17).

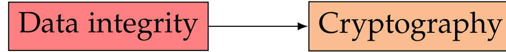
$$e \cdot d = 1 \mod \Phi(N) \tag{17}$$

Equation: Bézout's identity

Corrupting the modulus N can simplify the computation of $\Phi(N)$. The aim is either to force the modulus to be a prime, in this case $\Phi(N) = N - 1$ or to make it easily factorizable, simplifying the computation of $\Phi(N)$.

This technique was successfully used in the ClkScrew attack [65] for loading an un-trusted application in the TEE of a smartphone.

PERSISTENT FAULT ANALYSIS ON AES



The Persistent Fault Analysis (PFA) relies on the corruption of Substitution Boxes (SBoxes) in a persistent way. This means that the fault remains active until the target is reset. This able to realize several operations with the same faulted value.

This kind of fault was first used to attack the AES algorithm in [120] and a practical attack using UV light to corrupt AES SBoxes lookup tables was presented in [76].

Corrupting SBoxes implies that they are biased. The consequence is that for an entry x the output is $\tilde{y} = \tilde{SB}(x)$ instead of $y = SB(x)$, moreover, as the SB is initially a bijection, no other input value can output the y value in the faulted SBox $\tilde{SB}$.

Considering the last round of AES, the ith byte of the cipher C_i is computed with the secret key K and the previous round internal state i^{th} byte X_i as presented in equation (18).

$$\tilde{C}_i = \tilde{SB}(X_i) \oplus K_i \tag{18}$$

Equation: AES last round *SubBytes* and *AddRoundKey* operations in the presence of a corrupted SBox

Actually, because the SBox is faulted the internal state X is also a faulted value but it doesn't matter in the attack. The important thing is that there is a value for $\tilde{C}_i$ which is unreachable because of the SBox corruption. This value is the y introduced above. This unreachable value (named forbidden value in the literature) is easily identifiable by observing the output ciphers. We name this value C_i^* for the ith byte of the cipher and it is directly linked to the ith byte of the key as shown in equation (19).

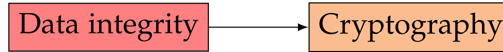
$$C_i^* = y \oplus K_i \tag{19}$$

$$\forall i, K_i = y \oplus C_i^* \tag{20}$$

Equation: Relation between the forbidden value and the secret key

At this point, y has only 2^8 possible values. By guessing the y value it is possible to directly recover all the key bytes as shown in equation (20). This reduces the number of possible keys to 2^8 values and the verification can be done by knowing a plaintext and its corresponding cipher.

DIFFERENTIAL FAULT ANALYSIS (DFA)



The DFA is a cryptanalysis technique which relies on the comparison of the correct cipher and a faulty cipher of the same (unknown) plaintext. In [55], the authors presents the attack for the DES algorithm. [121] introduces a DFA technique usable for the AES which was extended in [122] and [57] presents a DFA for ECC.

The DFA on AES is the most famous attack among all DFAs and was successfully exploited in [123] and even on a DFA protected AES by simultaneously injecting two faults in the implementation [124]. The principle of the attack is to realize a fault on a byte of the AES state before the last `MixColumns` operation, in the 9th round as presented in figure 29.

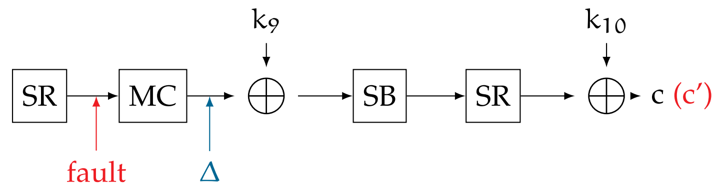


Figure 29: DFA Principle

Once the faulty ciphertext `c'` is obtained, the attacker can compute the value Δ such as presented in equation (21).

$$\Delta = SB^{-1}(SR^{-1}(c \oplus k_{10})) \oplus SB^{-1}(SR^{-1}(c' \oplus k_{10})) \tag{21}$$

Equation: Relation between the faulty ciphertext, the correct ciphertext and the possible faulty outputs of the *MixColumns* operation during a DFA on AES

Faulting one byte of the AES state before the last `MixColumns` implies that the faulted output of the `MixColumns` (Δ) has only 2^8 possible values. Regarding the equation (21), as the possible values for Δ are known, the only unknown value is k_{10} and at this point Δ has only four bytes (among sixteen) that are not zeros. The next step consists in testing all the values for these four bytes in k_{10} (2^{32} values in total) such as the corresponding Δ is among the possible values.

Given a faulty ciphertext c' this computation will give a set of possible values for the k_{10} bytes. The correct k_{10} value is the one that verify the equation (21) for every faulted ciphertexts. In [122], the authors demonstrate that the probability to recover the correct key with two faulty ciphertexts is around 98%.

Doing a fault on a byte will before the last `MixColumns` operation leads to a faulty diagonal in the AES as shown in figure 30.

In this figure, the `SubBytes` and `AddRoundKey` operations are not represented as they do not change the layout of the fault propagation.

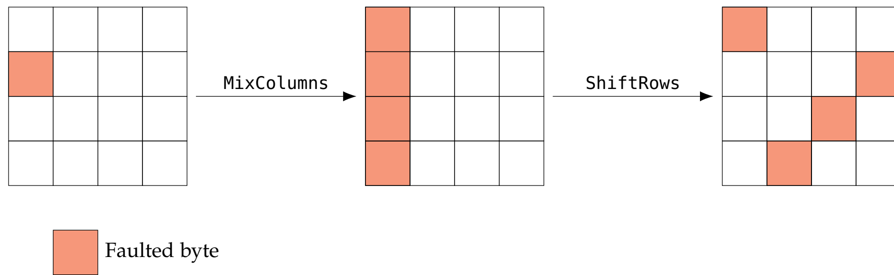
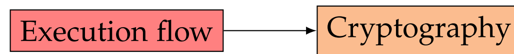


Figure 30: Propagation of a faulted byte before the last `MixColumns` operation in the AES

As one can see, the output has four faulted byte in the pattern of a diagonal. Comparing the correct cipher with the faulty one able to observe this diagonal and recover the corresponding bytes of the key.

Therefore, doing these steps for the 4 diagonals able to recover the 16 bytes of the key with only 8 faulty ciphertexts. Actually, 2 faulty ciphertexts can be enough because faulting a byte in the AES state result in a faulted diagonal. Therefore, by faulting 4 bytes in the AES state, it is possible to obtain 4 faulted diagonals with only one cipher. However, these faults are exploitable only if the four initial faulted bytes are on the four different columns of the AES state. Otherwise, there is too many possible values for Δ, reducing the efficiency of the attack.

AES LAST *addroundkey* SKIP



The last operation of the AES algorithm is the *AddRoundKey*. This operation is particularly critical. Indeed, if an attacker is able to skip it, she will obtain a faulty ciphertext $\tilde{c}$. The difference between this faulty ciphertext and the correct ciphertext is simply the `XOR` operation with the secret key k as shown in equation (22).

$$c = \tilde{c} \oplus k \tag{22}$$

Equation: Relation between the faulty ciphertext and the correct cipher text in the case of the skipping of the last *AddRoundKey* operation in AES

From this point, the secret key can be trivially recovered by `XOR`ing the faulty ciphertext with the correct one. This attack was successfully realized using a laser injection in [79].

4.4.1.2 *Fault attacks targeting memory partitioning*

Memory partitioning is a basic security feature in multi-application systems. It ensures that, despite the physical memory is shared, an application cannot access the memory of another application as shown in figure 31.

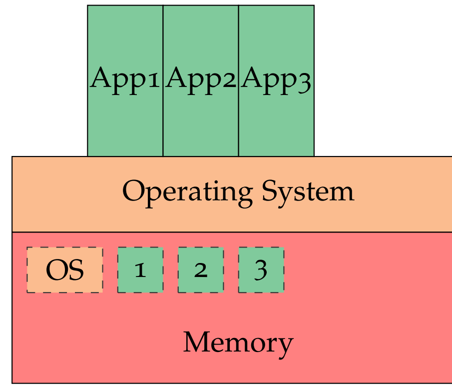
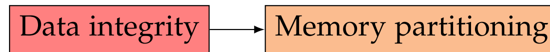


Figure 31: Memory partioning principle

This figure shows how the memory is managed in a multi-application system. Each application, even the OS itself, has a dedicated subpart of the memory. This partitioning is ensured by a dedicated component, the MMU or the MPU, and is configured by the OS at the startup. The OS can access all the memory as it configures this partitioning, this implies that while executing a sensitive application, the OS must be trusted, this trust is ensured by the secure boot. In security, escalating privileges can be done by accessing the OS memory and modifying its configuration.

ROWHAMMER ATTACKS.



Because of the memory partitioning, creating a malicious application that will access the OS memory and modify it is not possible. However, using a physical perturbation can induce modifications in this critical memory. This is a classical attack path in rowhammer attacks [87, 89].

For the example, in Linux-based OS, the memory partitioning is managed by a MMU, the MMU gives every application a pool of virtual addresses they can use to realize memory accesses. When the MMU receive a memory access, it translates the virtual address ($addr_{virt}$) into a physical address ($addr_{phys}$) using Page Table Entrys (PTEs) as shown in equation (23).

$$addr_{phys} = \mathsf{PTE} + addr_{virt}[11:0] \tag{23}$$

Equation: Relation between the virtual address and the physical address in Linux based OS

The PTE is stored in the OS memory. The MMU resolves the PTE's address using the upper bits of the virtual address and verifies that the application realizing the memory access can access this address. Therefore, an attacker able to corrupt the PTEs can force a memory

access to any part of the memory in particular, the OS memory. This was successfully done using rowhammer in [87, 88, 89] to obtained privileged rights.

4.4.1.3 *Fault attacks targeting secure boot*

As mentioned above, the secure boot is a critical security feature to ensure the trust in a OS or an application. The secure boot is the succession of verified stages, each stage is ciphered and signed. The aim of a stage is to verify the authenticity of the next stage, to uncipher it and to execute it . The only exception is the first stage which is *de facto* trusted and executed at the startup, it is the root of trust.

Bypassing the secure boot able to load an un-trusted OS on the target, this can be critical if a sensitive application is load on such OS, also, if the bypass happens soon enough, critical assets, as the device master key, might still be loaded in the memory. This security was bypassed on the XBOX360 [125] using a voltage glitch which force the return value the signature comparison function to zero. This exploit drove the apparition of modchips for this console with financial consequences for Microsoft.

The secure boot was also successfully bypassed on an ARM Cortex-M3 core using a glitch voltage [101]. The idea is to load a payload and its destination address in the device memory as shown on figure 32.

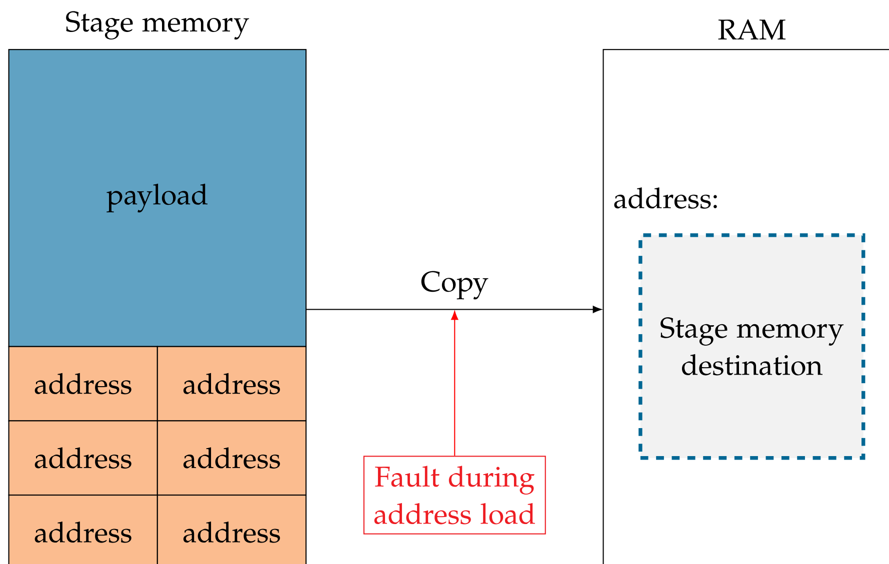
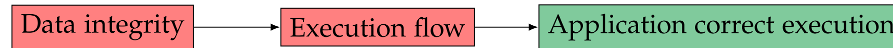


Figure 32: Boot loader stage for bypassing secure boot using a fault attack [101]

During the secure boot, this stage will be loaded in the RAM before being verified and un-ciphered. By faulting the instruction doing the memory load of the addresses, the authors are able to load this value in the Program Counter (PC) instead of the RAM and therefore execute their payload, bypassing the verification of the stage and then the secure boot.

4.4.1.4 *Fault attacks targeting application correct execution*

OPERAND STACK CORRUPTION ON THE JAVACARD VIRTUAL MACHINE.



The JCVM is the a part of the software layout powering SEs. As SEs provide a high level of security, many work not only focus on the hardware layer but also on the software layer regarding physical attacks.

The JCVM relies on a stack to manage operations such as branches, function and method calls, memory accesses, *etc*. Every application executing over a JCVM has its own stack and the JCVM ensures the memory partitioning between them. Fault attacks on JCVM focus on corrupting the stack and exploiting the multi-application architecture to achieve combined attacks [126, 127, 128].

In the JavaCard literature, combined attacks refer to both software (named logical attacks) and hardware attacks. In this research field, the attacker is often trying to bypass security mechanisms such as the ByteCode Verifier (BCV) or the memory partitioning via logical attacks (*i.e.* software only). Combined attacks on the JCVM focus on the loading of a malicious application that does not trigger the BCV and which is activated *via* a fault corrupting the stack to either load a malicious class leading to an instance confusion [126, 128] or taking control of the execution flow [127].

In this case, the fault targets the device data integrity which leads to a modification of the executed application behavior.

4.5 COUNTERMEASURES

As we presented how to inject a fault, how to model its effects, how to analyze implementations and how to exploit attack paths, the last but not least FA topic is the building of countermeasures.

Building countermeasures is the final goal of all that was presented before as we aim to built resistant devices. However, we will see that there is no countermeasures that can make a device totally fault proof and also that they all have a price, in money, space, memory or computation time.

This particularity imposes to developers and costumers to wisely adjust their products security depending on the performances they desire and the threats they identified. Despite it is a very interesting topic, we will not discuss about threat analysis but it is important to keep in mind that proposing and integrating countermeasures in a device is a long thought process.

Also, as the critical mechanisms of a device presented in figure 14 are the entry point of faults, countermeasures focus on hardening these mechanisms to make them resistant to faults.

4.5.1 *Space redundancy*

The space redundancy aims at protecting the device data integrity by storing it twice. Therefore, by regularly checking the difference between the two versions of the same data, the device can detect if a fault was induced on it. This solution is very naive and costly as it doubles the needed space for storing data. A refinement is to only duplicate sensitive data. However, as we have seen in section 4.4.1.2 for the memory partitioning, in some cases there are a lot of data to protect (all the PTEs in this case).

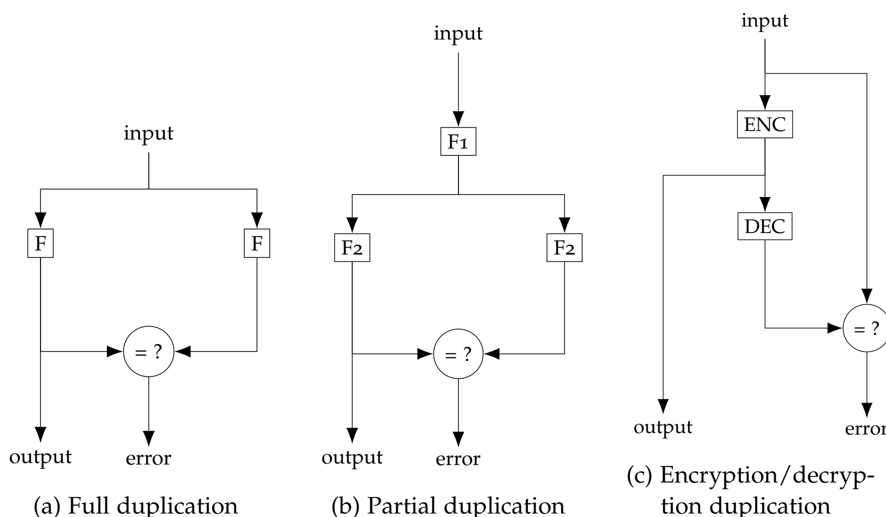
A less costly solution is to use error correcting codes. The most famous is the parity bit [129, 130]. This solution is usually used in DDR version 4 (DDR4)-Synchronous Random Access Memorys (SRAMs) and in the cache memories [131]. Also, to improve the security, better error correcting codes have been presented and use to protect AES implementations such as multiple parity bits [132], non-linear codes [133] or non-linera r-bits long codes [134].

Despite error correcting codes are an efficient countermeasure against faults, they do not protect against multiple faults. Indeed, it is possible for an attacker to fault both the data and the parity bit or its verification. This example shows that countermeasures against faults does not make FA totally inefficient but harder to achieve as they increase the number of faults needed to attack the system.

Moreover, in order to secure an implementation against all physical attacks (side-channel and perturbations), it is important to keep in mind that error correcting codes does not suits well with side-channel masking countermeasures [135].

4.5.2 *Operation duplication*

The operation duplication is in some way similar to the space redundancy but instead of duplicating a data it duplicates an operation. This can be done in several ways as shown in figure 33.



(a) Full duplication (b) Partial duplication (c) Encryption/decryption duplication

Figure 33: Operation duplication possible implementations

This figure presents three ways of implementing the duplication of an operation F. Figure 33a shows the most naive way which duplicates the full operation while figure 33b presents a partial duplication. The partial duplication is less costly than the full part and is usually used to secure a sensitive part of the algorithm. As this countermeasure is usually used to protect cryptographic algorithm, it is possible that both the encryption and the decryption are implemented, therefore it is possible to use both to realize a verification as presented in figure 33c. This duplication is interesting as it does not require to really duplicate a function.

As mentioned above, this countermeasure does not make FAs completely inefficient but make harder any FA targeting the operation as two faults are needed. Indeed, either both versions of the sensitive operation have to be faulted or a version of the operation and the compare operation.

4.5.3 *Infection countermeasure*

The infection countermeasure is a refinement of the duplication countermeasure which aims at protecting against the DFA when the compare operation is bypassed. The idea is to diffuse the fault effect in the output to avoid any information leakage due to the fault invariant. In this case, even with two faults, the DFA is not a viable attack.

In practice, the idea is to identify the fault between the outputs out_1 and out_2 and to use a diffusion function D to generate a mask from the fault which is applied to the outputs as presented in equations (24) and (25). This can also be done on intermediate states.

$$out'_1 = D(out_1 \oplus out_2) \oplus out_1 \tag{24}$$
$$out'_2 = D(out_1 \oplus out_2) \oplus out_2 \tag{25}$$
$$D(0) = 0 \tag{26}$$

Equation: Infection countermeasure

With a good diffusion function D, the fault will propagate in the output and no information will be usable by the attacker. This is mostly used in AES. The only constraint on D is shown in equation (26). Several diffusion functions are proposed in [136, 137] and the need of randomness in D was highlight in [138].

4.5.4 *Code hardening*

The code hardening is a particular kind of countermeasure which aim at adding software countermeasures in a code to protect it against control flow attacks. At first, these works aimed at detect and protect against software attacks such as Return-Oriented Programming (ROP) attacks [139] but appeared to be suitable for protecting against fault attacks targeting the control flow.

These countermeasures face many challenges both on their design and on their implementation. Indeed, the program is protected by the program and moreover, the integration of countermeasures must be compliant with the compilation process of software which realizes optimizations that can nullify the countermeasures efficiency.

As a consequence, the integration of such countermeasures has to be adapted to the compilation process as shown in figure 34 with the impact it can have on development time and optimization.

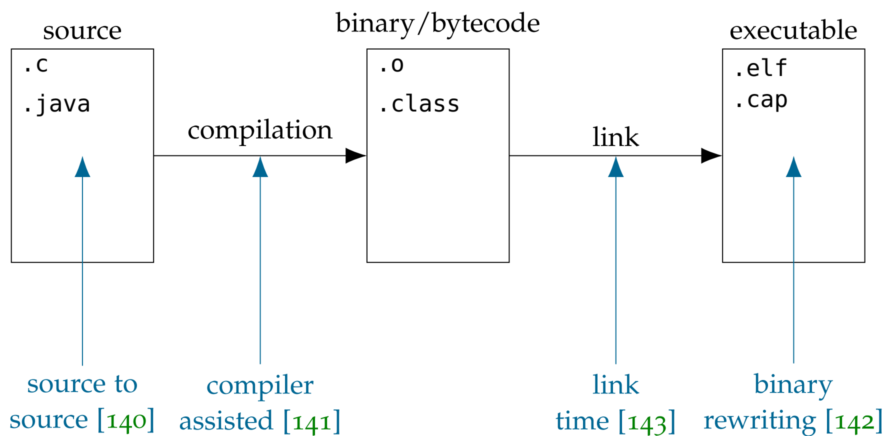


Figure 34: Software countermeasures integration in the compilation process.

This figure shows the different steps of the compilation for `.elf` (Linux executable file) and `.cap` (JavaCard executable file) files. This process is simplified but it able to highlight the different steps where a software countermeasure can be integrated into a software implementation.

Regarding the countermeasures themselves, several solutions are proposed.

One of them is the signature of blocks [144]. This method relies on the verification of signature at different moments of the program execution. The blocks can have different granularity, they can be basic blocks [145] or C statements [140]. Also, the signature verification can be implemented in different ways: it can rely on hardware mechanisms like the performance counter [146], it can be fully integrated in software in the program [147, 148] or in a virtual machine as proposed for the JCVM in [142, 149, 150].

Another solution aims at complementing the signature of blocks. Indeed, the signature of blocks able to detect control flow errors from a block to another but not inside the block. A solution to detect intra-block control flow error is the usage of step counters [151]. The principle of this solution is to initialize a counter N when entering a block which is equal to the number of instructions to execute in the block. Each time an instruction is executed, this counter is decreased by 1 and its value is check at the end of the block. At this moment, its value must be 0 otherwise it means that an instruction has been skipped or replayed. This countermeasure is summarized in listing 4.1.

```
N = I+1;
instruction0;
N--;
instruction1;
N--;
.
.
.
instructionI;
N--;
if(N != 0)
    error();
```

Listing 4.1: Step counter countermeasure on a block of I instructions

4.5.5 *Sensors*

Another way to protect a device or an implementation against FAs is to detect that there is a fault attempt. As mentioned in introduction, perturbation attacks rely on pushing the device out of its nominal operation point. Therefore, a proposed solution is to integrate sensors in the device which will detect if there is any abnormal perturbation targeting the device.

Nowadays, there is a sensor for almost every injection mediums: EMPs [152], supply voltage [153], temperature [154], light [155], FBBI [156] and even a (patented) package penetration sensor [157]. However, sensors are tough to calibrate as they aim at detecting with the best probability a perturbation attempt while having the lowest error rate possible. An important issue is that a perturbation might be detected without it to be malicious. Also, the physical environment of a chip depends of its usage and it can change during its lifetime making it very complicated to efficiently calibrate a perturbation sensor.

Finally, as the possible reaction of device detecting a perturbation is to destroy itself, it is possible to realize Deny Of Service (DOS) attacks.

4.6 CONCLUSION ON PERTURBATION ATTACKS

Perturbation attacks are an important threat for digital devices. They can modify the behavior of a device during its execution and therefore, break security mechanisms or break cryptographic algorithm security.

Despite they are an efficient way for attacking, they are not straightforward to understand, realize nor exploit. For this reason many works focus on discovering way to inject faults, characterizing their impact on the devices behavior, analyze this impact on secure implementations and either exploit them or protect against them. All these topics are a research field on their own showing the diversity of high level skills needed for fully understanding fault attacks.

The consequence is that, despite their efficiency, fault attacks are difficult to exploit and to mitigate, making the design and development of secure devices resistant to fault attacks a highly skilled process.

Part II

CONTRIBUTION

...but always keep going.

5 FAULT EFFECT CHARACTERIZATION ON SYSTEMS ON CHIP

When the world shoves you around, you just gotta stand up and shove back.
It's not like somebody's gonna save you if you start babbling excuses.

— Roronoa Zoro (One Piece)

ABSTRACT

This chapter presents my contribution regarding the fault model characterization on SoCs. The introduced elements are the SoC model, the attacker model, the device setup with the test codes and the tested targets, the perturbation benches and the analysis tools. A paper based on this work was published in WISTP 2019 Conference [158].

Contents

5.1 SoC modeling 66
5.2 Attacker model 66
5.3 Experimental method 67
5.3.1 Top-down approach 67
5.3.2 Target setup 68
5.4 Determining the faulted element 73
5.5 Conclusion 75

As this thesis aims at evaluating the security of SoCs against FAs, the first step consists in characterizing how these devices are perturbed as shown in figure 16.

However, because SoC are complex and embed many modules, my contribution only focus on characterizing the CPU behavior against FAs. As mentioned in section 2.3.2, the CPU is the cornerstone of the device and therefore an interesting target.

The characterization relies on a model of this target. The model provides information about the behavior of the device and therefore the elements that can be perturbed. The role of the characterization, is to determine which elements are perturbed and how.

Therefore, this chapter presents the modeling of a SoC, the attacker model we consider (and therefore our analysis frame), the top-down approach we use for characterizing fault effects and the target setup.

5.1 SOC MODELING

A modern CPU architecture is presented in figure 10 (section 2.3.2.1) and this model is used for our characterization process. The important things to have in mind are that:

- the CPU embeds a pipeline which fetches, decodes and executes instructions;
- the pipeline is connected to a bank of registers storing the data manipulated by the instructions;
- data and instructions are read from a memory subsystem composed of cache memories. These cache memories can either store data, instruction or both;
- the data and instructions addresses used by the core are virtual and translated by the MMU during memory accesses;
- everything is connected *via* buses.

All these Micro-Architectural Blocks (MABs) are involved during a program execution and can be perturbed during a FA. Our aim is to determine which elements are perturbed regarding different injection mediums and how.

5.2 ATTACKER MODEL

Regarding the CPU architecture, one cannot access and debug every MABs. Considering modern SoCs are embedded in smartphones, a realistic attacker aiming at using fault for weakness its security will start by characterizing the fault effects on a smartphone he bought itself.

In this situation, the attacker access to the device is limited to flashing the OS and executing self-made programs. As this thesis is a first step in SoC characterization, we consider an attacker which can only execute its programs on the target to characterize the device behavior.

As a consequence, our method match this constraints as we only execute user level programs. In this chapter we present the method using such programs and in the following chapters we demonstrate that this method able to deduce information about perturbation effects and the this information is relevant for attacking security mechanisms.

However, to challenge the method we present, we also realized a micro-architectural characterization using a custom OS and Join Test Action Group (JTAG).

5.3 EXPERIMENTAL METHOD

As mentioned above, the experiment method for fault characterization is based on the execution of user level programs. However, it aims at providing a information about the ISA and the micro-architectural behaviors of the target. To reach this objective, the method is based on a top-down analysis we link the MABs to the program elements (the data and the instructions) and discriminate the affected MABs using various test program.

5.3.1 *Top-down approach*

During a Fault Injection (FI), one or several MABs are disturbed. As they can all be perturbed during a fault injection, the full fault effect characterization can be a complicated process. However, according to the previous works on SEs, in most cases, the fault affects only a single MAB, like the cache or the pipeline. We actually verified this assumption on modern CPUs. Under this simplified paradigm, the fault characterization problem aims at determining which MAB is faulted and how.

To reach our objective, the proposed method consists in realizing a fault during a test program execution and in determining the micro-architectural fault that can explain the observed misbehavior.

Summarized in figure 35, the general idea is to apply a top-down approach. We start by determining whether the fault affects the data or the instructions, this corresponds to the ISA level. Once we know which element is affected, we determine which of its MABs is faulted. To discriminate which element is faulted, we repeatedly execute the same instructions on a known state CPU.

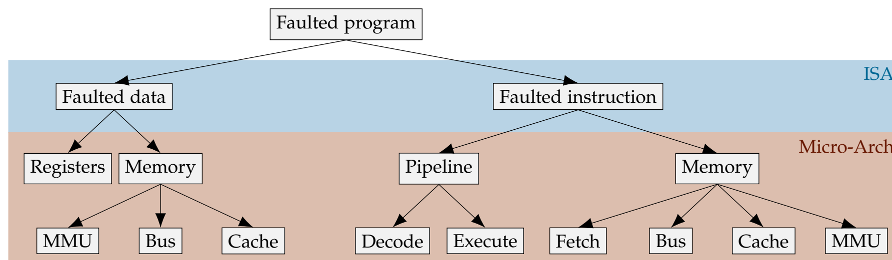


Figure 35: Fault effect characterization overview

Disturbing the program execution will give a distribution of faulted values. The next step consists in determining whether these faulted values come from a fault on the manipulated data or on the instructions.

Once we know if the data or the instructions are faulted, it is possible, from the fault model on these elements to determine which MABs have been faulted. In the case of a register corruption, it is straightforward that the registers are faulted. In the case the wrong instruction is fetched from the memory, either the cache has loaded the wrong data or the MMU has failed the address translation. If an instruction corruption is observed, the fault affects either one of the pipeline MABs or the cache or the instruction bus.

5.3.2 *Target setup*

For the fault model characterization, two important things must be defined: the test program and the initial values of the CPU registers. Each test program consists in the repetition of an instruction. However, as they execute on a complex system, they are wrapped in a program as described in figure 36.

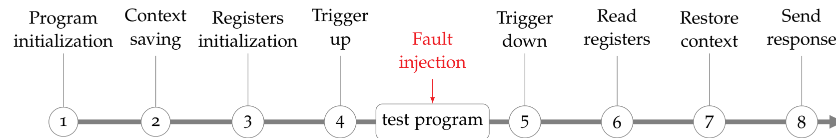


Figure 36: General program organization

This figure shows the program wrapping the test program. It aims at initializing the processor state before executing the test program and to send back the relevant data (usually the registers values) and processing the triggers. Also, as it has to communicate with an external component (usually the PC managing the experiment), it must save its execution context and restores it after the test as we do not want this part of the program to be faulted.

5.3.2.1 *Test programs*

The method for determining a fault model consists in executing a chosen program on a known state. By executing the program, the state will change in the succession of states. Perturbing the execution might create a fault. Then a state in the middle of the execution will be faulted, and all the successive states will also be faulted.

To determine a fault model, the method consists in comparing the expected final state with the faulty one. Repeating the comparison with various initial states or programs allow to determine the fault model.

This approach is effective but presents an issue. Once a faulty state appears in the computation, the program continues. This induces that the faulty final state is not a consequence of the fault, but a consequence of the fault and the execution of the program on it as shown

on Figure 37. However, in the ideal case, we want to compare the first faulty state with its expected state without it being modified by the execution of the program.

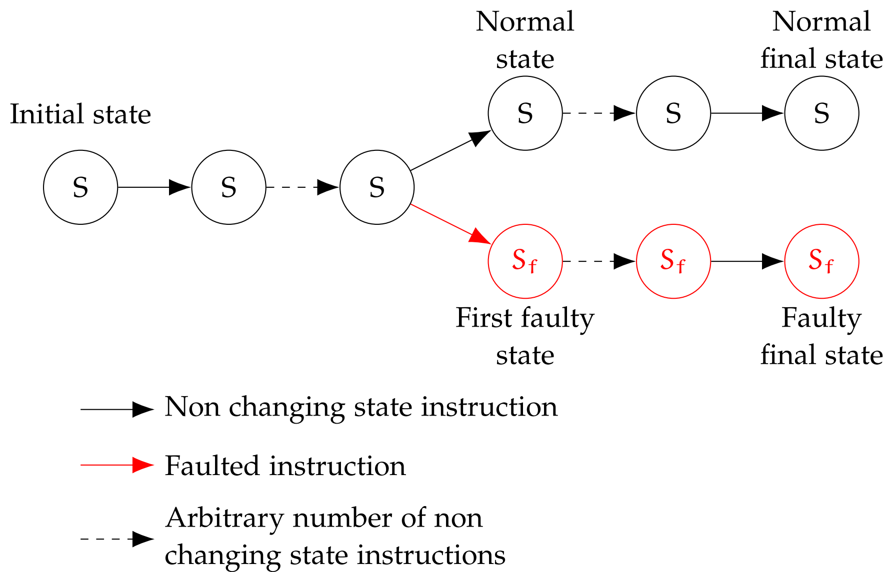


Figure 37: Succession of states during the execution of a program containing only non-changing state instructions

A workaround of this issue is to choose the program carefully. From now, the only constraint on the instructions to be executed is that they must only work on the registers. To avoid the aforementioned issue, we will choose programs executing instructions that do not change the state. Therefore, when such a program will execute, no change will be observed on the state. This means, if after the perturbation the state has changed, it's due to the perturbation. In our case, either the registers have been directly modified or the instruction has been. And, best of all, when a fault happened, it will not be modified by the program as shown in Figure 37. Therefore, the faulty final state is exactly the same as the first faulty state which is the ideal case to determine a fault model.

The good news is that it is possible to create a lot of such programs as it is possible to have a lot of non-changing state instructions. The most common is the `nop` instruction, however, as sometimes the fault depends from the instruction it is interesting to vary the instructions used in the test programs. Therefore, the instructions we use are `nop` equivalent instructions such as `mov rX, rX`, `and rX, rX` or `or rX, rX` where `rX` is a register from the processor state.

Using these kind of instructions, it is possible to create a test program such as shown in listing 5.1.

This test program, and this kind of test program in general, presents many advantages. The first one is that it uses very generic instructions, therefore it is usable on any processor architecture. The second one is that as we can arbitrary choose the number of time we repeat the instruction, we can make it large enough to have a wide injection window even on the fastest targets. The third one is that as the instructions do not change the processor state and that they are all

```
or r3, r3;
/*
 * Arbitrary number of repetitions
 */
or r3, r3;
```

Listing 5.1: Example of a test program for fault characterization

the same, it does not matter which one is perturbed, avoiding any synchronization issue. Finally, these instructions are "data processing instructions" which means that they do not fetch data from memory. Regarding our method presented in figure 35, this reduce the analysis frame as, in this case, the "memory" branch from the "faulted data" node cannot be reached.

However, this test program also presents some drawbacks. The first one is that, as it does not fetch data from the memory, we are not able to determine if the fault targets this path. This test requires another test program presented in listing 5.2.

```
str r2, [r4];
ldr r2, [r4];
/*
 * Arbitrary number of repetitions
 */
str r2, [r4];
ldr r2, [r4];
```

Listing 5.2: Example of a test program for fault characterization on memory accesses

This test program is a bit different from the previous one as it is based on two instructions. The reason is that memory access instruction are always changing state instructions. For instance, if the value in `r2` is different from the value at the address stored in `r4`, a load or a store instruction will erase this difference. The consequence is that if we only do store instructions, a fault in the memory will be overwritten, while if we only do load instructions, a fault in the registers will be overwritten. A solution could be to do two experiments, one with stores and one with loads. The test program presented in listing 5.2 allow us to determine if there is a fault in any case with one experiment. This is a consequence of the fact that the repetition of the store and the load is a non-changing state sequence of instructions.

The second drawback of the test program presented in listing 5.1 is that it is not able to detect instructions skips or repetitions. As these are a common fault model, it is important to take them into account. The detection of such faults is usually done by faulting a test program incrementing a register such as presented in listing 5.3.

This program is able to simply determine if an instruction, or several, was skipped or repeated by checking the value of `r8`. However, this test program presents some issues. The first issue is that some

```
add r8, #1;
/*
 * Arbitrary number of repetitions
 */
add r8, #1;
```

Listing 5.3: Example of a test program for instruction skip/repetition characterization

instruction corruptions can be considered as a skip or repetitions, in particular in the case the immediate value `#1` is forced to `#0`, `#ff` (-1) or any other value or if the incremented register is modified. Therefore, this test program must be used after it has been determined that this kind of faults does not appear. This is easily done using the test program presented in listing 5.1.

The second issue of the program presented in listing 5.3 is that the used instructions are no non-changing state instructions. The consequence is that if a different fault than instruction skip or repetition is achieve, it will be harder to identify it than with the first test program presented in listing 5.1.

The third issue of the test program presented in listing 5.3 is that the number of increments is limited by the register size to avoid overflows. This is more an issue on low-width processors and only impact instruction repetitions but it is important to keep it in mind.

The final issue of the test program presented in listing 5.3 is that it is not able to detect instruction repetition with replacement fault models. In this fault model, an instruction (or several) is repeated while another one (usually the immediately following one) is skip instead. Regarding the test program in listing 5.3, repeating a `add r8, #1` instruction which replaces a `add r8, #1` does not induce a fault. Therefore, the solution is to vary the incremented registers such as presented in listing 5.4.

```
add r5, #1
add r6, #1;
add r7, #1;
/*
 * Arbitrary number of repetitions
 */
add r5, #1;
add r6, #1;
add r7, #1;
```

Listing 5.4: Example of a test program for instruction repetition with replacement characterization

This program is able to determine if instruction where replaced by others simply by checking the registers values. However, most of the time, not only one instruction is replaced but several, this comes from the fetching of a pack of instruction. Therefore, this program

might not be able to detect large packs of instructions replaced as it is limited by the number of available registers.

With these test programs, it is possible to determine all the known fault model at the ISA level targeting data processing instructions and memory accesses instructions. However, these programs does not give information about branch instructions. The reason is that these instructions are complicated to analyze as a fault affecting them can lead to the execution of unknown code. However, data processing instructions and memory accesses instructions compose an important part of programs, making their analysis relevant.

5.3.2.2 *Target initial values*

To have a simple fault characterization, the initial values must fit some properties. They must be all different (equation (27)) and any logical or arithmetical simple operation between them must be identifiable (equation (28)). In other words, if we observe a faulted value $v \in [\![0, 2^n - 1]\!]$ where n is the bit size we consider, we want to be able to determine the involved initial values and the logical or arithmetical operation without any ambiguity.

$$\forall (i,j) \in [\![0, k-1]\!]^2,\ i \neq j \Longrightarrow x_i \neq x_j \tag{27}$$

Equation: Difference between all registers initial values property

$$\forall v \in [\![0, 2^n - 1]\!],\ \exists! (i,j,o) \in [\![0, k-1]\!]^2 \times O \text{ such as } o(x_i, x_j) = v \tag{28}$$

Equation: No arithmetical links between registers initial values property

where x_i is the ith initial value, k is the number of registers and O the set of operations we consider.

To match most of the modern CPUs the operations we consider for our initial values are the logical `OR`, `AND`, `XOR` and the arithmetical `ADD` and `SUB`. We realized the characterization on several targets and our constraints are $n = 32$ or $n = 64$ and $k = 10$ or $k = 16$.

Create a set of initial values matching the properties presented in equations (27) and (28) with these parameters is not complicated, randomly generating the values work. However, randomly generated values are not easy to analyze for a human, so we built the x_i as presented in equation (29).

$$x_i = 2^i + 2^n - 2^{n/2} - 2^{i+n/2} \tag{29}$$

Equation: Registers initial values construction

The corresponding initial values for our registers are presented in table 1. However, despite the fact these values fill the properties presented above and are easily identifiable for a human, their construction impose a constraint on the number of observable registers which is $k \leqslant n/2$. In other word, the number of register we can monitor must be lower than half of the bit size of these registers.

Register	Initial values (32 bits)	Initial values (64 bits)
r0	`0xfffe0001`	`0xfffffffe00000001`
r1	`0xfffd0002`	`0xfffffffd00000002`
r2	`0xfffb0004`	`0xfffffffb00000004`
r3	`0xfff70008`	`0xfffffff700000008`
r4	`0xffef0010`	`0xffffffef00000010`
r5	`0xffdf0020`	`0xffffffdf00000020`
r6	`0xffbf0040`	`0xffffffbf00000040`
r7	`0xff7f0080`	`0xffffff7f00000080`
r8	`0xfeff0100`	`0xfffffeff00000100`
r9	`0xfdff0200`	`0xfffffdff00000200`

Table 1: Target initial values for fault characterization considering ten registers

The issue with these values is that in the case of the execution of test programs for instruction skip and repetition characterization (listings 5.3 and 5.4), the registers overflow can be reached quickly. With these programs, it can be interesting to simply initialize the registers to 0.

5.4 DETERMINING THE FAULTED ELEMENT

To determine the faulted MAB, we rely on the available registers observation and the executed instructions knowledge. The way they are faulted gives information about the faulted element.

Disturbing the program execution will give a distribution of faulted values in the registers. The next step consists in determining whether these faulted values come from a fault on the manipulated data or on the instructions. Indeed, the execution of the nth program instruction by the CPU can be modeled such as in equation (30):

$$s_{n+1} = ins_n(s_n) \tag{30}$$

Equation: State computation in a CPU

where s_{n+1} is the CPU state after the execution of the nth instruction ins_n. An instruction is composed of three elements: an opcode encoding the operation to do, a reference to the destination register and reference(s) to the operand(s). These operands can be registers

or immediate values. Depending on the architecture, the encoding of this information may vary but they are always present.

When there is a fault during an instruction execution, we assume here that it either applies on the data or on the instruction. We experimentally verified this assumption. Therefore, the faulted instruction execution can be modeled such as in (31).

$$\tilde{s_{n+1}} = \tilde{ins}(\tilde{s_n}) \tag{31}$$

Equation: Faulted state computation in a CPU

where $\tilde{x}$ denotes the faulted representation of x. From this representation, we can define the fault model f_{data} on the data as introduced in (32), and the fault model f_{ins} on the instruction as presented in (33).

$$\tilde{s_n} = f_{data}(s_n) \tag{32}$$

Equation: Data fault model

$$\tilde{ins} = f_{ins}(ins) \tag{33}$$

Equation: Instruction fault model

These fault models can have different descriptions to match with the different underlying fault causes. The data fault types and their corresponding MABs are presented in table 2.

Faulted element	Data				
Fault type	Register corruption	Memory corruption		Bad fetch	
Faulted MAB	Registers	Cache	Data bus	Cache	MMU

Table 2: Data fault models

Based on the CPU model and table 2, it is possible, from these fault types, to determine which MABs has been faulted. In the case of a register corruption, it is straightforward that the registers are faulted. If there is a memory corruption, the cache storing the data or the data bus is faulted. In the the “bad fetch” case, either the cache has loaded the wrong data or the MMU has failed the address translation.

For the instructions, the fault types, presented in table 3, are corruption and bad fetch.

Faulted element	Instruction				
Fault type	Corruption			Bad fetch	
Faulted MAB	Pipeline	Cache	Bus	Cache	MMU

Table 3: Instruction fault models

If an instruction corruption is observed, the fault affects either one of the pipeline MABs, the cache or the instruction bus. In the case of a bad fetch, either the instruction cache has loaded the wrong instruction or the address translation has failed.

Regarding the test code presented above, the data fault models "memory corruption" and "bad fetch" cannot appear as there is no data fetched from the memory. Therefore, we can focus on the remaining fault models and this is enough for determining which element among the registers, the pipeline or the memory has been faulted.

5.5 conclusion

This chapter presented the method and the theory about the fault effect characterization on SoCs.

Despite several methods exists since a long time, this work is the first synthesis on fault characterization on a digital device. Indeed, even if it was presented in the scope of SoCs, this method perfectly fits with the analysis of a simpler device such as a SE. The main difference will come from the model of the target, but a model of a SE is presented in figure 4.

With this method, we were able to characterize faults at the ISA and micro-architectural level. Therefore, the next chapter presents how we applied this method on SoCs.

6 EXPERIMENTAL WORK

Hard work and efforts never betray.

— Gaï Maito (Naruto)

ABSTRACT

This chapter presents our experimental setup and the characterization results on the three chosen targets: the BCM2837, the BCM2711b0 and the Intel Core i3-6100T. The characterization aims at defining the optimal injection parameters and the associated effects at the ISA and micro-architectural level for each target.

Contents

6.1 Practical work setup 78
- 6.1.1 Attack benchs 78
- 6.1.2 Evaluated devices 81
- 6.1.3 Tools 85

6.2 BCM2837 characterization 91
- 6.2.1 Hot-spots maps 92
- 6.2.2 Analyzer results 94
- 6.2.3 Micro-architectural analysis using a test program 100
- 6.2.4 Micro-architectural analysis on a baremetal setup with JTAG 104
- 6.2.5 Conclusion on the BCM2837 characterization 107

6.3 BCM2711b0 characterization 108
- 6.3.1 Hot-spots maps 108
- 6.3.2 Analyzer results 109
- 6.3.3 Conclusion on the BCM2711b0 characterization 113

6.4 Intel Core i3 characterization 114
- 6.4.1 Hot-spots maps 114
- 6.4.2 Analyzer results 116
- 6.4.3 Conclusion on the Intel Core i3-6100T 119

6.5 Characterization conclusion 119

As presented in chapter 4, characterization is the former step in evaluating and securing devices against fault attacks. As we aim at evaluating the security of SoCs, we need to realize this characterization. The method we use for characterizing fault effects was presented in chapter 5. In this chapter, we present the practical application of this method by introducing our experimental setup, the targets we worked on and the characterization results we obtained on them.

The work presented in these chapters are part of several papers published at WISTP 2019 [158] or submitted for publication [159, 160].

6.1 PRACTICAL WORK SETUP

Before presenting our experimental results, it is important to introduce the tools we used for realizing our characterizations. As fault injections are probabilistic attacks and therefore, time consuming, the bench aims at autonomously managing fault injection campaigns on the Device Under Test (DUT).

6.1.1 *Attack benchs*

A generic attack bench is presented in figure 38.

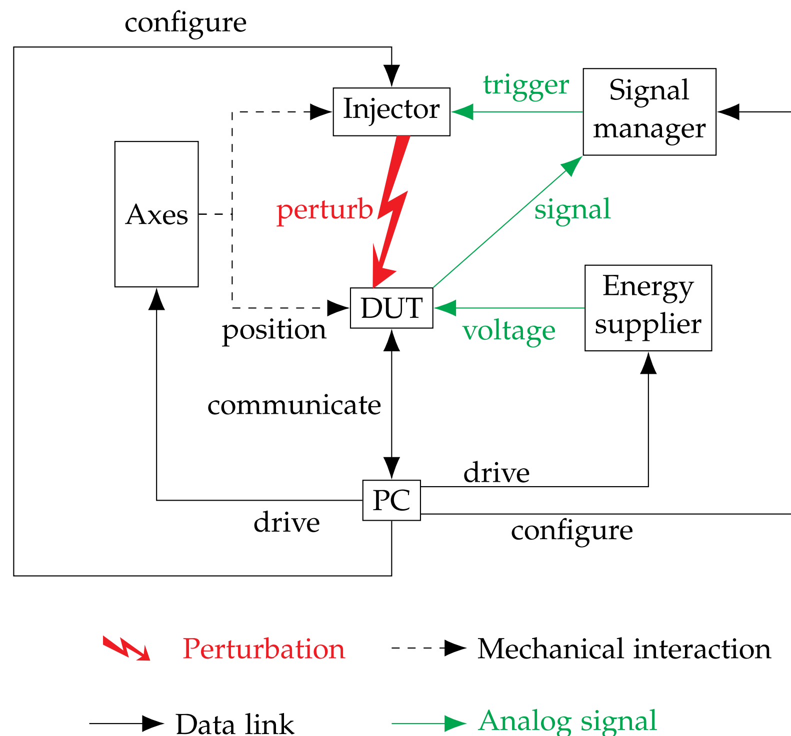


Figure 38: Generic attack bench organization and interactions

As the bench aims at perturbing the DUT, it has a central role in its organization. The DUT executes the test code we want to fault.

The perturbation is induced by the injector, there are different injectors regarding the injection medium, and, in the case of software induced fault attack, the injector is the DUT itself. The injector can be

configured to tweak the different injection parameters. These parameters vary from a medium to another. During this thesis, we focused on EM and laser perturbations.

In some cases, axes are needed to move either the DUT or the injector. This able to test different positions during EM, laser or FBBI injections. This is specific and quite time consuming as it implies a mechanical movement. Being able to optimize the path of the injector over the DUT can therefore be very time saving.

An important component of the bench is the energy supplier. Indeed, as the bench aims at being autonomous, it must manage the case where the perturbation crashes the DUT. When the DUT is crashed, a reboot of the system is needed and it is done by the energy supplier.

Another important component is the signal manager which determines when doing the injection and sends a trigger signal to the injector. In the ideal cases, the trigger signal directly comes from the DUT and is transmitted to the injector. However, in some complicated cases (on a closed platform for instance), such signal is not available. Therefore, the signal manager must analyze the DUT behavior and, *via* pattern recognition for instance, generates the trigger signal for the injector. This component is very important in current attack benchs as its performances (speed, success rate, jitter and accuracy) able a precise synchronization with the device. This component is where the most added value of an attack bench comes from. In our case, we always managed to output a trigger signal so we simply use an oscilloscope.

The last component is the PC, it aims at managing the fault injection campaigns, configures every components, drives the necessary elements and stores the results of the experiments. It keeps communicating with the DUT when it is possible to ensure that everything is going as expected.

Also, most of the time, an oscilloscope is added to acquire and observe the analog signals.

6.1.1.1 *ElectroMagnetic bench*

The first bench we used is the EM bench presented in figure 39.

This bench is composed with all elements introduced in figure 38. On this figure, the DUT is an Intel Core i3. The other evaluated targets are presented in the following section. As it is an electromagnetic bench, a probe is placed over the DUT to generate the EM waves. This probe is home-made and is composed of a copper wired rolled around a ferrite.

The energy supplier, which manages the reset of the DUT, is composed of an Arduino board.

The signal manager is handled by an oscilloscope, which also acquire the analog signals mentioned in figure 38. For the characterization, as we master the targeted program, we configure the DUT so it outputs a trigger signal just before starting the test program. Therefore, we do not need any pattern recognition mechanism in our signal manager.

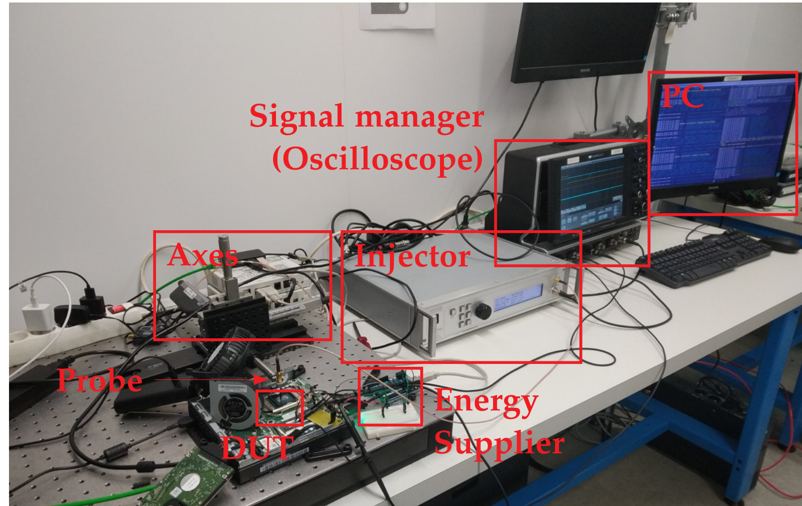


Figure 39: ANSSI's EM bench with Intel Core i3 DUT

The PC displays information about the running experiment. The communication with every bench component is done *via* a serial communication.

ELECTROMAGNETIC INJECTION SETUP. The probe is moved over the DUT *via* two-dimension axes and fed in voltage by an high voltage (800 V/16 A) AvTech pulse generator used as an injector.

The signal coming out of the pulse generator was measured using an attenuator, the results are presented in figures 40 and 41.

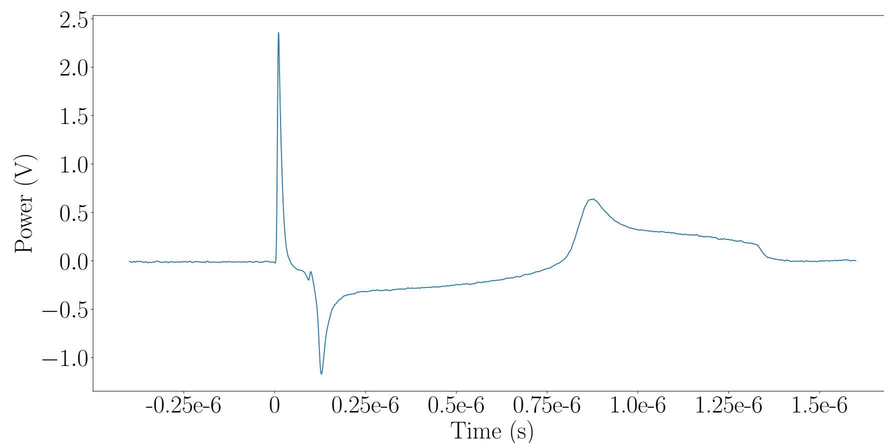


Figure 40: Pulse generated by the AvTech (100 V input)

This measure shows that the rising edge time of the first pulse is around 8.5 ns while the falling edging last for around 24 ns. However, this measures shows the signal when the output of the pulse generator (50 Ω) is connected to an adapated attenuator and an adapted scope as well. Therefore, when connected to a not necessary adapted probe, the signal might be different. A way to have an idea of the signal actually feeding the probe is to measure the probe's reflection coefficient using a Vector Network Analyzer (VNA) and compute its scattering parameters (also know as S-matrix)[161]. This approach is

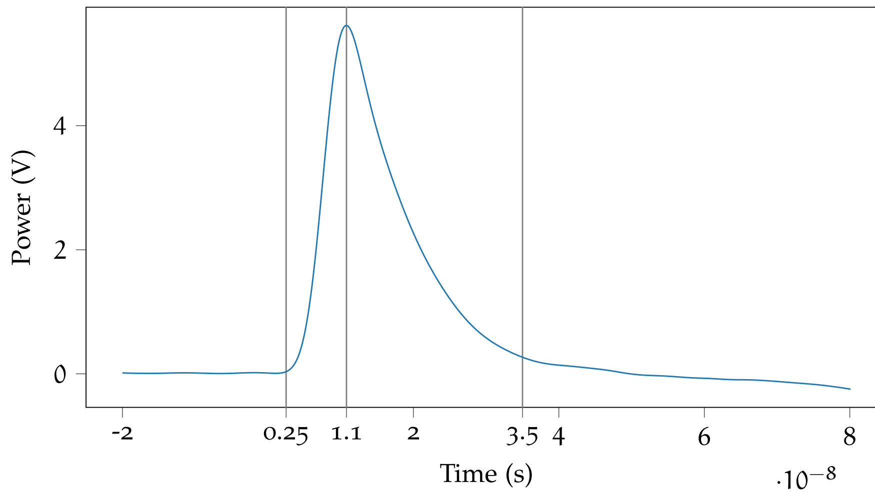


Figure 41: Pulse generated by the AvTech zoomed on the first peak (200 V input)

relevant considering the probe as an isolated system, but, during fault injections, the probe is close to the circuit and therefore it cannot be considered isolated, we are in a near-field situation regarding electromagnetic theory. In our case, instead of considering the probe as the system to characterize, we will consider the probe+circuit as the system. Indeed, during fault injections this system is isolated from external electromagnetic perturbations and we therefore can measure its scattering parameters. These parameters represent the reflection of the device for various input signal frequencies. Identifying frequencies where the reflection is minimal means the system absorption is maximal which means it is at this frequency that the system absorbs the highest amount of energy. This energy can therefore either be converted into heat *via* Joule's effect or it can influence the circuit behavior inducing faults. Doing so, we should be able the most efficient frequency to use for our injector to maximize our fault probability.

As mentioned, this approach focus on the probe+circuit system, which means it must be redone for every tested target. However, this is an on-going work and we do not have relevant results to present yet.

6.1.1.2 *Laser bench*

For LFIs, our bench is composed of a focused laser bench with a 50 x long distance lens and the wave length is around 970 nm. For being able to inject faults, the chip backside must be thinned around 150 µm. In our experiments, we used a pulse width of 20 ns.

6.1.2 *Evaluated devices*

Now that all the equipment are set up, we must determine targets to work on. As we want to evaluate the security of modern SoCs, we naturally target such devices. However, as we want to realize a characterization work, the targets must be open enough so we can develop

and execute our test programs on it. Also, we want the targets to be representative of the available SoCs on the market.

Therefore, the targets we have chosen are:

- The BCM2837 powering the Raspberry Pi 3 model B board, this board is widely used in IoT projects and has an important community.
- The BCM2711b0 powering the Raspberry Pi 4, this board is the latest version of the Raspberry Pi. It was chosen so we are able to compare the results we obtain on very similar devices (the BCM2837 and the BCM2711b0).
- The Intel Core i3-6100T powering a desktop PC, this target had been chosen to test our characterization method on a x86 architecture. This was the first time a x86 architecture device was characterized against fault injections.

With these three targets, we have a representative set of SoCs with both ARM and x86 architectures. During our experiments, these devices are powered by a Debian 9 OS. The influence of the OS is discussed in section 6.2.4.1.

6.1.2.1 *BCM2837 (ARM)*

The first target is the BCM2837 powering the Raspberry Pi 3 model B board presented in figure 42.

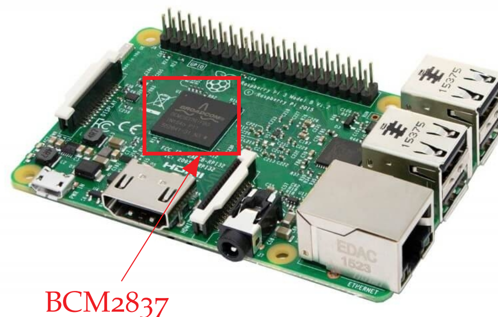


Figure 42: Raspberry Pi 3 model B board

This board embeds of 1 GB of RAM running at 900 MHz, a 40-pin header as General Purpose Inputs/Outputs (GPIOs), a 2.4 GHz wireless connection (Wi-Fi), an ethernet port, an High-Definition Multimedia Interface (HDMI) port, USB 2.0 ports, an analog audio-video jack port, a camera interface and a socket for a microSD card. Its modules show the multi-purpose orientation of its processor, the BCM2837.

This SoC is printed with a 28 nm technology, it is composed of four cores with four threads each. The cores are ARM Cortex-A53 cores and can run at a maximum frequency of 1.2 GHz. In total, this SoC has a cache memory of 512 kB. Its layout is in the figure 43.

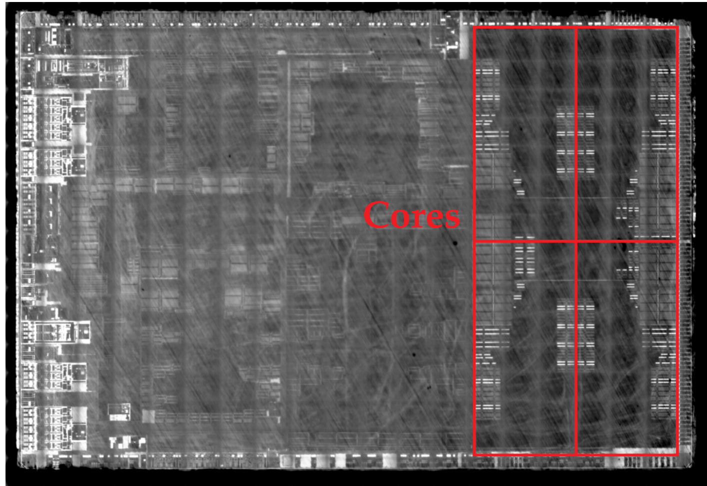


Figure 43: BCM2837 infrared backside layout image

This IR backside layout image was acquired after opening the BCM2837. The four cores are easily identifiable and covers around 25 % of the die surface. The rest of the die corresponds to the other integrated modules (GPU, video core, *etc*) and the buses.

It is interesting to notice that the die does not fill the complete chip package as shown in figure 44.

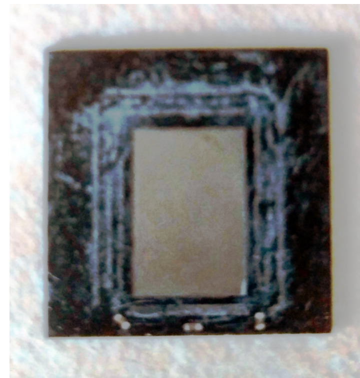


Figure 44: Open BCM2837 with the chip in its package

This figure shows that the die is placed in the center of the chip. The rest of the chip is constituted of wire-boudings used for connecting the die to the BGA. This is important as, during our experiments, we use EMPs to perturb the component over all its package.

6.1.2.2 *BCM2711b0 (ARM)*

The second target we worked on is the BCM2711b0 powering the Raspberry Pi 4 board presented in figure 45.

This board is also composed of a RAM (from 1 GB up to 4 GB), a 40-pin GPIOs as on the Raspberry Pi 3 model B, a Bluetooth and a Bluetooth Low Energy interface, a 2.4 GHz and 5 GHz Wi-Fi connections, two micro-HDMI ports, a gigabit Ethernet port and a socket for a microSD card. This is very similar with the Raspberry Pi 3 model B but with more up to date technologies.

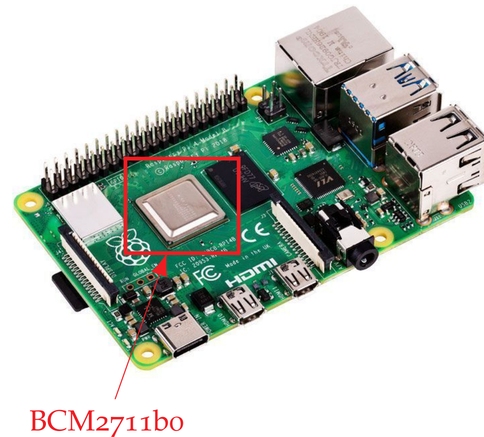


Figure 45: Raspberry Pi 4 board

The BCM2711b0 SoC, engraved in 28 nm as the BCM2837, embeds a four ARM Cortex-A72 cores with four threads each. The cores can run at a maximum frequency of 1.5 GHz. The processor integrates a total cache memory of 1 MB. Its layout is shown in figure 46.

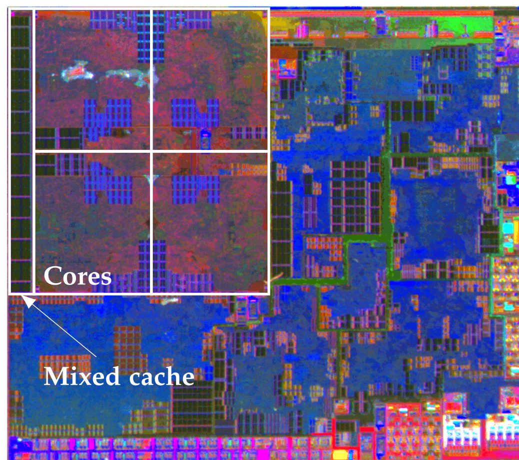


Figure 46: BCM2711b0 infrared backside layout image

This picture was acquired using infrared imaging after having removed the metal protection of the chip. The four cores are easily identifiable so is the mixed cache memory. As on the BCM2837, the rest of the die corresponds to the other integrated modules.

6.1.2.3 *Intel Core i3 (x86)*

The last evaluated target is the Intel Core i3-6100T introduced in figure 47.

It is printed in a 14 nm technology and is composed of two cores with four threads each. The maximum frequency of the cores is 3.2 GHz. The die also integrates a graphical processor thus the classical one and 3 MB of cache memory.

Figure 47: Intel Core i3 SoC

As this chip is modular, it is not link to a specific board. Therefore, for our experiments, we use a motherboard we modified to output a trigger signal when needed.

6.1.3 *Tools*

For the characterization step, we need several tools: a bench manager which drives the perturbation bench and a fault analyzer which analyze the experiments results.

6.1.3.1 *Bench manager*

The bench manager aims at driving every components composing the perturbation bench and storing the results in a `.csv` file. It is developed in Python 3 and aims at being modular. Therefore, it is organized around a `Manip` class which aims at organizing the experiments as shown in figure 48.

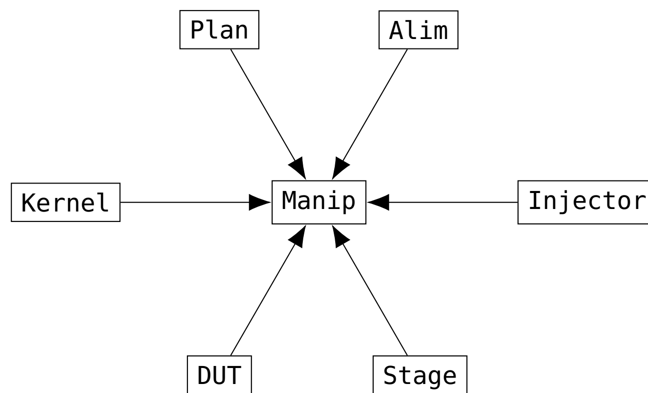


Figure 48: Bench manager software general organization

This figure shows all the modules needed by the `Manip` to realize experiment campaigns.

The `Alim` class corresponds to the energy supplier driver. It able to drive the energy supply of the DUT and therefore restart it when

needed. It usually implements methods such as `alim_on()`, `alim_-off()` and `restart()`.

The `Injector` class corresponds to the injector driver. It able to configure it to test different possible configurations *via* the `set("parameter", value)` method.

The `Stage` class corresponds to the axes driver and able to position the axes using the `set([x,y,z])` method.

The `DUT` class is the DUT driver. It implements different methods but the most important is the `start_test()` method. In some cases, it also implements a `get_temperature()` method.

The `Plan` class is the class supplying all the test parameters to use. It either stores or generates the injection parameters to use.

The `Kernel` class implements the test protocols. It implements the general progress of the experiment by organizing when to check the temperature of the device, when restart it, when check its behavior and when realize an injection test. For instance, an experiment progress is presented in figure 49 as an example.

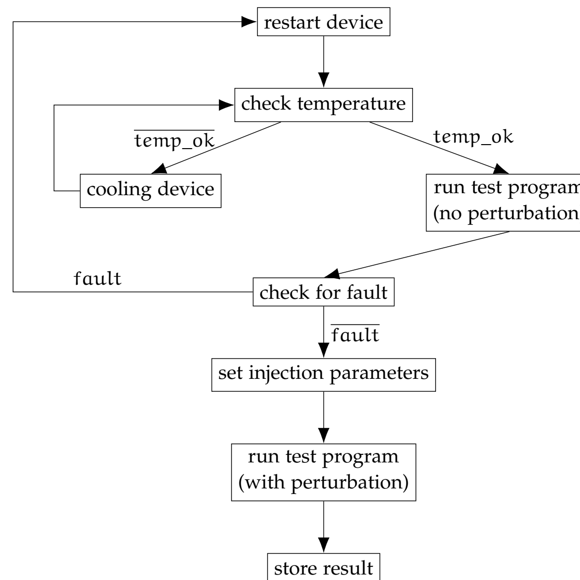


Figure 49: Example of an experiment process

This figure shows an example of an experiment process, however, depending on the situation it might be adapted to add more temperature check, check if the device is still up by running the test program without perturbation after the perturbed one, *etc*.

All the classes presented above and their parameters are initialized in a parameter file. Therefore, when we are doing a new experiment, only this file has to be edited and be passed as an argument to the bench manager process.

6.1.3.2 *Fault analyzer*

The fault analyzer is the software which helps in the analysis of the results we get from our experiments. It is also developed with Python3 and, there are two versions of this software, the old one is kept for

compatibility if the analysis of old experiments is required. The general architecture of the fault analyzer is presented in figure 50.

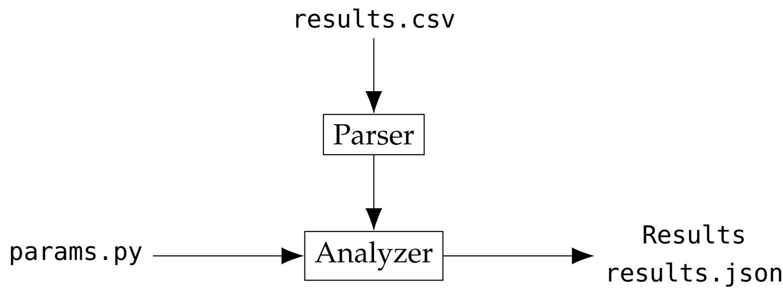


Figure 50: Fault analyzer software principle

The `results.csv` file is generated during an experiment which stores every injection attempt and the corresponding result it had on the target. The `params.py` file contains all the information needed to parse the information stored in the `results.csv` correctly, such as the initial values of the registers, the bit wide of the target, the names of the fields used in the `results.csv` file, *etc*. An example of such file is presented in listing 6.1.

```
params = {
    "default_values": DEFAULT_VALUES,
    "obs_names": ["r{}".format(i) for i in range(NB_REGISTERS)],
    "to_test": [True for i in range(NB_REGISTERS)],
    "reboot_name": "reboot",
    "log_name": "log",
    "log_separator": ";",
    "log_flag_begin": "FlagBegin",
    "log_flag_end": "FlagEnd",
    "nb_bits": NB_BITS,
    "reboot_name": "reboot",
    "coordinates_name": ["x_pos", "y_pos"],
    "carto_resolution": [7303, 6407],
    "chip_dimension": [7, 6],
    "dimension_unit": "mm"
}
```

Listing 6.1: Example of `params.py` file for the fault analyzer

This file contains the parameters for a cartography analysis done on the BCM2711b0 and presented below.

The parsing of the `results.csv` file is done via the `pandas`[1] python library. Therefore, the second input of the `Analyzer` class is a `dataframe` object from this library. As this library manage a large variety of files, we can easily interface the analyzer with other types of files if needed.

The output of the `Analyzer` is a `Results` class. A `Results` class contains a table of `Result` which is a class containing various statistical information about the experiment. There are different types

1 https://pandas.pydata.org/

of `Result` depending on the experiment specificities, the currently implemented ones are the basic `Result`, the `CartoResult` for cartography experiments and the `AESResult` for tests on AES. These classes can be used for storing (in `json`), printing (*via* a modified version of the `prettytable`[2] library) and plotting (*via* the self-developed `plotter`) the results. All the figures presenting results are realized with this tool. The `plotter` library was developed during this thesis and is available in open sources on `PyPA`[3] and on my `Github` page[4].

Also, as there are different type of results, the `Analyzer` class is designed to be modular. It can be composed of different classes integrated at the run time depending on the configuration in the `params.py` file. This modularity is implemented *via* a decorator pattern. Currently, the available modules for the analyzer are the `FaultAnalyzerBase` and the `FaultAnalyzerFaultModel`. They are also derivated in a `FaultAnalyzerCartoBase` and `FaultAnalyzerCartoFaultModel` which realize a map for every identified fault model, perturbation intensity and delay. However, as these analyses are very time consuming and have an important memory footprint. Therefore, they are rarely used and we only realize simple maps.

FAULT MODELS. Regarding the fault models, they are heuristically tested among a restricted set of fault models. The construction of this set is described in the next section. The analysis is done at the register level, *i.e.* the heuristics can identify if an observed faulted value can be obtained via the combination of registers such as the addition, the `xor` and so on. This can be done without any ambiguity because the tested default values respect the property presented in equation (28) which ensure that any simple operation between them gives a unique result.

The tested fault models are:

- **bit reset**: when the observed faulted value is all "0"
- **bit set**: when the observed faulted value is all "1"
- **bit flip**: when the observed faulted value has all its bits flipped with the initial value
- **other observed value**: when the observed faulted value is the same as one stored in another register
- **other observed complementary value**: when the observed faulted value is the complementary of a value stored in another register
- **and with other observed**: when the observed faulted value is the logical `AND` between its initial value and one stored in another register
- **or with other observed**: when the observed faulted value is the logical `OR` between its initial value and one stored in another register

2 https://pypi.org/project/PrettyTable/
3 https://pypi.org/project/plotter/
4 https://github.com/T-TROUCHKINE/plotter

- **xor with other observed**: when the observed faulted value is the logical XOR between its initial value and one stored in another register
- **add with other observed**: when the observed faulted value is the logical ADD between its initial value and one stored in another register
- **or between two other observed**: when the observed faulted value is the logical OR between two values stored in other registers
- same as above with AND, XOR and ADD

Regarding the initial values presented in table 1, we can easily determine which fault model gives an observed faulted value. Excepted when the faulted value has all its bits to 0. Indeed, in this case, no matter the initial values: $x \oplus x = 0$. Therefore, there is no way to discriminate if the fault is on forcing the register to 0 or the operand to XOR. Similarly, on some architectures (ARM at least), there exists a MVN instructions which move the not of a register into another one. In this case, we cannot discriminate if we observe bit flips directly from the register or from the instruction operand forced to MVN. For these reasons, we do not consider **bit reset**, **bit set** and **bit flip** fault models as faults on instructions.

FAULT MODELS HEURISTIC CONSTRUCTION. At first, the only considered fault models were the bit set, the bit reset and the bit flip. The other fault models were added while we were doing experiments. Indeed, the analyzer is built to store the faulted values for which it could not determine the fault model.

Therefore, after every analysis, we check this set of values, as presented in listing 6.2, to determine "by hand" new fault models to consider.

The listing 6.2 shows the faulted values which could not be explained by one of the aforementioned fault models on an experiment done on the BCM2837. The interest of this functionality of our analyzer is that it able to feed our fault model set as we realize experiments and therefore to have an incremental approach in our characterization work. In other words, our analysis is not based on a self interpretation of the results but on identified fault models which are reused from an experiment to another and not adapted to a specific case. This able to have a solid base which is mandatory to compare characterization works between them.

The addition of a new fault model is made easy and straightforward via a simple interface. All fault models are defined in the fault_models.py file. There are two types of fault models: the data fault model and the instruction fault model as shown in listing 6.3.

The name attribute corresponds to the name of the fault model while faulted_obs is the faulted observed register. In the case of an instruction fault model, the origin of the faulted value is also added.

Adding a new fault models consists in two steps:

```
 bcm2837_andR8_iv4_EM_fix_20200127 results
===========================================
Fault model unknown values
+------------+
|   Values   |
+------------+
| 0xe49de004 |
| 0xe28dd00c |
| 0xe12fff1e |
| 0xe92d000e |
| 0xe52de004 |
| 0xe24dd008 |
| 0xe28d3010 |
| 0xe1a02003 |
| 0xe59d100c |
| 0xe58d3004 |
| 0x00200000 |
| 0x000008e0 |
| 0x00000004 |
| 0x60000010 |
| 0x00100000 |
| 0x00000880 |
+------------+
```

Listing 6.2: Example of values for which the fault analyzer could not determine a fault model.

1. writing the function to test if the fault model is actually explaining the observed fault model
2. adding the fault model to the available fault models

The test function, presented as an example in listing 6.4, takes three arguments: the `fault`, the registers default values `default_values` and the bit size to consider `nb_bits`. The `fault` argument is a `Fault` class which stores the faulted value and the faulted observed register as shown in listing 6.5.

The available fault models are stored in two arrays, one for the data fault models and the other for the instruction fault models as shown in listing 6.6.

As mentioned, adding a fault model is very simple as it only consists in implementing the test function and adding the `(name, test)` information to the corresponding array. Therefore, the fault model will be automatically tested by calling the `get_fault_model(fault, default_values, nb_bits)` function which returns the matching fault model.

INTERFACES. There are two interfaces for the fault analyzer, a command line interface and a graphical one. They are presented in appendix B.

```
3  class FaultModel():
4      def __init__(self, name, faulted_obs):
5          self.name = name
6          self.faulted_obs = faulted_obs
7
8  class InstructionFaultModel(FaultModel):
9      def __init__(self, name, faulted_obs, origin):
10         super().__init__(name, faulted_obs)
11         self.origin = origin
12
13 class DataFaultModel(FaultModel):
14     def __init__(self, name, faulted_obs):
15         super().__init__(name, faulted_obs)
```

Listing 6.3: Types of fault models as defined in `fault_models.py`

```
17 def is_other_obs_fault_model(fault, default_values, nb_bits):
18     for i, dv in enumerate(default_values):
19         if (i != fault.faulted_obs) and (dv == fault.faulted_value):
20             return i
```

Listing 6.4: Or with other observed fault model test function as implemented in `fault_models.py`

6.2 BCM2837 CHARACTERIZATION

The BCM2837 powering the Raspberry Pi 3 model B board is the first SoC we characterize. Also, as it is the first, its characterization is deeper and more detailed than other targets as they were more study to confirm our hypothesis from this characterization and challenge our method.

The characterization process is organized as follow: first we determine the best injection parameters to observe faults, on the EM bench, the only parameters we can tweak are the voltage power feeding the probe, the X position and the Y position over the chip. The second step consists in realizing a campaign with fixed parameters and determine the ISA fault model using the fault analyzer. The last step aims at determining the micro-architectural elements that might be faulted regarding the ISA fault model. On the BCM2837, this was deeply studied and challenged with an analysis on a baremetal setup and the use of JTAG. This micro-architectural analysis is the topic of a paper currently submitted at JCEN2020 [159].

During a fault injection, we consider three cases:

- no fault is observed, in this case the tested program runs as if nothing happened
- a fault is observed, in this case the tested program runs normally but a value of an observed register has been modified

```
1  class Fault():
2      def __init__(self, faulted_obs, faulted_value):
3          self.faulted_obs = faulted_obs
4          self.faulted_value = faulted_value
```

Listing 6.5: `Fault` class as implemented in `fault.py`

```
92  instr_fault_models = [
93      {
94          "name": "Other observed value",
95          "test": is_other_obs_fault_model
96      },
97      {
98          "name": "Other observed complementary value",
99          "test": is_other_obs_comp_fault_model
100     },
```

Listing 6.6: Two first instruction fault models as implemented in `fault_-models.py`

- the device needs a reboot, in this case the fault effect drove the device in a state where we cannot recover any information about the registers and we need to restart it

On the BCM2837, we used two test codes based on the repetition of `orr r5,r5` and `and r8,r8` instructions as introduced in section 5.3.2.1.

6.2.1 *Hot-spots maps*

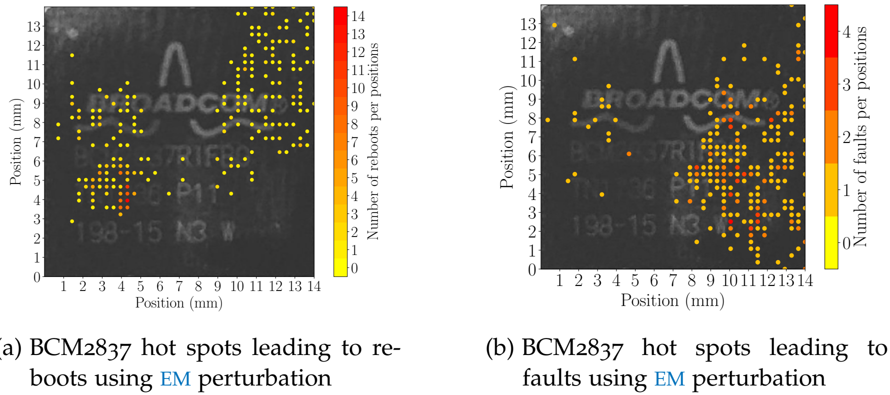
As mentioned above, our first aim is to determine the optimal injection parameters. To do so we realize two experiments. The first one aims at determining the (X, Y) position of the probe where effects and faults are obtained. The second one aims at determining the best voltage input for obtaining faults.

6.2.1.1 *Spatial location*

The hot spots maps are presented in figure 51. For this experiment, the chip was divided in a 40×40 grid and every position was tested 30 times leading to 48000 injections in total. The X-axis and Y-axis represent the position of the probe over the chip in mm.

The figure 51a displays the positions where reboots were obtained. Two areas are highlight, one on the top right and one on the left. As mentioned above, the die does not fill the whole package therefore we can see that the effects are not obtained when the probe is over the die but over the wire bounding. Also, the area on the left seems to be very sensitive with almost 50 % of reboots.

The figure 51b shows the probe positions where a fault is obtained. They do not perfectly overlay with the positions where reboots where obtained as they are more concentrated in the bottom right of the



(a) BCM2837 hot spots leading to reboots using EM perturbation

(b) BCM2837 hot spots leading to faults using EM perturbation

Figure 51: Hot spots of the BCM2837 regarding EM perturbation

chip. However, faults seems less probable to obtain as the maximum of faults we could observe for a position is 4 (13 %). However, this probabilities are not relevant with only 30 test per positions, but the difference with reboots remains interesting.

Regarding these maps, we decide to fix the probe position to $X = 11$ mm and $Y = 5$ mm for the other experiments.

6.2.1.2 *Input voltage*

The other injection parameter we want to tweak is the voltage pulse amplitude feeding the probe. For the determination of this value we tested the voltage amplitude between 400 V and 600 V with a step of 10 V and we realized 1000 for each value leading to 21000 tests. Realizing injections below 400 V does not provoke any effect and injections above 600 V leads to an very high number of reboots, which does not interest us.

This campaign was realized on both experiments (`orr r5,r5` and `and r8,r8`). The results are presented in figure 52.

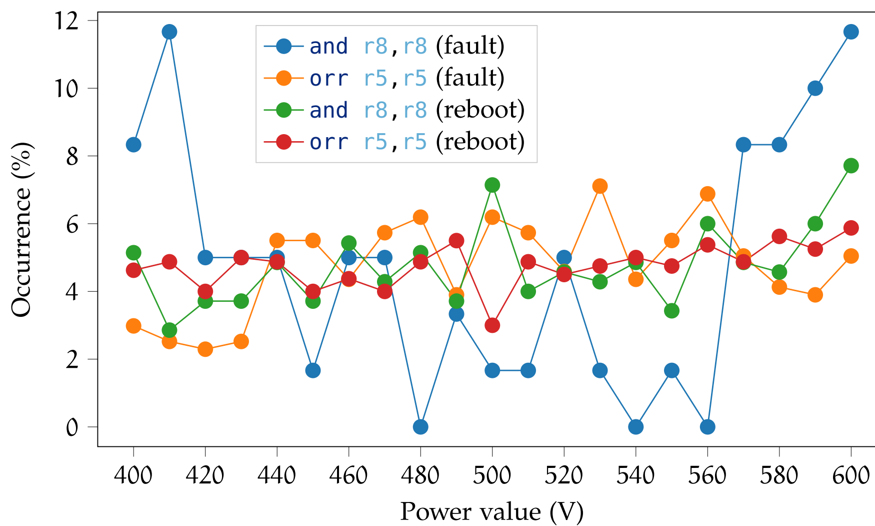


Figure 52: Input voltage amplitude effect on BCM2837 during EM perturbation

These results does not highlight a particular behavior of the chip regarding the voltage amplitude. Also, the fault probability seems to be different regarding the tested program, which is counter intuitive. However, we did not make further researches to understand this phenomena and decided to keep sweeping the input voltage from 400 V to 600 V.

6.2.2 *Analyzer results*

After having determined the injections parameters, we realize a campaign with the aforementioned parameters. Once the experiments are done, the fault analyzer presented in section 6.1.3.2 gives some statistical information we can use to determine the fault model.

6.2.2.1 *Fault probability*

One observation from these experiments is that the fault is dependent of the executed instructions. The experiments with a `orr` are faulted with a probability around 3 % while the experiments with a `and` are faulted with a probability around 1 %. As mentioned above, this behavior was not more deeply studied.

6.2.2.2 *Faulted values distribution*

An interesting information to have is the distribution of the faulted values we observed. Indeed, this helps to determine if the fault has a tendency to be random or deterministic. The results for both experiments are presented in figures 53 and 54.

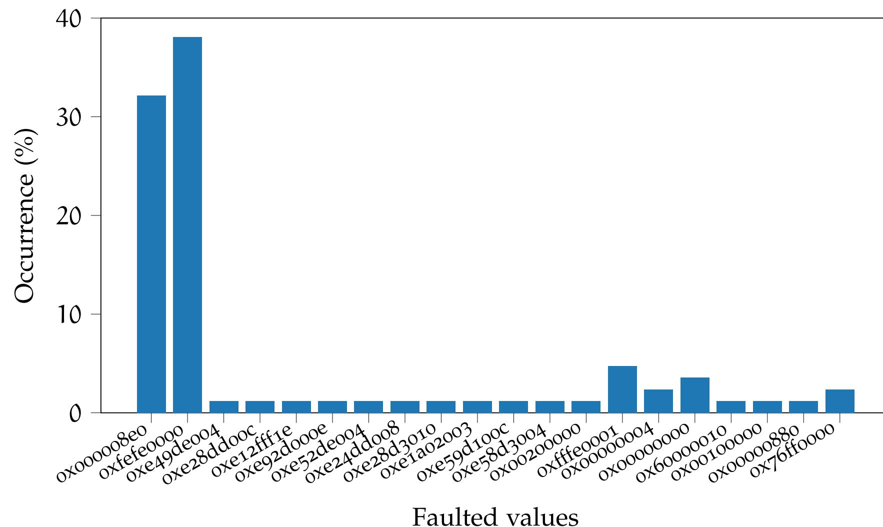


Figure 53: `and r8,r8` faulted values distribution on BCM2837 using EM perturbation

These results show that the fault effects are relatively deterministic. Indeed in both experiments, there are values that are outstandingly recurrent, *i.e.* they are observed with a probability above 35 %.

Figure 54: `orr r5,r5` faulted values distribution on BCM2837 using EM perturbation

These results give us the information that, despite being probabilistic, the fault effect is quite repeatable and therefore we aim at understanding what is its effect, or at least the most probable effect. Also, the origin of these faulted values will be discussed below while determining the fault model.

6.2.2.3 *Targeted register*

Another interesting information is the probability for a register to be faulted. The results are presented in figure 55.

Figure 55: Probability of observed registers to be faulted for both experiments on BCM2837 using EM perturbation

It shows that, in the experiments with the `orr r5,r5` instruction, the `r5` is faulted in around 87.5 % of the cases. In the case of the `and r8,r8` instruction, the `r8` register is faulted in around 65.5 % of the cases. This suggests that the most faulted register is directly linked with the executed instruction and that the fault effect is highly repeatable.

Moreover, in any experiment, the `r0` register is always significantly faulted with a probability varying between 10 % and 25 % of the cases.

Other registers might be faulted but with a probability always lower than 2 %.

6.2.2.4 *Fault model*

The fault analyzer presented above able to heuristically determine how the observed faulted values are obtained. As explained, it aims as determining simple relations between the faulted values and the initial values of the registers. The figure 56 introduces the results of such analysis on the experiments made on the BCM2837.

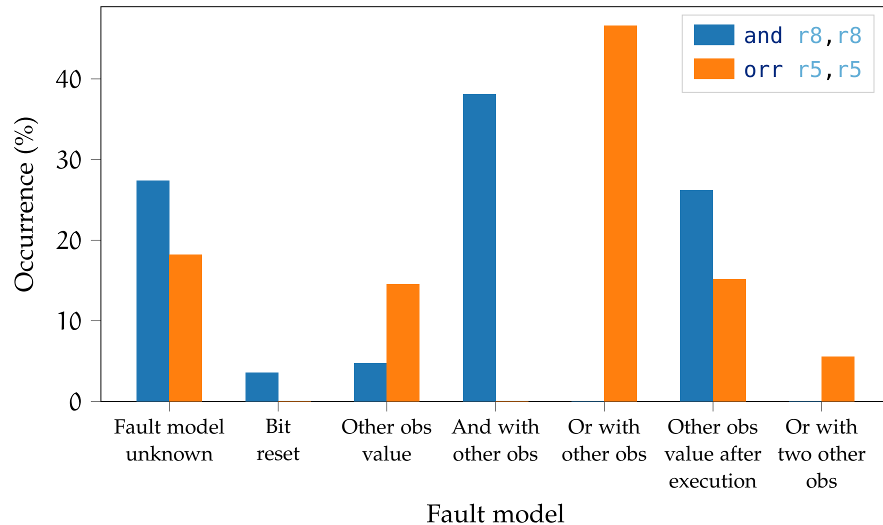


Figure 56: Probability of observing the different fault models for both experiments on BCM2837 using EM perturbation

As the analysis is heuristically done, there are some cases in which the fault analyzer could not determine a fault model. In this case the fault can be considered completely random. However in other cases, a simple relation between the values could be determine. It is interesting to note that regarding the experiments, the most probable fault model is either "And with other obs" (in the `and r8,r8` experiment) or "Or with other obs" (in the `orr r5,r5` experiment). This suggests that the fault model is directly linked to the executed instruction.

Based on these results, and regarding the method presented in section 5.3.1 with the figure 35, we want to determine if the fault targets the data (*i.e.* the registers) or the instructions. Actually, the fault targets both, but with different probabilities, our aim is to determinate how these effects behave.

6.2.2.5 *Bit reset fault model*

One observed fault model is a complete reset of the faulted register. This happen only during the `and r8,r8` experiment with a relatively low (3.27 %) probability. As mentioned above, it can be the register forced to `0` or a `XOR` between a register and itself. However, due to its low probability, we do not focus more on this fault model.

6.2.2.6 *Instruction loading and shift*

In some cases, the faulted value is computed from the executed instruction such as in equation (34).

$$v_f = (i \ \mathtt{rot}_l \ 8) \wedge \mathtt{0xfff} \tag{34}$$

with v_f is the faulted value, i the executed instruction, and $\mathtt{rot}_l$ the bit rotation to the left operation. This fault is observed with various instructions.

This fault model is complicated. As it fetches an instruction into a register, we can suppose that there is a load from the memory with the PC as base address register such as `ldr r0, PC`. This way, the value targeted by the address stored in the PC register will be loaded in `r0`. However, the left shift mechanism is difficult to explain. There exists a rolling mechanism on the operand in the ARM instruction set, for instance `ldr r0, [lsl PC, #8]`, but it does not fully explain our observations. This fault model appears between 20 % and 35 % of the cases for each experiment. This is a high probability regarding other fault models probability, however, as it is complicated, it is complicated to exploit it.

6.2.2.7 *Instruction corruption*

The other observed fault model is an instruction corruption.

Based on the figure 56, the most important fault model for the `orr r5,r5` instruction is the "Or with other obs". Regarding the executed instruction, this fault model corresponds to modify the operands or the destination of the instruction. With this fault model, the faulted register is `r5` in 100 % of the cases, meaning that the destination of the instruction is not modified. Therefore, the fault modifies an operand used by the instruction. To determine which effect the fault has on the instruction, we simply determine the instruction that explain our observations. The result is given in table 4.

Faulted instruction	Occurrence (%)
`orr r5,r1`	92.54 %
`orr r5,r0`	6.14 %
`orr r5,r7`	1.32 %

Table 4: `orr r5,r5` corruptions regarding the "Or with other obs" fault model

This table shows that the fault has a high probability to force an operand to `r1`. In some cases, the operand is forced to `r0` or `r7`.

Regarding the `and r8,r8` instruction experiment, the same kind of corruption is observed, it corresponds to the "And with other obs" fault model. Moreover, in this case there is only one corruption which happen in 100 % of the cases and the corresponding faulted instruction is `and r8,r0`. In this case, the operand is forced to `r0` instead of `r1` but the behavior remains the same. However, the reason why

we observe `r0` or `r1` is not identified, we suppose it comes from the initial register involved in the instruction but we did not confirm this hypothesis. We can say that the fault force the operand to low hamming weight values, which match the force to zero behavior of the EM injection medium.

While looking at the `orr r5,r5` instruction experiment we also observe the "Or with two other obs" fault model. It is similar to the previous one but, in this case, the two operands are modified. The results corresponding to these corruption are presented in table 5.

Faulted instruction	Occurrence (%)
`orr r5,r4,r1`	51.86 %
`orr r5,r0,r1`	48.14 %

Table 5: `orr r5,r5` corruptions regarding the "Or with two other obs" fault model

This table shows that one of the operand is always forced to `r1` while the destination remains unchanged and the other operand is either `r4` or `r0`.

The last behavior we observed correspond to the "Other obs value" in this fault model the operands are modified but also the operation code. In this case, it corresponds to a `mov` operation. Considering this operation, two information are important, the origin of the value and its destination. In the `orr r5,r5` experiment, the results are presented in table 6.

Faulted instruction	Occurrence (%)
`mov r5, r0`	46.48 %
`mov r5, r4`	30.99 %
`mov r5, r1`	21.12 %
`mov r0, r5`	1.41 %

Table 6: `orr r5,r5` corruptions regarding the "Other obs value" fault model

These results show that the registers `r0`, `r1` and `r4` already involved in the other fault models remains the ones the fault forces to use and they all are low hamming weight registers. However, the `r1` register is not the more probable in this case. We did not find any explanation for the higher presence of `r0` in this case.

Regarding the `and r8,r8` experiment, the results are presented in table 7.

Faulted instruction	Occurrence (%)
`mov r8, r0`	72.73 %
`mov r0, r8`	18.18 %
`mov r4, r0`	9.09 %

Table 7: `and r8,r8` corruptions regarding the "Other obs value" fault model

These results show some similarities with the previous ones. The fault force the registers to low hamming weight ones (`r0` and `r4`).

Regarding these experiments, we can conclude that our fault most of the time corrupts the operands of the executed instruction and force them to low hamming weight values. However, sometimes it infers a register corruption or a modification of the instruction opcode. This correspond to the ISA fault model however, as mentioned above, we are interested in the micro-architectural fault model to determine if these behaviors come from some SoC specificities and to be able to build efficient countermeasures against them.

6.2.2.8 *Number of faulted instructions.*

When working with modern CPUs with high frequencies ($> 1\,\mathrm{GHz}$), one important fault parameter is the spreading. We determined the fault affects instructions but we cannot determine how many instructions are actually faulted.

Regarding the BCM2837 CPU frequency ($1.2\,\mathrm{GHz}$) and the injector first peak duration (3.25×10^{-9} s), we can suppose that a fault perturbs around 4 ($1.2 \times 10^9 \cdot 3.25 \times 10^{-9} = 3.9$) instructions. To confirm this hypothesis we decided to fault a test program composed of the repetition of the `mov rX,rX` with $X \in [\![0, 9]\!]$. We observed that in 84.34 % of the cases, the fault corrupts the instruction, the faulted instruction becoming `mov rX,rY` with $(X, Y) \in [\![0, 9]\!]^2$.

Faulting such program gave us the information that, on average, the fault affects 1.45 instructions. As this result is different from the expected one, it can be explained by the fact that the CPU does not run at maximum speed all the time.

Regarding this result, we can focus our analysis on the corruption of one or two instructions.

6.2.2.9 *Correspondence between the fault model and the instruction encoding*

Before doing the micro-architectural analysis, it is interesting to confront the instruction corruption we observe regarding how instructions are encoded. The figure 57 presents the encoding of the data processing instructions in the ARM architecture.

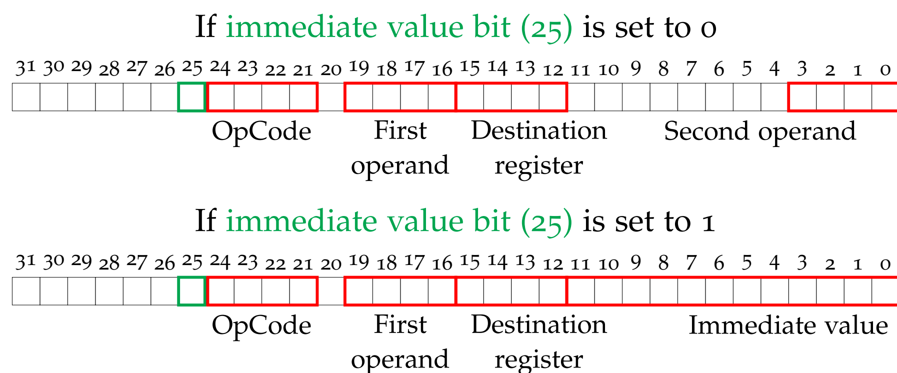


Figure 57: Data processing instruction encoding on ARM

This figure shows on which bits the information composing a data processing instruction (operands, opcode, destination) are encoded. As we know the initial instruction we execute, we are able to determine which bits are faulted by our perturbation.

As we observe a fault on the operands or the opcode, we know that our fault is equivalent to modify the bits 0 to 3 (second operand), the bits 16 to 19 (first operand) or the bits 21 to 24 (opcode).

As we mentioned above, the fault seems to set bits to 0 as we observe low hamming weight faulted values. The only counter-example is when we observed a corruption of the second operand from `r5` (`0b0101`) to `r7` (`0b0111`). Also, we observed a corruption of the opcode, and by checking the binary values of the original and the faults opcodes presented in table 8, we concluded that they are not forced to low hamming weight values.

Original opcode	Faulted opcode
`orr` (`0b1100`)	`mov` (`0b1101`)
`and` (`0b0000`)	`mov` (`0b1101`)

Table 8: Binary values of the observed opcodes.

This table shows that the faulted opcode always have a higher hamming weight than the original instruction. This observation lead to two conclusions. The first one is that the opcode and the operands are not faulted in the same way, this explain the probability difference between faulting an opcode and faulting an operand. The other one is that considering the way instructions are encoded to determine the fault model is not relevant because the fault does not affect their encoding but another micro-architectural mechanism of the core.

This second conclusion seems to be the most relevant for us, especially due to the fact we observed similar faults on another architecture (x86 presented in section 6.4) because a different architecture means a different instruction encoding.

6.2.3 *Micro-architectural analysis using a test program*

Now that we have determined the ISA fault model, we want to determine which micro-architectural blocks are faulted and can explain our observations. To do so, we realize two experiments based on test programs specifically adapted to our previous observations.

6.2.3.1 *Fault during instruction fetch determination*

We saw that the fault usually perturbs the instruction second operand. However, regarding the instruction encoding presented in figure 57, the second operand can either be a register or an immediate value. As we want to determine which part of the micro-architecture is faulted we want to test our injection setup on program working with immediate value rather than registers. Indeed, if the fault is the same in both cases, it means that it targets a part of the core which does not distin-

guish registers from immediate value, in other words, every blocks before the decoding stage of the pipeline. If the fault is only observe on the registers, it means that it targets a block which specifically works differently regarding if we use registers or immediate values. The only block doing so is the execute stage of the pipeline.

The main issue is that our test codes are not suitable for testing the manipulation of immediate values as they would overwrite a fault. Therefore, we had to realize a different kind of test program presented in listing 6.7.

```
          cmp r3, #255
          bne fault
          b nofault
fault:    mov r9, #170
          b end
nofault:  mov r9, #85
end:      nop
```

Listing 6.7: BCM2837 immediate value test program

This program realizes a comparison between the register `r3` and the value `0xff`. If they are equal, the processor will pass the `bne` instruction and then branch to the `nofault` tag. Then the register `r9` is loaded with the value `0x55` and the program terminate. If `r3` and `0xff` are not equal, then the program will branch to the `fault` tag. `r9` will be loaded with `0xaa`, and then the processor will branch to the `end` tag and terminates.

The attack aims to fault the `cmp r3, #255` instruction. As `r3` is initialized with the value `0xff`, the first scenario must happen. However, according to our fault model, we should modify the second operand (which is an immediate value in this case) and force it to `0`. Therefore, the processor will follow the second scenario. If it happens, we achieved to modify the control flow of the program with our ElectroMagnetic Fault Injection (EMFI) and actually faulted an immediate value. The observation of `r9` gives us this information.

table 9 gives the results for this experiment. This table shows that in more than 90% of the cases we effectively fault the immediate value. This experiment confirms that the fault model we determined is portable on other instructions, in particular, instruction using immediate values.

Fault	Fault (`r9 = 0xaa`)	`r9 = 0xfffcb924`	Unknown
Appearance rate	94%	4%	2%

Table 9: Fault distribution on `cmp` test code

It is interesting to note that the probability of modifying the control flow is very high. Greater than 90%. This high probability can be explained with the fact that many faults on the `cmp r3, #255` instruction can lead to the second scenario, such as first operand corruption

or opcode corruption. However, regarding our previous experiment, we are confident the consequence of corrupting the second operand.

Therefore, this observation on corrupting an immediate value gives some information on the effect of the fault in the pipeline. Indeed, pipelines are built on three main functions: the *fetch* which reads the instruction and compute the PC, the *decode* which decodes the instruction and set the flags in the pipeline and the *execute* which executes the decoded instruction and stores the result.

From these functions, only *execute* behaves differently if the second operand is an immediate value or a register. Regarding the electronic way, the multiplexer of the second operand in the ALU select either the immediate value or a register.

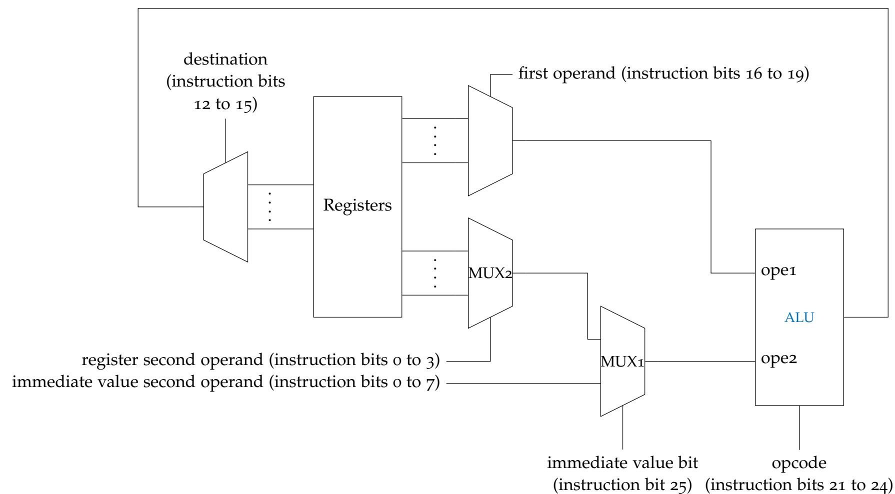


Figure 58: Pipeline execute stage architecture with ARM instruction (figure 57) corresponding bits

Figure 58 shows the big map of the pipeline *execute* stage architecture. Multiplexers are used to feed the ALU with the operands. The flags and value used are the same as the one exposed in Figure 57 and come from the *decode* stage. The multiplexer `MUX1` is driven by the immediate value bit.

Regarding our fault model, second operand corruption either it is a register or an immediate value, and based on the figure 58, we can see that it matches with corrupting the `MUX2` selector signal (modifying the register second operand) or corrupting the `MUX1` down input (modifying the used immediate value). In both cases, this matches with corrupting the lower bits of the instructions. Therefore, we can conclude that these bits are corrupted before they are decoded and used in the execute stage. This means that the fault occurs before the instruction reaches the decode stage, in other words, the fault happen during the instruction fetch and might target any micro-architectural block from the external memory to the fetch stage, including the caches.

6.2.3.2 *Fault in instruction path determination*

In this section, we aim at identifying which MAB is faulted and how. Regarding the previous experiments, we suppose that the fault affect the memory subsystem. Regarding the figure 35 in section 5.3.1, this means that the micro-architectural blocks targeted by the fault can be the fetch stage, the buses, the (L1i or L2 as we fault instructions) caches and the MMU.

To determine which one is faulted, we decide to realize a memory access experiment with the code presented in listing 6.8. Actually, this experiment is not suitable for determining information about the mixed cache memory as repeating memory accesses at the same address will make the cache copy this address and the memory access will only reach the L1 data cache memory. However, as it gives interesting information about the fault model, we keep presenting the results.

```
str r8, [r9] // Several
ldr r8, [r9] //     times
```

Listing 6.8: BCM2837 memory test code

To do so, we initialize a page of memory (4 kB), then we set the observed registers values to addresses in this page and fault the test program.

This code realizes memory loads and stores at/from the address stored in `r9` to/from the register `r8`. As the memory page is initialized with known values, the expected value in `r8` are known. Also, the ISA fault model is known, therefore we can't identify and anticipate faults that correspond to this model and focus on faults corrupting the data coming from the memory.

By faulting this program we observed a fault probability of 3.36 %, which is coherent with our previous observations.

An unanticipated fault is that with a probability around 25 %, the faulted value is the `ldr r8, [r9]` instruction encoded value. In this case, the faulted instruction is `ldr r8, [PC]` which corresponds to set the operand to `0xff`. This is the highest hamming weight the operand can have which does not correspond with our previous observations.

For the other faults (74.4% exactly), the observed faulted value is always $b_{ad} + 50$ where b_{ad} is the page memory base address. This is the value stored in `r2`. In this case, the faulted instruction is `mov r8, r2`. This is consistent with the previously determined fault model. Moreover, the fault does not only modify the second operand but also the opcode, forcing the instruction into a data processing instruction instead of memory loading instruction. As we already tested data processing instructions, we did not see this fault effect. This shows the importance of testing different types of instructions for determining the complete fault effect.

During this experiment, we did not observe faults on the fetched data. Therefore, we conclude that the fault targets the dedicated to the instruction part of the memory subsystems. This corresponds to the

L1 instruction cache, its connected buses, the MMU and the fetch MAB. However, using only test programs, we cannot more deeply determine which MAB is faulted. Therefore, we decide to use a baremetal setup with a JTAG to realize deeper characterization as introduced in [159].

6.2.4 *Micro-architectural analysis on a baremetal setup with JTAG*

This section focus on determining the fault model at the micro-architectural level on the BCM2837 using a baremetal setup (*i.e.* no OS) and a JTAG connection. This method was only be done on the BCM2837 because it is very time consuming. Indeed, unlike the method presented in section 5.3, this method consists in faulting a program, wait for a fault, stall the program when a fault appear and manually debug the micro-architectural elements such as the cache memory or the MMU. This debug step is time consuming, requires an expert each time and therefore cannot be repeated an important amount of time to confront the observation with a statistical analysis.

However, this characterization gives information about what can happen while faulting the BCM2837 and we can suppose that the fault we observe are the more likely to happen. Moreover, this characterization work was done within a collaboration with the LHS (Laboratoire de Haute Sécurité) of the INRIA of Rennes. This team developed the baremetal setup and realized the characterized experiments, however, their injection setup is not the same as ours but, as we focus on characterizing the faults and not how to achieve them, we decided to abstract this difference during the analysis. The reason is that, during this collaboration, we focused on comparing the fault model observed on the Linux and the baremetal setups.

6.2.4.1 *Baremetal and Linux setup behavior differences against EM fault injection*

This characterization did not highlight a different behavior between the baremetal and the Linux setup regarding the fault models. Without realizing a deep characterization like above, we simply present the observed fault model while faulting a test program repeating the `orr r3,r3` instruction in figure 59.

This result shows that the most observed fault model is the "Or with other obs" with a probability of 86.46 %. This perfectly match our observations on the Linux setup and we therefore can conclude that the Linux OS has no impact on the observed fault model.

6.2.4.2 *Target setup for micro-architectural characterization*

As we aim at characterizing the fault effect at the micro-architectural level, the test program presented in section 5.3.2.1 cannot help us. Therefore, we decide to fault a more complete program presented in listing 6.9.

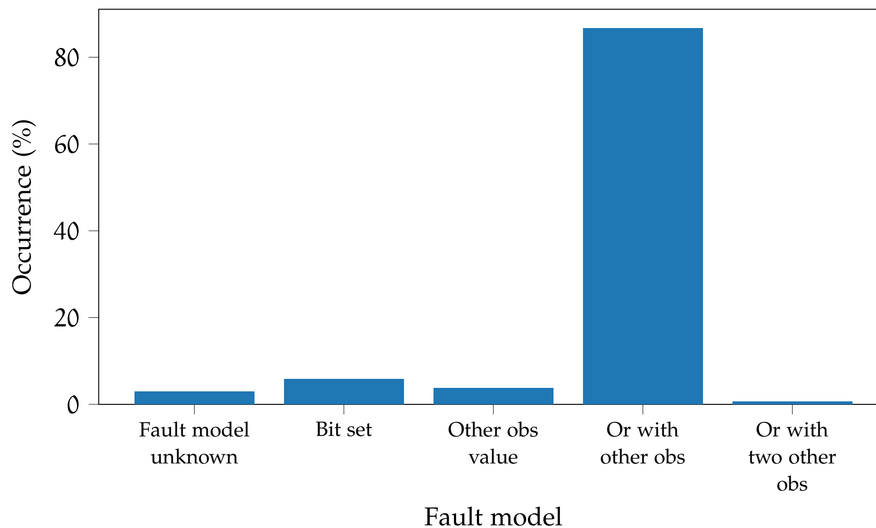


Figure 59: Probability of observing the different fault models for `orr r3,r3` experiment on BCM2837 with a baremetal setup using EM perturbation

```
trigger_up();
//wait to compensate bench latency
wait_us(1);
invalidate_icache();
for(int i = 0; i<50; i++) {
  for(int j = 0; j<50; j++) {
    cnt++;
  }
}
trigger_down();
```

Listing 6.9: BCM2837 baremetal test program

This test program is composed of two intricate loops. As we aim at verifying that we fault the instruction cache, it is invalidated before starting the loops, therefore, it is possible to precisely target the moment the instructions are fetched from the memory and loaded in the instruction cache.

Actually, by faulting at different moments of the program execution we are able to fault different micro-architectural elements: the L1 instruction cache, the MMU and the L2 mixed cache.

6.2.4.3 *Fault on the instruction cache*

A fault occurs when the `cnt` variable value at the end of the program is not equalled to 2500. Since a fault is detected, we use the JTAG to re-execute our loop **`for`**, in the listing 6.9, by directly setting the PC value at the start of the loops. Executing instruction by instruction, we monitor the expected side effects. This execution is done without fault injection. Each instructions was well-executed except for the `add` instruction at address `0x48a08` on listing 6.10.

```
...
0x48a04:  0xb94017a0       ldr       x0, [x29,#20]
0x48a08:  0x11000400       add       x0, x0, #0x1
0x48a0c:  0xb90017a0       str       x0, [x29,#20]
0x48a10:  0xb9401ba0       ldr       x0, [x29,#24]
0x48a14:  0x11000400       add       x0, x0, #0x1
0x48a18:  0xb9001ba0       str       x0, [x29,#24]
0x48a1c:  0xb9401ba0       ldr       x0, [x29,#24]
0x48a20:  0x7100c41f       cmp       x0, #0x31
0x48a24:  0x54ffff0d       b.le 48a04 <loop+0x48>
...
```

Listing 6.10: BCM2837 assembly code of the loop test program

By monitoring the `x0` register before and after the `cnt` incrementing instruction, we observe that the value is kept unchanged: the increment is not executed. Since the fault is still present after the EM injection, we can conclude that a wrong instruction value is stored in the L1 instruction cache. We confirm this fault model by invalidating the instruction cache using the `ic iallu` instruction. By re-executing our application, the fault has disappeared.

We can infer that the injected fault has affected a part of the instruction cache. However, it is impossible to access (read) the new incremented value. Since the fault happens during the cache filling, we can suppose that the memory transfer had been altered. Moreover, the ISA fault model determined above can explain the absence of increment in the case the instruction `add x0, x0, #1` is forced to `add x0, x0, #0`.

This confirms that the instruction cache can be faulted in a way that matches the fault model we determined above.

6.2.4.4 *Fault on the MMU*

During the experiments on the baremetal setup, we observed that the MMU can be faulted. As presented in section 4.4.1.2 the MMU rely on PTEs to operate. These PTEs are stored in a Translation Lookaside Buffer (TLB) which is basically a cache for PTEs. In our baremetal setup, the PTEs are configured to realize an identity mapping, *i.e.* the virtual address used by the CPU is the same as the physical address used by the RAM.

To reconstruct the memory mapping, we use a pair of instructions computing the physical address (and the corresponding metadata) for a given virtual one and a script has been designed to extract the memory mapping.

With this method, we compare the memory mappings with (listing 6.12) and without (listing 6.11) a fault.

Three different effects can be observed depending on the page:

1. Pages are correct with an identity mapping up to `0x70000`. Remarkably these are all the pages used to map our application in memory. Therefore, a hypothesis is that the corresponding

```
VA      -> PA
0x0     -> 0x0      0x80000 -> 0x80000
0x10000 -> 0x10000  0x90000 -> 0x90000
0x20000 -> 0x20000  0xa0000 -> 0xa0000
0x30000 -> 0x30000  0xb0000 -> 0xb0000
0x40000 -> 0x40000  0xc0000 -> 0xc0000
0x50000 -> 0x50000  0xd0000 -> 0xd0000
0x60000 -> 0x60000  0xe0000 -> 0xe0000
0x70000 -> 0x70000  0xf0000 -> 0xf0000
```

Listing 6.11: BCM2837 with baremetal setup with correct identity memory mapping

```
VA      -> PA
0x0     -> 0x0      0x80000 -> 0x0
0x10000 -> 0x10000  0x90000 -> 0x0
0x20000 -> 0x20000  0xa0000 -> 0x0
0x30000 -> 0x30000  0xb0000 -> 0x0
0x40000 -> 0x40000  0xc0000 -> 0x80000
0x50000 -> 0x50000  0xd0000 -> 0x90000
0x60000 -> 0x60000  0xe0000 -> 0xa0000
0x70000 -> 0x70000  0xf0000 -> 0xb0000
```

Listing 6.12: BCM2837 with baremetal setup with faulted memory mapping

translations are present in caches and are not impacted by the fault.

2. Pages are incorrectly mapped to `0x0`. A read at `0x80000` reads, with success, physical memory at `0x0`.

3. Pages are shifted. A read at `0xc0000` gives the physical memory value at `0x80000`.

If we invalidate the TLB after a fault, nothing changes: the mapping stills modified. We conclude that the fault does not affect the cache mechanism of address translation (at least what can be invalidated by software) but directly the MMU configuration.

This fault model is very powerful as it able to break the memory partitioning. However, we did not observed it at the ISA level as its impact at this layer corresponds to fetch the wrong instruction or the wrong data. Considering that we do not initialize the RAM to known value, the fault is likely to fetch a random data, this can explain why the analyzer could not determine a fault model for all faults.

6.2.5 *Conclusion on the BCM2837 characterization*

In this section we realized the characterization of faults on the BCM2837 CPU. We determined that the fault mainly modify the instruction operands by targeting the memory cache. We also determine that the MMU and the mixed cache can be affected.

The memory is known to be faulted on SE but these results able us to conclude that the large integration of cache memory make modern CPUs sensitive to faults. The next experiments will able to confirm this

observation on a more up-to-date device using a laser and even on another architecture using EM perturbation.

However, because it is time consuming, no micro-architectural debugging has been done on the following targets. We only focus on comparing the results at the ISA level and considering that if the ISA observations are the same, therefore, the micro-architectural behavior is the same.

6.3 BCM2711B0 CHARACTERIZATION

The BCM2711b0 was characterized while perturbed using a laser. The analysis process is the same as the one presented in above with the BCM2837. As we could only access a laser bench during a restricted duration, we only tested the program based on the `orr r5,r5` instructions. However, this timing constraints helped to confirm the relevance of having a well defined method we could use to analyze and characterize our device.

6.3.1 *Hot-spots maps*

As mentioned above, our laser injection setup has its parameters fixed. Therefore, the only injection parameter we have to drive is the position of the laser spot over the chip. However, as the laser spot is around 1 µm and that the chip surface is around 42 mm², this step is very time consuming. Therefore, to reduce this step, we only focused on the cores. The results are presented in figure 60.

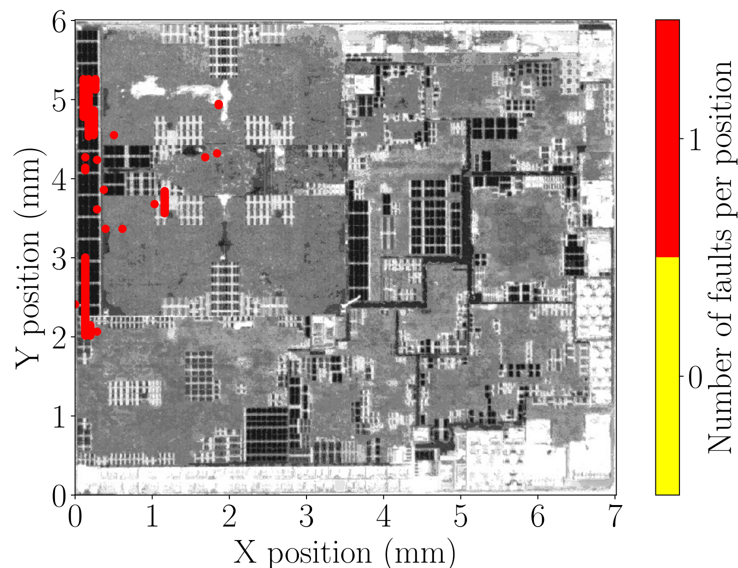


Figure 60: BCM2711b0 hot spots leading to faults using laser perturbation

It is important to note that the dot size on the figure does not match the laser spot size. Also, as these experiments were done with our bench manager, we do not have results about the reboots of the target.

The figure 60 shows that the cores are sensitive to laser fault injection and, in particular, the cache memory. This is a first interesting

information as we determined on the BCM2837 CPU that the cache is faulted. However, as we did not have much time to realize the experiments, we decided to analyze the results of the cartography experiment, this might be imprecise as we do not target a specific location where we observed an interesting behavior but it will gives information about the general behavior of the device against laser fault injection.

6.3.2 *Analyzer results*

The analyzer gives a statistical analysis of the faulted device. However, as we do not fault the same position, the probabilities must be carefully considered as if the spot was fixed at a specific location, the probabilities might be different. In particular, the fault probability is not relevant as we test different positions for the laser spot.

Despite these issues, we propose to realize the same analysis as done on the BCM2837, abstracting the fact that the experiment was actually a cartography.

6.3.2.1 *Faulted values distribution*

As mentioned above, looking at the faulted values distribution helps us to determine if the fault effect is consistent or very variable. The results for the BCM2711b0 are presented in figure 54.

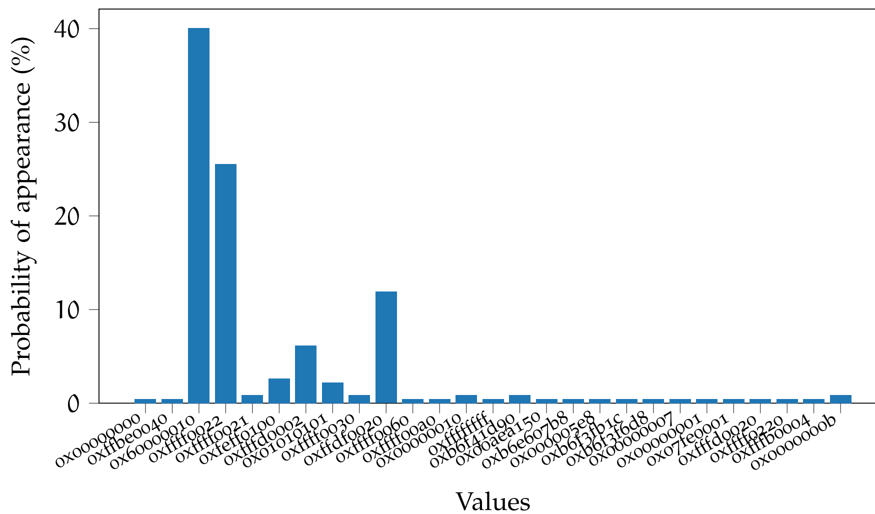


Figure 61: `orr r5,r5` faulted values distribution on BCM2711b0 using laser perturbation

These results shows that, despite we are doing a cartography, some values are very likely (between 10 % to 40 %) to appear. This suggests that the device is perturbed in a consistent way.

6.3.2.2 *Targeted register*

As with the BCM2837, another interesting information is the distribution of the faulted register. This result for the BCM2711b0 is given in figure 62.

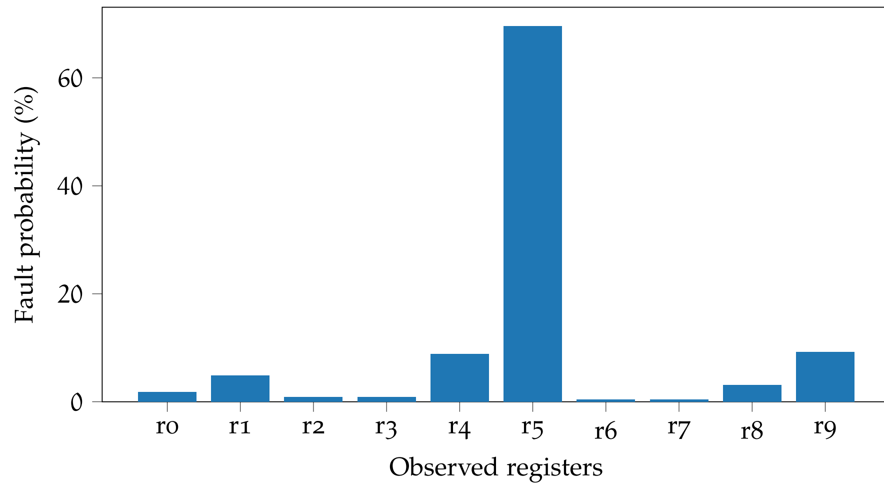


Figure 62: Probability of observed registers to be faulted for `orr r5,r5` experiment on BCM2711b0 using laser perturbation

This result shows that the `r5` register is faulted in more than 65 % of the cases. This actually match our observation while perturbing the BCM2837 using EM perturbations. This strongly suggests that the faulted register is linked with the executed instruction, which is `orr r5,r5` in this case. Moreover, we can suppose that the reason we have the same observation on both targets is the same, *i.e.* that the fault model on both targets is the same.

This hypothesis is very interesting as, if it is actually the case, this means the fault model is independent from the used injection medium.

6.3.2.3 *Fault model*

The fault model distribution observed on the BCM2711b0 is presented in figure 63.

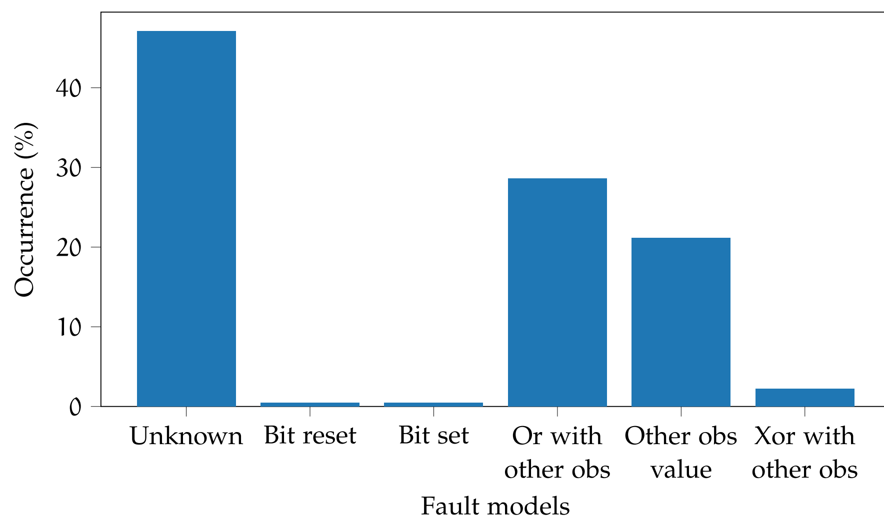


Figure 63: Probability of observing the different fault models for `orr r5,r5` experiment on BCM2711b0 using laser perturbation

A major issue one can easily identify is that the analyzer could not determine the fault model in around 50 % of the observed fault. This is quite important compared with the other experiments. However, some of them have important probability of appearance and in particular the "Or with other obs" fault model. Also, one must keep in mind that the experiment is actually a cartography and that complex behavior might come from the fact that we are testing various positions.

It is interesting to note the presence of the "Or with other obs" fault model as it correspond to the fault model observed on the BCM2837 executing the `orr r5,r5` instruction presented in figure 56 in section 6.4.2.4. This suggests that, on some positions, the BCM2837 and the BCM2711b0 are perturbed in similar way independently of the used injection medium.

Actually, some fault models are in common for both target, the only new ones are the "Bit set" which force all bits of the register to 1 and the "Xor with other observed".

6.3.2.4 *Maps per fault model*

An interesting information is to analyze the probe location regarding the fault model. Considering the observed fault models, the corresponding maps are presented in figures 64 and 65.

These figures show where the laser spot was located when we observed the different fault models. An interesting observation is that some fault models are specific to a determined location such as the "Bit set", "Bit reset" and "Xor with other obs" fault models. For the other fault models, despite they are located at several positions, they do not really overlap. This suggests that the fault models are due to the targeted element of the die. Moreover, knowing that they are independent makes our analysis relevant despite we do not analyze fix the laser spot at a specific location.

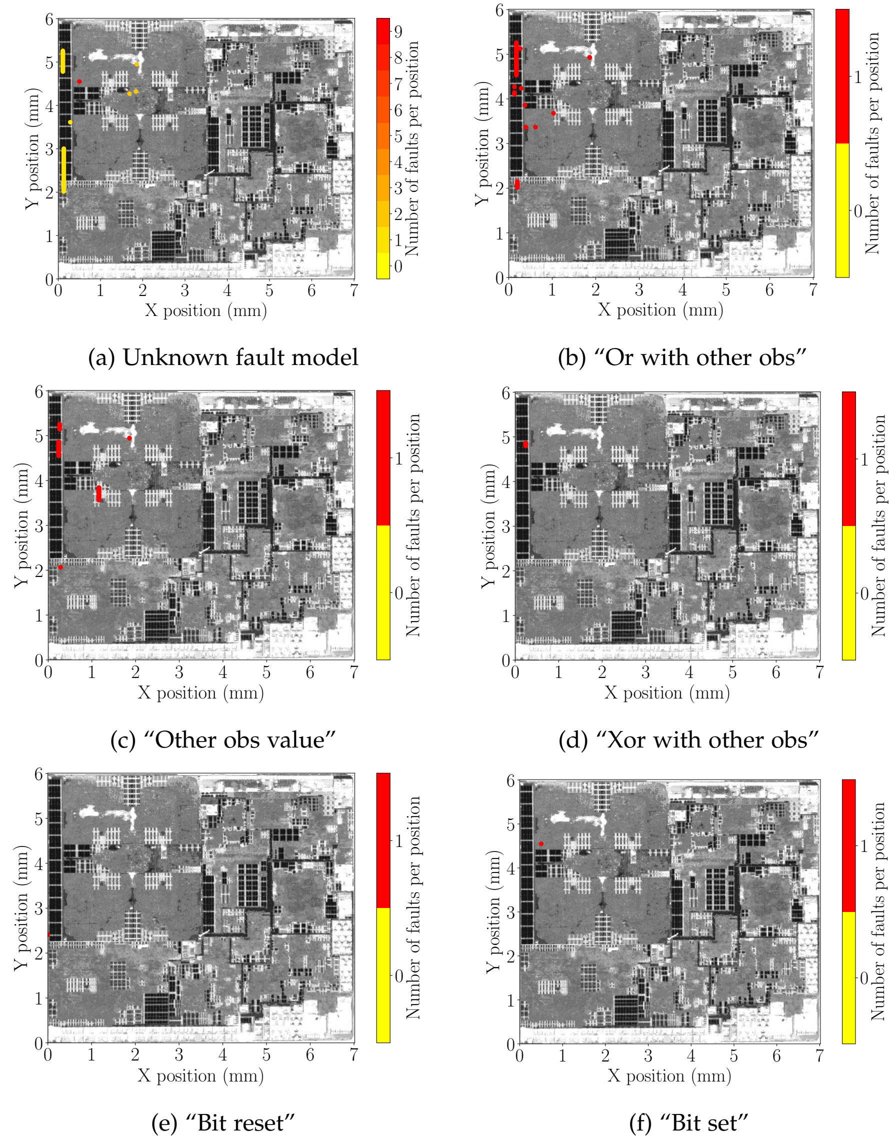
6.3.2.5 *Register corruption*

The fault model distribution suggest that some register corruption happened. A register bit reset and a register bit set. Despite they appear with a very low probability (less than 0.5 %) they are present. It could be interesting to fix the laser spot to the position they were observe to determine if it is consistent.

6.3.2.6 *Instruction corruption*

Regarding instruction corruption, the analysis gives interesting information. Indeed, the three remaining fault models "Or with other obs", "Other obs value" and "Xor with other obs" correspond to instruction corruption. Moreover, they match the instruction fault model we already observed on the BCM2837 in section 6.2.2.7.

It is interesting to determine in which way the instruction is corrupted and if it matches with our observations on the BCM2837.



(a) Unknown fault model

(b) "Or with other obs"

(c) "Other obs value"

(d) "Xor with other obs"

(e) "Bit reset"

(f) "Bit set"

Figure 64: BCM2711b0 hot spots leading to the different observed fault models

Regarding the "Or with other obs" fault model the only observed instruction was `orr r5,r1`. This corresponds perfectly with our observation on the BCM2837 on which this instruction was observed in 92.54 % of this fault model.

Considering the "Other observed value", the faulted instructions are presented in table 10.

This table shows that when the instruction is forced to a `mov` the source register is always `r8`, this is similar to some observations made on the BCM2837. However, regarding the destination register, the most probable is `r9` while the others are similar to previous observations on the BCM2837. Again, as this experiment is a cartography, some results may be influenced by the laser position but we observe some similarities between the BCM2837 and the BCM2711b0 behaviors under perturbations.

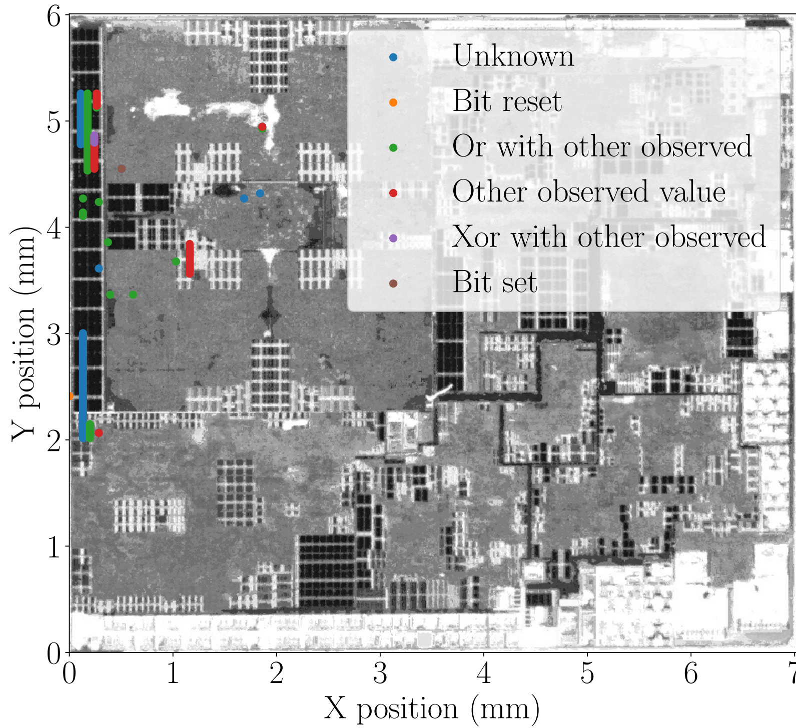


Figure 65: BCM2711b0 hot spots leading to the different observed fault models (alternative version)

Faulted instruction	Occurrence (%)
`mov r9, r8`	41.67 %
`mov r4, r8`	39.58 %
`mov r1, r8`	16.67 %
`mov r0, r8`	2.08 %

Table 10: `orr r5,r5` corruptions regarding the "Other obs value" fault model on the BCM2711b0 perturbed with a laser

Finally, for the "Xor with other obs" fault model, the only observed instruction is `xor r8,r0`. This is totally specific to the BCM2711b0 experiments.

6.3.3 *Conclusion on the BCM2711b0 characterization*

This section presented the characterization work we have done on the BCM2711b0. Due to time constraints a full characterization with fixed parameters could not be done. However, the analysis of the cartography experiments gives interesting information, in particular when compared with the results observed on the BCM2837.

The fault stability is comparable on both targets, and this is actually very true as we do not work with fixed parameters. Among the observed faults, some are very similar between the BCM2837 and the

BCM2711b0, and in particular the instruction corruption which corrupt the instruction operand in the same way on both targets.

As we cannot really conclude that the fault behave in the exact same way on both targets, this similarity strongly suggests that the fault model is not really depending of the fault injection medium but the targeted micro-architectural element. Indeed, despite we do not realize a micro-architectural characterization on the BCM2711b0, the laser precision able us to confirm that most of the faults we observed are obtain while targeting the cache memory. This cache memory is the element we identified as the perturbed one on the BCM2837.

Therefore, we conclude that the fault model we observe comes from the cache misbehavior on both targets and that these cache react in the same way to EM and laser perturbations.

6.4 INTEL CORE I3 CHARACTERIZATION

The Intel Core i3-6100T is the last target we characterize and we used EM perturbation against it. This target appeared to be harder to fault than the previous target. This difference was not investigated and might be due to the manufacturing technology or the architecture of the device.

For this target the two tested programs were based on the repetition of the `mov rbx,rbx` and `orr rbx,rbx` instructions. To be compliant with the Intel nomenclature, the register are named after the x86 specification. However, it does not change anything about the characterization method.

6.4.1 *Hot-spots maps*

As with the previous target, the first characterization step consists in determining the injection parameter we want to use. This step was very time consuming on the Intel Core i3 because it is harder to fault than the other targets.

6.4.1.1 *Spatial location*

The first parameter we worked on is the spatial position of the probe over the target. As the Intel Core i3 is covered with a metal package, we had to remove it to use EM perturbations. Doing so, we directly access the die of the chip. This die is visible on figure 66 and was divided in a 20×40 grid, each position was tested 30 times leading to 24000 operations.

As mentioned, faulting the Intel Core i3 is harder than the previous target, the consequence is that the fault map is not relevant to determine any interesting positions. However, by using an important input power, we were able to observe areas leading to reboots. This result is presented in figure 66.

This figure shows four sensitive areas for the Intel Core i3. Despite we do not know the layout of the die, we know that this target embeds two cores. These areas might therefore match these cores, the

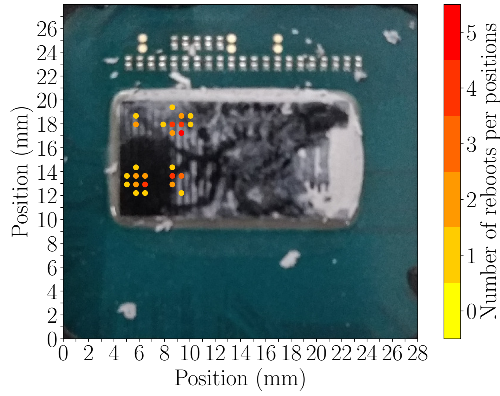


Figure 66: Intel Core i3 hot spots leading to reboots using EM perturbation

symmetrical organization of the area suggests it is the case but this hypothesis needs confirmation.

To obtain faults, we decided to place the probe at the position X = 9 mm and Y = 13 mm and to test various input voltage.

6.4.1.2 *Input voltage*

The input voltage was tested by sweeping it between 650 V to 750 V with a step of 10 V. The results are presented in figure 67.

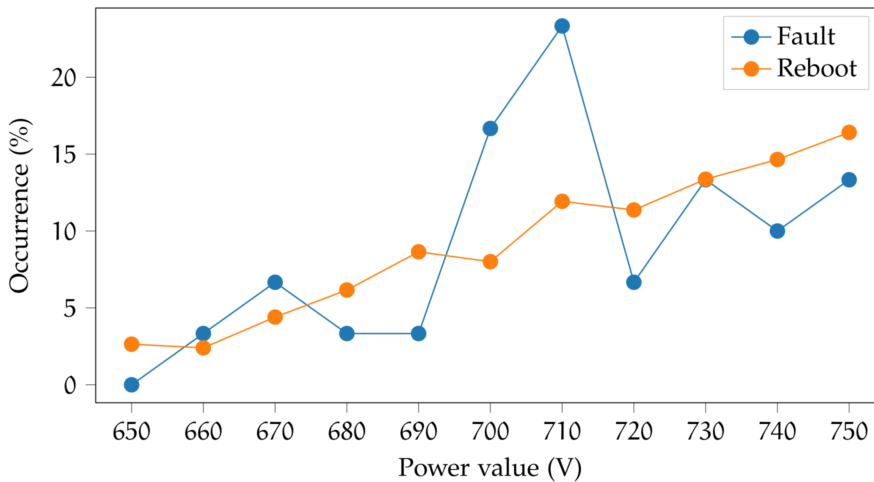


Figure 67: Input voltage amplitude effect on Intel Core i3 during EM perturbation

This figure shows that the number of reboots seems to grow linearly with the input voltage. Regarding the fault probability, the highest probability are observed around 710 V and decrease rapidly with lower and higher values.

This highlight that the Intel Core i3 is harder to fault in comparison with the BCM2837 on which we observed faults with a input voltage of 300 V (even if we actually characterized it with an input voltage around 500 V). As mentioned, no investigation for determining this sensibility difference was done.

6.4.2 *Analyzer results*

As on the previous target, we used the analyzer to determine how the Intel Core i3 behave against EM perturbations.

6.4.2.1 *Fault probability*

The fault probability on the Intel Core i3 was observed to be between 0.31 % and 0.39 % which is up to ten times lower that the fault probability on the BCM2837.

Also, this explained why we could not observe faults during the spatial cartography as we did only 30 tests per position while there is a fault every 250 attempts on average.

6.4.2.2 *Faulted values distribution*

Regarding the faulted values distribution, the results for both experiments are presented in figures 68 and 69.

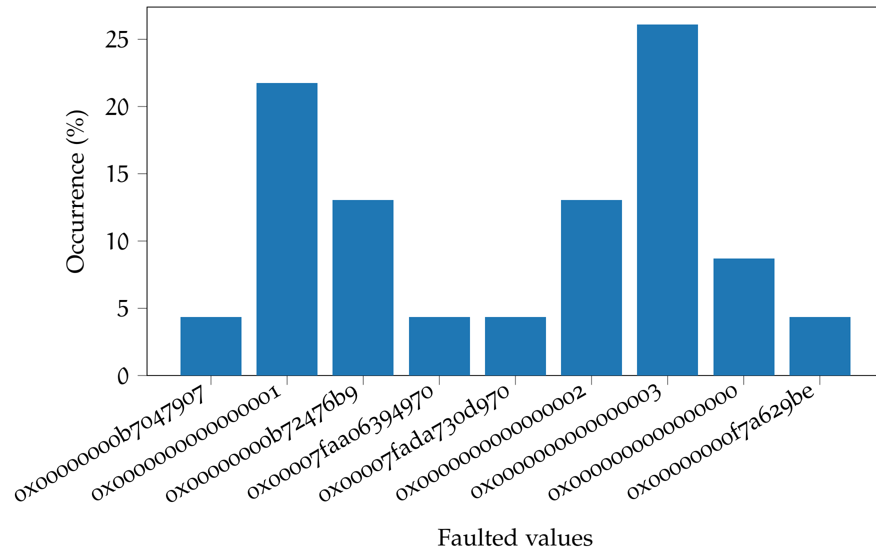


Figure 68: `mov rbx,rbx` faulted value distribution on Intel Core i3 using EM perturbation

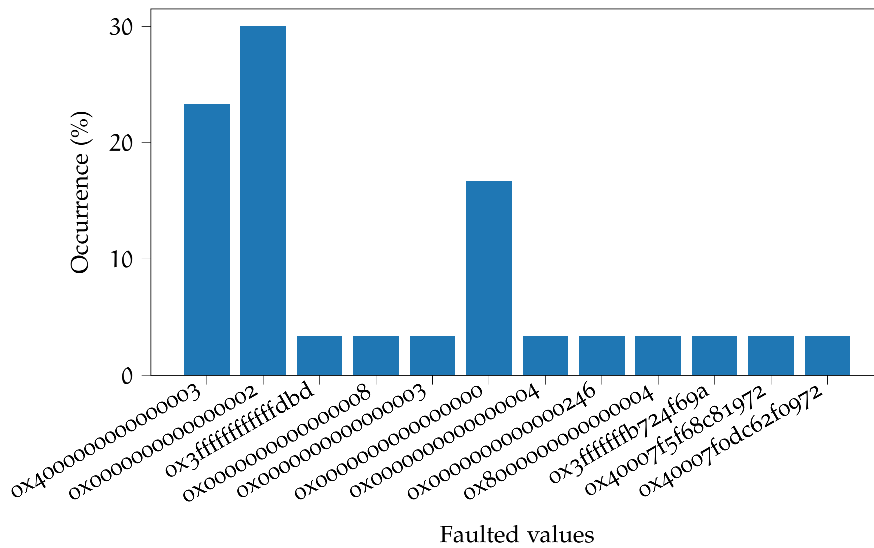


Figure 69: `orr rbx,rbx` faulted value distribution on Intel Core i3 using EM perturbation

These results show that the number of observed faulted value is really lower than the number we observed on the previous targets. This shows that, despite being hard to fault, the faults observed on the Intel Core i3 are consistent. This suggests that the fault does not have a random effect on the device.

6.4.2.3 *Targeted register*

Regarding the targeted register, faulted one is always the one manipulated by the instruction, *i.e.* `rbx`. As we only execute instructions involving this register, this strongly suggests that we are in a similar situation than we were on the previous targets.

6.4.2.4 *Fault model*

For the fault model analysis, the analyzer was not able to determine the fault model for between 35 % to 45 % of the cases. This is quite similar to our observations on the BCM2837. The fault model distribution is shown in figure 70.

Despite the unknown fault model, the only present fault models are the “Bit reset”, the “Other obs value” and the “Or with other obs”. The “after execution” mention comes from the fact that the fault models are computed on the values after the faulty execution of the test program. Indeed, the observed faulted value could be computed using these fault models on the observed faulted values instead of the initial values.

This suggests that the faults first corrupts some registers then corrupts the executed instruction.

However, without this particularity, the observed fault models seem to match with the observation we made on the previous targets. Indeed, the “Or with other obs” fault model only appear during the `orr rbx,rbx` experiment and the “Other obs value” is very present in the `mov rbx,rbx`.

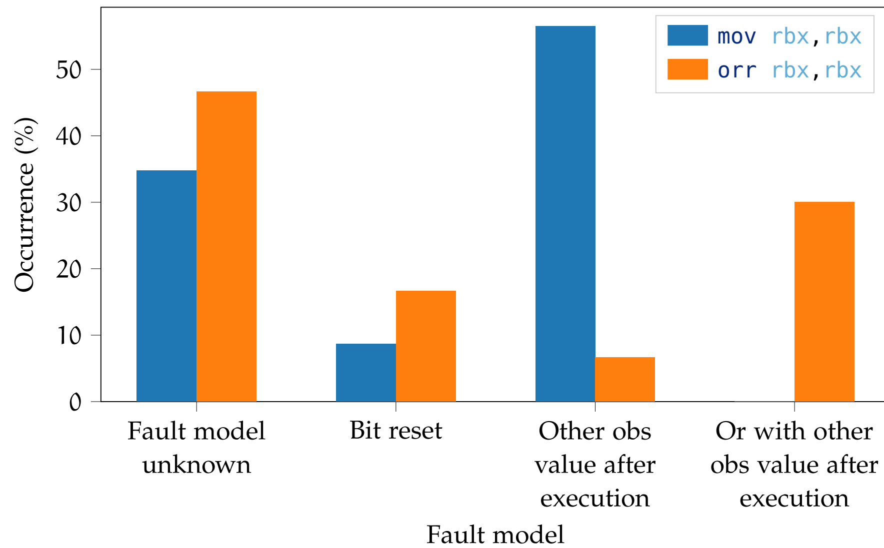


Figure 70: Probability of observing the different fault models for both experiments on Intel Core i3 using EM perturbation

6.4.2.5 *Register corruption*

As mentioned above, the observed fault always involve a register corruption, however, most of them could not be identified and are considered random.

But some "Bit reset" have been observed. This was already observed on the BCM2837 and as said before, it matches the EM perturbation medium behavior.

6.4.2.6 *Instruction corruption*

The "Other obs value" fault model is observed on both experiment. As one is based on a `orr` instruction, it seems that, at least in this case, the fault corrupts the instruction opcode. The corresponding faulted instructions are `mov rbx,rdi` (50 %) and `mov rbx,r11` (50 %). This shows that the operand is also faulted as on the previous targets.

Considering the `mov rbx,rbx` experiment, the "Other obs value" fault model is explained with the corruption of the register source or destination in the instruction. The observed faulted instructions are presented in table 11.

Faulted instruction	Occurrence (%)
`mov rbx, rdi`	46.15 %
`mov rbx, rax`	38.46 %
`mov rbx, rcx`	15.38 %

Table 11: `mov rbx,rbx` corruptions regarding the "Other obs value" fault model

This table shows that the destination register is never faulted but the source register is. This corresponds to the instruction operand and perfectly matches our observations on the BCM2837. As this target

and the Intel Core i3 are very different: different architecture, different instruction encoding, different process technology and different cache management, this similarity is very interesting. Despite these differences, as the fault are similar we can conclude that the causes are the also similar.

The last fault model we observed is the "Or with other obs" fault model and only appear during the `orr rbx,rbx` experiment. This is again similar to our observations on the BCM2837 and suggests an operand corruption. The corrupted observed instruction are presented in table 12.

Faulted instruction	Occurrence (%)
`orr rbx, rax`	77.78 %
`orr rbx, rcx`	22.22 %

Table 12: `orr rbx,rbx` corruptions regarding the "Other with other obs" fault model

This table shows that the observed fault model actually comes from a corruption of the operand such as on the BCM2837. This strongly suggests that the fault is independent from the instruction encoding and effectively targets a mechanism manipulating the operands. However, we identified the cache to be the target micro-architectural, this suggest that we also perturb it on the Intel Core i3 as we observe the same behavior.

6.4.3 *Conclusion on the Intel Core i3-6100T*

This section presents the characterization work we have done on the Intel Core i3-6100T target using our method. This target appeared to be hard to fault than the BCM2837 and the BCM2711b0 as we needed a stronger EM perturbation to observe faults.

However, contrary to the previous targets, we observe less different faults, in other words, the number of observed faulted value is around ten on the Intel target while it is more around twenty-five on the BCM targets. However, we did not look into determining the root cause of this difference.

A more interesting observation is that the fault model on the Intel target is often a second operand corruption, just as on the BCM targets. This suggest that these targets, while being different, are faulted in a similar way.

Therefore, we can suppose that the Intel target and the BCMs share a mechanism which is faulted in the same way on all targets. While it is harder to fault it on the Intel target.

6.5 Characterization conclusion

This chapter presented the characterization experiments realized during this thesis. The characterized target are chosen to be representative of common SoCs one can find in modern digital devices. The

chosen target are the BCM2837 powering the Raspberry Pi 3 model B, the BCM2711b0 powering the Raspberry Pi 4 and the Intel Core i3-6100T powering a PC.

These targets represent the two most widespread architectures in SoCs: ARM (BCM2837 and BCM2711b0) and x86 (Intel Core i3-6100T). They were tested against EM perturbation (BCM2837 and Intel Core i3-6100T) and laser perturbation (BCM2711b0).

Despite several constraints, the three targets were characterized at the ISA level. The conclusions are that faults on all the targets are able to modify the operands used by the executed instructions. Also, the CPU registers were proven to be corruptible. An in-depth characterization at the micro-architectural level able to identify that the memory cache is faulted.

Moreover, the targets show a similar faulted behavior despite being manufactured in different technologies, implementing different architecture and being faulted using different injection medium. This observation leads us to the conclusion that the fault model on these targets does not depend of these elements but a mechanism they have in common. Even if we know that we fault the cache, we could not precisely identified the involved mechanism.

Having a common to all SoCs mechanism which behaves in the same way to different perturbation make these targets quite weak. Indeed, if the faults are repeatable from a SoC to another with a correct probability, any attack realized on a program on a SoC might be applicable to any other SoC using this program.

This observation make the mitigation of such faults a important step. However, in this thesis, to assess the relevance of our characterization, we did not focus on the design of a countermeasure but on the exploitability of the characterized faults regarding classical security softwares embed in SoCs.

7 FAULT MODEL EXPLOITABILITY

Life happens wherever you are, whether you make it or not.

— Uncle Iroh (Avatar: The Last Airbender)

ABSTRACT

This chapter presents our work to evaluate Linux programs against the faults we characterized in the previous section. We present a DFA on the OpenSSL implementation of AES, a PFA on our baremetal AES using persistent faults in the cache. We also realize an analysis of the `sudo` program which involves a user authentication mechanism. This analysis highlight the complexity of such programs and the need of an analysis and fault simulation software we also introduce. The works presented in this chapter were submitted in the JCEN journal [159] and in the CHES conference [162].

Contents

7.1 DFA on the OpenSSL AES implementation 123
- 7.1.1 Source code location 123
- 7.1.2 Static analysis 124
- 7.1.3 Fault attack on the OpenSSL AES 126
- 7.1.4 DFA software 128
- 7.1.5 Conclusion on the OpenSSL AES DFA 132

7.2 Baremetal AES PFA [159] 132
- 7.2.1 PFA Result 132
- 7.2.2 Conclusion on the PFA on our baremetal AES 134

7.3 `sudo` authentication 134
- 7.3.1 Attack model 135
- 7.3.2 `sudo` analysis 135
- 7.3.3 Code analysis 141
- 7.3.4 Program setup 142
- 7.3.5 Side channel analysis 144
- 7.3.6 Exploitation 145
- 7.3.7 Conclusion on the forced authentication 147

7.4 Analysis tools 147
- 7.4.1 The function analyzer 148
- 7.4.2 Fault simulator 149

7.5 Evaluation conclusion 151

In this chapter, we aim at evaluating the security of classical Linux programs regarding our fault model. The evaluation of such programs is very different from the ones executing on smartcards. Indeed, as they execute on a modern CPU, their execution is not sequential but parallel due to the presence of optimizations as mentioned in section 2.3.2.2. This make the synchronization of the perturbation with a particular moment of the program execution trickier.

Moreover, Linux programs rely on mechanisms coming from the Linux kernel like the shared libraries. These libraries are either included in the executable (static linking), loaded at the program startup (dynamic linking) or loaded during the program execution (module loading). Therefore, as a Linux program is composed of an executable and several libraries it relies on, its attack surface is important but it is also more complex to analyze it. In particular, due to a lot of runtime mechanisms, a static analysis of such program may be superficial compared with what actually happen at runtime, in particular considering faults. Actually, the execution of such programs involve the execution of the Linux kernel dynamic linker as presented in figure 71.

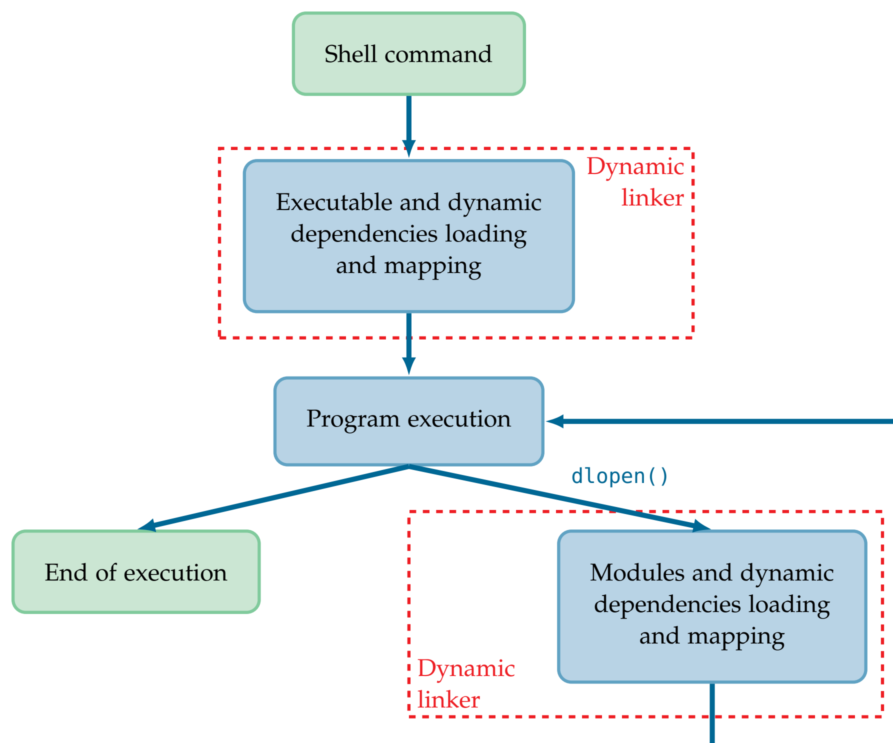


Figure 71: Linux dynamic linker interventions during program execution

This figure shows that, on Linux systems, the execution of a program depending on dynamic libraries involves the execution of a kernel piece of code named the dynamic linker which aims at loading and mapping the needed libraries for the program to execute. Also, a program can call the dynamic linker to load a shared object for executing it at runtime using the `dlopen()` function. Therefore, only analyzing the binary of a program is not enough to asses its security but also some kernel mechanisms must be checked. In particular the

dynamic linker as the loading of the wrong library may cause security issues.

Because the security of such programs against fault attacks is a novel topic, there are no fault security analysis method, no fault analysis tools and only few works that target such systems.

Therefore, this chapter presents the first steps of such security evaluation. For the synchronization part, we present two works targeting AES. On the hand, a work using a DFA in section 7.1 which forces us to successfully target the ninth round. On the other hand, another work using a PFA and a persistent fault in the CPU cache to help us understand the fault effects on our baremetal AES implementation in section 7.2. Moreover, in section 7.3, we aim at faulting a complete Linux program, namely the `sudo` program. Then this chapter is concluded by a description of the tools developed from these works and aiming at helping us in the analysis of complex Linux programs in section 7.4.

7.1 DFA ON THE OPENSSL AES IMPLEMENTATION

OpenSSL[1] is an open source general-purpose cryptography library. It is used in various programs needing cryptographic mechanisms such as web navigator, mail boxes, video-conferences, user authentication, *etc*.

The AES algorithm [163] is a symmetric algorithm based on ten rounds executing the `SubBytes`, `ShiftRows`, `MixColumns` and `AddRoundKey` operations. It is one of the most used symmetric algorithm and it is involved in a lot of applications like securing internet connections, encrypt disks, *etc*. Therefore, it is a very interesting target to consider. In this section, we focus on attacking the OpenSSL `AES` implementation using fault attacks. The target is the BCM2837 powered by a Linux OS.

For the setup, the program is organized as presented in figure 36 in section 5.3.2 with the test program being the `AES_encrypt()` function from the OpenSSL library. Regarding this setup, our test program can be considered as a Linux program with dependencies of the `libssl.so` and `libcrypto.so` libraries which are loaded at the startup by the linker. However, there is no shared library loaded at runtime. Therefore, perturbing the program during its execution will not affect the dynamic linker.

7.1.1 *Source code location*

The OpenSSL source code is available on the OpenSSL website[2]. As we focus on the AES, we study the `AES_encrypt()` function which is available in the `crypto/aes/aes_core.c` file. Architecture specific implementations are available, for instance, on the Raspberry Pi 3 the executed function is the `_armv4_AES_encrypt()` function available in the

1 `https://www.openssl.org/`
2 `https://www.openssl.org/source/gitrepo.html`

crypto/aes/asm/aes-armv4.pl file. However, our analysis on assembly code will not come from this file but directly from the disassembly of the program. The reason is that the ARMv4 specific implementation is embedded in a Perl script which realizes some optimizations. Therefore, disassembling the binary gives the closest-to-reality assembly code.

7.1.2 *Static analysis*

Before evaluating the AES implementation, we realize a static analysis of the source code and binary to determine whether the determined fault model reveals exploitable vulnerabilities.

7.1.2.1 *AES round optimization*

The AES_encrypt() function is optimized to have the lowest execution time. This optimization is obtained by using pre-computed tables for every SubBytes and MixColumns operations. The listing 7.1 presents a part of these tables.

```
static const u32 Te0[256] = {
    0xc66363a5U, 0xf87c7c84U, 0xee777799U, 0xf67b7b8dU,
    0xfff2f20dU, 0xd66b6bbdU, 0xde6f6fb1U, 0x91c5c554U,
    0x60303050U, 0x02010103U, 0xce6767a9U, 0x562b2b7dU,
    0xe7fefe19U, 0xb5d7d762U, 0x4dababe6U, 0xec76769aU,
```

Listing 7.1: OpenSSL AES pre-computed tables (partial)

The Te0 table corresponds to the operation presented in equation (35) with S the SubBytes operation.

$$Te0[x] = S(x) \cdot [2, 1, 1, 3] \tag{35}$$

Equation: Pre-computation of the Te tables in the OpenSSL AES

This operation realizes the computation of the first column of the next AES state. Corresponding tables Te1, Te2, Te3 are also pre-computed for the others columns. In the end, the outputs of these tables a recombined to obtain the current round AES state. The listing 7.2 presents how the states are recombined.

The recombination of the states consists in XORing the output of the tables between them and with the current round key rk. The table access is done by switching and masking the current state (s0, s1, s2, s3). The full computation consists in a loop repeating these operations alternating between the tX and the sX states as input/output of the tables.

```
1     t0=Te0[(s0>>24)] ^ Te1[(s1>>16)&0xff] ^
    ↪  Te2[(s2>>8)&0xff] ^ Te3[s3&0xff] ^ rk[4];
2     t1=Te0[(s1>>24)] ^ Te1[(s2>>16)&0xff] ^
    ↪  Te2[(s3>>8)&0xff] ^ Te3[s0&0xff] ^ rk[5];
3     t2=Te0[(s2>>24)] ^ Te1[(s3>>16)&0xff] ^
    ↪  Te2[(s0>>8)&0xff] ^ Te3[s1&0xff] ^ rk[6];
4     t3=Te0[(s3>>24)] ^ Te1[(s0>>16)&0xff] ^
    ↪  Te2[(s1>>8)&0xff] ^ Te3[s2&0xff] ^ rk[7];
```

Listing 7.2: OpenSSL AES round computation (C)

7.1.2.2 *OpenSSL AES vulnerability analysis.*

Regarding the source code of an AES round presented in listing 7.2, we can see that the involved operations are: the logical right shift, the logical `AND`, the logical `XOR` and a memory access to the tables. As a consequence, the assembly code for a round will only use `ldr`, `and`, `eor` and `lsr` instructions. This is confirmed by the disassembled code of the `AES_encrypt()` function available in appendix E.

These instructions are very similar to the instructions we used in section 5.3.2.1 for characterizing the fault model on our targets. Therefore, we know that we have a high probability to modify the second operand of these instructions.

Also, for the DFA (presented in section 4.4.1.1), we want to fault only one byte in the AES state before the last `MixColumns`. As many instructions manipulate 32 bits wide register, faulting the second operand will mainly fault 4 bytes. If the 4 bytes are in different columns, the obtained cipher is still usable for a DFA and even leak information on 4 bytes of the key. However, as we presented earlier, the operations are done column by column and therefore, faulting a register will mainly modify the entire column making the faulted cipher not exploitable. This eliminate any fault on the `ldr`, `eor` and `lsr` instructions.

However, there are the `and` instructions that remain and they are used to apply a byte wide mask on the register. Faulting the second operand of these instructions (i.e. the used mask) will results in applying a byte wide fault on the result. This corresponds to the fault we want to obtain for the DFA.

According to the `AES_encrypt()` assembly code available in appendix E the `and` instructions represent 23 % of the instructions composing an AES round. As presented in section 6.2.2.1, we have 1 % fault probability on `and` instructions. Therefore, in first approximation (by considering all instructions have the same execution time), we can suppose that the probability to obtain a usable cipher for a DFA is around 0.23 % per injection. In other words, around 400 fault injections are needed before having a usable cipher. As 8 ciphers are needed for a complete DFA, we can suppose that around 3200 injections are needed for obtaining all the ciphers needed for the DFA.

In practice, the instructions does not have the same execution time, in particular the memory access instructions (`ldr`) are, in general, slower than the data processing instructions (`and`, `eor` and `lsr`). However, estimating the execution time is a tricky job as, due to CPU optimizations [164, 165] (cache, fill buffers, *etc*), this time is quite variable. However, these optimizations aim at making a memory access as fast as a data processing instruction, making our hypothesis relevant.

7.1.3 *Fault attack on the OpenSSL AES*

Based on the analysis, we know that our fault model is relevant for attack the OpenSSL AES implementation in 3200 injections. The next step naturally consists in testing if we actually can realize this exploitation on an implementation running on the BCM2837.

7.1.3.1 *Synchronization*

As explained in section 2.3, SoCs are multi-core and multi-threaded systems with various optimizations. This architecture brings a non sequential execution of instructions composing a program. Therefore, synchronizing a fault injection with a particular moment of the program execution (in our case: before the last `MixColumns` operation) is a tricky business.

As we do not have information nor a method to synchronize our perturbation with the program, we decided to sweep over the AES execution and analyze the fault probability and the number of faults diagonals regarding the delay between the start of the AES computation and the moment we perturb it. These results are presented in figures 72 and 73.

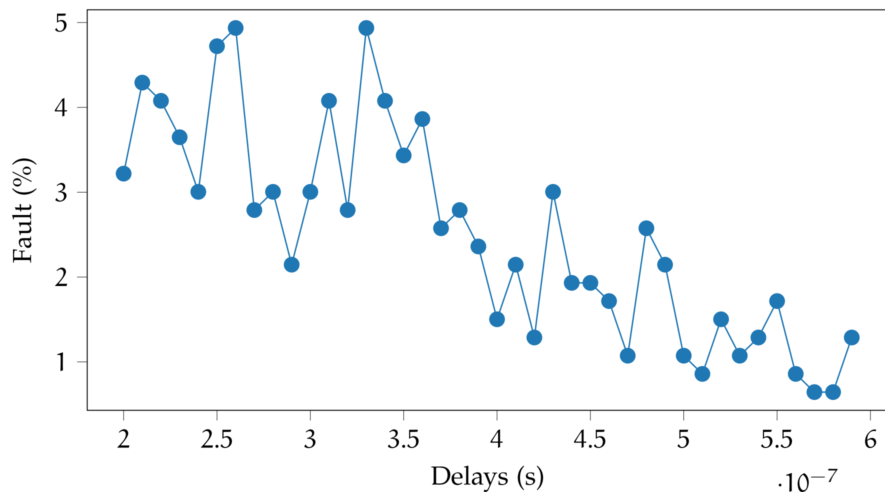


Figure 72: Impact of the delay on the probability while faulting an OpenSSL AES encryption using EM perturbation on BCM2837.

The figure 72 shows that the fault probability has a tendency to decrease while we increase the delay before doing the perturbation. However, we did not make a further researches to determine the origins of this phenomena.

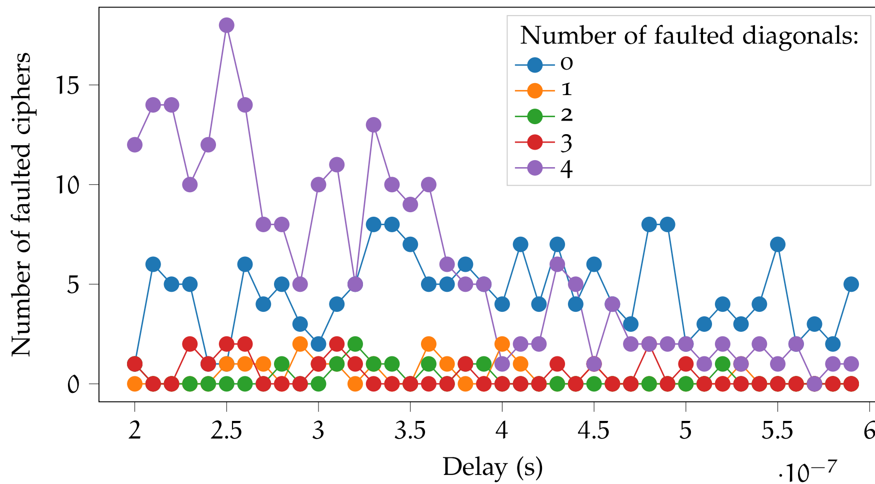


Figure 73: Impact of the delay on the number of faulted diagonals while faulting an OpenSSL AES encryption using EM perturbation on BCM2837.

The figure 73 presents the number of faulted ciphers with a specific number of faulted diagonals regarding the delay before the perturbation. 0 diagonals faulted means that the cipher is faulted but not with a diagonal pattern.

These results shows us that the number of ciphers with less than 4 faulted diagonals faulted is quite constant and therefore independent of the injection timing. However, the number of ciphers with 4 faulted diagonals decreases when the delay before the perturbation increase. This is understandable because, in the AES, any fault that appears on at least a byte is diffused on the whole cipher after the execution of a `MixColumns`, `ShiftRows` and an other `MixColumns` sequence. In other word, any fault before the before-last `MixColumns` (the one during the 8th round) leads to a fully faulted cipher, *i.e.* the four diagonals are faulted, but this kind of cipher is not suitable for a DFA.

When the perturbation occurs after the before-last `MixColumns`, the diffusion of the fault is limited to the diagonals containing the initial faulted bytes. Supposing that faulting four bytes with each byte on different diagonal is less probable than faulting a single byte. The number of cipher with four faulted diagonals should drop when the perturbation occurs after this `MixColumns`. This seems to happen around 400 ns of delay. Also, we could expect that, after this delay, the number of ciphers with less than 4 faulted diagonals will increase, but we do not observe such phenomena.

As mentioned above, the number of ciphers with less than 4 faulted diagonals remains almost constant independently of the delay. This observation is complicated to explain as before the before-last `MixColumns`, we should observe only fully faulted ciphers and after this `MixColumns`, a more regular distribution should be observed. But it is not the case. Regarding the figure 72, our observation seems to correlate with the fault probability, *i.e.* after 400 ns of delay the probability to obtain ciphers with 4 faulted diagonals decrease while the probability to obtain ciphers with less than 4 faulted diagonals remains constant.

This correlation is unnatural as there is no reason which might explain that the probability of having 4 faulted diagonals decrease while the probability of having less than 4 faulted diagonals remains constant, and must be studied carefully. Our opinion is that several phenomena occur. The first one is that, because we passed the before-last `MixColumns`, the distribution of the number of faulted diagonals is quite balanced (when the delay is greater than 400 ns). The second one is that, for an unknown reason, passing this delay the fault probability decrease.

Despite we think this is the best explanation of these observations, we do not know why the fault probability would decreased after this delay and we cannot explain why we observe ciphers we less than 4 faulted diagonals before this delay. This result is an example of the complexity induced by the CPU architecture regarding synchronization and our lack of knowledge about how programs are executed within these CPUs.

However, we were able to obtain ciphers with only one faulted diagonal making the DFA possible. Therefore, we focused on using these faulted ciphers to realize our DFA.

7.1.3.2 *Exploitability*

The faults campaign consisted of 3000 injections (up to 3200 are required for a complete DFA regarding our analysis) and around an hour is needed to achieve them with our setup. Among these injections, we obtained 466 faults (15.54%). By all of these faults, only 16 have a single faulted diagonal (4.348%) and considering these faulted ciphers; only 8 correspond to a one-byte fault before the `MixColumns` operation. Also, faults appear with the same probability on every diagonal.

In the end, the probability of obtaining a suitable faulted cipher for the DFA is 0.34% which corresponds to 1 cipher every 294 injections which is a bit better than the rate we extrapolated from the analysis above (0.23 %). This difference might be due to the hypothesis we made that any fault on instructions other than `and` instruction will not give any interesting faulted cipher. Indeed, it is possible by faulting a 4 *bytes* register that the fault affects only one byte.

Considering an injection needs 2 seconds, we obtain a usable cipher every 10 minutes. As 8 ciphers are needed for the complete DFA and because every diagonal has the same probability to be faulted, 3 hours of injection are enough to obtain the needed ciphers.

7.1.4 *DFA software*

To do the DFA, we developed a C program which aims at realizing it. As the DFA able to retrieve the key bytes diagonal per diagonal, our program works only on a diagonal. Therefore, its inputs are the faulted ciphers and the diagonal index to work on.

After our experiments, we sort the suitable ciphers and stored them in different files named `ciphers_diagX.txt` where X is the diagonal

index. Our DFA program then reads the ciphers contained in the file and use them to realize the DFA as explained on figure 29 in section 4.4.1.1.

The possible Δ values are pre-computed using a Python script and stored in a header file. Then the program computes the Δ value for every possible key diagonal (2^{32} values) for every faulted ciphers stored in the input file. The loop doing this is presented in listing 7.3.

```
137   for(uint64_t value = 0; value < MAX_VAL_32BITS; value++){
       ↪ /* Key bytes */
138     /* Computing the reversed good cipher */
139     int_to_bytes((uint32_t)value, k_diag, AES_DIAG_SIZE);
140     xor_array(c_diag, k_diag, c_xor, AES_DIAG_SIZE);
141     inv_s_box_array(c_xor, c_sub, AES_DIAG_SIZE);
142
143     k_is_good_candidate = 1; /* Check if we have a good
         ↪ candidate */
144     for(i=0; i<nb_faulted_ciphers; i++){ /* Test over all
         ↪ faulted ciphers */
145       f_str = faulted_ciphers[i];
146       hex_str_to_bytes(f_str, f_bytes, AES_NB_BYTES);
147       extract_diag(f_bytes, f_diag, diag, AES_DIAG_SIZE);
148       /* Computing the reversed faulty cipher */
149       xor_array(f_diag, k_diag, f_xor, AES_DIAG_SIZE);
150       inv_s_box_array(f_xor, f_sub, AES_DIAG_SIZE);
151       /* Computing the delta */
152       xor_array(c_sub, f_sub, delta, AES_DIAG_SIZE);
153       /* Check if delta is in possible values */
154       delta_val = (uint32_t)bytes_to_int(delta,
           ↪ AES_DIAG_SIZE);
155       k_is_good_candidate = k_is_good_candidate &
156                             is_in_array(mix_col_diff_list,
157                                         delta_val,
158                                         NB_MIX_COL_VALUES);
159     }
160
161     if(k_is_good_candidate){ /* Print the key diagonal */
162       printf("\rKey diagonal %d = 0x%08x\n", diag,
           ↪ (uint32_t)value);
163     }
164
165     printf("\r0x%08x", (uint32_t)value); /* Print the last
         ↪ tested key */
166     fflush(stdout);
167   }
```

Listing 7.3: DFA program main loop

This listing shows the loop iterating over the possible key values (line 137) and then computing the correct cipher last round reverse: the

left term of equation (21) in section 4.4.1.1 (lines 139 to 141). The second part consists in computing the right term of equation (21) for every faulted ciphers (line 144 to 150) and then compute the delta value (line 152). Then we check if the key candidate is a good one by checking if the delta value is among the possible output values of a faulted `MixColumns` (line 155).
As one can note, despite we work on 32 bits, we have to convert our variables into byte arrays. The reason is the `SubBytes` operation which involves SBoxes. Despite manipulating byte arrays is slower than manipulating 32 bits data, the storing of the SBoxes requires 256 B (2^8 B) of memory when working with bytes and 16 GB (4×2^{32} B) of memory when working with 32 bits data. We therefore decided to work with bytes array to lower the memory footprint of our program.
However, we developed a Python script which generates the 32 bits SBoxes but we did not tested it. As a consequence we do not know how much faster a fully 32 bits program would be. Also, regarding listing 7.3, the program is not optimized in memory as we use every buffer only once while we could reuse them between two operations. This choice was made to keep a good clarity in the source code.

7.1.4.1 *Usage and results*

The usage of the program is straightforward, the program will look for the `ciphers_diagX.txt` files and computing the correct diagonal using the passed argument such as in listing 7.4.

```
> ./aes_dfa 0
> Key diagonal 0 = 0x132ba717
```

Listing 7.4: Example usage and result of the DFA program on the first diagonal

Currently the correct cipher is hard coded in the program and is presented in listing 7.5.

```
112    char* c_str = "69c4e0d86a7b0430d8cdb78070b4c55a";
```

Listing 7.5: AES correct key as implemented in the DFA program

However, this is a bad coding habit and a program enhancement will be to pass this key as a parameter. The results presented in listing 7.4 were obtained using the faulted ciphers introduced in listing 7.6.

```
9fc4e0d86a7b04aed8cda980708dc55a
11c4e0d86a7b0446d8cd60807094c55a
```

Listing 7.6: Faulted ciphers used in our DFA (`ciphers_diag0.txt`)

7.1.4.2 *Performance*

From 2 faulted ciphertexts with the same faulted diagonal, our program is able to recover the 4 corresponding bytes of the key in an

hour on average. The computer used for this computation is powered by an Intel(R) Core(TM) i7-8550U CPU clocked at, at least, 1.80 GHz with 16 GB of memory.
As our implementation works per diagonals, it is possible to run four instances of the program and therefore realize the cryptanalysis on the four diagonals in parallel. However, this require four times more memory at the moment, indeed the pre-computed SBoxes are not shared between these instances making the usage of byte arrays even more relevant. But, sharing the SBoxes using multiple threads will divide the memory footprint by 4, making it a great improvement to do.
Finally, once we obtained the faulted ciphers, only 1 hour is needed to recover the key. Adding the time needed to obtained these faulted ciphers, the complete cryptanalysis can be achieved in less than 4 hours.
This timing considers that the hot spots determination and the fault characterization are already done. These steps require at least a week of work but the results are reusable on every target powered by the characterized device. Also, the algorithm consider that every provided faulted cipher is relevant for realizing the DFA, which is, in practice, not the case.

7.1.4.3 *Detecting the relevance of faulted ciphers*

To do a DFA, we use ciphers with one diagonal faulted. However, this diagonal must be faulted with a specific pattern, *i.e.* the fault must comes from a one byte faulted before the `MixColumns` operation. However, it happens that we observe a cipher with a diagonal faulted while it does not comes from a fault on a byte before the `MixColumns`. In can be only the four bytes faulted independently for instance. In such case, the cipher is not exploitable for a DFA.
In our case, as we fully master the implementation, we could verify if a faulted cipher match a fault on a byte before the `MixColumns`. However, this verification needs the knowledge of the key, which is irrelevant in a practical attack but relevant in the case we just want to assert our faults are suitable for a DFA.
Therefore, to adapt our DFA program to avoid this verification is to implement an enumeration protocol which tests several faulted ciphers pairs. Indeed, when we realize the DFA, the correct key is recovered in 98 % of the cases with two ciphers. Therefore, when a key does not match with two given ciphers, there is two cases: either it is not the correct key or the used faulted cipher does not match a fault on a byte before the `MixColumns` and more ciphers must be tested.
In practice, the ratio of the number of suitable for DFA faulted ciphers against the number of ciphers with only one diagonal faulted can only be determined with the knowledge of the key or by successfully realize the attack. In our experiment, we determine that 50 % of the ciphers with one faulted diagonal are suitable for a DFA.
Therefore, the last improvement we can do for our DFA software is the implementation of a smart way to test the different ciphers with-

out knowing if they are actually suitable for a DFA. Of course, this will make the algorithm slower depending on our capacity to have suitable faulted ciphers.

7.1.5 *Conclusion on the OpenSSL AES DFA*

In this section, we successfully attack the OpenSSL AES running on a BCM2837 using a DFA. We saw, *via* a static analysis, that the OpenSSL implementation of the AES is sensitive to faults regarding our fault model. We computed a success rate from this analysis.
By faulting the AES, we observed a greater success rate than the one we initially computed (ignoring instruction execution time variability and CPU optimizations). We also observe interesting behavior regarding our moment of injection. Indeed, the fault probability and the number of ciphers with four faulted diagonals have a tendency to decrease while we increase the delay between the start of the AES and the perturbation. We did not make further research to explain this phenomena but it shows that the CPU complexity can lead to interesting observation regarding synchronization.
Despite these undetermined behaviors, we were able to obtain suitable faults for realizing the DFA with a self-programmed software in around an hour.

7.2 BAREMETAL AES PFA [159]

The work presented in this section aims at assessing that the persistent faults in the data cache observed in section 6.2.4.3 are suitable for an exploitation. The reason is that these faults are particularly powerful regarding our target as they do not need to be synchronized with the program execution but only have to exist. However, due to time constraints we did not realized this experiment on a Linux system.
Therefore, in this section, we realize a PFA (introduced in section 4.4.1.1) on an AES we implemented on our baremetal setup on the BCM2837.
Our AES implementation is a naive one without countermeasures. The SBoxes are implemented with a lookup table. Before the actual encryption, we pre-warm the cipher. In other words, to limit timing leakages due to caches, we invalidate the instruction and data caches, then we perform a dummy AES encryption to fill the caches with data and instructions of interest. This is the step where we inject the fault. In a second step, we encrypt 10 000 random plaintexts, without any new fault injection, and try to deduce the key from the observed faulty ciphers.

7.2.1 *PFA Result*

To detect the forbidden values, we simply count the number of occurrences for all bytes values in the ciphertext, for the first byte, then the second byte, etc. On figure 74, one can observe that 2 values are

forbidden. Therefore, in our case, two bytes have been faulted in the SBox lookup table.

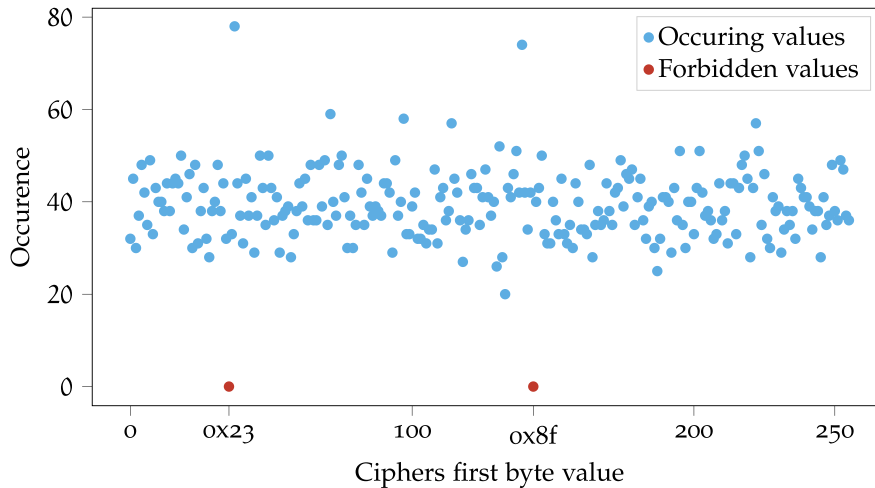


Figure 74: Distribution of the first byte for 10 000 faulted ciphers on our baremetal AES on the BCM2837.

Therefore, the analysis presented in section 4.4.1.1 must be adapted to the two faults case. For each byte index from 0 to 15, we get the two forbidden values. They are shown in Table 13.

Index	0	1	2	3	4	5	6	7
Forbidden values	23 8f	21 8d	2d 81	4f e3	7f d3	08 a4	7a d6	27 8b
Index	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
Forbidden values	6f c3	37 9b	3b 97	17 bb	7d d1	1b b7	00 ac	59 f5

Table 13: Forbidden values for every byte of the observed ciphers.

Since we have 2 forbidden values y_1 and y_2 per byte, we have 2^{24} key hypotheses. First, we have 2^8 guesses for y_1 (or y_2 as they can swap but only one is needed) then, among the two forbidden values for each byte, we must find the one giving the key regarding our y_1 (2^{16} possible choices). The possible key bytes for a correctly guessed forbidden value $y_1 = \texttt{0x30}$ are displayed in table 14. In other words, table 14 is table 13 XORed with y_1. Finding the key then consists in choosing the correct value (shown in red on table 14) out of the two, for each byte index.
The correct key is `13 11 1d 7f e3 94 4a 17 f3 07 a7 8b 4d 2b 30 c5`. In total, we have 2^{24} key hypotheses, with a pair of plaintext and corresponding ciphertext to validate the correct key, recovering the AES key can be done on a desktop computer in a few minutes. It takes 145 s on an Intel Xeon E5-1620 v3 CPU to find $y_1 = \texttt{0x30}$, so it would take by extrapolation 757 s, or less than fifteen minutes, to test all keys.

Index	0	1	2	3	4	5	6	7
Forbidden values	13 bf	11 bd	1d b1	7f d3	4f e3	38 94	4a e6	17 bb
Index	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
Forbidden values	5f f3	07 ab	0b a7	27 8b	4d e1	2b 87	30 9c	69 c5

Table 14: Possible key bytes for the correct guess $y_1 =$ `0x30` with the correct key in red.

7.2.2 *Conclusion on the PFA on our baremetal AES*

In this section, we presented a PFA we realized on a self-developed AES on our BCM2837 baremetal setup. The fault is coming from a corruption of the data cache and is persistent. In other words, the fault remains in place while the cache is not invalidated.
This fault able us to obtain 10 000 ciphers from different plaintexts and identify forbidden values for each of their bytes. However, during our experiments we observed two forbidden values per byte which means that we faulted two SBoxes stored in the cache. This shows that the EM injection medium have an important spreading.
But, by adapting the PFA presented in section 4.4.1.1, we were able to reduce the number of possible keys from 2^{128} to 2^{24}.

7.3 `sudo` AUTHENTICATION

In this section, we aim at evaluating the feasibility of faulting a complex Linux program, *i.e.* a program that both load shared libraries at startup and during its runtime. Also, we want to target a security oriented program.
Therefore, we decided to work on the `sudo`-based authentication process of a Linux system. This is an interesting target because the Linux kernel allows several users on the same device. Usually, for every days tasks, the usual user has a limited access to the system, this ensure that he cannot mess up the system configuration while working or access another user protected files. Also, many other users can be configured for different purpose, a famous one is the `www-data` user which is allowed to execute HTTP requests, *i.e.* requests coming from unknown person *via* internet.
The critical operations, like updates or security configuration for instance, must be done by an unrestricted user, usually known as `root`. A restricted user can be allowed to obtain the `root` privileges for a moment to realize some critical operations if it belongs to the correct group: `sudoers` or `wheel` for instance. Some Linux binaries allow to change the current user like `su` or `sudo`, but they require an authentication.

7.3.1 *Attack model*

The program we decide to target is `sudo`. This program allows a user to fake another user identity if he is allowed to. Most of the time, it is used for granting `root` privileges for a specific operation. Users allowed to have the `root` privileges are called `sudoers`.
Regarding this program, there are three attack models.

1. The attacker has access to a `sudoer` user session but does not know its password. This situation can be achieved by being able to execute arbitrary code via a user application. In this case, the attacker can execute code as the compromised user but the Linux segregation mechanism prevents him to access the other users domain.
2. The attacker has the credentials of a user on the system which is not a `sudoer`. In this case, the user is not allowed to fake the `root` identity and the attacker is therefore limited to the compromised user domain. This is the typical situation on shared devices.
3. The attacker has access to a user session without knowing its credentials and which is not a `sudoer`.

In this work, we focus on the first case which corresponds to force the authentication of the compromised user asked by the `sudo` binary without knowing the user's password.

7.3.2 *`sudo` analysis*

Contrary to what we have done with the OpenSSL AES in section 7.1, we do not develop a test program for the `sudo` program. We focus on the binary available in the Debian distribution. This is challenging because, to evaluate such program, we need to fully understand its architecture and to identify its interesting pieces of code, both in the binary and in the code source. Regarding our work, an interesting piece of code is a part of the code that might lead to an unwanted authentication if it is perturbed.
As a first step, we want to analyze the functions that are involved in the authentication process, in other words, the function that actually realize the password verification and all the calling functions that, at a moment, manage the information (an integer in the `sudo`) that the authentication succeed or failed.
Then our first goal is identify these functions in the source code and identify in which binary their assembly code is stored for execution. To do so, there are two strategies:

1. the static top down approach, we find the `main` function of the program in the source code and get down through the different calls until we find the function that generates the information that the authentication succeed or not. This might seems to be a good approach, but as presented in figure 71, on Linux systems,

a program can load a shared library at runtime and call a function from it. This actually happen early in the `main` of the `sudo` program. Therefore, from this point, the best we can do is betting on the loaded shared library and the called function which require an important knowledge about the evaluated program.

2. The other approach is a bottom up analysis at runtime. This analysis consists in using a debugger on the target program. The idea is to identify a function that we now (for sure) is called in the process and put a break point on it. Once this function is executed we identify which function called it and we repeat this until we reach the `main` function. This approach is very effective as it able to identify all the called functions, it also able to identify the shared libraries that are involved. This is very important because the knowledge of which libraries a program relies on is not necessary a common knowledge. The only drawback is that we must know a function that is called during the process we want to analyze. Also, this function must be the lowest possible in the function calls so we do not miss any of them. If it happens that our starting function still calls other ones, we can try to identify them using the source code, and hoping there are not runtime resolution anymore.

Regarding these strategies, the one we have chosen is the second. The reason is that we think that analyzing the execution of a program at runtime is more relevant and less time consuming than analyzing its source code. For instance, the only `sudo` source code is composed of 2492 files (`find ./ -type f | wc -l`) worth 73 MB of code (`du -sh ./`) and the `sudo` program relies on many other libraries.

7.3.2.1 *Runtime analysis*

To realize the runtime analysis, we placed ourselves on a fully mastered setup (*i.e.* with `root` privileges) and we used the `gdb` program.
As mentioned above, the analysis consists in putting a break point on a function we know for sure is called during our process. Considering the `sudo` program, we know that it will compare two hashes (one obtained from the user typed password and one from the `/etc/shadow` file). Therefore, we are confident that the `strcmp` function will be called and is the lowest function in the call tree. So, we start the `sudo` program in a terminal using the `sudo ls` command for instance. Then, start `gdb` and attack the `sudo` process to it using `gdb -pid $(pidof sudo)`. Then we put a break point on `strcmp` using the `b strcmp` command and let the `sudo` program run using the `cont` command. After having typed a password as `sudo` ask, our debugger must reach the break point.
At this moment, the process is stopped at the start of the `strcmp` function and all the already called functions can be shown using the `where` command. However, it turns out that the `strcmp` function is called many times in the `sudo` program (for checking that the user asking for root privileges is actually in the `sudoers` group for instance). There-

fore, we must determine if we are in the correct call of the `strcmp` function, *i.e.* the one that actually compare the hashes. To do so, we display the arguments given to the function. According to the Intel calling convention, during a function call, the first two arguments are stored in `rdi` and `rsi` (`r0` and `r1` for ARM). The content of these register can be displayed using the `x/1s $REGISTER_NAME` command (`x/1s $rdi` and `x/1s $rsi` in our case).
If the inputs looks like we are comparing hashes, it means we have reached our point of interest. In the case we did not, we can let the program executes and reach the next `strcmp` call using the `cont` command.
As `strcmp` is called many times, it is interesting to automatize these steps using the `commands` command as shown in listing 7.7.

```
1  # On first terminal
2  $ sudo ls
3
4  # On second terminal
5  # As root
6  $ gdb --pid $(pidof sudo)
7  # Break if the first character in rdi is '$'
8  (gdb) b strcmp if *(char*)$rdi == 0x24
9  (gdb) commands
10 > x/1s $rdi
11 > x/1s $rsi
12 > where
13 > cont
14 > end
15 (gdb) cont
```

Listing 7.7: Commands to identify the `strcmp` calls of the `sudo` program using `gdb`.

Using these commands, many results will show up but one of them will appear to compare hashes as presented in listing 7.8. Also, as we know that we want to compare hashes and we know that they start with the `$` character, we can customize the break point to break only if the first character of the `rdi` register starts with `$` such as down in listing 7.7 on line 8.
This result shows that we are calling the `__strcmp_avx2()` function from the `libc.so` (line 1). It also shows the two passed arguments on the lines 2 and 3 which are hashes with a common prefix. The next lines (4 to 17) show the functions that actually are on the function call stack, in other words, all the functions called from the `main` program. We can note that we are in an easy case because we are able to effectively identify the correct `strcmp` function. But it might happen that a process creates a new one (using `fork()` or `clone()`) which is used for doing the comparison for instance. In this case, we could not find the correct `strcmp` call because the child process would have a new

```
1  Breakpoint 1, 0x00007f7630680f60 in __strcmp_avx2 () from
   ↪ /usr/lib/libc.so.6
2  0x55cbf54a8b00:       "$6$UxZjWMS1PnDbraCq$O74AxFWWRjkEZnSqO9jobFgl9yn
   ↪ O8cxq8/S5f1tu7ZEPxmXoPOq/ZFtgoKzD3LbAl/fXdZvZ.oCYbFea5cgvo1"
3  0x55cbf547db40:       "$6$UxZjWMS1PnDbraCq$nwD.rOgcz/bgOtx6dpUABwRkCgZ
   ↪ Srlj0O9RhmodmC.bmS.n5TxdXG803C95HnHAs9hM4yJ2Li80pMnDUPJ1b60"
4  #0  0x00007f7630680f60 in __strcmp_avx2 () from /usr/lib/libc.so.6
5  #1  0x00007f763074a60a in ?? () from /usr/lib/security/pam_unix.so
6  #2  0x00007f7630749d9f in ?? () from /usr/lib/security/pam_unix.so
7  #3  0x00007f7630747b46 in pam_sm_authenticate () from
   ↪ /usr/lib/security/pam_unix.so
8  #4  0x00007f7630038882 in ?? () from /usr/lib/libpam.so.0
9  #5  0x00007f7630038181 in pam_authenticate () from /usr/lib/libpam.so.0
10 #6  0x00007f7630052a12 in ?? () from /usr/lib/sudo/sudoers.so
11 #7  0x00007f7630051b45 in ?? () from /usr/lib/sudo/sudoers.so
12 #8  0x00007f7630053d44 in ?? () from /usr/lib/sudo/sudoers.so
13 #9  0x00007f7630070f2e in ?? () from /usr/lib/sudo/sudoers.so
14 #10 0x00007f7630068e0e in ?? () from /usr/lib/sudo/sudoers.so
15 #11 0x000055cbf4a9f520 in ?? ()
16 #12 0x00007f763054b152 in __libc_start_main () from /usr/lib/libc.so.6
17 #13 0x000055cbf4aa178e in ?? ()
```

Listing 7.8: `gdb` output of a `strcmp` call comparing two hashes.

`PID`. `PID` which we used to attach the process to `gdb` using the `gdb --pid $(pidof sudo)` command.
Using this result, we are able to identify that there are thirteen functions that manipulate the information about the authentication and even the shared libraries they are in (`libc.so`, `pam_unix.so`, `libpam.so` and `sudoers.so`). However, we do not have the name of every function (some are displayed with `??`). The reason is that the shared libraries only store the function name of the functions that are supposed to be exported and used by other programs. This name is useful for the dynamic linker to resolve the function calls. The other functions are private to the shared library and therefore are statically linked in the library itself. Therefore, not storing their name save some memory space.
To identify these functions, there are two strategies:

1. the static analysis on the source code. By knowing some functions involved in the call tree, we can try to determine in the source code which functions are calling them and so on. Regarding the result in listing 7.8, this may easily work for the `pam_sm_authenticate()` but hardly with the `strcmp` one as it is called a lot.

2. The other solution is to redo the runtime analysis after having replaced the used shared libraries by the same but with debug symbols. These debug symbols will be identified by `gdb` and therefore displayed. However, this require to recompile these libraries.

The solution we have chosen is the runtime analysis with the debug symbols. The reason is that we think it is more relevant to analyze the process at runtime.

7.3.2.2 *Recompile Debian package from source*

As mentioned, we aim to recompile the packages used in the `sudo` authentication with debug symbols. The Debian distribution allows to do it easily with various tools. The files we want to analyze are the `pam_unix.so`, the `libpam.so`, the `sudoers.so` and the `sudo` executable. To recompile a package, we must start by identify it using the `dpkg -search` command. For instance to determine the package containing the `sudoers.so` file we use `dpkg -search sudoers.so`. To have all the interesting shared object recompile with debug symbols we must recompile the following packages (on a Debian 10.1.0 distribution): `sudo`, `libpam0g:amd64` and `libpam-modules:amd64`. Some of them have the mention `amd64` which is an indication of the supported architecture and might change depending on the target we are recompiling on.
Once we have identified the packages to recompile, we must install the dependencies needed for the package using the `apt-get build-dep` command. For instance, for the `sudo` package we do `apt-get build-dep sudo`.
Then we can download and compile the package using the following command: `DEB_BUILD_OPTIONS="sotrip noopt" fakeroot apt-get -b source sudo` (for the `sudo` package).
Once we have compiled all our packages, we can install them using the `dpkg -i` command such as `dpkg -i sudo_1.8.27-1+deb10u2_-amd64.deb` for the `sudo` package.
Finally, we can do the manipulation presented in listing 7.7 and we obtain a result similar to what is presented in listing 7.9.

```
1  Breakpoint 1, __strcmp_avx2 () at ../sysdeps/x86_64/multiarch/strcmp-avx2.S:92
2  92        ../sysdeps/x86_64/multiarch/strcmp-avx2.S: No such file or directory.
3  0x5637b173b050:        "$6$hH.15uU5laaxuXHY$wtSOcCKWmY1JmyY2CWlVs/8ixyON36ZxQV2RpMJkITzqkIM18lyXNMICoYNIVDe
   ↪  UVXqH0Fs390nl6Lw8m5ArZ0"
4  0x5637b173a200:        "$6$hH.15uU5laaxuXHY$4b7acwY3u21L9Wd8TxQeCIkpmasNufgDzIrScjXreP8oFQA4c.0nZmcYJB2zf5p
   ↪  6rDvPdBC0Fo6JWvquBKaVc."
5  #0  __strcmp_avx2 () at ../sysdeps/x86_64/multiarch/strcmp-avx2.S:92
6  #1  0x00007fb956a006a0 in verify_pwd_hash (p=0x0, hash=0x5637b173a200 "$6$hH.15uU5laaxuXHY$4b7acwY3u21L9Wd8
   ↪  TxQeCIkpmasNufgDzIrScjXreP8oFQA4c.0nZmcYJB2zf5p6rDvPdBC0Fo6JWvquBKaVc.", nullok=0x0) at passverify.c:124
7  #2  0x00007fb956a00050 in _unix_verify_password (pamh=0x5637b173fd30, name=0x5637b173fed0 "toto",
   ↪  p=0x5637b173aad0 "", ctrl=0x40800400) at support.c:792
8  #3  0x00007fb9569fc6fb in pam_sm_authenticate (pamh=0x5637b173fd30, flags=0x8000, argc=0x1,
   ↪  argv=0x5637b1734620) at pam_unix_auth.c:177
9  #4  0x00007fb956a1c024 in _pam_dispatch_aux (pamh=0x5637b173fd30, flags=0x8000, h=0x5637b173c980,
   ↪  resumed=PAM_FALSE, use_cached_chain=0x0) at pam_dispatch.c:110
10 #5  0x00007fb956a1c62c in _pam_dispatch (pamh=0x5637b173fd30, flags=0x8000, choice=0x1) at pam_dispatch.c:411
11 #6  0x00007fb956a1b68b in pam_authenticate (pamh=0x5637b173fd30, flags=0x8000) at pam_auth.c:34
12 #7  0x00007fb956a351f9 in sudo_pam_verify (pw=0x5637b1731318, prompt=0x5637b1740f10 "[sudo] password for
   ↪  toto: ", auth=0x7fb956a94220 <auth_switch>, callback=0x7fff76368d50) at
   ↪  ../../../plugins/sudoers/auth/pam.c:187
13 #8  0x00007fb956a3477e in verify_user (pw=0x5637b1731318, prompt=0x5637b1740f10 "[sudo] password for toto:
   ↪  ", validated=0x2, callback=0x7fff76368d50) at ../../../plugins/sudoers/auth/sudo_auth.c:328
14 #9  0x00007fb956a36a76 in check_user_interactive (validated=0x2, mode=0x1, auth_pw=0x5637b1731318) at
   ↪  ../../../plugins/sudoers/check.c:153
15 #10 0x00007fb956a36d01 in check_user (validated=0x2, mode=0x1) at ../../../plugins/sudoers/check.c:223
16 #11 0x00007fb956a5356c in sudoers_policy_main (argc=0x1, argv=0x7fff76369230, pwflag=0x0, env_add=0x0,
   ↪  verbose=0x0, closure=0x7fff76368ef0) at ../../../plugins/sudoers/sudoers.c:388
17 #12 0x00007fb956a4e516 in sudoers_policy_check (argc=0x1, argv=0x7fff76369230, env_add=0x0,
   ↪  command_infop=0x7fff76368fd8, argv_out=0x7fff76368fe0, user_env_out=0x7fff76368fe8) at
   ↪  ../../../plugins/sudoers/policy.c:866
18 #13 0x00005637b0270d5d in policy_check (plugin=0x5637b0285580 <policy_plugin>, argc=0x1,
   ↪  argv=0x7fff76369230, env_add=0x0, command_info=0x7fff76368fd8, argv_out=0x7fff76368fe0,
   ↪  user_env_out=0x7fff76368fe8) at ../../src/sudo.c:1162
19 #14 0x00005637b026c753 in main (argc=0x2, argv=0x7fff76369228, envp=0x7fff76369240) at ../../src/sudo.c:250
```

Listing 7.9: `gdb` output of a `strcmp` call comparing two hashes with libraries compiled with debug symbols. A bigger representation is available in appendix C.

This results gives more information than the one presented in listing 7.8. First we can identify all the functions involved in the management of the information used to confirm that the authentication succeed or not. More, we also have the source files containing these functions.
At this point, we have the information about every used shared libraries and all the involved functions. But, we do not know if these libraries are loaded at the startup of the program or during the runtime.

7.3.2.3 *Determining the load time of the shared libraries*

Determining the libraries that are loaded at the startup of a program is easily done using the `ldd` command. Therefore, if a library does not appear in the listed ones, we can conclude it is loaded at runtime.
As the `ldd` command takes a file as an input, we must use the real paths of the libraries. On Debian 10, they are at the locations presented in table 15.

File	Path
`sudo`	`/bin/sudo`
`sudoers.so`	`/usr/lib/sudo/sudoers.so`
`libpam.so`	`/lib/x86_64-linux-gnu/libpam.so`

Table 15: Library paths

Using the `ldd` command we could determine that the `sudoers.so` is loaded at runtime but depends on `libpam.so` (which is therefore loaded at the same time) and that `libpam.so` loads during its execution the `pam_unix.so` library.

7.3.2.4 *`sudo` architecture regarding user authentication*

After all the work presented above, we were able to obtain a clear view of the `sudo` architecture which is summarized in figure 75.
This figure shows that the `sudoers.so` library is loaded at runtime, however it is loaded very early in the program execution as it contains the plugin used for the authentication. `sudoers.so` depends on the `libpam.so` and load it at its startup. One may note that "at startup" is a confusing term here as it is not load at the startup of the program (`sudo`) but at the startup of the `sudoers.so` plugin. The important thing to keep in mind is that `sudoers.so` and `libpam.so` are loaded at the same time. Finally, during its execution, `libpam.so` load the `pam_unix.so` library which calls the famous `strcmp` function comparing the password hashes *via* the `verify_pwd_hash` function.
With this overview, we now know which functions of the code we must analyze against our characterized fault model and where to find both their binary and source code.

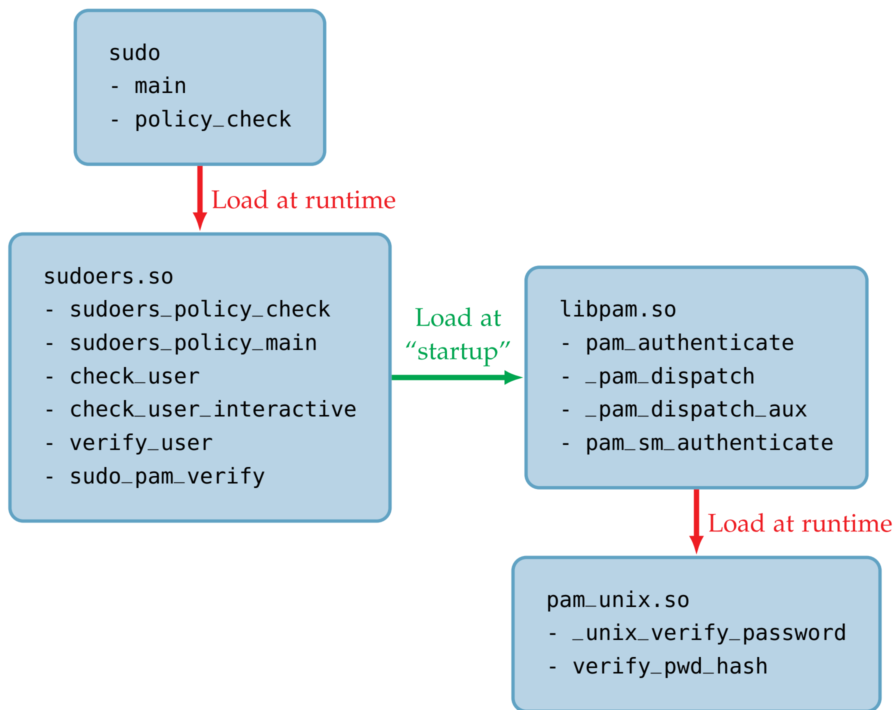


Figure 75: `sudo` calling architecture and dependencies regarding the user authentication process.

7.3.3 *Code analysis*

Regarding the figure 75, we know some functions that it is interesting to fault. Therefore, the next step consists in analyzing them both at source code and binary to highlight attack paths regarding our fault model.
The listing 7.10 shows the part of the `main` function of the `sudo` binary which calls the `policy_check` function and check the returned value.

```
259    ok = policy_check(&policy_plugin, nargc, nargv,
        ↪  env_add,
260                        &command_info, &argv_out,
                           ↪  &user_env_out);
261    sudo_debug_printf(SUDO_DEBUG_INFO, "policy plugin
        ↪  returns %d", ok);
262    if (ok != 1) {
263      if (ok == -2)
264                      usage(1);
265      exit(EXIT_FAILURE); /* plugin printed error message
          ↪  */
266    }
```

Listing 7.10: `sudo` binary `policy` return value checking in `main` function from `sudo.c`

We can see that this piece of code is really simple, the returned value must be equal to `1` to successfully authenticate, otherwise the program terminates with an error. At this point we can identify several

attack paths, forcing the return value to `1` before the critical `if` or avoiding the execution of the `exit` function.
Regarding our fault model, forcing the returned value to `1` seems to be unlikely to happen. Avoiding the execution of the `exit` function might be possible as we can modify the second operand of a comparison instruction. However, its feasibility cannot be determined until we check the binary code.
Actually, this example is quite easy and straightforward, however, among the functions presented in figure 75, some are very complex with hundreds of lines of C code, and thousands of binary. Therefore, doing the analysis by hand is really time consuming and it can be inaccurate.
The solve this analysis problem, we decided to develop an analyzer and simulator software suitable for analyzing Linux programs. In other word, this software aims at automatizing the steps done in section 7.3.2, identify attacks paths regarding a given fault model and simulating actual attacks to determine their relevance. This software is presented in section 7.4.
At this moment, the analysis software is still under development and, because of its youth, it does not already fulfill all of its goals. Therefore, we do not have relevant results to present using it already.
In parallel of the analyzer and simulator development, we attempt to actually force an authentication on the `sudo` of some of our targets.

7.3.4 *Program setup*

As mentioned in section 7.3.1 we placed ourselves in the scenario where the attacker is able to execute code as a legitimate user but without knowing its credentials. We imagined a possible scenario where an attacker is able to execute its own user code and try to force a privilege escalation using a fault attack.
Therefore, our program setup is built to run without privilege access. The tricky part is to be able to synchronize the injection with the actual execution of the functions presented in figure 75.
To achieve the triggering, we decided to use the multi-processing architecture of a Linux system. Two process are created by the user: the first one will handle the communication with the attack bench and send it a signal when the authentication process is supposed to happen. The second process is driven by the first one, *i.e.* the standard input `stdin` of the second process is connected to a pipe shared with the first process so it can write in (for sending the password) while the standard output `stdout` of the second process is connected to the standard output of the first process which is read by the bench during the experiment. With this setup, the bench will have the information if the authentication succeed or failed.
Once everything is set up, the first process start the execution of the `sudo` program using the `execlp` command.
The figure 76 introduces the execution flow of our program.

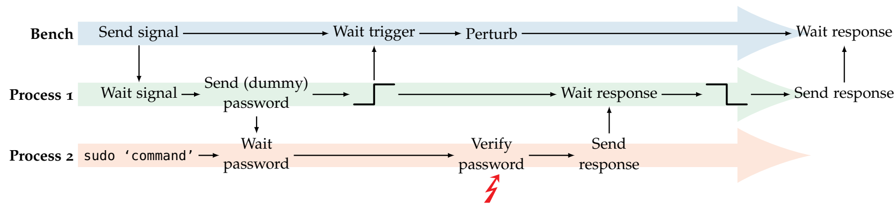


Figure 76: Target program execution flow

This figure presents the general architecture of our setup. The communication between the first process and the bench is done *via* a trigger signal. On our target, we are able to write on the IOs from the user land, using the `/dev/gpiomem` file on ARM targets and using a network LED accessible using the `/sys/class/leds/b43-phy0::radio/brightness` file on Intel targets. As mentioned, the first process communicates with the second one after creating it using the `fork` function and redirect its standard inputs and outputs using the `pipe` and `dup2` functions as presented in listing 7.11.

```
303 if(pipe(pc_fd) < 0)
304   return EXIT_FAILURE;

    /* Some code... */

311 pid = fork();

    /* Some code... */

315 /* Child process */
316 else if(pid == 0){
317   close(cp_fd[0]); /* Close the read end of the child to
      ↪ parent pipe */
318   dup2(cp_fd[1], STDOUT_FILENO); /* Redirect STDOUT to the
      ↪ write end of the child to parent pipe */
319   close(pc_fd[1]); /* Close the write end of the parent to
      ↪ child pipe */
320   dup2(pc_fd[0], STDIN_FILENO); /* Redirect STDIN to the
      ↪ read end of the parent to child pipe */
321   execlp("sudo", "sudo", "-S", "whoami", NULL);
322 }
```

Listing 7.11: C code for the child process in the `sudo` evaluation setup.

This code shows the connecting process of the standard input and output of the child to pipes. Also, one can see that the `sudo` program will execute the `whoami` command. This command returns the user name which is `root` in the case we have the `root` privileges (`UID == 0`). However, any command can be used instead to realize a complete privilege escalation.

Once the second process is set up, the first one can control it by writing into the pipe connected to the standard input and reading from the one connected to the standard output as shown in listing 7.12.

```
325 close(cp_fd[1]); /* Close the write end of the child to
    ↪  parent pipe */
326 close(pc_fd[0]); /* Close the read end of the parent to
    ↪  child pipe */

    /* Some code... */

356 write(pc_fd[1], "pwd\n", 4);

    /* Some code... */

370 read(cp_fd[0], buf, BUFSIZE);
```

Listing 7.12: Part of the C code for the parent process communicating with the child process.

With this setup, we fully automatize a user land application to control a process executing `sudo` while communicating to the attack bench using IOs.
However, we need to evaluate how relevant this setup is before trying to do any attack using it.

7.3.5 *Side channel analysis*

To test our setup, we decided to do a side channel analysis. The goal is to determine if the trigger signal we are outputting to the bench is relevant regarding the authentication process.
As presented in figure 76, the trigger is raised up right after the password is sent. According to the `gdb` trace presented in listing 7.9 the prompt message is display in the `verify_user` and `sudo_pam_verify` functions. This means that the trigger rise up after them.
We can conclude that, at most, the functions from `libpam.so` are executed, then the dynamic linker load `pam_unix.so` and then its functions are executed before backing up the information about the success of the failure of the authentication and executing the command or not.
The figure 77 presents the EM activity of the CPU during both a failed and a succeed authentication.
These traces were acquired on the BCM2711b0 (Raspberry Pi 4) target. The acquisition of these traces require to success the authentication, so we realized them on a mastered target. Therefore, an accessible device is needed for this characterization. The blue signal corresponds to the EM activity and the orange signal corresponds to the trigger signal received by the bench.
To obtained these traces, we placed the side channel probe at the position X = 6.5 mm and Y = 5.9 mm regarding the BCM2711b0 layout available in figure 60.

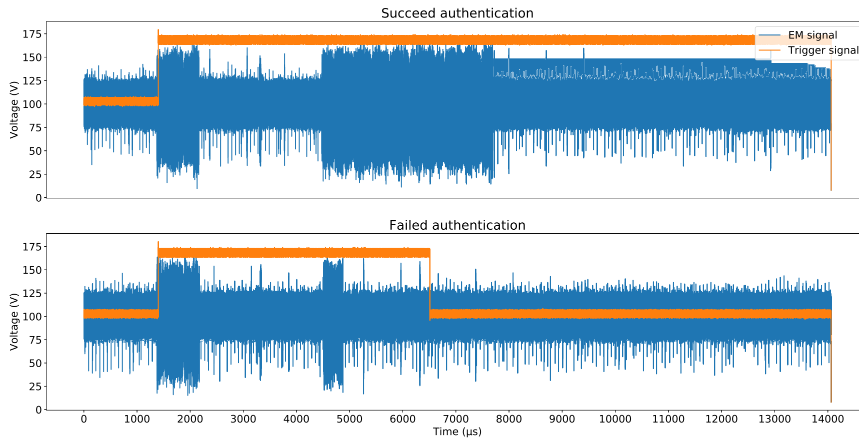


Figure 77: EM activity of the CPU during a user authentication on BCM2711b0

Regarding the traces on figure 77, we can identify two activity areas. One constant in time and one time variable depending on the success or the failure of the authentication. At this point we can suppose that the first activity area corresponds to the load of the `pam_unix.so` library in the memory while the second area matches with the hash comparison.
To confirm this hypothesis, we can check the compared hashes. Indeed, as shown in listing 7.9, the two hashes have a common prefix `$6$hH.15uU5laaxuXHY$` which is added to the real password hash by the Linux kernel before the comparison. As this prefix is always the same, the `strcmp` never fail while comparing the first part of the hashes. Also, this prefix is 20 B long while the compared hash (prefix included) are 106 B long. Therefore, comparing two different hashes must be five times faster than comparing the same hashes, if the first character after the prefix is different as it is the case in listing 7.9.
Regarding the figure 77, we can see that in the case of a failed authentication, the second activity area lasts for around 400 ms (from 4500 ms to 4900 ms). While, in the case of a succeed authentication, the second activity area lasts for around 2200 ms (from 4500 ms to 7700 ms). As the ratio of the execution time of these areas ($400/2200 = 0.182$) is close to the ratio of identity between the hashes ($20/106 = 0.189$), we can consider that they are correlated and that we actually observe the hash comparison. However, we cannot confirm more this hypothesis as it would require us to be able to forge a hash which partially match the real one. But as the hash algorithm is SHA512 (identified by the \$6\$ identifier in the common prefix) obtaining hashes which partially collide require to brute force them. This is too much time consuming to actually doing it, for instance, we need to generate $256^{10} = 2^{80}$ passwords to have a collision with 10 common bytes.

7.3.6 *Exploitation*

Regarding our side channel analysis, we are confident that our setup able us to observe a relevant trigger regarding the `sudo` authentication

and in particular the hash comparison. However, we are limited to a sub-part of the whole process execution because the trigger is raised up just after the password is sent.
Another strategy would have been to rise the trigger just before starting the `sudo` program. However, this would lead to a long execution window making it hard to attack.
This is why we decided to stick with the first solution, as we can observe the program behavior with a reasonable time window, we decided to attack the authentication with a perturbation (laser on BCM2711b0 and EM on Intel Core i3). As we did not already fully analyzed the program, we sweep over the time window and observe the consequences.
However, this method did not give us interesting results already. This might comes from the fact that, despite we have the smallest time window possible, it remains large regarding the CPU frequency. Therefore, to have a not to long experiment we had to use a large step for the sweep and we therefore must have missed the interesting areas to fault.
This large time window shows how important it is to analyze such program. The goal is to reduce the interesting time window to accelerate the sweeping process.

7.3.6.1 *Enhancing existing attack*

Another interesting way to evaluate a system is to enhance proofs of concept attacks to turn them into real attack.
In [166], the authors successfully force two different hashes to be considered as the same by the `strcmp` function used in the `verify_pwd_hash` function of the `pam_unix.so` file. Contrary to us, they target the `su` program, which is close to the `sudo` program as it able to change the user identity (and therefore its privilege level).
However, to achieve this attack, the authors modified the `libpam` library by adding a trigger in the `libpam_misc` module. As this addition requires to have the `root` privilege, this attack is not a privilege escalation but a proof of concept of the weakness of the `strcmp` function against the instruction skip fault model. The attack consists in exiting the comparing loop of the `strcmp` function by skipping a `cbz` instruction (Compare and Branch on Zero in ARM).
Regarding this work and, using our attack setup and analysis, we should be able to realize a real privilege escalation as we are able to identify the `strcmp` function execution, with `root` privileges, using a user land trigger and side channels. However, we did not identified the same fault model as theirs (instruction skip) and their target is not clearly identified.
Another enhancement would be to identify the `strcmp` function execution with a pattern recognition method. Doing so, the attack would be portable on any target leaking the library loading and `strcmp` execution.

7.3.7 *Conclusion on the forced authentication*

In this section, we presented our work aiming at evaluating the user authentication process of the `sudo` program. This program is an interesting example of a complete Linux program, it works with Linux credentials management and it involves shared libraries and the dynamic linker in several ways.
The variety of mechanisms involved in the execution of a Linux program and the important size of both the source code and the binary code of the target program make an exhaustive analysis really time consuming. Therefore, we focused on determining the interesting parts of our program such as the pieces of code manipulating the information storing if the authentication succeed or failed.
We have done this analysis by recompiling the target program with debug symbols and using the `gdb` tool. This way, we were able to identify the architecture of the `sudo` program regarding the user authentication. However, for a more precise analysis of the involved functions, and due to the important size of both the code and the binary, we need a more automatized tool.
In parallel of the code analysis, we realized a software setup which able to attack the `sudo` program from the user land and we analyzed it using side channels. This analysis gives us the information that we are able to have a trigger around the dynamic linker execution and the hash comparison. However, we were not able to achieve an attack forcing the authentication due to our lack of precision regarding the important time window the program executes in. This highlights the need of a tool that helps us to determine the interesting pieces of code to fault and to time our perturbation with them.
However, a recent work [166], successfully forced an authentication on a modified `libpam` with an instruction skip fault model on the `strcmp` function. As our setup does not need any `root` privilege and that we are able to identify the `strcmp` execution *via* side channels, if we are able to realize instruction skips, we might be able to realize a real privilege escalation.
Despite this would lead to a successful attack, it is only one identified attack path and there is a lot of work remaining to actually assess the security of such program.

7.4 ANALYSIS TOOLS

As we have seen in the above section, analyzing a Linux program is a complex task due to its size and some kernel mechanisms involved in the execution. Also, due to their large execution time, blindly sweep over the execution time to find an interesting moment to attack is a very time consuming step. Therefore, being able to simulate an attack regarding a fault model on a whole program will save a lot of time in the evaluation of such programs.
For these reasons, we decided to develop an analysis and fault simulation tool which is adapted for Linux programs. This tool aims at

automatizing the steps we have done in the `sudo` program analysis (section 7.3). As we have done it with `gdb` one can think than we can re-use it and scripts it to realize both the analysis and simulation part. The main issue in using `gdb` is that it is architecture dependent and if, for instance, we want to analyze an ARM compiled program we have to work on an ARM system or use an emulation solution. Therefore, we decided to base our tool on a architecture independent debugging solution: `angr`[3].
Using this solution, we want to realize at least the same operation we have done with `gdb` in section 7.3 but also set up a simulation environment suitable and actually simulate fault in during the program execution. The global architecture of our wanted solution is presented in figure 78.

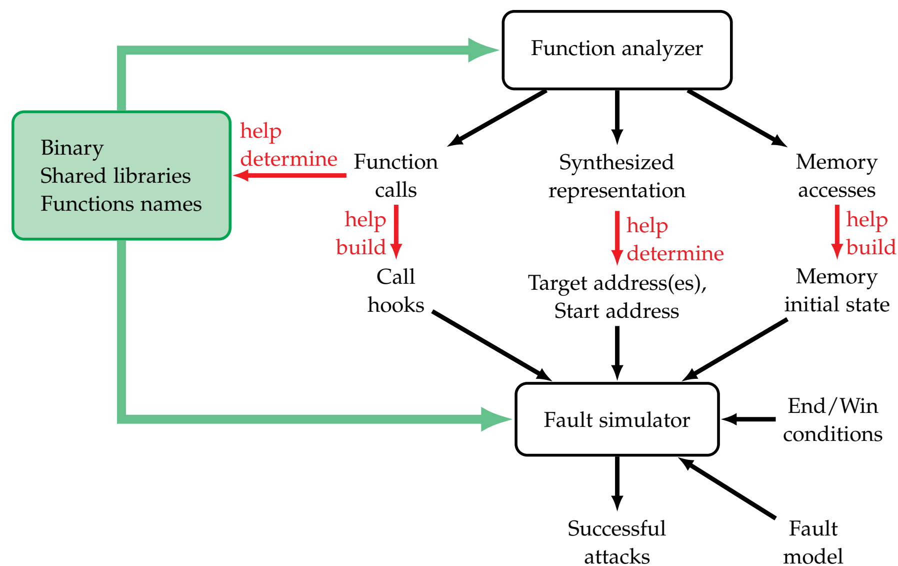


Figure 78: Analyzer and simulator software architecture.

This figure presents the global architecture of the analysis and simulation software we aim to have. It is composed of two main parts: the function analyzer and the fault simulator. Both take as the main inputs the binary file, the shared libraries and the functions to analyze of simulate.

7.4.1 *The function analyzer*

The function analyzer aims at determining the architecture of the target program such as we have done with `sudo` in section 7.3. It will also help in getting the information needed to setup the simulation environment. To do so it is composed of three main functions.
The first one is a function call analyzer which aims at determining all the calls that are done during the program execution. This helps to determine the involved functions and shared libraries. At the very beginning this block is only aware that there is a `main` function and build the call tree from this point. Then we might be able to add the shared

3 `https://angr.io/`

libraries and functions we want to analyze and so on. However, it is important to keep in mind that for the analysis to be done efficiently the shared libraries have to be compiled with debug symbols. Otherwise the calls cannot be resolved. Regarding the simulation part, it helps to determine the calls we want to hook during the simulation, this aims at reducing the actual simulated code and focus only on the interesting part. Indeed, some functions such as error printing are not needed to be correctly simulated and it is therefore interesting to hook them to fasten the simulation.
The second function is a function printer. It aims at providing a synthesized but complete overview of a given function. This helps at understanding the architecture of a function and therefore determine the interesting instructions to fault. This able us to avoid to test a fault on all instructions of the program but only focus on interesting ones. However, in the case of very large functions (like the `sudoers_policy_main` in `sudoers.so`), this tool is not efficient enough to provide a view which is actually exploitable for determining interesting instructions to fault by a human. Therefore, automatizing this step would give a huge added value to our software and is an interesting topic to work on. This function summary also helps at identifying a start address for the simulation. Indeed, simulating the whole program is useless if we aim at targeting a precise piece of code and would be really time consuming. Therefore, identifying the address where to start the simulation is important. Doing so also imply the necessity of given an initial state for the simulation environment as the simulation will not start from the `main` function. The printing function can display the disassembly in a shell as in figure 79 with arrows for the branch instructions and by coloring the basic blocs, in Tikz as in appendix D with also arrows for the branch instructions and basic blocs in actual blocs and in HTML[4] with basic blocs in actual blocs and clickable links for branches with a highlight of the targeted basic bloc.
The third function aims at determining all the memory accesses done during the program execution. This is useful to initialize the relevant parts of the simulation memory, in particular when we do not start the program from the `main` function as suggested above.
All the tools presented in this section are developed with Python3 and are based on the `angr` library which is itself based on `capstone`[5] and `unicorn`[6].

7.4.2 *Fault simulator*

The fault simulator aims at testing a program security by injecting faults in it at runtime. The main issue about simulating a Linux program is the management of the shared libraries. However, `angr` man-

4 Live examples with `sudoers_policy_check` and `sudoers_policy_main` available at https://thomas.trouchkine.com/demo/
5 http://www.capstone-engine.org/
6 https://www.unicorn-engine.org/docs/

```
                            0x419314:   mov     r3, fp
                            0x419318:   str     r2, [sp]
                            0x41931c:   bl      sudoers_policy_main @ 0x42029c
                            0x419320:   cmp     r0, #1
                            0x419324:   mov     r6, r0
                        +-- 0x419328:   beq     #0x419370
+---+---+---+-------+---|-> 0x41932c:   ldr     r0, [pc, #0xb8]
|   |   |   |       |   |   0x419330:   ldr     r1, [pc, #0xb8]
|   |   |   |       |   |   0x419334:   add     r0, pc, r0
|   |   |   |       |   |   0x419338:   mov     r3, r5
|   |   |   |       |   |   0x41933c:   add     r0, r0, #0x70
|   |   |   |       |   |   0x419340:   ldr     r2, [pc, #0xac]
|   |   |   |       |   |   0x419344:   str     r6, [sp]
|   |   |   |       |   |   0x419348:   add     r1, pc, r1
|   |   |   |       |   |   0x41934c:   bl      sudo_debug_exit_int_v1 @ 0x403f88
|   |   |   |       |   |   0x419350:   ldr     r3, [sp, #0xc]
|   |   |   |       |   |   0x419354:   ldr     r2, [sp, #0x1c]
|   |   |   |       |   |   0x419358:   mov     r0, r6
|   |   |   |       |   |   0x41935c:   ldr     r3, [r3]
|   |   |   |       |   |   0x419360:   cmp     r2, r3
|   |   |   |   +---|---|-- 0x419364:   bne     #0x4193cc
|   |   |   |   |   |   |   0x419368:   add     sp, sp, #0x24
|   |   |   |   |   |   |   0x41936c:   pop     {r4, r5, r6, r7, r8, sb, sl, fp, pc}
|   |   |   |   |   |   +-> 0x419370:   ldr     r3, [pc, #0x80]
|   |   |   |   |   |       0x419374:   ldr     r2, [pc, #0x80]
|   |   |   |   |   |       0x419378:   add     r3, pc, r3
|   |   |   |   |   |       0x41937c:   ldr     r3, [r3]
|   |   |   |   |   |       0x419380:   cmp     r3, r2
|   |   |   |   |   +------ 0x419384:   bls     #0x41932c
|   |   |   |   |           0x419388:   ldr     r3, [pc, #0x70]
|   |   |   |   |           0x41938c:   ldr     r3, [r4, r3]
|   |   |   |   |           0x419390:   ldr     r2, [r3, #0x82c]
|   |   |   |   |           0x419394:   cmp     r2, #0
|   |   |   +---|---------- 0x419398:   bne     #0x41932c
|   |   |       |           0x41939c:   ldr     r2, [r3, #0x848]
|   |   |       |           0x4193a0:   cmp     r2, #0
|   |   +-------|---------- 0x4193a4:   bne     #0x41932c
|   |           |           0x4193a8:   ldr     r3, [r3, #0x880]
|   |           |           0x4193ac:   cmp     r3, #0
|   +-----------|---------- 0x4193b0:   bne     #0x41932c
|               |           0x4193b4:   bl      sudo_auth_needs_end_session @ 0x40480c
|               |           0x4193b8:   cmp     r0, #0
|               |           0x4193bc:   ldreq   r3, [pc, #0x40]
|               |           0x4193c0:   ldreq   r3, [r4, r3]
|               |           0x4193c4:   streq   r0, [r3, #0xc]
+---------------|---------- 0x4193c8:   b       #0x41932c
                +---------> 0x4193cc:   bl      __stack_chk_fail @ 0x403898
```

Figure 79: `sudoers_policy_check` function partial disassembly printed in a shell.

age them smoothly as it automatically initialize a suitable memory for the program and able to hook the unresolved functions.

However, as mentioned above in some cases, we do not want to start the program execution from the start so we have to provide the target address and modify the memory initialization. Actually, by default `angr` allocate a memory but set every address as `UNINITIALIZED`.

Also, as we want to simulate fault injection, we need to inject faults. This is done *via* a fault generator. Currently, the fault generator can only generates fault instructions. The reason is that we focused on developing what we actually observed. The faulted instruction replaces the actual instruction via a hook at the target address.

The last things to configure in the fault simulator are the ending condition and the winning condition. The ending condition is useful when we do not want to wait for the program to be fully simulated but prematurely terminating it when we know that we can verify if the attack succeed or not, usually we simply check that we reached a certain instruction (the return of a specific function for instance). The winning condition able the simulator to verify if the attack was successful or not, if it is the case, it stores the fault, *i.e.* the fault in-

struction and the address and try with another fault and/or another target address. This process is summarized in figure 80.

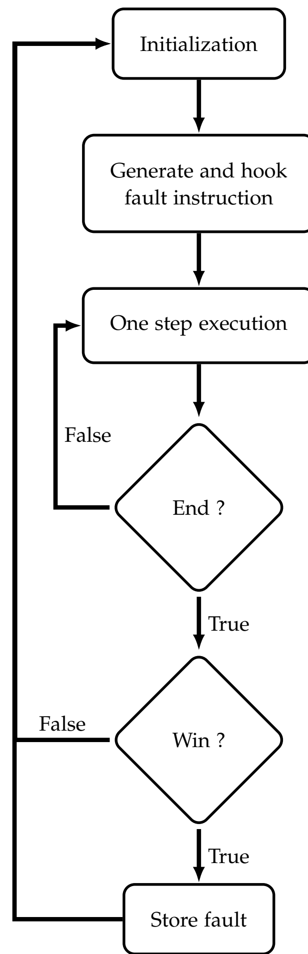


Figure 80: Fault simulation process.

This figure shows the main process of the fault simulator, as one can see, the execution is step by step and the end condition is check after every step. Therefore, our simulator is instruction accurate, this is mandatory using `angr`. Another issue with the simulator is that, in the case of dynamic linking, the dynamic linker behavior is not simulated. Therefore, it is possible that our analysis miss the case the dynamic linker is faulted.
However, we think that adding the simulation of the dynamic will be too time consuming. An interesting way would be to analyze the dynamic linker as a standalone program (hoping it does not do a dynamic linking itself) and assess its security against faults. If it appears that the dynamic linker is sensitive to faults, it might be interesting to model these faults an integrate them in the possible fault models of our simulator.

7.5 EVALUATION CONCLUSION

This chapter presented our work to evaluate the security of some Linux programs against the faults we characterized on our targets.

We managed to successfully realize a DFA on the OpenSSL implementation of the AES and a PFA on our AES implemented on our Raspberry Pi 3 bare metal setup. Regarding the `sudo` program we identified an exploitable attack path from a fault attack presented in **??** which is suitable with our program attack setup. However, the authors have a different fault model than the one we characterized, therefore, we did not succeed to do the attack when we tried. But this is only one attack path and many more might be available.
The attack on the AES gives us the information that despite CPU optimizations, it is still possible to synchronize a fault injection with a specific moment of the program execution. However, the results we observe does not match with a sequential execution of the AES. Regarding the `sudo` program, our side channel analysis confirm that the parallel execution is negligible as we were able to observe the timing differences on the comparison of similar or different hashes in the `strcmp` function.
We also defined an analysis method for complex Linux programs like the `sudo` one and we started to develop an software which aims at analyzing such programs but also simulate them while injecting faults to assess their resilience against faults.
This kind of software highlight many challenges as it aims at simulating a complex system with shared library and a dynamic linker. At the moment, we do not simulate the dynamic linker. We think the best thing to do is to analyze it as a standalone software and assess its security against faults. Therefore, not considering it in the simulation will be relevant. Also, all shared libraries are not loaded during the simulation but some calls are simply hooked at runtime. Despite this reduces the coverage of the simulation, it makes it faster than an exhaustive one. But to do so, we need to identify the relevant functions to actually simulate and the one we can hook, this is what the analysis part of our software is designed for.

Part III

CONCLUSION

This is how it ends.

8 CONCLUSION

During this thesis, we aim at providing elements that would help the evaluation of modern devices, also known as SoCs, against physical attacks and in particular, perturbation attacks.
In chapter 1 we presented the need of security evaluation in general and the special need regarding new technologies, in particular mobile devices. Indeed, these mobile devices propose a wide variety of services, including sensitive ones such as identification, payment and healthcare. Doing so, they slowly replace security oriented, dedicated and evaluated devices, also known as SEs. Therefore, to match new usages and not reduce our security level, we need to be able to evaluated the security of these new devices.
Because, we already evaluate SEs, we have a good idea on the existing physical attacks. However, the differences between SEs and modern SoCs rise the question about the portability of the evaluation process. This is why we presented in chapter 2 the main differences between these devices and we highlighted that modern SoCs have a more complex CPU than SEs, in many ways. More cores with more optimization, which also execute a more complex architecture.
By considering this complexity, we developed a fault characterization method we presented in chapter 5 and that we used in chapter 6 on three targets representing the available modern devices with two ARM and one Intel devices. This characterization work helped us to determine that all the targets are faulted in the same way independently of the injection medium, here EM and laser perturbations, and independently of their architecture. Despite, the exact fault model may vary from a target to another, we were able to modify the second operand of a target instruction on all of them. This gives us the information that these targets, and maybe many more, share a similar mechanism which behave in the same way to perturbations. This is interesting in many ways, the first one is that if we are able to identify the mechanism, we can propose a countermeasure and apply it to all the devices with these mechanism. However, this also means that if an attack using a fault on this mechanism works on a device, there is great chances that it works on another devices with the same mechanism.
However, we did not focused on identifying the mechanism and proposing a countermeasure but we kept working on the evaluation of SoCs. Indeed, in the chapter 7 we focused on assessing the exploitability of the fault model we characterized. As we worked on the Linux ecosystem, we identified all the specific elements of a program execution in the environment, in particular the dynamic libraries, the dynamic linker and the asynchronous process execution. Despite these mechanisms, we were able to attack the AES algorithm in two different ways, one with the operand corruption fault model

doing a DFA and one with a persistent corruption in the cache using a PFA. The DFA confirmed us that we are actually able to target a specific round of the AES, even in an asynchronous and time variable execution environment while the PFA able to highlight that synchronization issues may be avoided if a persistent fault is set in the memory subsystem of the device.
The final evaluation we have done is on the sudo program. Our aim was to evaluate the Linux authentication process regarding our fault model. However, the complexity and size of such program makes the analysis of such software a tricky and time consuming business requiring debug tools, binary analysis and fault simulator software. However, as this is a new topic, there is no such tools available for modern devices. Therefore, we presented the analysis method and the first development of tools which aim at easing the evaluation of such software regarding fault attacks.
Answering all these questions gave us a large but precise overview of the requirements for evaluating a modern device. However, and despite it is important, we did not discuss about specific to SoCs countermeasures in this work as it would be to time consuming regarding the time available for a thesis.

IDENTIFIED RESEARCH TOPICS AND FUTURE WORKS

This work helped us to identify several relevant research topics and future works which would help the scientific community to keep working on the SoC evaluation against perturbation attacks.

An interesting topic is the development of an open signal manager for attack benchs. Indeed, as mentioned in section 6.1.1, analyzing the signals coming from a DUT and being able to produce a relevant trigger signal for synchronizing a perturbation with a moment of the target execution will be very helpful for the analysis of modern devices. The main reason is that most of them, in particular smartphones, are closed devices and producing an efficient trigger is a tricky business. Pattern matching methods are known to be efficient but their implementations remain under intellectual property. An open and public initiative on this topic would help researchers in testing and evaluating relevant attacks while increasing the repeatability of experiments on closed targets.

Regarding our characterization, as we identify similar fault models on different targets using different injection mediums, it is important to determine which mechanism is common to these targets, why its perturbation induces an operand corruption on the executed instructions. Also, regarding this fault model, an interesting topic would be the development of a solution to protect instructions against corruptions.

From the software point of view, our work highlight the importance of the dynamic linker in the execution of a program in the Linux environment. An interesting topic would be to analyze the behavior of this piece of code regarding perturbation attacks. Moreover, identifying security concerned pieces of code in the Linux kernel and propose an analysis against faults and even countermeasure should be an important research topic in the future.

Finally, the last interesting topic we identified is the development of a tool which helps in the analysis of programs running in the Linux environment against faults. Indeed, currently existing code analyzers work on reduced pieces of code in a simple environment (limited links and sequential execution).

SCIENTIFIC COMMUNICATION

International conferences

1. Thomas Trouchkine, Guillaume Bouffard, and Jessy Clediere. "Fault Injection Characterization on modern CPUs – From the ISA to the Micro-Architecture." In: *WISTP 2019, Paris, France*. 2019

International journals

1. Thomas Trouchkine et al. "Electromagnetic Fault Injection against a complex CPU, toward a new micro-architectural fault models." In: *Journal of Cryptographic Engineering (JCEN)*. 2020

National conferences

1. Thomas Trouchkine, Guillaume Bouffard, and Jessy Clediere. "EM Injection Vs. Modern CPU - Fault Characterization And AES Differential Fault Analysis." In: *Comptatibilité électromagnétique France 2020*. 2020

Workshops

1. Thomas Trouchkine, Guillaume Bouffard, and Jessy Clediere. "Perturbation attack on modern CPUs, from the fault model to the exploitation." In: *Journée thématique sur les attaques par injection de fautes (JAIF)*. 2020
2. Guillaume Bouffard et al. "Radically secure computing." In: *SILM seminar 2020*. 2020
3. Thomas Trouchkine et al. "Do not trust modern System-on-Chips." In: *PHISIC 2019*. 2019

4. Thomas Trouchkine et al. "Do not trust modern System on Chip." In: *Journée thématique sur les attaques par injection de fautes (JAIF)*. 2019

5. Thomas Trouchkine. "EM fault injection on Raspberry Pi 3 and ZedBoard." In: *GDR SoC2 research group*. 2018

6. Thomas Trouchkine. "Problems and state of the art of faults injection on Systems on Chip." In: *PHISIC 2018*. 2018

All the articles and slides are available on my website at `https://thomas.trouchkine.com/publications/`

OPEN SOURCED PROJECTS

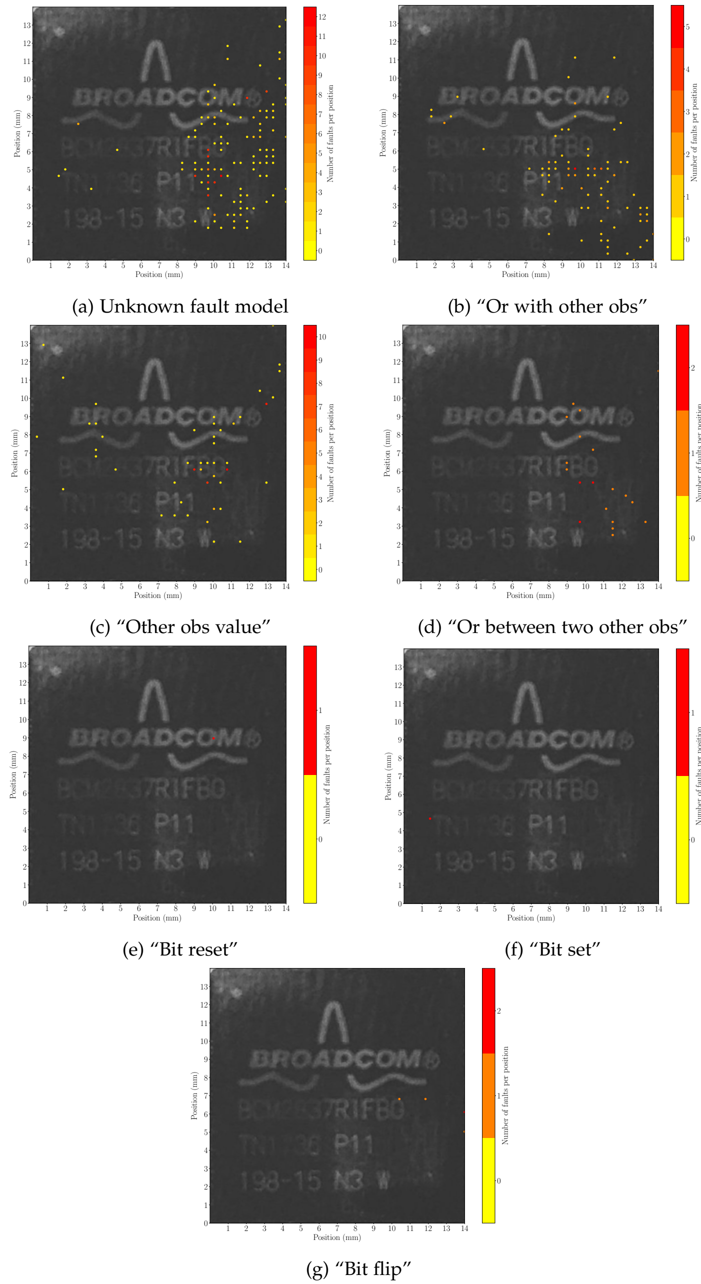
- Plotter python package (`https://plotter-doc.xyz/`)

Part IV

APPENDIX

A

BCM2837 MAPS PER FAULT MODEL



(a) Unknown fault model

(b) “Or with other obs”

(c) “Other obs value”

(d) “Or between two other obs”

(e) “Bit reset”

(f) “Bit set”

(g) “Bit flip”

Figure 81: BCM2837 hot spots leading to the different observed fault models

B

FAULT ANALYZER INTERFACE

Currently, the fault analyzer (presented in section 6.1.3.2) proposes two interfaces, a command line interface based on the Cmd framework[1] and a graphical interface based on the Gtk3 framework[2].
The listings B.1 to B.3 shows how the command line interface look like.
The listing B.1 presents the first view which lists the available experiments to analyze. The start next to the index means that the experiment is already analyzed and therefore, that the results can be displayed. To start an analysis, the user just has to type `analyze <index>` and to print the available results `print <index>`. Which will display something like presented in listing B.2.

```
fa> print

Manips
========
[0]* bcm2837_andR8_iv4_EM_fix_20200127
[1]* bcm2837_orrR5_iv3_EM_fix_20200124
[2]  bcm2711b0_orrR5_iv5_laser_20x_4v_20ns_carto_zoom_20200210
[3]  bcm2711b0_orrR5_iv5_laser_20x_4v_20ns_carto_full_20200210
[4]  bcm2711b0_orrR5_iv5_laser_20x_4v_20ns_carto_core_20200211
[5]  bcm2711b0_sudo_debut_laser_20x_4v_20ns
```

Listing B.1: Fault analyzer command line interface (List of experiments). Experiments with a star (*) are already analyzed.

```
fa> print 0

bcm2837_andR8_iv4_EM_fix_20200127 available results
=====================================================
[0] General statistics
[1] Effect of the power value
[2] Effect of the delay
[3] Observed statistics
[4] Faulted values statistics
[5] Fault model statistics
```

Listing B.2: Fault analyzer command line interface (List of results)

This listing presents the available results for the selected experiment, the results vary from an experiment to another but some are always available such as the general statistics. To display a specific result, the user has to type `print <experiment_index> <result_index>` or

1 https://docs.python.org/3/library/cmd.html
2 https://python-gtk-3-tutorial.readthedocs.io/en/latest/

```
fa> print 0 0

bcm2837_orrR5_iv3_EM_fix_20200124 results
===========================================
General statistics
+-----------------------------------------+-------+
|                Statistic                | Value |
+-----------------------------------------+-------+
|        Number of operations to do       | 21000 |
|         Number of operation done        | 13538 |
|           Percentage done (%)           | 64.47 |
|            Number of reboots            |  800  |
|        Percentage of reboots (%)        | 5.909 |
|     Number of responses bad formated    |  3145 |
| Percentage of responses bad formated (%) | 23.23 |
|             Number of faults            |  436  |
|         Percentage of faults (%)        | 3.221 |
|          Number of faulted obs          |  489  |
|       Average faulted obs per fault     | 1.122 |
+-----------------------------------------+-------+
```

Listing B.3: Fault analyzer command line interface (Result)

`print <experiment_index> a` to display all of them. This display the results in the form of a table as presented in listing B.3.
This listing presents a specific result displayed in the form of a table. It is possible to display the HTML or the LaTeX version of this table via the `-html` or `-latex` flags. Also, it is possible to plot a result *via* the `plot` command or to merge to results in a new one *via* the `merge` command. This is useful for comparing the results of two different experiments for instance.
The second interface, is the graphical interface based on the Gtk3 framework and is presented in figure 82.

Figure 82: Fault analyzer graphical interface

This interface is divided in two parts, the left part lists the available experiments and the right part displays the selected results. However, this interface is not mature and does not implement all the functions such as the plotting or the merge.
The implementation of such interfaces is made as simple and straightforward as possible. To have a functional interface the only needed thing is to implement functions that are mapped on the analyzer `Core`

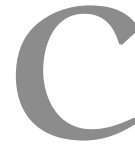
functions, such as `print`, `plot`, `analyze`, *etc*. Then the interface must simply get the `Core` class as argument such as presented in listing B.4.

```
if __name__ == "__main__":
    c = Core(**CONFIG)

    # Argument parsing...

    if args.gui:
        interface = Gtk3FaultAnalyzer(c)
        interface.start_interface()
    else:
        interface = Cmdline(c)
        interface.cmdloop()
```

Listing B.4: `main` of the fault analyzer



GDB ANALYSIS OF `SUDO` WITH DEBUG SYMBOLS

```
1  Breakpoint 1, __strcmp_avx2 () at
   ↪  ../sysdeps/x86_64/multiarch/strcmp-avx2.S:92
2  92        ../sysdeps/x86_64/multiarch/strcmp-avx2.S: No such file or
   ↪  directory.
3  0x5637b173b050:        "$6$hH.15uU5laaxuXHY$wtSOcCKWmY1JmyY2CWlVs/8ixyO
   ↪  N36ZxQV2RpMJkITzqkIM18lyXNMICoYNIVDeUVXqH0Fs390nl6Lw8m5ArZ0"
4  0x5637b173a200:        "$6$hH.15uU5laaxuXHY$4b7acwY3u21L9Wd8TxQeCIkpmas
   ↪  NufgDzIrScjXreP8oFQA4c.0nZmcYJB2zf5p6rDvPdBC0Fo6JWvquBKaVc."
5  #0  __strcmp_avx2 () at ../sysdeps/x86_64/multiarch/strcmp-avx2.S:92
6  #1  0x00007fb956a006a0 in verify_pwd_hash (p=0x0, hash=0x5637b173a200
   ↪  "$6$hH.15uU5laaxuXHY$4b7acwY3u21L9Wd8TxQeCIkpmasNufgDzIrScjXreP8oFQ
   ↪  A4c.0nZmcYJB2zf5p6rDvPdBC0Fo6JWvquBKaVc.", nullok=0x0) at
   ↪  passverify.c:124
7  #2  0x00007fb956a00050 in _unix_verify_password (pamh=0x5637b173fd30,
   ↪  name=0x5637b173fed0 "toto", p=0x5637b173aad0 "", ctrl=0x40800400)
   ↪  at support.c:792
8  #3  0x00007fb9569fc6fb in pam_sm_authenticate (pamh=0x5637b173fd30,
   ↪  flags=0x8000, argc=0x1, argv=0x5637b1734620) at pam_unix_auth.c:177
9  #4  0x00007fb956a1c024 in _pam_dispatch_aux (pamh=0x5637b173fd30,
   ↪  flags=0x8000, h=0x5637b173c980, resumed=PAM_FALSE,
   ↪  use_cached_chain=0x0) at pam_dispatch.c:110
10 #5  0x00007fb956a1c62c in _pam_dispatch (pamh=0x5637b173fd30,
   ↪  flags=0x8000, choice=0x1) at pam_dispatch.c:411
11 #6  0x00007fb956a1b68b in pam_authenticate (pamh=0x5637b173fd30,
   ↪  flags=0x8000) at pam_auth.c:34
12 #7  0x00007fb956a351f9 in sudo_pam_verify (pw=0x5637b1731318,
   ↪  prompt=0x5637b1740f10 "[sudo] password for toto: ",
   ↪  auth=0x7fb956a94220 <auth_switch>, callback=0x7fff76368d50) at
   ↪  ../../../plugins/sudoers/auth/pam.c:187
13 #8  0x00007fb956a3477e in verify_user (pw=0x5637b1731318,
   ↪  prompt=0x5637b1740f10 "[sudo] password for toto: ", validated=0x2,
   ↪  callback=0x7fff76368d50) at
   ↪  ../../../plugins/sudoers/auth/sudo_auth.c:328
14 #9  0x00007fb956a36a76 in check_user_interactive (validated=0x2,
   ↪  mode=0x1, auth_pw=0x5637b1731318) at
   ↪  ../../../plugins/sudoers/check.c:153
15 #10 0x00007fb956a36d01 in check_user (validated=0x2, mode=0x1) at
   ↪  ../../../plugins/sudoers/check.c:223
16 #11 0x00007fb956a5356c in sudoers_policy_main (argc=0x1,
   ↪  argv=0x7fff76369230, pwflag=0x0, env_add=0x0, verbose=0x0,
   ↪  closure=0x7fff76368ef0) at ../../../plugins/sudoers/sudoers.c:388
17 #12 0x00007fb956a4e516 in sudoers_policy_check (argc=0x1,
   ↪  argv=0x7fff76369230, env_add=0x0, command_infop=0x7fff76368fd8,
   ↪  argv_out=0x7fff76368fe0, user_env_out=0x7fff76368fe8) at
   ↪  ../../../plugins/sudoers/policy.c:866
18 #13 0x00005637b0270d5d in policy_check (plugin=0x5637b0285580
   ↪  <policy_plugin>, argc=0x1, argv=0x7fff76369230, env_add=0x0,
   ↪  command_info=0x7fff76368fd8, argv_out=0x7fff76368fe0,
   ↪  user_env_out=0x7fff76368fe8) at ../../src/sudo.c:1162
19 #14 0x00005637b026c753 in main (argc=0x2, argv=0x7fff76369228,
   ↪  envp=0x7fff76369240) at ../../src/sudo.c:250
```

Listing C.1: `gdb` output of a `strcmp` call comparing two hashes with libraries compiled with debug symbols.

D SUDOERS_POLICY_CHECK() DISASSEMBLY

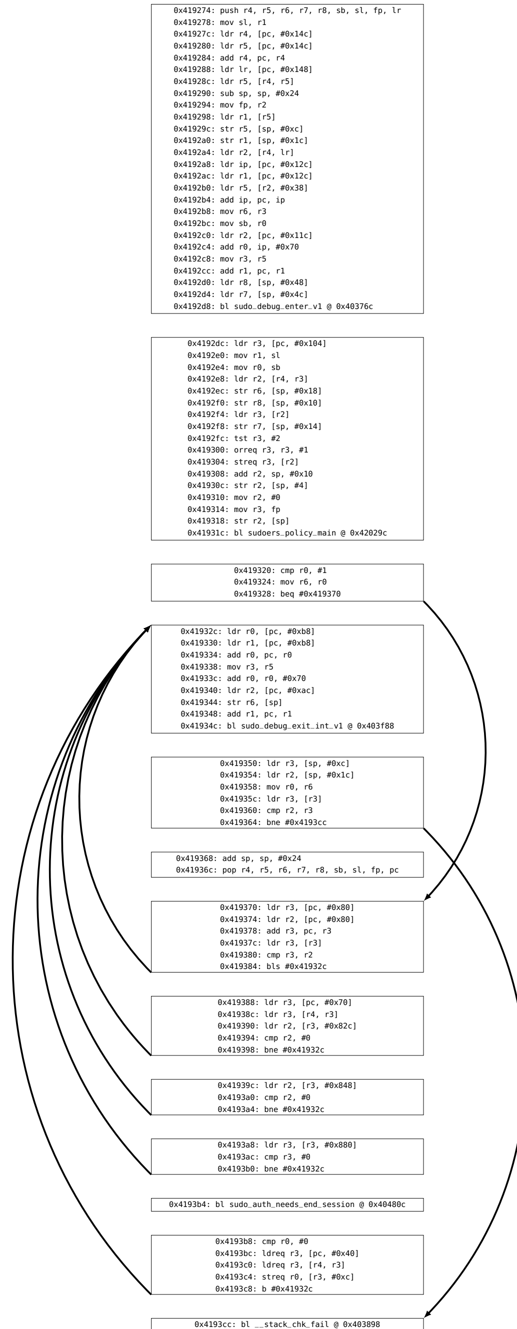


Figure 83: `sudoers_policy_check()` Tikz disassembly.

E

OPENSSL AES AES_ENCRYPT() ROUND FUNCTION (DISSASSEMBLED)

This appendix presents the disassembled code corresponding to the round introduced in listing 7.2. As this sample doesn't not present the full function, in particular the initialization steps, it is not enough to understand the whole implementation of the round. However, some information may help to understand.

The registers `r0` to `r3` are initialized with the first AES state `XOR`ed with the first round key. The `ip` register stores the remaining number of rounds (line 53).

The registers `r0` and `r7` to `r9` contains the shifted and masked values of the current state which are used as input for the pre-computed tables.

The `sl` register corresponds to the stack limit of the current process. It is the base address for the pre-computed tables.

E.1 OPENSSL AES ROUND COMPUTATION (DISASSEMBLED)

```
1  537b4:   e79a4107   ldr  r4, [sl, r7, lsl #2]
2  537b8:   e00e7821   and  r7, 0xff, r1, lsr #16
3  537bc:   e79a5108   ldr  r5, [sl, r8, lsl #2]
4  537c0:   e00e8001   and  r8, 0xff, r1
5  537c4:   e79a6109   ldr  r6, [sl, r9, lsl #2]
6  537c8:   e00e9421   and  r9, 0xff, r1, lsr #8
7  537cc:   e79a0100   ldr  r0, [sl, r0, lsl #2]
8  537d0:   e1a01c21   lsr  r1, r1, #24
9  537d4:   e79a7107   ldr  r7, [sl, r7, lsl #2]
10 537d8:   e79a8108   ldr  r8, [sl, r8, lsl #2]
11 537dc:   e79a9109   ldr  r9, [sl, r9, lsl #2]
12 537e0:   e0200467   eor  r0, r0, r7, ror #8
13 537e4:   e79a1101   ldr  r1, [sl, r1, lsl #2]
14 537e8:   e00e7422   and  r7, 0xff, r2, lsr #8
15 537ec:   e0255468   eor  r5, r5, r8, ror #8
16 537f0:   e00e8822   and  r8, 0xff, r2, lsr #16
17 537f4:   e0266469   eor  r6, r6, r9, ror #8
18 537f8:   e00e9002   and  r9, 0xff, r2
19 537fc:   e79a7107   ldr  r7, [sl, r7, lsl #2]
20 53800:   e0211c64   eor  r1, r1, r4, ror #24
21 53804:   e79a8108   ldr  r8, [sl, r8, lsl #2]
22 53808:   e1a02c22   lsr  r2, r2, #24
23 5380c:   e79a9109   ldr  r9, [sl, r9, lsl #2]
24 53810:   e0200867   eor  r0, r0, r7, ror #16
25 53814:   e79a2102   ldr  r2, [sl, r2, lsl #2]
26 53818:   e00e7003   and  r7, 0xff, r3
27 5381c:   e0211468   eor  r1, r1, r8, ror #8
```

```
28  53820:   e00e8423   and  r8, 0xff, r3, lsr #8
29  53824:   e0266869   eor  r6, r6, r9, ror #16
30  53828:   e00e9823   and  r9, 0xff, r3, lsr #16
31  5382c:   e79a7107   ldr  r7, [sl, r7, lsl #2]
32  53830:   e0222865   eor  r2, r2, r5, ror #16
33  53834:   e79a8108   ldr  r8, [sl, r8, lsl #2]
34  53838:   e1a03c23   lsr  r3, r3, #24
35  5383c:   e79a9109   ldr  r9, [sl, r9, lsl #2]
36  53840:   e0200c67   eor  r0, r0, r7, ror #24
37  53844:   e49b7010   ldr  r7, [fp], #16
38  53848:   e0211868   eor  r1, r1, r8, ror #16
39  5384c:   e79a3103   ldr  r3, [sl, r3, lsl #2]
40  53850:   e0222469   eor  r2, r2, r9, ror #8
41  53854:   e51b400c   ldr  r4, [fp, #-12]
42  53858:   e0233466   eor  r3, r3, r6, ror #8
43  5385c:   e51b5008   ldr  r5, [fp, #-8]
44  53860:   e0200007   eor  r0, r0, r7
45  53864:   e51b6004   ldr  r6, [fp, #-4]
46  53868:   e00e7000   and  r7, 0xff, r0
47  5386c:   e0211004   eor  r1, r1, r4
48  53870:   e00e8420   and  r8, 0xff, r0, lsr #8
49  53874:   e0222005   eor  r2, r2, r5
50  53878:   e00e9820   and  r9, 0xff, r0, lsr #16
51  5387c:   e0233006   eor  r3, r3, r6
52  53880:   e1a00c20   lsr  r0, r0, #24
53  53884:   e25cc001   subs ip, ip, #1
54  53888:   1affffc9   bne  537b4   <_armv4_AES_encrypt+0x34>
```

BIBLIOGRAPHY

[1] Keith Mayes and Konstantinos Markantonakis. "Smart Cards, Tokens, Security and Applications." In: Springer International Publishing, 2017. ISBN: 9783319505008.

[2] *JavaCard Specification*. 2020. URL: https://www.oracle.com/java/technologies/javacard-specs-downloads.html.

[3] Moritz Lipp et al. "Meltdown." In: *meltdownattack.com* (2018). URL: https://meltdownattack.com/meltdown.pdf.

[4] Paul Kocher et al. "Spectre Attacks: Exploiting Speculative Execution." In: *meltdownattack.com* (2018). URL: https://spectreattack.com/spectre.pdf.

[5] R. M. Tomasulo. "An Efficient Algorithm for Exploiting Multiple Arithmetic Units." In: *IBM Journal of Research and Development* 11.1 (1967), pp. 25–33.

[6] Eric Brier, Christophe Clavier, and Francis Olivier. "Correlation Power Analysis with a Leakage Model." In: *Cryptographic Hardware and Embedded Systems - CHES 2004: 6th International Workshop Cambridge, MA, USA, August 11-13, 2004. Proceedings*. Ed. by Marc Joye and Jean-Jacques Quisquater. Vol. 3156. Lecture Notes in Computer Science. Springer, 2004, pp. 16–29. DOI: 10.1007/978-3-540-28632-5_2. URL: https://doi.org/10.1007/978-3-540-28632-5_2.

[7] Paul C. Kocher. "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems." In: *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*. Ed. by Neal Koblitz. Vol. 1109. Lecture Notes in Computer Science. Springer, 1996, pp. 104–113. DOI: 10.1007/3-540-68697-5_9. URL: https://doi.org/10.1007/3-540-68697-5_9.

[8] Daniel Moghimi et al. "TPM-FAIL: TPM meets Timing and Lattice Attacks." In: *29th USENIX Security Symposium (USENIX Security 20)*. Boston, MA: USENIX Association, Aug. 2020. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/moghimi.

[9] Gildas Avoine and Loïc Ferreira. "Attacking GlobalPlatform SCP02-compliant Smart Cards Using a Padding Oracle Attack." In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2018.2 (2018), pp. 149–170. DOI: 10.13154/tches.v2018.i2.149-170. URL: https://doi.org/10.13154/tches.v2018.i2.149-170.

[10] Andrew Kwong et al. "RAMBleed: Reading Bits in Memory Without Accessing Them." In: *41st IEEE Symposium on Security and Privacy (S&P)*. 2020.

[11] Michael Schwarz et al. "ZombieLoad: Cross-Privilege-Boundary Data Sampling." In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*. Ed. by Lorenzo Cavallaro et al. ACM, 2019, pp. 753–768. DOI: 10.1145/3319535.3354252. URL: https://doi.org/10.1145/3319535.3354252.

[12] Jo Van Bulck et al. "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution." In: *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*. Ed. by William Enck and Adrienne Porter Felt. USENIX Association, 2018, pp. 991–1008. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/bulck.

[13] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. "Differential Power Analysis." In: *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*. Ed. by Michael J. Wiener. Vol. 1666. Lecture Notes in Computer Science. Springer, 1999, pp. 388–397. DOI: 10.1007/3-540-48405-1_25. URL: https://doi.org/10.1007/3-540-48405-1_25.

[14] Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. "Template Attacks." In: *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers*. Ed. by Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar. Vol. 2523. Lecture Notes in Computer Science. Springer, 2002, pp. 13–28. DOI: 10.1007/3-540-36400-5_3. URL: https://doi.org/10.1007/3-540-36400-5_3.

[15] David Oswald and Christof Paar. "Breaking Mifare DESFire MF3ICD40: Power Analysis and Templates in the Real World." In: *Cryptographic Hardware and Embedded Systems - CHES 2011 - 13th International Workshop, Nara, Japan, September 28 - October 1, 2011. Proceedings*. Ed. by Bart Preneel and Tsuyoshi Takagi. Vol. 6917. Lecture Notes in Computer Science. Springer, 2011, pp. 207–222. DOI: 10.1007/978-3-642-23951-9_14. URL: https://doi.org/10.1007/978-3-642-23951-9_14.

[16] Timo Kasper, David Oswald, and Christof Paar. "EM Side-Channel Attacks on Commercial Contactless Smartcards Using Low-Cost Equipment." In: *Information Security Applications, 10th International Workshop, WISA 2009, Busan, Korea, August 25-27, 2009, Revised Selected Papers*. Ed. by Heung Youl Youm and Moti Yung. Vol. 5932. Lecture Notes in Computer Science. Springer, 2009, pp. 79–93. DOI: 10.1007/978-3-642-10838-9_7. URL: https://doi.org/10.1007/978-3-642-10838-9_7.

[17] Daniel Genkin et al. "ECDSA Key Extraction from Mobile Devices via Nonintrusive Physical Side Channels." In: *IACR Cryp-*

tol. ePrint Arch. 2016 (2016), p. 230. URL: http://eprint.iacr.org/2016/230.

[18] Eyal Ronen et al. "IoT Goes Nuclear: Creating a ZigBee Chain Reaction." In: *IACR Cryptol. ePrint Arch.* 2016 (2016), p. 1047. URL: http://eprint.iacr.org/2016/1047.

[19] Benedikt Gierlichs, Lejla Batina, and Pim Tuyls. "Mutual Information Analysis - A Universal Differential Side-Channel Attack." In: *IACR Cryptol. ePrint Arch.* 2007 (2007), p. 198. URL: http://eprint.iacr.org/2007/198.

[20] Guillaume Dabosville, Julien Doget, and Emmanuel Prouff. "A New Second-Order Side Channel Attack Based on Linear Regression." In: *IEEE Trans. Computers* 62.8 (2013), pp. 1629–1640. DOI: 10.1109/TC.2012.112. URL: https://doi.org/10.1109/TC.2012.112.

[21] Houssem Maghrebi, Thibault Portigliatti, and Emmanuel Prouff. "Breaking Cryptographic Implementations Using Deep Learning Techniques." In: *Security, Privacy, and Applied Cryptography Engineering - 6th International Conference, SPACE 2016, Hyderabad, India, December 14-18, 2016, Proceedings*. Ed. by Claude Carlet, M. Anwar Hasan, and Vishal Saraswat. Vol. 10076. Lecture Notes in Computer Science. Springer, 2016, pp. 3–26. DOI: 10.1007/978-3-319-49445-6_1. URL: https://doi.org/10.1007/978-3-319-49445-6_1.

[22] Mathieu Carbone et al. "Deep Learning to Evaluate Secure RSA Implementations." In: *IACR Cryptol. ePrint Arch.* 2019 (2019), p. 54. URL: https://eprint.iacr.org/2019/054.

[23] *A Hacker Guide To Deep Learning Based Side Channel Attacks*. https://elie.net/talk/a-hackerguide-to-deep-learning-based-side-channel-attacks/. Accessed: 2020-06-11.

[24] Diego F. Aranha et al. *LadderLeak: Breaking ECDSA With Less Than One Bit Of Nonce Leakage*. Cryptology ePrint Archive, Report 2020/615. https://eprint.iacr.org/2020/615. 2020.

[25] Peter Pessl et al. "DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks." In: *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*. Ed. by Thorsten Holz and Stefan Savage. USENIX Association, 2016, pp. 565–581. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/pessl.

[26] Clémentine Maurice et al. "Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud." In: *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society, 2017. URL: https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/hello-other-side-ssh-over-robust-cache-covert-channels-cloud/.

[27] Berk Gülmezoglu et al. “A Faster and More Realistic Flush+Reload Attack on AES.” In: *Constructive Side-Channel Analysis and Secure Design - 6th International Workshop, COSADE 2015, Berlin, Germany, April 13-14, 2015. Revised Selected Papers*. Ed. by Stefan Mangard and Axel Y. Poschmann. Vol. 9064. Lecture Notes in Computer Science. Springer, 2015, pp. 111–126. DOI: 10.1007/978-3-319-21476-4_8. URL: https://doi.org/10.1007/978-3-319-21476-4_8.

[28] Project Zero team. *Reading privileged memory with a side-channel*. Accessed: 2020-06-14. 2018. URL: https://googleprojectzero.blogspot.co.uk/2018/01/reading-privileged-memory-with-side.html.

[29] Robert M Tomasulo. “An efficient algorithm for exploiting multiple arithmetic units.” In: *IBM Journal of research and Development* 11.1 (1967), pp. 25–33.

[30] Saad Islam et al. “SPOILER: Speculative Load Hazards Boost Rowhammer and Cache Attacks.” In: *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*. Ed. by Nadia Heninger and Patrick Traynor. USENIX Association, 2019, pp. 621–637. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/islam.

[31] Giorgi Maisuradze and Christian Rossow. “ret2spec: Speculative Execution Using Return Stack Buffers.” In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*. Ed. by David Lie et al. ACM, 2018, pp. 2109–2122. DOI: 10.1145/3243734.3243761. URL: https://doi.org/10.1145/3243734.3243761.

[32] Daniel Gruss et al. “Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR.” In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. Ed. by Edgar R. Weippl et al. ACM, 2016, pp. 368–379. DOI: 10.1145/2976749.2978356. URL: https://doi.org/10.1145/2976749.2978356.

[33] Michael Schwarz et al. “NetSpectre: Read Arbitrary Memory over Network.” In: *Computer Security - ESORICS 2019 - 24th European Symposium on Research in Computer Security, Luxembourg, September 23-27, 2019, Proceedings, Part I*. Ed. by Kazue Sako, Steve Schneider, and Peter Y. A. Ryan. Vol. 11735. Lecture Notes in Computer Science. Springer, 2019, pp. 279–299. DOI: 10.1007/978-3-030-29959-0_14. URL: https://doi.org/10.1007/978-3-030-29959-0_14.

[34] Matthieu Rivain, Emmanuelle Dottax, and Emmanuel Prouff. “Block Ciphers Implementations Provably Secure Against Second Order Side Channel Analysis.” In: *Fast Software Encryption, 15th International Workshop, FSE 2008, Lausanne, Switzerland, February 10-13, 2008, Revised Selected Papers*. Ed. by Kaisa Ny-

berg. Vol. 5086. Lecture Notes in Computer Science. Springer, 2008, pp. 127–143. DOI: 10.1007/978-3-540-71039-4_8. URL: https://doi.org/10.1007/978-3-540-71039-4_8.

[35] Rajat Subhra Chakraborty and Swarup Bhunia. "Hardware protection and authentication through netlist level obfuscation." In: *2008 International Conference on Computer-Aided Design, ICCAD 2008, San Jose, CA, USA, November 10-13, 2008*. Ed. by Sani R. Nassif and Jaijeet S. Roychowdhury. IEEE Computer Society, 2008, pp. 674–677. DOI: 10.1109/ICCAD.2008.4681649. URL: https://doi.org/10.1109/ICCAD.2008.4681649.

[36] Rajat Subhra Chakraborty and Swarup Bhunia. "HARPOON: An Obfuscation-Based SoC Design Methodology for Hardware Protection." In: *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 28.10 (2009), pp. 1493–1502. DOI: 10.1109/TCAD.2009.2028166. URL: https://doi.org/10.1109/TCAD.2009.2028166.

[37] Rajat Subhra Chakraborty and Swarup Bhunia. "RTL Hardware IP Protection Using Key-Based Control and Data Flow Obfuscation." In: *VLSI Design 2010: 23rd International Conference on VLSI Design, 9th International Conference on Embedded Systems, Bangalore, India, 3-7 January 2010*. IEEE Computer Society, 2010, pp. 405–410. DOI: 10.1109/VLSI.Design.2010.54. URL: https://doi.org/10.1109/VLSI.Design.2010.54.

[38] Dongfang Li et al. "Hardware IP Protection through Gate-Level Obfuscation." In: *14th International Conference on Computer-Aided Design and Computer Graphics, CAD/Graphics 2015, Xi'an, China, August 26-28, 2015*. IEEE, 2015, pp. 186–193. DOI: 10.1109/CADGRAPHICS.2015.39. URL: https://doi.org/10.1109/CADGRAPHICS.2015.39.

[39] Jiliang Zhang. "A Practical Logic Obfuscation Technique for Hardware Security." In: *IEEE Trans. Very Large Scale Integr. Syst.* 24.3 (2016), pp. 1193–1197. DOI: 10.1109/TVLSI.2015.2437996. URL: https://doi.org/10.1109/TVLSI.2015.2437996.

[40] Shweta Malik et al. "Development of a Layout-Level Hardware Obfuscation Tool." In: *2015 IEEE Computer Society Annual Symposium on VLSI, ISVLSI 2015, Montpellier, France, July 8-10, 2015*. IEEE Computer Society, 2015, pp. 204–209. DOI: 10.1109/ISVLSI.2015.118. URL: https://doi.org/10.1109/ISVLSI.2015.118.

[41] Md Shahed Enamul Quadir and John A. Chandy. "Key Generation for Hardware Obfuscation Using Strong PUFs." In: *Cryptogr.* 3.3 (2019), p. 17. DOI: 10.3390/cryptography3030017. URL: https://doi.org/10.3390/cryptography3030017.

[42] John Melngailis. "Focused ion beam technology and applications." In: *Journal of Vacuum Science & Technology B: Microelectronics Processing and Phenomena* 5.2 (1987), pp. 469–495. DOI: 10.1116/1.583937. eprint: https://avs.scitation.org/doi/

pdf/10.1116/1.583937. URL: https://avs.scitation.org/doi/abs/10.1116/1.583937.

[43] D. Binder, E. C. Smith, and A. B. Holman. "Satellite Anomalies from Galactic Cosmic Rays." In: *IEEE Transactions on Nuclear Science* 22.6 (1975), pp. 2675–2680.

[44] J. C. Pickel and J. T. Blandford. "Cosmic Ray Induced in MOS Memory Cells." In: *IEEE Transactions on Nuclear Science* 25.6 (1978), pp. 1166–1171.

[45] G. A. Sai-Halasz. "Cosmic ray induced soft error rate in VLSI circuits." In: *IEEE Electron Device Letters* 4.6 (1983), pp. 172–174.

[46] T. J. O'Gorman. "The effect of cosmic rays on the soft error rate of a DRAM at ground level." In: *IEEE Transactions on Electron Devices* 41.4 (1994), pp. 553–557.

[47] W. A. Kolasinski et al. "The Effect of Elevated Temperature on Latchup and Bit Errors in CMOS Devices." In: *IEEE Transactions on Nuclear Science* 33.6 (1986), pp. 1605–1609.

[48] T. C. May and M. H. Woods. "Alpha-particle-induced soft errors in dynamic memories." In: *IEEE Transactions on Electron Devices* 26.1 (1979), pp. 2–9.

[49] R. Koga, W. A. Kolasinski, and S. Imamoto. "Heavy Ion Induced Upsets in Semiconductor Devices." In: *IEEE Transactions on Nuclear Science* 32.1 (1985), pp. 159–162.

[50] R. Koga and W. A. Kolasinski. "Heavy ion induced snapback in CMOS devices." In: *IEEE Transactions on Nuclear Science* 36.6 (1989), pp. 2367–2374.

[51] Jim Colvin. "Functional Failure Analysis by Induced Stimulus." In: 2002.

[52] D. H. Habing. "The Use of Lasers to Simulate Radiation-Induced Transients in Semiconductor Devices and Circuits." In: *IEEE Transactions on Nuclear Science* 12.5 (1965), pp. 91–100.

[53] Arjen K. Lenstra. "Memo on RSA signature generation in the presence of faults." In: (1996). manuscript. URL: http://infoscience.epfl.ch/record/164524.

[54] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. "On the Importance of Checking Cryptographic Protocols for Faults (Extended Abstract)." In: *Advances in Cryptology - EUROCRYPT '97, International Conference on the Theory and Application of Cryptographic Techniques, Konstanz, Germany, May 11-15, 1997, Proceeding*. Ed. by Walter Fumy. Vol. 1233. Lecture Notes in Computer Science. Springer, 1997, pp. 37–51. DOI: 10.1007/3-540-69053-0_4. URL: https://doi.org/10.1007/3-540-69053-0_4.

[55] Eli Biham and Adi Shamir. “Differential Fault Analysis of Secret Key Cryptosystems.” In: *Advances in Cryptology - CRYPTO ’97, 17th Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 1997, Proceedings*. Ed. by Burton S. Kaliski Jr. Vol. 1294. Lecture Notes in Computer Science. Springer, 1997, pp. 513–525. DOI: 10.1007/BFb0052259. URL: https://doi.org/10.1007/BFb0052259.

[56] Feng Bao et al. “Breaking Public Key Cryptosystems on Tamper Resistant Devices in the Presence of Transient Faults.” In: *Security Protocols, 5th International Workshop, Paris, France, April 7-9, 1997, Proceedings*. Ed. by Bruce Christianson et al. Vol. 1361. Lecture Notes in Computer Science. Springer, 1997, pp. 115–124. DOI: 10.1007/BFb0028164. URL: https://doi.org/10.1007/BFb0028164.

[57] Ingrid Biehl, Bernd Meyer, and Volker Müller. “Differential Fault Attacks on Elliptic Curve Cryptosystems.” In: *Advances in Cryptology - CRYPTO 2000, 20th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 2000, Proceedings*. Ed. by Mihir Bellare. Vol. 1880. Lecture Notes in Computer Science. Springer, 2000, pp. 131–146. DOI: 10.1007/3-540-44598-6_8. URL: https://doi.org/10.1007/3-540-44598-6_8.

[58] Sergei P. Skorobogatov and Ross J. Anderson. “Optical Fault Induction Attacks.” In: *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers*. Ed. by Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar. Vol. 2523. Lecture Notes in Computer Science. Springer, 2002, pp. 2–12. DOI: 10.1007/3-540-36400-5_2. URL: https://doi.org/10.1007/3-540-36400-5_2.

[59] Sergei Skorobogatov. “Semi-invasive attacks-A new approach to hardware security analysis.” In: (Jan. 2005).

[60] Loïc Zussa et al. “Investigation of timing constraints violation as a fault injection means.” In: *27th Conference on Design of Circuits and Integrated Systems (DCIS)*. Avignon, France, Nov. 2012, pas encore paru. URL: https://hal-emse.ccsd.cnrs.fr/emse-00742652.

[61] Sébastien Ordas, Ludovic Guillaume-Sage, and Philippe Maurine. “Electromagnetic fault injection: the curse of flip-flops.” In: *J. Cryptographic Engineering* 7.3 (2017), pp. 183–197. DOI: 10.1007/s13389-016-0128-3. URL: https://doi.org/10.1007/s13389-016-0128-3.

[62] Cyril Roscian et al. “Fault Model Analysis of Laser-Induced Faults in SRAM Memory Cells.” In: *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography, Los Alamitos, CA, USA, August 20, 2013*. Ed. by Wieland Fischer and Jörn-Marc Schmidt. IEEE Computer Society, 2013, pp. 89–98. DOI: 10.

1109/FDTC.2013.17. URL: https://doi.org/10.1109/FDTC.2013.17.

[63] Bettina Rebaud et al. "Setup and Hold Timing Violations Induced by Process Variations, in a Digital Multiplier." In: *IEEE Computer Society Annual Symposium on VLSI, ISVLSI 2008, 7-9 April 2008, Montpellier, France*. IEEE Computer Society, 2008, pp. 316–321. DOI: 10.1109/ISVLSI.2008.70. URL: https://doi.org/10.1109/ISVLSI.2008.70.

[64] Michel Agoyan et al. "When Clocks Fail: On Critical Paths and Clock Faults." In: *Smart Card Research and Advanced Application, 9th IFIP WG 8.8/11.2 International Conference, CARDIS 2010, Passau, Germany, April 14-16, 2010. Proceedings*. Ed. by Dieter Gollmann, Jean-Louis Lanet, and Julien Iguchi-Cartigny. Vol. 6035. Lecture Notes in Computer Science. Springer, 2010, pp. 182–193. DOI: 10.1007/978-3-642-12510-2_13. URL: https://doi.org/10.1007/978-3-642-12510-2_13.

[65] Adrian Tang, Simha Sethumadhavan, and Salvatore J. Stolfo. "CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management." In: *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*. Ed. by Engin Kirda and Thomas Ristenpart. USENIX Association, 2017, pp. 1057–1074. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/tang.

[66] B. Lasbouygues et al. "Timing analysis in presence of supply voltage and temperature variations." In: *Proceedings of the 2006 International Symposium on Physical Design, ISPD 2006, San Jose, California, USA, April 9-12, 2006*. Ed. by Louis Scheffer. ACM, 2006, pp. 10–16. DOI: 10.1145/1123008.1123012. URL: https://doi.org/10.1145/1123008.1123012.

[67] Thomas Korak and Michael Hoefler. "On the Effects of Clock and Power Supply Tampering on Two Microcontroller Platforms." In: *2014 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2014, Busan, South Korea, September 23, 2014*. Ed. by Assia Tria and Dooho Choi. IEEE Computer Society, 2014, pp. 8–17. DOI: 10.1109/FDTC.2014.11. URL: https://doi.org/10.1109/FDTC.2014.11.

[68] Claudio Bozzato, Riccardo Focardi, and Francesco Palmarini. "Shaping the Glitch: Optimizing Voltage Fault Injection Attacks." In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2019.2 (2019), pp. 199–224. DOI: 10.13154/tches.v2019.i2.199-224. URL: https://doi.org/10.13154/tches.v2019.i2.199-224.

[69] Michael Hutter and Jörn-Marc Schmidt. "The Temperature Side Channel and Heating Fault Attacks." In: *Smart Card Research and Advanced Applications - 12th International Conference, CARDIS 2013, Berlin, Germany, November 27-29, 2013. Revised Selected Papers*. Ed. by Aurélien Francillon and Pankaj Rohatgi. Vol. 8419. Lecture Notes in Computer Science. Springer, 2013,

pp. 219–235. DOI: 10.1007/978-3-319-08302-5_15. URL: https://doi.org/10.1007/978-3-319-08302-5_15.

[70] Sébastien Ordas, Ludovic Guillaume-Sage, and Philippe Maurine. "EM Injection: Fault Model and Locality." In: *2015 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2015, Saint Malo, France, September 13, 2015*. Ed. by Naofumi Homma and Victor Lomné. IEEE Computer Society, 2015, pp. 3–13. DOI: 10.1109/FDTC.2015.9. URL: https://doi.org/10.1109/FDTC.2015.9.

[71] Mathieu Dumont, Mathieu Lisart, and Philippe Maurine. "Electromagnetic Fault Injection : How Faults Occur." In: *2019 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2019, Atlanta, GA, USA, August 24, 2019*. IEEE, 2019, pp. 9–16. DOI: 10.1109/FDTC.2019.00010. URL: https://doi.org/10.1109/FDTC.2019.00010.

[72] François Poucheret et al. "Local and Direct EM Injection of Power Into CMOS Integrated Circuits." In: *2011 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2011, Tokyo, Japan, September 29, 2011*. Ed. by Luca Breveglieri et al. IEEE Computer Society, 2011, pp. 100–104. DOI: 10.1109/FDTC.2011.18. URL: https://doi.org/10.1109/FDTC.2011.18.

[73] Lionel Rivière et al. "High precision fault injections on the instruction cache of ARMv7-M architectures." In: *IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2015, Washington, DC, USA, 5-7 May, 2015*. IEEE Computer Society, 2015, pp. 62–67. DOI: 10.1109/HST.2015.7140238. URL: https://doi.org/10.1109/HST.2015.7140238.

[74] Ang Cui and Rick Housley. "BADFET: Defeating Modern Secure Boot Using Second-Order Pulsed Electromagnetic Fault Injection." In: *11th USENIX Workshop on Offensive Technologies, WOOT 2017, Vancouver, BC, Canada, August 14-15, 2017*. Ed. by William Enck and Collin Mulliner. USENIX Association, 2017. URL: https://www.usenix.org/conference/woot17/workshop-program/presentation/cui.

[75] E. R. Fossum and D. B. Hondongwa. "A Review of the Pinned Photodiode for CCD and CMOS Image Sensors." In: *IEEE Journal of the Electron Devices Society* 2.3 (2014), pp. 33–43.

[76] Jörn-Marc Schmidt, Michael Hutter, and Thomas Plos. "Optical Fault Attacks on AES: A Threat in Violet." In: *Sixth International Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2009, Lausanne, Switzerland, 6 September 2009*. Ed. by Luca Breveglieri et al. IEEE Computer Society, 2009, pp. 13–22. DOI: 10.1109/FDTC.2009.37. URL: https://doi.org/10.1109/FDTC.2009.37.

[77] David Samyde et al. "On a New Way to Read Data from Memory." In: *Proceedings of the First International IEEE Security in Storage Workshop, SISW 2002, Greenbelt, Maryland, USA, December 11, 2002*. IEEE Computer Society, 2002, pp. 65–69. DOI: 10.

1109/SISW.2002.1183512. URL: https://doi.org/10.1109/SISW.2002.1183512.

[78] Sergei Skorobogatov. "Optical Fault Masking Attacks." In: *2010 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2010, Santa Barbara, California, USA, 21 August 2010*. Ed. by Luca Breveglieri et al. IEEE Computer Society, 2010, pp. 23–29. DOI: 10.1109/FDTC.2010.18. URL: https://doi.org/10.1109/FDTC.2010.18.

[79] Jakub Breier, Dirmanto Jap, and Chien-Ning Chen. "Laser Profiling for the Back-Side Fault Attacks: With a Practical Laser Skip Instruction Attack on AES." In: *Proceedings of the 1st ACM Workshop on Cyber-Physical System Security, CPSS 2015, Singapore, Republic of Singapore, April 14 - March 14, 2015*. Ed. by Jianying Zhou and Douglas Jones. ACM, 2015, pp. 99–103. DOI: 10.1145/2732198.2732206. URL: https://doi.org/10.1145/2732198.2732206.

[80] Shahin Tajik et al. "Laser Fault Attack on Physically Unclonable Functions." In: *2015 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2015, Saint Malo, France, September 13, 2015*. Ed. by Naofumi Homma and Victor Lomné. IEEE Computer Society, 2015, pp. 85–96. DOI: 10.1109/FDTC.2015.19. URL: https://doi.org/10.1109/FDTC.2015.19.

[81] Aurélien Vasselle et al. "Laser-Induced Fault Injection on Smartphone Bypassing the Secure Boot." In: *2017 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2017, Taipei, Taiwan, September 25, 2017*. IEEE Computer Society, 2017, pp. 41–48. DOI: 10.1109/FDTC.2017.18. URL: https://doi.org/10.1109/FDTC.2017.18.

[82] K. von Arnim et al. "Efficiency of body biasing in 90-nm CMOS for low-power digital circuits." In: *IEEE Journal of Solid-State Circuits* 40.7 (2005), pp. 1549–1556.

[83] Noemie Beringuier-Boher et al. "Body Biasing Injection Attacks in Practice." In: *Proceedings of the Third Workshop on Cryptography and Security in Computing Systems, CS2@HiPEAC, Prague, Czech Republic, January 20, 2016*. Ed. by Martin Palkovic et al. ACM, 2016, pp. 49–54. DOI: 10.1145/2858930.2858940. URL: https://doi.org/10.1145/2858930.2858940.

[84] Philippe Maurine et al. "Yet Another Fault Injection Technique : by Forward Body Biasing Injection." In: (Sept. 2012).

[85] Stéphanie Anceau et al. "Nanofocused X-Ray Beam to Reprogram Secure Circuits." In: *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*. Ed. by Wieland Fischer and Naofumi Homma. Vol. 10529. Lecture Notes in Computer Science. Springer, 2017, pp. 175–188. DOI: 10.1007/978-3-319-66787-4_9. URL: https://doi.org/10.1007/978-3-319-66787-4_9.

[86] Yoongu Kim et al. “Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors.” In: *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*. IEEE Computer Society, 2014, pp. 361–372. DOI: 10.1109/ISCA.2014.6853210. URL: https://doi.org/10.1109/ISCA.2014.6853210.

[87] *Exploiting the DRAM rowhammer bug to gain kernel privileges*. https://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html. Accessed: 2020-06-30.

[88] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. “Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript.” In: *CoRR* abs/1507.06955 (2015). arXiv: 1507.06955. URL: http://arxiv.org/abs/1507.06955.

[89] Victor van der Veen et al. “Drammer: Deterministic Rowhammer Attacks on Mobile Platforms.” In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. Ed. by Edgar R. Weippl et al. ACM, 2016, pp. 1675–1689. DOI: 10.1145/2976749.2978406. URL: https://doi.org/10.1145/2976749.2978406.

[90] Pietro Frigo et al. “Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU.” In: *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. IEEE Computer Society, 2018, pp. 195–210. DOI: 10.1109/SP.2018.00022. URL: https://doi.org/10.1109/SP.2018.00022.

[91] Andrei Tatar et al. “Throwhammer: Rowhammer Attacks over the Network and Defenses.” In: *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*. Ed. by Haryadi S. Gunawi and Benjamin Reed. USENIX Association, 2018, pp. 213–226. URL: https://www.usenix.org/conference/atc18/presentation/tatar.

[92] Kit Murdock et al. “Plundervolt: Software-based Fault Injection Attacks against Intel SGX.” In: *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 2020, pp. 1466–1482. DOI: 10.1109/SP40000.2020.00057. URL: https://doi.org/10.1109/SP40000.2020.00057.

[93] Sikhar Patranabis et al. “A Biased Fault Attack on the Time Redundancy Countermeasure for AES.” In: *Constructive Side-Channel Analysis and Secure Design - 6th International Workshop, COSADE 2015, Berlin, Germany, April 13-14, 2015. Revised Selected Papers*. Ed. by Stefan Mangard and Axel Y. Poschmann. Vol. 9064. Lecture Notes in Computer Science. Springer, 2015, pp. 189–203. DOI: 10.1007/978-3-319-21476-4_13. URL: https://doi.org/10.1007/978-3-319-21476-4_13.

[94] Bilgiday Yuce, Patrick Schaumont, and Marc Witteman. "Fault Attacks on Secure Embedded Software: Threats, Design and Evaluation." In: *CoRR* abs/2003.10513 (2020). arXiv: 2003.10513. URL: https://arxiv.org/abs/2003.10513.

[95] 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: *2011 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2011, Tokyo, Japan, September 29, 2011*. Ed. by Luca Breveglieri et al. IEEE Computer Society, 2011, pp. 105–114. DOI: 10.1109/FDTC.2011.9. URL: https://doi.org/10.1109/FDTC.2011.9.

[96] Sébanjila Kevin Bukasa et al. "Let's shock our IoT's heart: ARMv7-M under (fault) attacks." In: *Proceedings of the 13th International Conference on Availability, Reliability and Security, ARES 2018, Hamburg, Germany, August 27-30, 2018*. Ed. by Sebastian Doerr et al. ACM, 2018, 33:1–33:6. DOI: 10.1145/3230833.3230842. URL: https://doi.org/10.1145/3230833.3230842.

[97] Fabien Majéric, Eric Bourbao, and Lilian Bossuet. "Electromagnetic security tests for SoC." In: *2016 IEEE International Conference on Electronics, Circuits and Systems, ICECS 2016, Monte Carlo, Monaco, December 11-14, 2016*. IEEE, 2016, pp. 265–268. DOI: 10.1109/ICECS.2016.7841183. URL: https://doi.org/10.1109/ICECS.2016.7841183.

[98] Nicolas Moro et al. "Experimental evaluation of two software countermeasures against fault attacks." In: *2014 IEEE International Symposium on Hardware-Oriented Security and Trust, HOST 2014, Arlington, VA, USA, May 6-7, 2014*. IEEE Computer Society, 2014, pp. 112–117. DOI: 10.1109/HST.2014.6855580. URL: https://doi.org/10.1109/HST.2014.6855580.

[99] Alexandre Menu et al. "Experimental Analysis of the Electromagnetic Instruction Skip Fault Model." In: *15th Design & Technology of Integrated Systems in Nanoscale Era, DTIS 2020, Marrakech, Morocco, April 1-3, 2020*. IEEE, 2020, pp. 1–7. DOI: 10.1109/DTIS48698.2020.9081261. URL: https://doi.org/10.1109/DTIS48698.2020.9081261.

[100] Jakub Breier and Dirmanto Jap. "Testing Feasibility of Back-Side Laser Fault Injection on a Microcontroller." In: *Proceedings of the 10th Workshop on Embedded Systems Security, WESS 2015, Amsterdam, The Netherlands, October 8, 2015*. Ed. by Stavros A. Koubias and Thilo Sauter. ACM, 2015, p. 5. DOI: 10.1145/2818362.2818367. URL: https://doi.org/10.1145/2818362.2818367.

[101] Niek Timmers, Albert Spruyt, and Marc Witteman. "Controlling PC on ARM Using Fault Injection." In: *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2016, Santa Barbara, CA, USA, August 16, 2016*. IEEE Computer Society,

2016, pp. 25–35. DOI: 10.1109/FDTC.2016.18. URL: https://doi.org/10.1109/FDTC.2016.18.

[102] Nicolas Moro et al. "Electromagnetic fault injection: towards a fault model on a 32-bit microcontroller." In: *CoRR* abs/1402.6421 (2014). arXiv: 1402.6421. URL: http://arxiv.org/abs/1402.6421.

[103] *Characterization of EM faults on ATmega328p*. Zenodo, July 2019. DOI: 10.5281/zenodo.2647298. URL: https://doi.org/10.5281/zenodo.2647298.

[104] Alexandre Menu et al. "Precise Spatio-Temporal Electromagnetic Fault Injections on Data Transfers." In: *2019 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2019, Atlanta, GA, USA, August 24, 2019*. IEEE, 2019, pp. 1–8. DOI: 10.1109/FDTC.2019.00009. URL: https://doi.org/10.1109/FDTC.2019.00009.

[105] Dilip S. V. Kumar et al. "An In-Depth and Black-Box Characterization of the Effects of Laser Pulses on ATmega328P." In: *Smart Card Research and Advanced Applications, 17th International Conference, CARDIS 2018, Montpellier, France, November 12-14, 2018, Revised Selected Papers*. Ed. by Begül Bilgin and Jean-Bernard Fischer. Vol. 11389. Lecture Notes in Computer Science. Springer, 2018, pp. 156–170. DOI: 10.1007/978-3-030-15462-2_11. URL: https://doi.org/10.1007/978-3-030-15462-2_11.

[106] Brice Colombier et al. "Laser-induced Single-bit Faults in Flash Memory: Instructions Corruption on a 32-bit Microcontroller." In: *IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2019, McLean, VA, USA, May 5-10, 2019*. IEEE, 2019, pp. 1–10. DOI: 10.1109/HST.2019.8741030. URL: https://doi.org/10.1109/HST.2019.8741030.

[107] Julien Proy et al. "A First ISA-Level Characterization of EM Pulse Effects on Superscalar Microarchitectures: A Secure Software Perspective." In: *Proceedings of the 14th International Conference on Availability, Reliability and Security, ARES 2019, Canterbury, UK, August 26-29, 2019*. ACM, 2019, 7:1–7:10. DOI: 10.1145/3339252.3339253. URL: https://doi.org/10.1145/3339252.3339253.

[108] Bilgiday Yuce, Nahid Farhady Ghalaty, and Patrick Schaumont. "Improving Fault Attacks on Embedded Software Using RISC Pipeline Characterization." In: *2015 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2015, Saint Malo, France, September 13, 2015*. Ed. by Naofumi Homma and Victor Lomné. IEEE Computer Society, 2015, pp. 97–108. DOI: 10.1109/FDTC.2015.16. URL: https://doi.org/10.1109/FDTC.2015.16.

[109] Stjepan Picek et al. "Evolving genetic algorithms for fault injection attacks." In: *37th International Convention on Information and Communication Technology, Electronics and Microelectronics, MIPRO 2014, Opatija, Croatia, May 26-30, 2014*. IEEE, 2014, pp. 1106–1111. DOI: 10.1109/MIPRO.2014.6859734. URL: https://doi.org/10.1109/MIPRO.2014.6859734.

[110] Olivier Faurax et al. "PAFI: Outil d'analyse de circuit pour l'accélération de l'injection de fautes en simulation." In: (July 2020).

[111] Kun jun Chang and Yung yuan Chen. *System-Level Fault Injection in SystemC Design Platform*.

[112] Andrea Höller et al. "QEMU-Based Fault Injection for a System-Level Analysis of Software Countermeasures Against Fault Attacks." In: *2015 Euromicro Conference on Digital System Design, DSD 2015, Madeira, Portugal, August 26-28, 2015*. IEEE Computer Society, 2015, pp. 530–533. DOI: 10.1109/DSD.2015.79. URL: https://doi.org/10.1109/DSD.2015.79.

[113] Jean-Baptiste Machemie et al. "SmartCM a smart card fault injection simulator." In: *2011 IEEE International Workshop on Information Forensics and Security, WIFS 2011, Iguacu Falls, Brazil, November 29 - December 2, 2011*. IEEE Computer Society, 2011, pp. 1–6. DOI: 10.1109/WIFS.2011.6123124. URL: https://doi.org/10.1109/WIFS.2011.6123124.

[114] Lucien Goubet et al. "Efficient Design and Evaluation of Countermeasures against Fault Attacks Using Formal Verification." In: *Smart Card Research and Advanced Applications - 14th International Conference, CARDIS 2015, Bochum, Germany, November 4-6, 2015. Revised Selected Papers*. Ed. by Naofumi Homma and Marcel Medwed. Vol. 9514. Lecture Notes in Computer Science. Springer, 2015, pp. 177–192. DOI: 10.1007/978-3-319-31271-2_11. URL: https://doi.org/10.1007/978-3-319-31271-2_11.

[115] Maria Christofi. "Let's (TL)-FACE some code!" In: *Séminaires en sécurité numérique, 2013*. 2013.

[116] Marie-Laure Potet et al. "Lazart: A Symbolic Approach for Evaluation the Robustness of Secured Codes against Control Flow Injections." In: *Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014, March 31 2014-April 4, 2014, Cleveland, Ohio, USA*. IEEE Computer Society, 2014, pp. 213–222. DOI: 10.1109/ICST.2014.34. URL: https://doi.org/10.1109/ICST.2014.34.

[117] Pascal Berthomé et al. "High Level Model of Control Flow Attacks for Smart Card Functional Security." In: *Seventh International Conference on Availability, Reliability and Security, Prague, ARES 2012, Czech Republic, August 20-24, 2012*. IEEE Computer Society, 2012, pp. 224–229. DOI: 10.1109/ARES.2012.79. URL: https://doi.org/10.1109/ARES.2012.79.

[118] Jörn-Marc Schmidt and Michael Hutter. "Optical and EM Fault-Attacks on CRT-based RSA: Concrete Results." English. In: *Austrochip 2007, 15th Austrian Workhop on Microelectronics, 11 October 2007, Graz, Austria, Proceedings*. Verlag der Technischen Universität Graz, 2007, pp. 61–67. ISBN: 978-3-902465-87-0.

[119] Jörn-Marc Schmidt and Christoph Herbst. "A Practical Fault Attack on Square and Multiply." In: *Fifth International Workshop on Fault Diagnosis and Tolerance in Cryptography, 2008, FDTC 2008, Washington, DC, USA, 10 August 2008*. Ed. by Luca Breveglieri et al. IEEE Computer Society, 2008, pp. 53–58. DOI: 10.1109/FDTC.2008.10. URL: https://doi.org/10.1109/FDTC.2008.10.

[120] Fan Zhang et al. "Persistent Fault Analysis on Block Ciphers." In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2018.3 (2018), pp. 150–172. DOI: 10.13154/tches.v2018.i3.150-172. URL: https://doi.org/10.13154/tches.v2018.i3.150-172.

[121] Christophe Giraud. "DFA on AES." In: *IACR Cryptol. ePrint Arch.* 2003 (2003), p. 8. URL: http://eprint.iacr.org/2003/008.

[122] Christophe Giraud and Adrian Thillard. "Piret and Quisquater's DFA on AES Revisited." In: *IACR Cryptol. ePrint Arch.* 2010 (2010), p. 440. URL: http://eprint.iacr.org/2010/440.

[123] Gilles Piret and Jean-Jacques Quisquater. "A Differential Fault Attack Technique against SPN Structures, with Application to the AES and KHAZAD." In: *Cryptographic Hardware and Embedded Systems - CHES 2003, 5th International Workshop, Cologne, Germany, September 8-10, 2003, Proceedings*. Ed. by Colin D. Walter, Çetin Kaya Koç, and Christof Paar. Vol. 2779. Lecture Notes in Computer Science. Springer, 2003, pp. 77–88. DOI: 10.1007/978-3-540-45238-6_7. URL: https://doi.org/10.1007/978-3-540-45238-6_7.

[124] Bodo Selmke, Johann Heyszl, and Georg Sigl. "Attack on a DFA Protected AES by Simultaneous Laser Fault Injections." In: *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2016, Santa Barbara, CA, USA, August 16, 2016*. IEEE Computer Society, 2016, pp. 36–46. DOI: 10.1109/FDTC.2016.16. URL: https://doi.org/10.1109/FDTC.2016.16.

[125] Brett Giller. "Implementing Practical Electrical Glitching Attacks." In: BlackHat, 2015.

[126] Guillaume Barbu, Guillaume Duc, and Philippe Hoogvorst. "Java Card Operand Stack: Fault Attacks, Combined Attacks and Countermeasures." In: *Smart Card Research and Advanced Applications - 10th IFIP WG 8.8/11.2 International Conference, CARDIS 2011, Leuven, Belgium, September 14-16, 2011, Revised Selected Papers*. Ed. by Emmanuel Prouff. Vol. 7079. Lecture Notes in Computer Science. Springer, 2011, pp. 297–313. DOI:

10.1007/978-3-642-27257-8_19. URL: https://doi.org/10.1007/978-3-642-27257-8_19.

[127] Guillaume Bouffard, Julien Iguchi-Cartigny, and Jean-Louis Lanet. "Combined Software and Hardware Attacks on the Java Card Control Flow." In: *Smart Card Research and Advanced Applications - 10th IFIP WG 8.8/11.2 International Conference, CARDIS 2011, Leuven, Belgium, September 14-16, 2011, Revised Selected Papers*. Ed. by Emmanuel Prouff. Vol. 7079. Lecture Notes in Computer Science. Springer, 2011, pp. 283–296. DOI: 10.1007/978-3-642-27257-8_18. URL: https://doi.org/10.1007/978-3-642-27257-8_18.

[128] Julien Lancia. "Java Card Combined Attacks with Localization-Agnostic Fault Injection." In: *Smart Card Research and Advanced Applications - 11th International Conference, CARDIS 2012, Graz, Austria, November 28-30, 2012, Revised Selected Papers*. Ed. by Stefan Mangard. Vol. 7771. Lecture Notes in Computer Science. Springer, 2012, pp. 31–45. DOI: 10.1007/978-3-642-37288-9_3. URL: https://doi.org/10.1007/978-3-642-37288-9_3.

[129] Ramesh Karri, Grigori Kuznetsov, and Michael Gössel. "Parity-Based Concurrent Error Detection of Substitution-Permutation Network Block Ciphers." In: *Cryptographic Hardware and Embedded Systems - CHES 2003, 5th International Workshop, Cologne, Germany, September 8-10, 2003, Proceedings*. Ed. by Colin D. Walter, Çetin Kaya Koç, and Christof Paar. Vol. 2779. Lecture Notes in Computer Science. Springer, 2003, pp. 113–124. DOI: 10.1007/978-3-540-45238-6_10. URL: https://doi.org/10.1007/978-3-540-45238-6_10.

[130] Kaijie Wu et al. "Low Cost Concurrent Error Detection for the Advanced Encryption Standard." In: *Proceedings 2004 International Test Conference (ITC 2004), October 26-28, 2004, Charlotte, NC, USA*. IEEE Computer Society, 2004, pp. 1242–1248. DOI: 10.1109/TEST.2004.1387397. URL: https://doi.org/10.1109/TEST.2004.1387397.

[131] Daniele Rossi et al. "Error correcting code analysis for cache memory high reliability and performance." In: *Design, Automation and Test in Europe, DATE 2011, Grenoble, France, March 14-18, 2011*. IEEE, 2011, pp. 1620–1625. DOI: 10.1109/DATE.2011.5763257. URL: https://doi.org/10.1109/DATE.2011.5763257.

[132] Guido Bertoni et al. "Error Analysis and Detection Procedures for a Hardware Implementation of the Advanced Encryption Standard." In: *IEEE Trans. Computers* 52.4 (2003), pp. 492–505. DOI: 10.1109/TC.2003.1190590. URL: https://doi.org/10.1109/TC.2003.1190590.

[133] Mark G. Karpovsky, Konrad J. Kulikowski, and Alexander Taubin. "Differential Fault Analysis Attack Resistant Architectures for the Advanced Encryption Standard." In: *Smart Card Research and Advanced Applications VI, IFIP 18th World*

Computer Congress, TC8/WG8.8 & TC11/WG11.2 Sixth International Conference on Smart Card Research and Advanced Applications (CARDIS), 22-27 August 2004, Toulouse, France. Ed. by Jean-Jacques Quisquater et al. Vol. 153. IFIP. Kluwer/Springer, 2004, pp. 177–192. DOI: 10.1007/1-4020-8147-2_12. URL: https://doi.org/10.1007/1-4020-8147-2_12.

[134] Mark G. Karpovsky, Konrad J. Kulikowski, and Alexander Taubin. "Robust Protection against Fault-Injection Attacks on Smart Cards Implementing the Advanced Encryption Standard." In: *2004 International Conference on Dependable Systems and Networks (DSN 2004), 28 June - 1 July 2004, Florence, Italy, Proceedings*. IEEE Computer Society, 2004, pp. 93–101. DOI: 10.1109/DSN.2004.1311880. URL: https://doi.org/10.1109/DSN.2004.1311880.

[135] Laurie Genelle, Christophe Giraud, and Emmanuel Prouff. "Securing AES Implementation against Fault Attacks." In: *Sixth International Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2009, Lausanne, Switzerland, 6 September 2009*. Ed. by Luca Breveglieri et al. IEEE Computer Society, 2009, pp. 51–62. DOI: 10.1109/FDTC.2009.29. URL: https://doi.org/10.1109/FDTC.2009.29.

[136] Marc Joye, Pascal Manet, and Jean-Baptiste Rigaud. "Strengthening hardware AES implementations against fault attacks." In: *IET Information Security* 1.3 (2007), pp. 106–110. DOI: 10.1049/iet-ifs:20060163. URL: https://doi.org/10.1049/iet-ifs:20060163.

[137] Jacques J. A. Fournier et al. "Design and characterisation of an AES chip embedding countermeasures." In: *IJIEI* 1.3/4 (2011), pp. 328–347. DOI: 10.1504/IJIEI.2011.044101. URL: https://doi.org/10.1504/IJIEI.2011.044101.

[138] Victor Lomné, Thomas Roche, and Adrian Thillard. "On the Need of Randomness in Fault Attack Countermeasures - Application to AES." In: *2012 Workshop on Fault Diagnosis and Tolerance in Cryptography, Leuven, Belgium, September 9, 2012*. Ed. by Guido Bertoni and Benedikt Gierlichs. IEEE Computer Society, 2012, pp. 85–94. DOI: 10.1109/FDTC.2012.19. URL: https://doi.org/10.1109/FDTC.2012.19.

[139] Tyler K. Bletsch, Xuxian Jiang, and Vincent W. Freeh. "Mitigating code-reuse attacks with control-flow locking." In: *Twenty-Seventh Annual Computer Security Applications Conference, ACSAC 2011, Orlando, FL, USA, 5-9 December 2011*. Ed. by Robert H'obbes' Zakon, John P. McDermott, and Michael E. Locasto. ACM, 2011, pp. 353–362. DOI: 10.1145/2076732.2076783. URL: https://doi.org/10.1145/2076732.2076783.

[140] Jean-François Lalande, Karine Heydemann, and Pascal Berthomé. "Software Countermeasures for Control Flow Integrity of Smart Card C Codes." In: *Computer Security - ESORICS 2014 - 19th European Symposium on Research in Computer*

Security, Wroclaw, Poland, September 7-11, 2014. Proceedings, Part II. Ed. by Miroslaw Kutylowski and Jaideep Vaidya. Vol. 8713. Lecture Notes in Computer Science. Springer, 2014, pp. 200–218. DOI: 10.1007/978-3-319-11212-1_12. URL: https://doi.org/10.1007/978-3-319-11212-1_12.

[141] Julien Proy et al. "Compiler-Assisted Loop Hardening Against Fault Attacks." In: *TACO* 14.4 (2017), 36:1–36:25. DOI: 10.1145/3141234. URL: https://doi.org/10.1145/3141234.

[142] Guillaume Bouffard, Bhagyalekshmy Thampi, and Jean-Louis Lanet. "Security Automaton to Mitigate Laser-based Fault Attacks on Smart Cards." In: *International Journal of Trust Management in Computing and Communications* 2 (Jan. 2014), pp. 185–205. DOI: 10.1504/IJTMCC.2014.064158.

[143] Ronald De Keulenaer et al. "Link-time smart card code hardening." In: *Int. J. Inf. Sec.* 15.2 (2016), pp. 111–130. DOI: 10.1007/s10207-015-0282-0. URL: https://doi.org/10.1007/s10207-015-0282-0.

[144] Nahmsuk Oh, Philip P. Shirvani, and Edward J. McCluskey. "Control-flow checking by software signatures." In: *IEEE Trans. Reliab.* 51.1 (2002), pp. 111–122. DOI: 10.1109/24.994926. URL: https://doi.org/10.1109/24.994926.

[145] A. Murat Fiskiran and Ruby B. Lee. "Runtime Execution Monitoring (REM) to Detect and Prevent Malicious Code Execution." In: *22nd IEEE International Conference on Computer Design: VLSI in Computers & Processors (ICCD 2004), 11-13 October 2004, San Jose, CA, USA, Proceedings*. IEEE Computer Society, 2004, pp. 452–457. DOI: 10.1109/ICCD.2004.1347961. URL: https://doi.org/10.1109/ICCD.2004.1347961.

[146] Yubin Xia et al. "CFIMon: Detecting violation of control flow integrity using performance counters." In: *IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2012, Boston, MA, USA, June 25-28, 2012*. Ed. by Robert S. Swarz, Philip Koopman, and Michel Cukier. IEEE Computer Society, 2012, pp. 1–12. DOI: 10.1109/DSN.2012.6263958. URL: https://doi.org/10.1109/DSN.2012.6263958.

[147] Yuqun Chen et al. "Oblivious Hashing: A Stealthy Software Integrity Verification Primitive." In: *Information Hiding, 5th International Workshop, IH 2002, Noordwijkerhout, The Netherlands, October 7-9, 2002, Revised Papers*. Ed. by Fabien A. P. Petitcolas. Vol. 2578. Lecture Notes in Computer Science. Springer, 2002, pp. 400–414. DOI: 10.1007/3-540-36415-3_26. URL: https://doi.org/10.1007/3-540-36415-3_26.

[148] Olga Goloubeva et al. "Soft-Error Detection Using Control Flow Assertions." In: *18th IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT 2003), 3-5 November 2003, Boston, MA, USA, Proceedings*. IEEE Computer Society, 2003, pp. 581–588. DOI: 10.1109/DFTVS.2003.1250158. URL: https://doi.org/10.1109/DFTVS.2003.1250158.

[149] Guillaume Bouffard, Bhagyalekshmy N. Thampi, and Jean-Louis Lanet. "Detecting Laser Fault Injection for Smart Cards Using Security Automata." In: *Security in Computing and Communications - International Symposium, SSCC 2013, Mysore, India, August 22-24, 2013. Proceedings*. Ed. by Sabu M. Thampi et al. Vol. 377. Communications in Computer and Information Science. Springer, 2013, pp. 18–29. DOI: 10.1007/978-3-642-40576-1_3. URL: https://doi.org/10.1007/978-3-642-40576-1_3.

[150] Ahmadou Sere, Julien Iguchi-Cartigny, and Jean-Louis Lanet. "Evaluation of Countermeasures Against Fault Attacks on Smart Cards." In: 5 (Jan. 2011).

[151] Bogdan Nicolescu, Yvon Savaria, and Raoul Velazco. "SIED: Software Implemented Error Detection." In: *18th IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT 2003), 3-5 November 2003, Boston, MA, USA, Proceedings*. IEEE Computer Society, 2003, pp. 589–596. DOI: 10.1109/DFTVS.2003.1250159. URL: https://doi.org/10.1109/DFTVS.2003.1250159.

[152] David El-Baze, Jean-Baptiste Rigaud, and Philippe Maurine. "A fully-digital EM pulse detector." In: *2016 Design, Automation & Test in Europe Conference & Exhibition, DATE 2016, Dresden, Germany, March 14-18, 2016*. Ed. by Luca Fanucci and Jürgen Teich. IEEE, 2016, pp. 439–444. URL: http://ieeexplore.ieee.org/document/7459351/.

[153] Asier Goikoetxea Yanci, Stephen Pickles, and Tughrul Arslan. "Detecting Voltage Glitch Attacks on Secure Devices." In: *2008 ECSIS Symposium on Bio-inspired, Learning, and Intelligent Systems for Security, BLISS 2008, Edinburgh, UK, 4-6 August 2008*. Ed. by Adrian Stoica et al. IEEE Computer Society, 2008, pp. 75–80. DOI: 10.1109/BLISS.2008.26. URL: https://doi.org/10.1109/BLISS.2008.26.

[154] J. J. L. Franco et al. "Ring oscillators as thermal sensors in FPGAs: Experiments in low voltage." In: *2010 VI Southern Programmable Logic Conference (SPL)*. 2010, pp. 133–137.

[155] Rodrigo Possamai Bastos et al. "A bulk built-in sensor for detection of fault attacks." In: *2013 IEEE International Symposium on Hardware-Oriented Security and Trust, HOST 2013, Austin, TX, USA, June 2-3, 2013*. IEEE Computer Society, 2013, pp. 51–54. DOI: 10.1109/HST.2013.6581565. URL: https://doi.org/10.1109/HST.2013.6581565.

[156] David El-Baze, Jean-Baptiste Rigaud, and Philippe Maurine. "An Embedded Digital Sensor against EM and BB Fault Injection." In: *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2016, Santa Barbara, CA, USA, August 16, 2016*. IEEE Computer Society, 2016, pp. 78–86. DOI: 10.1109/FDTC.2016.14. URL: https://doi.org/10.1109/FDTC.2016.14.

[157] Joseph Unsworth and Michael Mapson. *Electro-active Cradle Circuits For The Detection Of Access Or Penetration*. WO/1991/005306. 1991. URL: https://patentscope.wipo.int/search/en/detail.jsf?docId=WO1991005306&tab=PCTBIBLIO.

[158] Thomas Trouchkine, Guillaume Bouffard, and Jessy Clediere. "Fault Injection Characterization on modern CPUs – From the ISA to the Micro-Architecture." In: *WISTP 2019, Paris, France*. 2019.

[159] Thomas Trouchkine et al. "Electromagnetic Fault Injection against a complex CPU, toward a new micro-architectural fault models." In: *Journal of Cryptographic Engineering (JCEN)*. 2020.

[160] Thomas Trouchkine, Guillaume Bouffard, and Jessy Clediere. "Fault Characterization and Exploitation on modern CPUs." In: 2020.

[161] Kaneyuki Kurokawa. "Power waves and the scattering matrix." In: *Microwave Theory and Techniques, IEEE Transactions on* 13.2 (1965). Lien, pp. 194–202. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1125964 (visited on 09/14/2015).

[162] Thomas Trouchkine, Guillaume Bouffard, and Jessy Clédière. "SoCs Security evaluation against fault attacks." In: 2020.

[163] "Advanced Encryption Standard (AES)." In: 2001. URL: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf.

[164] "Intel® 64 and IA-32 ArchitecturesOptimization Reference Manual." In: 2020. URL: https://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-optimization-reference-manual.html.

[165] "Arm® Cortex®-A75 Software Optimization Guide." In: 2018. URL: https://developer.arm.com/documentation/101398/0200.

[166] Clément Gaine et al. "Electromagnetic Fault Injection as a New Forensic Approach for SoCs." In: *IEEE WIFS 2020*. 2020.

[167] Thomas Trouchkine, Guillaume Bouffard, and Jessy Clediere. "EM Injection Vs. Modern CPU - Fault Characterization And AES Differential Fault Analysis." In: *Comptatibilité électromagnétique France 2020*. 2020.

[168] Thomas Trouchkine, Guillaume Bouffard, and Jessy Clediere. "Perturbation attack on modern CPUs, from the fault model to the exploitation." In: *Journée thématique sur les attaques par injection de fautes (JAIF)*. 2020.

[169] Guillaume Bouffard et al. "Radically secure computing." In: *SILM seminar 2020*. 2020.

[170] Thomas Trouchkine et al. "Do not trust modern System-on-Chips." In: *PHISIC 2019*. 2019.

[171] Thomas Trouchkine et al. “Do not trust modern System on Chip.” In: *Journée thématique sur les attaques par injection de fautes (JAIF)*. 2019.

[172] Thomas Trouchkine. “EM fault injection on Raspberry Pi 3 and ZedBoard.” In: *GDR SoC2 research group*. 2018.

[173] Thomas Trouchkine. “Problems and state of the art of faults injection on Systems on Chip.” In: *PHISIC 2018*. 2018.

ABSTRACT

Since the democratization of mobile devices, sensitive operations like payment, identification or healthcare, usually done using security evaluated smartcards, are handled by these devices. However, mobile devices neither are designed for security nor security evaluated. Therefore, their resistance against powerful attacks, like physical attacks is questionable.
In this thesis, we aim at evaluating the security of mobile devices against physical attacks, in particular perturbation attacks. These attacks aims at modifying the execution environment of the device to induce bugs during its computation. These bugs are called faults. These faults can compromise the security of a device by allowing the cryptanalysis of its secret or forcing an unauthorized authentication for instance.
Mobile devices are powered by modern processors, which are the heart of this work, and are never evaluated against fault attacks. However, our knowledge about fault attacks on smartcards is not relevant as the processors powering smartcards are way less complex, in terms of number of modules, technology node and optimization mechanisms, than modern processors.
Regarding this situation, we aim at providing rationals on the security of modern processors against fault attacks by defining a fault characterization method, using it on representative modern processors and analyzing classical security mechanisms against the characterized faults.
We characterized three devices, namely the BCM2837, BCM2711b0 and the Intel Core i3-6100T against fault attacks using two different injection mediums: electromagnetic perturbations and a laser. We determined that these devices, despite having different architecture and using different mediums are faulted in similar ways. Most of the time, a perturbation on these devices modify their executed instructions.
As this is a powerful fault, we also analyzed classical security mechanisms embedded in such devices. We successfully realized a differential fault analysis on the AES implementation of the OpenSSL library, which is used in every Linux based operating system. We also analyzed the Linux user authentication process involved in the `sudo` program. This work highlights the lack of tools to efficiently analyze Linux programs, which are rather complex with dynamic linking mechanisms, against fault attacks.

RÉSUMÉ

De nos jours, nos appareils mobiles sont utilisés pour réaliser des opérations sensibles telles que du paiement, de l'identification ou la gestion de services santé. Historiquement, ces opérations sont réalisées par des appareils conçus et évalués pour résister à diverses attaques: les éléments sécurisés. En revanche, les appareils mobiles sont conçus pour fournir la meilleure performance possible et ne subissent aucune évaluation de sécurité. Cet état de fait interroge sur la résistance de ces appareils face aux attaques classiques contre lesquelles se protègent les éléments sécurisés.
Parmi ces attaques, nous nous proposons, dans cette thèse, d'étudier les attaques par perturbations. Ces attaques consistent à modifier les conditions d'exécution du circuit ciblé afin d'induire des erreurs dans son fonctionnement. Ces erreurs volontaires, communément appelées fautes, permettent de créer des failles dans la cible pouvant aller jusqu'à la cryptanalyse d'un algorithme de chiffrement ou l'authentification d'un utilisateur non autorisé.
Bien que ces méthodes d'attaques soient connues et étudiées sur les éléments sécurisés, les appareils modernes reposent sur des processeurs modernes présentant des différences par rapport aux processeur des éléments sécurisés. Cela peut être le nombre de module qu'ils embarquent, leur finesse de gravure ou des optimisations.
L'impact de ces différences sur la sécurité des processeur n'a pas été étudié en prenant en compte la possibilité d'induire des fautes. C'est ce que nous réalisons dans cette thèse. Nous définissons une méthode permettant de caractériser les effets de perturbations sur un processeur moderne que nous appliquons sur trois processeurs représentatifs des appareils existants: le BCM2837, le BCM2711b0 et l'Intel Core i3-6100T. Nous avons également utilisés deux moyens de perturbation classiques: l'injection d'onde électromagnétique et l'utilisation d'un laser. L'étude de ces cibles, en variant les moyens d'injections de faute, nous a permis de déterminer qu'elles réagissent toutes de manière similaire aux différentes perturbations malgré leur différentes architectures. L'effet le plus marquant étant la modification des instructions exécutées.
Ce type de faute est très fort car il permet de modifier une partie du programme exécuté pendant son exécution. Vérifier le programme avant de l'exécuter ne protège en rien face à ce type de fautes, par exemple. C'est pourquoi nous avons également étudié la résistance des mécanismes de sécurité présents dans ces cibles face à ce type de faute. Nous avons notamment réussi à cryptanalyser l'implémentation de l'algorithme de chiffrement AES de la bibliothèque OpenSSL, très utilisé dans les systèmes utilisant Linux. Nous avons également étudié la résistance du mécanisme d'authentification des utilisateurs d'un système Linux en regardant le programme `sudo`. Cette étude nous a, en particulier, révélé que la communauté manque d'outils efficace pour analyser ce type de programmes face aux fautes. En effet, les programmes s'exécutent dans un environnement Linux bénéficient de nombreux mécanismes liés au noyau Linux qui rendent l'exécution d'un programme difficile à étudier.